
O Proszę nie czytać tego !

Zaprzyjaźnijmy się !

Jeśli mamy ze sobą spędzić parę godzin, to proponuję – przejdźmy na „ty”. Moje nazwisko zobaczyłeś już na okładce, dodam więc tylko, że książka ta powstała na podstawie moich doświadczeń jako programisty w Hahn-Meitner Institut^{†)} w Berlinie – dawniej Zachodnim. Jestem oczarowany jasnością, prostotą i logiką programowania w języku C++ i dlatego proponuję Ci spędzenie ze mną paru chwil w krainie programowania obiektowo orientowanego.

Wstępne założenie

Książka ta jest napisana z myślą o czytelnikach, którzy znają choćby jeden język programowania – wszystko jedno, czy to będzie BASIC czy język C. Muszę przyznać, że początkowo odczuwałem pokusę napisania książki dla czytelników znających język C, jednak gdy zorientowałem się, że na polskim rynku „C” ma opinię jakiegoś trudnego języka dla specjalistów – postanowiłem zacząć prawie od zera.

Prawdę mówiąc obecnie język C jest już językiem starym. Na przestrzeni lat dostrzegano jego liczne niedoskonałości. Powstała też rewolucyjna idea programowania obiektowo orientowanego, która oprócz swej elegancji ma aspekt wymierny – pozwala oszczędzić tysiące dolarów wydawanych na pracę zespołów programistów pracujących nad dużymi projektami. Środowisko programistów od dawna oczekiwało na pojawienie się czegoś (jeszcze) lepszego niż C. Gdy pojawił się język C++ łączący w sobie prostotę programowania klasycznego C i możliwość programowania obiektowo orientowanego – przyjęty został z ogromnym entuzjazmem.

†) instytut badań jądrowych

Czym jest programowanie techniką obiektowo orientowaną? Dlaczego to taka sensacja i rewolucja?

Najkrócej mówiąc polega ono na zmianie sposobu myślenia o programowaniu. Przypomnij sobie jak to jest, gdy musisz napisać program w znanych Ci do tej pory językach programowania. Wszystko jedno czy to będzie program na sterowanie lotem samolotu, czy program na obsługę eksperymentu fizycznego. Otóż najpierw starasz się ten kawałek rzeczywistości, który masz oprogramować, wyrazić za pomocą liczb – zmiennych występujących w programie. Dziesiątki takich zmiennych są (luźno) rozrzucone po Twoim programie, a samo napisanie programu, to napisanie sekwencji działań na tych liczbach.

Ten sam problem w ujęciu obiektowo orientowanym rozwiązuje się tak, że w programie buduje się małe i większe modele obiektów świata realnego, po czym wyposaża się je w zachowania i po prostu pozwala się im działać. Obiektem może być ster kierunku samolotu, czy też układ elektroniczny przetwarzający jakiś sygnał. Gdy takie modele już w programie istnieją, wydają sobie nawzajem polecenia – np. obiekt drążek sterowy wydaje polecenie sterowi kierunku, aby wychylił się 7 stopni w prawo.

Obiekty nie pouczają się *jak* coś trzeba zrobić, tylko mówią *co* trzeba zrobić. Programowanie tą techniką – poza tym, że jest logiczne, bo odzwierciedla relacje istniejące w świecie rzeczywistym – programowanie tą techniką dostarcza po prostu dobrej zabawy.

Argument o dobrej zabawie

nie przekonałby oczywiście nigdy szefów dużych zespołów programistów — dla nich najważniejszy jest fakt, że ta technika pozwala, by programiści znali się tylko na swoim kawałku pracy bez konieczności opanowywania całości olbrzymiego projektu. Pozwala ona też na bardzo łatwe wprowadzanie poważnych modyfikacji do programu – bez konieczności angażowania w tę pracę całości zespołu. Dla tego ostatniego – programowanie obiektowo orientowane jest wręcz jakby stworzone. Te cechy są łatwo przeliczane na pieniądze – nakłady finansowe na pracę zespołu.

Jest także powód dla którego język C++ zrobił karierę nawet wśród programistów pracujących pojedynczo, a także wśród amatorów. Otóż język ten pozwala na łagodne przejście z klasycznych technik programowania na technikę obiektowo orientowaną.

Ta cecha nazywana jest hybrydowością języka C++. W programie można technikę OO stosować w takim stopniu w jakim się ją opanowało. Tak też dzieje się najczęściej – programiści wybierają sobie z niego najpierw tylko same „rodzynki” i je stosują, a w każdym następnym programie sięgają po coś więcej. W języku C++ można pisać nawet programy nie mające z techniką OO nic wspólnego. Tylko, że to tak, jakby zwiedzać Granadę mając zamknięte oczy^{†)}.

†) „...kobieto, daj jałmużnę ślepcowi, bo nie ma większego nieszczęścia niż być ślepcem w Granadzie...”

Dość na tym. O technice OO porozmawiały jeszcze wielokrotnie. Teraz chciałbym wytłumaczyć się z paru spraw.

Dlaczego ta książka jest taka gruba?

Nie wynika to wcale z faktu by język C++ był tak trudny. Uznałem tylko, że to, co uczy naprawdę – to przykłady. W książce więc oprócz „teorii” są setki przykładowych programów, a każdy jest szczegółowo omówiony. Przykłodom towarzyszą wydruki pokazujące wygląd ekranu po wykonaniu programu. To wszystko właśnie sprawia, że książka jest tak obszerna. Jednak wydruki te załączam w przeświadczeniu, że często łatwiej zorientować się „co program robi” – rzucając okiem na taki wydruk (ekran). Dopiero potem radzę analizować sam program.

Do wykonania przykładowych programów użyłem kompilatora Borland C++ v 3.1 (komputer IBM PC) gdyż sądzę, że jest to kompilator, z którym przeciętny polski czytelnik może się spotkać najczęściej. Oczywiście w książce zajmujemy się samym językiem C++ – więc powinna Ci ona pomóc niezależnie od tego, z którym komputerem i typem kompilatora masz do czynienia.

Wersja języka

Sam język C++ ciągle się unowocześnia. Opisuję go tu bez tzw. *templates* i *exception handling*, jako że w tej wersji istnieją na razie tytułem eksperymentu[†]. Absolutnym autorytetem w sprawie języka była dla mnie książka (twórcy tego języka) Bjarne’a Stroustrup’a i Margaret A. Ellis – *The Annotated C++ Reference Manual*. W razie jakichkolwiek niejasności odsyłam do niej. (Uprzedzam jednak, że jest napisana w bardzo sformalizowanym stylu i nie jest pomyślana jako podręcznik do nauki).

W tekście czasem wspominam o wersji języka. Pamiętać należy, że termin *wersja języka* nie ma nic wspólnego z *wersją kompilatora*. Wersja kompilatora to wersja konkretnego dostarczonego Ci produktu. Tymczasem wersja języka to reguły gry, według których ten kompilator postępuje.

Pióro

Ktoś kiedyś powiedział, że są dwa style pisania – pierwszy to: „popatrzcie jaki ja jestem mądry” – a drugi to: „popatrzcie jakie to proste”. Ja w tej książce wybieram ten drugi wariant w przeświadczeniu, że prędzej zaprowadzi do celu. Z drugiej strony wiem, że bezpośredni, wręcz kolokwialny styl tej książki, zupełnie naturalny w książkach na Zachodzie – w Polsce może zaskoczyć niektórych czytelników.

†) Uwaga do bieżącego wydania: Ponieważ dziś już wiadomo, że ten eksperyment był udany i owe pojęcia weszły do C++ na stałe – poświęciłem im osobną książkę pod tytułem „Pasja C++”. Są w niej omówione szablony funkcji, szablony klas, klasy pojemnikowe i obsługa sytuacji wyjątkowych. Zajrzyj do niej jeśli już przeczytasz „Symfonię C++” i ją polubisz.

Dla kogo jest ta książka?

Pisząc tę książkę musiałem najpierw określić sobie czytelnika, dla którego jest ta książka. Otóż jest ona dla tzw. szerokiego grona czytelników. Pisałem ją tak, by była podręcznikiem dla czternastoletniego programisty amatora, jak i dla zawodowego informatyka. Trudno te sprawy pogodzić, więc są w niej kompromisy w obu kierunkach. Co do jednego z takich kompromisów mam największej obaw:

Czołem bracia angliści!

Gdy słyszałem, jak początkujący programiści wymawiają występujące w języku C++ angielskie słowa takie jak np. „unsigned”, „volatile”, „width” — postanowiłem w miejscach, gdzie się pojawiają po raz pierwszy – zamieścić adnotacje o ich wymowie. Wymowa podana jest nie w transkrypcji fonetycznej, ale za pomocą polskich liter. Wiem, że fakt ten może razić wielu czytelników. Proszę wtedy o wyrozumiałość – pamiętajcie, że ta książka ma służyć również małuczkim. Wymowa podana jest chyłkiem – w przypisie, w nawiasie, a w dodatku jeszcze w cudzysłowie – więc nie trzeba jej koniecznie czytać. Spodziewam się, że fakt zamieszczenia wymowy nie będzie przeszkadzał Czytelnikom, którzy językiem angielskim posługują się na codzień od lat – ale na pewno wzbudzi protest tych, którzy angielskiego nauczyli się przedwczoraj.

A swoją drogą, to nawet wśród moich kolegów fizyków i programistów – nie spotkałem poprawnie wymawiających słowo: „width”.

A teraz o strukturze tej książki

Można ją podzielić na dwie zasadnicze części – tę klasyczną, opisującą zwykłe narzędzia programowania, i drugą (zaczynającą się od rozdziału o klasach) opisującą narzędzia do programowania obiektowo orientowanego. Po tym następuje rozdział omawiający operacje wejścia/wyjścia – czyli sposoby pracy z takimi urządzeniami zewnętrznymi jak klawiatura, ekran, dyski magnetyczne. Wreszcie następuje rozdział o projektowaniu programu obiektowo orientowanego – zawierający szczegółowy instruktaż. Uznałem te sprawy za bardzo ważne, gdyż często programista mając w ręce to doskonałe narzędzie programowania, jakim jest C++, nie wie, co z nim począć.

Metodą kolejnych przybliżeń

Nie liczę na to, że czytając tę książkę zrozumiesz wszystko od razu. Wręcz przeciwnie – w tekście wielokrotnie sugeruję, byś opuścił niektóre paragrafy przy pierwszym czytaniu książki. Nie wszystkie aspekty są ważne już na samym początku nauki. Lepiej mieć najpierw ogólny pogląd, a dopiero potem zagłębiać się w trudniejsze szczegóły. Licząc na to, że po pierwszym czytaniu całości w niektóre miejsca wrócisz – pewne fragmenty tekstu opatrzyłem adnotacją: „dla wtajemniczonych”. Są tam uwagi wybiegające nieco do przodu, ale dotyczące spraw, które po pierwszym czytaniu będziesz już przecież znał.

Nie szanuj tej książki. Przygotuj sobie kolorowe (tzw. niekryjące) flamastry i czytając zakreślaj ważne dla Ciebie wyrazy czy myśli. Na marginesach pisz ołówkiem swoje uwagi i komentarze. Chociażby miały to być teksty typu „Co za bzdura!” albo „Nie rozumiem dlaczego ...” albo „porównaj dwie strony

wcześniej". Ta aktywność zaprocentuje Ci natychmiast, bo mając tak osobisty stosunek do tekstu, łatwiej koncentrować na nim uwagę – co jest warunkiem *sine qua non* szybkiej nauki.

Jeśli znasz język C „klasyczny” (czyli tzw. ANSI C)

to zapewne pierwsze rozdziały będą dla Ciebie wyraźnie za łatwe. Mimo to radzę je przejrzeć chociaż pobieżnie, gdyż występują pewne różnice w językach C i C++ – nawet na tym etapie (oczywiście na korzyść C++).

Gorąco natomiast zachęcam do uważnego przeczytania rozdziału o wskaźnikach. Z doświadczenia wiem, że te sprawy zna się zwykle najgorzej. Tymczasem zarówno w klasycznym C jak i w C++ wskaźnik jest bardzo ważnym i pożytecznym narzędziem. Dlatego ten rozdział tak rozbudowałem.

Podziękowania

Winien jestem wdzięczność przyjaciółom, którzy w jakiś sposób przyczynili się do powstania tej książki. Byli to: Sven Bläser, Olaf Boebel, Johannes Diemer, Jeffrey Erxmeyer, Bengt Skogvall, Kai Sommer, Klaus Spohr, Joachim Steiger i Detlef Weidenhammer. Zupełnie wyjątkowe podziękowania należą się Sycylijczykowi Pierpaolo Figuera za wiele godzin, które mi poświęcił. Ponieważ ta książka pisana była od razu po polsku, dlatego żaden z nich nigdy jej nie czytał. Wszystko, co dobre w tej książce, jest jednak ich zasługą, wszystko co złe – to wyłącznie moja wina. Jestem winien wdzięczność pierwszym czytelnikom tej książki: Mirkowi Zięblińskiemu i Bożenie Potempie. Ich uwagi krytyczne sprawiły, że książka stała się lepsza, ich uwagi pozytywne – zawsze wprawiały mnie w dobry nastrój.

Oczywiście na pewno niejedno da się w tej książce poprawić. Jeśli będziesz miał jakies uwagi, to proszę przyslij mi je pocztą elektroniczną na adres:

grebosz@bron.ifj.edu.pl

lub pocztą tradycyjną na adres wydawnictwa. Z góry wyrażam moją wdzięczność za nawet najdrobniejsze przyczynki.

Kod źródłowy przykładowych programów z tej książki można sobie łatwo sprowadzić z moich stron WWW w Internecie. Jej obecny adres to:

<http://www.ifj.edu.pl/~grebosz>

Nawet jeśli ten adres się kiedyś zmieni – łatwo znaleźć nową lokalizację robiąc tzw. search hasła "SYMFONIA C++", lub mojego nazwiska.

Na koniec wypada mi się wytłumaczyć z tytułu tego wstępu. Chciałem tu powiedzieć parę ważnych, ale i trochę nudnych spraw. Wiedząc, że czytelnicy najczęściej opuszczają wstępy – dałem taki tytuł podejrzewając, że zakazany owoc najlepiej smakuje.



1 Startujemy !

Aby pisać, trzeba pisać.... – Taką radę zwykło się dawać początkującym literatom. Tę samą zasadę można zastosować do piszących programy. Programować w danym języku można nauczyć się tylko pisząc w nim programy. Dlatego bez dalszych wstępów napiszmy nasz

1.1 Pierwszy program

```
#include <iostream.h>
main()
{
    cout << "Witamy na pokładzie" ;
}
```

 **Wykonanie tego programu spowoduje pojawienie się na ekranie tekstu:**

Witamy na pokładzie

 **Przyjrzyjmy się bliżej temu programowi**

W każdym programie w języku C++ musi być specjalna funkcja zwana `main`.^{†)} Od tej funkcji zaczyna się wykonywanie programu. Treść tej funkcji – jej ciało – czyli innymi słowy instrukcje wykonywane w ramach tej funkcji – zawarte są między dwoma nawiasami klamrowymi: { }

^{†)} `main` – ang. główna (czytaj: „mejn”).

Uwaga zecera: jeśli z przyczyn typograficznych przypis nie może pojawić się u dołu strony, należy go szukać na stronie następnej.

W naszym wypadku w funkcji `main` jest tylko jedna instrukcja

```
cout << "Witamy na pokładzie" ;
```

która sprawia, że na standardowym urządzeniu wyjściowym `cout` - czyli po prostu na ekranie - ma się pojawić tekst, zamieszczony tu w cudzysłowie. Skrót `cout`^{†)} wymawia się po polsku *Si-aut*. (Błagam, tylko nie: *Śi-aut*!)

Znaki `<<` oznaczają właśnie akcję, którą ma podjąć `cout` - czyli wyprowadzić na ekran tekst. Umieszczony na końcu średnik jest znakiem końca instrukcji.

Operacje wejścia/wyjścia

Należy tu wyraźnie podkreślić, że operacje związane z wprowadzaniem i wyprowadzaniem informacji na urządzenia takie, jak np. ekran - czyli tzw. operacje wejścia/wyjścia - nie są częścią definicji języka C++. Podprogramy odpowiedzialne za to są w jednej ze standardowych bibliotek, w które zwykle wyposażane są kompilatory. Abyśmy mogli z takiej biblioteki skorzystać w programie, musimy na początku umieścić liniijkę

```
#include <iostream.h>
```

która oznacza, że życzymy sobie, by kompilator przed przystąpieniem do pracy nad dalszymi linijkami programu wstawił^{††)} w tym miejscu tak zwany plik nagłówkowy biblioteki `iostream`.

Dla zainteresowanych dodam, że tenże plik nagłówkowy biblioteki zawiera dokładne deklaracje funkcji bibliotecznych, z których ewentualnie można korzystać. Dzięki temu, że kompilator zapoznaje się z tymi deklaracjami może od tej pory sprawdzać nas czy posługujemy się tymi funkcjami poprawnie. To bardzo korzystna cecha.

Biblioteką tą zajmiemy się bliżej pod koniec tej książki.

Wolny format zapisu programu

A teraz uwaga natury ogólnej. Program pisze się umieszczając kolejne instrukcje w liniijkach jedna pod drugą. Otóż w niektórych językach programowania (np. FORTRAN'ie, BASIC'u) obowiązują ściśle reguły określające pozycję (w liniijkę) na której dany składnik instrukcji może się znaleźć.

Np. w FORTRAN'ie - Jeśli chcemy umieścić znak komentarza, to stawiamy go w kolumnie pierwszej, jeśli ma to być numer etykiety - to do tego służą kolumny 1-5, jeśli chcemy umieścić znak kontynuacji z poprzedniej liniijki - to umieszczamy go w kolumnie szóstej.

Podobnie w BASIC'u - linia instrukcji musi się zacząć od numeru etykiety.

W języku C++ jest inaczej. Język C++ (podobnie jak C) jest językiem o tzw. wolnym formacie. Krótko mówiąc - nie ma żadnych przymusów. Wszystko może znaleźć się w każdym miejscu linii, a nawet zostać rozpisane na 10 liniijek. Poza nielicznymi sytuacjami, w dowolnym miejscu instrukcji można przejść d

†) od ang. C-onsole OUT-put

††) ang. include (czytaj: „inklud“)

nowej linii i tam kontynuować pisanie. To dlatego, że koniec instrukcji określany jest nie przez koniec linii, ale przez średnik, który stawiamy na końcu.

Białe znaki

Wewnątrz instrukcji można postawić dodatkowe znaki spacji i tabulatory, czy nawet znaki nowej linii. Są to tzw. białe znaki – białe, bo na papierze objawiają się jako niezadrukowane. Znaki te napotkane wewnątrz instrukcji są prawie zawsze ignorowane.

Zatem nasza instrukcja równie dobrze mogłaby wyglądać tak:

```
cout <<
    "witamy na pokładzie"
    ;
```

Wstawianie białych znaków służy nie kompilatorowi lecz programiście. Pomaga w tym, żeby program wyglądał czytelnie. Jeśli nam na tym wcale nie zależy, to możemy równie dobrze napisać program tak:

```
#include <iostream.h>
main(){cout<<"witamy na pokładzie";}
```

Nikt rozsądny jednak tak nie robi z dwóch powodów:

- ❖ program staje się wtedy nieprzejrzysty,
- ❖ korzystając z różnych specjalnych narzędzi do uruchamiania programów (tzw. debuggerów)^{†)} mamy możliwość śledzenia wykonywania programu krokowo: linijka za linijką. Dobrze jest więc mieć w jednej linijce tylko jedną instrukcję.

Kompilator i Linker

W ten sposób omówiliśmy sobie pierwszy program. Oczywiście program w takiej postaci, jak napisaliśmy, jest dla komputera niezrozumiały. Musi zostać przetłumaczony na język maszyny. Służy do tego kompilator. Nasz program poddajemy więc kompilacji i otrzymujemy wersję skompilowaną.

Taka skompilowana wersja jest jeszcze niepełna – musi zostać połączona z bibliotekami.

Ten proces łączenia wykonywany jest przez program zwany zwykle *linkerem* (link – ang. łączenie), a w żargonie programistów określane jest to jako *linkowanie*. Nie słyszałem by ktoś nazywał to inaczej. Poprawne, lansowane niegdyś określenie „konsolidacja” i „konsolidowanie” chyba się nie przyjęło. My używać będziemy sformułowania *linkowanie* względnie *łączenie*.

Przyłączenie funkcji bibliotecznych następuje dopiero w czasie linkowania. Nasza dyrektywa (instrukcja) `#include` zapoznała kompilator jedynie z samym nagłówkiem biblioteki. Potrzebny był on po to, by kompilator mógł

†) bug – ang. owad ; stąd debugger (czytaj: „debager”) – jakby: „odpluskwiacz”

sprawdzić poprawność naszego odnoszenia się do biblioteki. Natomiast sama treść funkcji bibliotecznych dołączana jest dopiero na etapie linkowania.

A teraz do dzieła. W rezultacie linkowania otrzymaliśmy program w wersji nadającej się do uruchomienia.

Zachęcam Cię czytelniku, byś teraz spróbował uruchomić nasz program na swoim komputerze. Mimo, że program jest prymitywny. Ważne tu jest, byś opanował technikę kompilacji i linkowania. Niestety nie mogę Ci tu nic pomóc. Istnieje bardzo wiele typów komputerów i wiele różnych typów kompilatorów. Jak Twojego kompilatora używać – musisz przeczytać w swojej dokumentacji lub zapytać kolegę.

Wszystko to nie jest trudne, najczęściej wcale nie musisz myśleć o linkowaniu, bo kompilator sam to uruchamia. Wtedy jest to tylko jedna komenda, a np. w kompilatorze Borland C++ naciśnięcie jednego tylko klawisza.

Uwaga: a kompilatory, które mogą kompilować programy napisane w klasycznym C, a także w C++. Wówczas trzeba im powiedzieć w jakim języku jest napisany program przedstawiony właśnie do kompilacji. Jednym ze sposobów powiedzenia tego może być rozszerzenie nazwy naszego programu. Jeśli rozszerzeniem jest CPP, wówczas kompilator podejmie do pracy jak do programu w języku C++. Czyli nasz program powinien się nazywać na przykład

```
moj_progr.cpp
```



Kiedy już program zadziała poprawnie i na ekranie pojawi się nasz tekst Witamy... wówczas możemy spróbować zmodyfikować ten program. Działania te pozwolą nam bliżej zapoznać się z techniką wypisywania na ekran. To się nam bardzo szybko przyda. A zatem w środek tekstu ujętego w cudzysłów wpiszmy znaki \n

```
"Witamy \nna pokładzie"
```

Zmiana ta powoduje, że tekst wypisany na ekranie wyglądał będzie następująco:

```
Witamy
na pokładzie
```

Znak \n (n – jak: new line, czyli: nowa linia) powoduje, że w trakcie wypisywania tekstu na ekranie następuje przejście do nowej linii i dalszy ciąg tekstu wypisywany jest poniżej.

Ewentualna następna instrukcja wyprowadzania tekstu zacznie go wyprowadzać od miejsca, w którym poprzednia skończyła. Zatem dwie instrukcje

```
cout << "Witamy \nna pokładzie" ;
cout << "Lecimy na " << "wysokosci 3500 stop" ;
```

spowodują pojawienie się na ekranie tekstu

Witamy
na pokładzieLecimy na wysokosci 3500 stop

Nie ma też znaczenia czy napiszemy

```
cout << "lecimy na " ;
cout << "wysokosci 3500 stop" ;
```

czy też może złożymy te instrukcje i zapiszemy krócej

```
cout << "lecimy na " << "wysokosci 3500 stop" ;
```

W obu sytuacjach rezultat na ekranie będzie ten sam:

```
lecimy na wysokosci 3500 stop
```

Dlaczego właściwie możliwe jest tak wygodne składanie – nie mogę Ci jeszcze teraz wyjaśnić. Musisz być cierpliwy i doczytać do rozdziału o bibliotece iostream.

1.2 Drugi program

To było wypisywanie informacji na ekranie. Natomiast z wczytywaniem danych z klawiatury spotykamy się w naszym drugim programie.

Oto on:

```
/* -----
Program na przeliczanie wysokosci podanej
w stopach na wysokosc w metrach.
Cwiczymy tu operacje wczytywania z klawiatury
i wypisywania na ekranie
-----*/
#include <iostream.h>
main()
{
    int          stopy ;                // to do przechowywania
                                        //          liczby stop
    float        metry ;                // do wpisania wyniku
    float        przelicznik = 0.3 ;    // przelicznik:
                                        //          stopy na metry

    cout << "Podaj wysokosc w stopach : " ;
    cin >> stopy ;                      // przyjecie danej
                                        //          z klawiatury

    metry = stopy * przelicznik; // wlasciwe przeliczenie

    cout << "\n" ;                      // to samo co: cout << endl ;

    // -----wypisanie wynikow
    cout << stopy << " stop - to jest : "
        << metry << " metrow\n" ;
}
```


Komentarze

Już na pierwszy rzut oka widać, że w programie pojawiły się opisy w „ludzkim” języku. Są to **komentarze**. Komentarze są tekstami zupełnie ignorowanymi przez kompilator, ale za to są bardzo pożyteczne dla programisty, bo przypominają nam co w danym miejscu programu chcieliśmy zrobić.

W języku C++ komentarze można umieszczać dwojako.

- ❖ Pierwszy sposób to ograniczenie jakiegoś tekstu znakami /* (z lewej) oraz */ (z prawej).
Sposób ten zastosowaliśmy na początku programu. Komentarz taki może się ciągnąć przez wiele linii – to także widzimy w przykładzie.
- ❖ Drugi sposób to zastosowanie znaków // (dwa ukośniki). Kompilator po napotkaniu takiego znaku ignoruje resztę znaków do końca linii – traktując je jako komentarz.

Uwaga:

Komentarze typu /* ... */ nie mogą być w sobie zagnieżdżane.

To znaczy, że niepoprawna jest taka forma

```
/* KOMENTARZ "ZEWNETRZNY"
TAK SIE CIAGNIE /* komentarz wewnetrzny */ DOKONCZENIE
ZEWNETRZNEGO */
```

Chciałoby się powiedzieć: niestety! Całe szczęście niektóre kompilatory, mimo zaleceń ANSI, pozwalają na to[†]). Możliwość zagnieżdżenia komentarzy jest czasem bardzo pomocna.

Wyobraźmy sobie kawałek większego programu, wyposażonego już w komentarze. Nagle chcielibyśmy zrezygnować chwilowo z 10 linii. Usunąć je z procesu kompilacji, ale nie skasować zupełnie. Naturalnym odruchem jest wtedy przerobienie tego fragmentu na komentarz.

Robi się to przez ujęcie tych linii w symbole komentarza: / oraz */ – stawiając pierwszy symbol na początku, a drugi na końcu tych 10 linii. Jeśli jednak w rzeczonym fragmencie programu są już jakieś komentarze typu /* ... */, to kompilator (niepozwalający na zagnieżdżanie ich) uzna to za błąd.*

Jak powiedziałem, postąpi tak kompilator niepozwalający na zagnieżdżanie, bo są kompilatory, które po wybraniu odpowiedniej opcji pozwalają na to.

Ograniczenie o zagnieżdżaniu komentarzy nie dotyczy współzycia komentarzy typu /* ... */ z komentarzami typu //

```
cout << "Uwaga pasazerowie : \n" ;
/* chwilowo rezygnuje z tych dwóch linii -----
cout << "Pali sie drugi silnik \n" ; // opis sytuacji i
cout << "Nie ma szans ratunku \n" ; // niech sie modla
```

[†]) ANSI – American National Standards Institute – amerykański instytut normalizacyjny.

```
cout << "Proszę zapiąć pasy...\n" ;
```

Jeśli Twój kompilator nie pozwala na zagnieżdżanie komentarzy, to możesz sobie poradzić stosując tak zwaną kompilację warunkową. (Patrz rozdz. o preprocesorze, str 122).

Na koniec dobra rada natury ogólnej:

Czas zużyty na pisanie komentarzy nigdy nie jest czasem straconym. Bardzo szybko zauważysz, że czas ten odzyskasz z nawiązką w trakcie uruchamiania programu lub przy późniejszych jego modyfikacjach.

Opisuj znaczenie każdej zmiennej, opisuj funkcje i ich argumenty, opisuj też to, co w danym fragmencie programu robisz. Nawet jeśli wtedy, gdy to piszesz, jest to jeszcze dla Ciebie jasne. Opisuj przede wszystkim wszystkie „kruczki”, które zastosowałeś. Dla samego siebie i dla tych, którzy modyfikować będą Twoje programy wtedy, gdy Ty będziesz już pisał w języku C++ systemy operacyjne.

Zmienne

Dosyć już o komentarzach, wróćmy do naszego drugiego programu. W funkcji `main` zauważamy linijki

```
int      stopy ;
float    metry ;
```

Są to **definicje zmiennych** występujących w programie. Zmiennym tym nadałiśmy nazwy: `stopy` oraz `metry`. Nazwy te są, jak widać, tak przez nas wybrane, by określały zastosowanie zmiennych. Można jednak zapytać ogólniej:

Co może być nazwą w języku C++ ?

Może to być dowolnie długi ciąg liter, cyfr oraz znaków podkreślenia `'_'`. Nazwa nie może się zacząć od cyfry. Małe i wielkie litery w nazwach są rozróżniane.

Nazwa nie może być identyczna z żadnym ze słów kluczowych języka C++. Te słowa kluczowe to:

<code>asm</code>	<code>auto</code>	<code>break</code>	<code>case</code>	<code>catch</code>	<code>char</code>
<code>class</code>	<code>const</code>	<code>continue</code>	<code>default</code>	<code>delete</code>	<code>do</code>
<code>double</code>	<code>else</code>	<code>enum</code>	<code>extern</code>	<code>float</code>	<code>for</code>
<code>friend</code>	<code>goto</code>	<code>if</code>	<code>inline</code>	<code>int</code>	<code>long</code>
<code>new</code>	<code>operator</code>	<code>private</code>	<code>protected</code>	<code>public</code>	<code>register</code>
<code>return</code>	<code>short</code>	<code>signed</code>	<code>sizeof</code>	<code>static</code>	<code>struct</code>
<code>switch</code>	<code>template</code>	<code>this</code>	<code>throw</code>	<code>try</code>	<code>typedef</code>
<code>union</code>	<code>unsigned</code>	<code>virtual</code>	<code>void</code>	<code>volatile</code>	<code>while</code>

Zatem jeśli wymyślamy swoją nazwę, to musimy unikać powyższych słów.

Nazwy są nam potrzebne po to, by za ich pomocą odnosić się w programie do naszych zmiennych, czy stałych - ogólnie mówić będziemy: do obiektów.



W języku C++ każda nazwa musi zostać zadeklarowana zanim zostanie użyta.

Tę złotą myśl możesz napisać sobie na ścianie. Swoją drogą – jeśli o tym zapomnisz, to kompilator Ci natychmiast o tym przypomni.

We wspomnianym fragmencie programu *deklarujemy*, że

- zmienna `stopy` służy do przechowywania liczby typu całkowitego `int`^{†)}.

Deklarujemy też, że

- metry oraz przelicznik są zmiennymi do przechowywania liczby rzeczywistej – inaczej: liczby zmiennoprzecinkowej – `float`^{††)}.

Definicja a deklaracja

Jest subtelna różnica między tymi dwoma pojęciami. Załóżmy, że chodzi o naszą nazwę `stopy`.

Deklaracja w momencie, gdy napotka ją kompilator mówi mu:

„Jakbyś zobaczył kiedyś słowo `stopy`, to wiedz, że oznacza ono obiekt typu całkowitego.”

Definicja zaś mówi mu:

„A teraz zarezerwuj mi w pamięci miejsce na obiekt typu całkowitego o nazwie `stopy`”

Nasz fragment programu zawiera w sobie jednocześnie i jedną i drugą wypowiedź. Rezerwujemy miejsce w pamięci i od razu oznajmiamy co to za obiekt. Zatem:

Definicja jest równocześnie deklaracją. Ale nie odwrotnie.

Może być bowiem deklaracja, która nie jest definicją. Możemy przecież tylko zadeklarować, że konkretna nazwa oznacza obiekt jakiegoś typu, ale obiektu tego w danym miejscu nie definiujemy. Chociażby dlatego, że już jest zdefiniowany w zupełnie innym module programu.

Przy definicji obiektu przelicznik widzimy taki zapis

```
float    przelicznik = 0.3 ;
```

Jest to nie tylko zwykła definicja rezerwująca miejsce w pamięci, ale taka definicja, która to miejsce w pamięci dodatkowo inicjalizuje wpisując tam wartość 0.3

†) integer – ang. liczba całkowita (czytaj: „intidżer”)

††) float (czytaj: „flout”) od słów: floating point – ang. zmienny przecinek.

Wczytywanie danych z klawiatury

Przejdźmy dalej. W naszym programie instrukcja

```
cin >> stopy ;
```

jest operacją związaną z klawiaturą, czyli mówiąc inaczej ze standardowym urządzeniem wejściowym `cin` – skrót od: C-onsole IN-put.^{†)}

Instrukcja ta umożliwia wczytanie z klawiatury liczby. Wartość tej liczby zostaje umieszczona w zmiennej `stopy`. Przy odrobinie wyobraźni można powiedzieć, że to właśnie sugeruje kierunek strzałek >>

Dygresja dla programistów klasycznego C.

Programujących w C klasycznym, ucieszył zapewne fakt tak prostego wprowadzania danych z klawiatury. Nie trzeba tu myśleć kiedy podać do funkcji wczytującej (`scanf`) nazwę zmiennej, a kiedy adres zmiennej. Jeszcze kilka tak miłych niespodzianek czeka nas w C++.

Następne linijki programu to przeliczenie stóp na metry i wydruk wyniku.

Przykładowe wykonanie programu spowoduje wydruk na ekranie poniższego tekstu

(Thustszym drukiem zaznaczone jest echo tekstu wpisywanego przez użytkownika programu.)

```
Podaj wysokosc w stopach : 3500
3500 stop - to jest : 1050 metrow
```

Spójrz jeszcze raz na tekst programu i zauważ jak to się stało, że na ekranie pojawiła się liczba 1050 – będąca obliczoną wartością umieszczoną w zmiennej `metry`. W komentarzach wyjaśnione są poszczególne kroki.

Zwróć też uwagę, na instrukcję

```
cout << "\n" ; // cout << endl ;
```

W komentarzu zaznaczyłem inny sposób zapisu tego samego. Jeśli chcemy na ekran wyprowadzić sam znak `'\n'` to możemy go zastąpić skrótem `endl` co oznacza: end of line (koniec linii). Mimo, że jest to dokładnie tyle samo znaków do wystukania na klawiaturze, to jednak łatwiej się to jakoś pisze. W rozdziale poświęconym operacjom wypisywania na ekran poznamy dalsze takie sztuczki.



Tak więc wyglądał nasz program do przeliczania stóp na metry.

No cóż – pomyślałeś zapewne: –Tyle pracy z wpisywaniem tekstu programu, po to, by otrzymać ten wynik... Prościej byłoby obliczyć to „na piechotę”.

Masz rację. Cud programowania objawia się dopiero w pętlach. Mówiąc szerzej: w instrukcjach sterujących.

†) Skrót ten wymawia się po polsku jako „Si-yn”. Z twardym „S”!

2 Instrukcje sterujące

Są to bardzo przydatne polecenia służące do sterowania przebiegiem programu. Ponieważ założyliśmy, że znasz jakikolwiek inny język programowania, zatem bez dalszych komentarzy przystępujemy do prezentacji takich instrukcji. Występują one w każdym języku programowania i nawet wszędzie mają podobny wygląd.

W instrukcjach sterujących podejmowane są decyzje o wykonaniu tych czy innych instrukcji programu. Decyzje te podejmowane są w zależności od spełnienia lub niespełnienia jakiegoś warunku. Inaczej mówiąc, od prawdziwości lub fałszywości jakiegoś wyrażenia. Najpierw więc wyjaśnijmy sobie co to jest prawda, a co fałsz w języku C++

2.1 Prawda – Fałsz

W języku C++ nie ma specjalnego typu określającego zmienne logiczne – czyli takie, które przyjmują wartości: prawda - fałsz.

Za to do przechowywania takiej informacji nadaje się każdy typ. Zasada jest genialnie prosta:

Sprawdza się czy wartość danego obiektu – np. zmiennej – jest równa zero, czy różna od zera.



Wartość zero – odpowiada stanowi: fałsz

Wartość inna niż zero – odpowiada stanowi: prawda

Nie musi to być nawet zawartość jednego obiektu. Może to być także bardziej skomplikowane wyrażenie, które trzeba obliczyć, aby przekonać się jaka jest jego wartość.

Co ciekawe – wynik nie musi być wcale liczbą. Nawet obiekt przechowujący znaki alfanumeryczne może być w ten sposób sprawdzany. Sprawdza się wów-

czas kod liczbowy złożonego tam znaku. Jeśli jest różny od zera, to wyrażenie odpowiada rezultatowi „prawda”, jeśli kod jest zerowy (czyli tzw. znak NULL) – odpowiada to rezultatowi „fałsz”.

2.2 Instrukcja warunkowa `if`

Instrukcja `if`^{†)} może mieć 2 formy:

```
if (wyrażenie)    instrukcja1 ;
```

lub

```
if (wyrażenie)    instrukcja1 ;
else              instrukcja2 ;
```

Wyrażenie to tutaj coś, co ma jakąś wartość. Może być to po prostu obiekt wybrany przez nas do przechowywania zmiennej logicznej, ale może to być też naprawdę wyrażenie, które najpierw trzeba obliczyć, by w rezultacie tego poznać jego wartość.

Najpierw zatem obliczana jest wartość wyrażenia. Jeśli jest ona niezerowa (prawda), to wykonywana jest instrukcja1. Jeśli wartość wyrażenia jest zero (fałsz), to instrukcja1 nie jest wykonywana.

W drugiej wersji instrukcji `if` widzimy dodatkowo słowo `else`^{††)}, co można przetłumaczyć jako: „w przeciwnym razie”. A zatem jeśli w tej drugiej sytuacji wartość wyrażenia jest niezerowa (prawda), to zostanie wykonana instrukcja1 w przeciwnym razie (`else!`), czyli gdy wartość wyrażenia jest zerowa (fałsz), zostanie wykonana instrukcja2.

Powtarzam – wynik wyrażenia może być różnego typu (całkowity, rzeczywisty itd). Sprawdza się tylko czy jest równy 0 czy nie.

Oto prosty przykład:

```
int i ;                // definicja obiektu int o nazwie i
cout << "Podaj jakas liczbe: " ;
cin >> i ;
if (i - 4) cout << " zmienna i miala wartosc inna niz 4";
else      cout << " zmienna i miala wartosc rowna 4";
```

Założmy, że podaliśmy liczbę 15.

Wyrażeniem było tu: `i-4`. Obliczana jest więc jego wartość $15 - 4 = 11$, a to jest różne od 0 (zatem: prawda), więc wykonana zostaje instrukcja pierwsza.

Gdybyśmy podali liczbę 4, wówczas do zmiennej `i` podstawione zostałyby 4. Wyrażenie `i-4` miałoby wartość 0 (czyli: fałsz) i wtedy wykonana zostałaby instrukcja druga.

†) `if` – ang. jeśli (czytaj: „yf”)

††) (czytaj: „els”)

Blok instrukcji

Często się zdarza, że chodzi nam o wykonanie warunkowe nie jednej instrukcji, a całego bloku instrukcji. Stosujemy wówczas instrukcję składaną zwaną inaczej blokiem. Są to po prostu zwykłe instrukcje ograniczone nawiasami { }. Zauważ, że po klamrze nie trzeba stawiać średnika.

```
{
    instr1 ;
    instr2 ;
    instr3 ;
}
```

Oto przykład programu, w którym stosujemy instrukcje składane.

```
#include <iostream.h>
main()
{
    int wys, punkty_karne ;           // definicja dwóch zmiennych
    // typu int. Obie są tego samego typu więc
    // wystarczy przecinek oddzielający nazwy

    cout << "Na jakiej wysokości lecimy ? [w metrach] : " ;
    cin >> wys ;

    // rozważamy sytuacje -----
    if(wys < 500)
    {
        cout << "\n" << wys << " metrów to za nisko !\n" ;
        punkty_karne = 1 ;
    }
    else
    {
        cout << "\nNa wysokości " << wys
            << " metrów jesteś już bezpieczny \n" ;
        punkty_karne = 0 ;
    }

    // ocena Twoich wyników -----
    cout << "Masz " << punkty_karne
        << " punktów karnych \n" ;
    if(punkty_karne) cout << "Popraw się !" ;
}
```

Przypominam, że różnicowane odstępów od lewego marginesu (wypełnione białymi znakami) nie mają dla kompilatora żadnego znaczenia. Pomagają natomiast programiście. Dzięki nim program staje się bardziej czytelny.

Oto przykładowy wygląd ekranu po wykonaniu tego programu

```
Na jakiej wysokości lecimy ? [w metrach] : 2500
Na wysokości 2500 metrów jesteś już bezpieczny
Masz 0 punktów karnych
```

Jeśli na zadane pytanie odpowiemy inaczej, to ekran może wyglądać tak:

```
Na jakiej wysokości lecimy ? [w metrach] : 100
100 metrów to za nisko !
```


Masz 1 punktów karnych
Popraw sie !

Zauważ jak prosto wypisuje się na ekranie wartość zmiennej `wys`. Wystarczyła instrukcja

```
cout << wys ;
```

Wybór wielowariantowy

Przy użyciu słowa `else` mieliśmy więc możliwość dwuwariantowego wyboru: Robimy *to*, w przeciwnym razie robimy *tamto*.

Możemy jednak pójść dalej – koło słowa `else` możemy postawić następną instrukcję `if`. Dzięki temu zyskamy możliwość wyboru wielowariantowego

```
if (warunek1) instrukcja1 ;  
else if (warunek2) instrukcja2 ;  
else if (warunek3) instrukcja3 ;  
else if (warunek4) instrukcja4 ;
```

(Inną możliwością wykonania wyboru wielowariantowego jest instrukcja `switch`, o której niebawem.)

2.3 Instrukcja `while`

Instrukcja `while`^{†)} ma formę:

```
while (wyrażenie) instrukcja1 ;
```

Najpierw oblicza się wartość wyrażenia. Jeśli wynik jest zerowy, wówczas instrukcja1 nie jest wcale wykonywana. Jeśli jednak wartość wyrażenia jest niezerowa (prawda), wówczas wykonywana jest instrukcja1, po czym ponownie obliczana jest wartość wyrażenia. Jeśli nadal wartość tego wyrażenia jest niezerowa, wówczas ponownie wykonywana jest instrukcja1, i tak dalej, dopóki (`while`!) wyrażenie ma wartość niezerową. Jeśli w końcu kiedyś obliczone wyrażenie będzie miało wartość zerową, wówczas dopiero pętla zostanie przerwana.



Zwracam uwagę, że obliczenie wartości wyrażenia odbywa się **przed** wykonaniem instrukcji1

```
#include <iostream.h>  
main()  
{  
    int ile ;  
  
    cout << "Ile gwiazdek ma miec kapitan ? : " ;  
    cin >> ile ;
```

†) `while` – ang. podczas gdy, dopóki (czytaj: „łajł”)

```

cout << "\n No to narysujmy wszystkie "
      << ile << " : " ;

// petla while rysujaca gwiazdki
while(ile)
{
    cout << "*" ;
    ile = ile - 1 ;
}
// na dowod ze mial prawo przerwac petle
cout << "\n Teraz zmienna ile ma wartosc " << ile
}

```

A oto przykładowy wygląd ekranu po wykonaniu tego programu

```

Ile gwiazdek ma miec kapitan ? : 4
No to narysujmy wszystkie 4 : ****
Teraz zmienna ile ma wartosc 0

```

2.4 Pętla do...while...

Słowa te oznaczają po angielsku: Rób... Dopóki...^{†)} Pętla taka ma formę

```
do instrukcja1 while(wyrażenie) ;
```

Czyli jakby po polsku

```
rób instrukcja1 dopóki (wyrażenie) ;
```

Działanie jej jest takie: Najpierw wykonywana jest instrukcja1. Następnie obliczona zostaje wartość wyrażenia. Jeśli jest ono niezerowe (prawda), to wykonanie instrukcji1 zostanie powtórzone, po czym znowu obliczone zostanie wyrażenie... i tak w kółko, *dopóki* wyrażenie będzie różne od zera.

Jak widać działanie tej pętli przypomina tę opisaną poprzednio. Różnica polega tylko na tym, że wartość wyrażenia obliczana jest nie przed, ale **po** wykonaniu instrukcji1. Wynika stąd, że instrukcja1 zostanie wykonana co najmniej raz. Czyli nawet wtedy, gdy wyrażenie nie będzie nigdy prawdziwe.

Na przykład:

```

#include <iostream.h>
main()
{
    char litera ;
    do {
        cout << "Napisz jakas litere : " ;
        cin >> litera ;
        cout << "\n Napisaales : " << litera << " \n" ;
    } while(litera != '\0') ;
}

```

^{†)} (czytamy: „du...łajl”)

```

}while(litera != 'K');
cout << "\n Skoro Napisales K to konczymy !" ;
}

```

A oto przykładowy wygląd ekranu po wykonaniu tego programu

```

Napisz jakas litere : A
Napisales : A
Napisz jakas litere : K
Napisales : K

```

Skoro Napisales K to konczymy !

Program nasz oczekuje na napisanie litery, pętla wczytywania liter odbywa się *dopóki* nie podamy litery K (wielkiej). Wtedy to wykonywanie pętli zakończy się. Zwracam uwagę – pętla wczytująca znaki zostanie wykonana przynajmniej raz.

W programie – w miejscu, które oznaczyłem jako ❶ – pojawił się w wyrażeniu nieznamy nam jeszcze dotąd operator != który oznacza: „różny od ...”. Zatem zapis

```
while(litera != 'K')
```

rozumiany jest jako „dopóki litera jest różna od K”. Temu i innym operatorom przyjrzymy się dokładnie później.

2.5 Pętla for

Ma ona formę

```
for(instr_ini ; wyraz_warun ; instr_krok) treść_pętli ;
```

co w przykładzie może wyglądać choćby tak:

```

for(i=0 ; i < 10 ; i=i+1)
{
    cout << "Ku-ku !" ;
}

```

Wyjaśnijmy co oznaczają poszczególne człony:

- ❖ *for* – (ang. dla...) oznacza: dla takich warunków rób...
- ❖ *instr_ini* – jest to instrukcja wykonywana zanim pętla zostanie po raz pierwszy uruchomiona.
 - W naszym przykładzie jest to podstawienie $i = 0$.
- ❖ *wyraz_warun* – jest to wyrażenie, które obliczane jest przed każdym obiegiem pętli. Jeśli jest ono różne od zera, to wykonywane zostają instrukcje będące treścią pętli.
 - U nas wyrażeniem warunkowym jest wyrażenie: $i < 10$. Jeśli rzeczywiście i jest mniejsze od 10, wówczas wykonywana

zostaje instrukcja będąca treścią pętli, czyli wypisanie tekstu "Ku-ku!"

- ❖ *instr_krok* – to instrukcja wykonywana na zakończenie każdego obiegu pętli. Jest to jakby ostatnia instrukcja, wykonywana bezpośrednio przed obliczeniem wyrażenia *wyraz_warun*.

- U nas jest to po prostu $i = i + 1$

Praca tej pętli odbywa się więc jakby według takiego harmonogramu:

- 1) Najpierw wykonują się instrukcje inicjalizujące pracę pętli.
- 2) Obliczane jest wyrażenie warunkowe. Jeśli jest równe 0 – praca pętli jest przerywana.
- 3) Jeśli powyżej okazało się, że wyrażenie było różne od zera, wówczas wykonywane zostają instrukcje będące treścią pętli.
- 4) Po wykonaniu treści pętli wykonana zostaje instrukcja *instr_krok*, po czym powtarzana jest akcja 2).



Oto kilka ciekawostek:

instr_ini - nie musi być tylko jedną instrukcją. Może być ich kilka, wtedy oddzielone są przecinkami. Podobnie w wypadku *instr_krok*

Wyszczególnione elementy: *instr_ini*, *wyraz_warun*, *instr_krok* - nie muszą wystąpić. Dowolny z nich można opuścić, zachowując jednak średnik oddzielający go od sąsiada.

Opuszczenie wyrażenia warunkowego traktowane jest tak, jakby stało tam wyrażenie zawsze prawdziwe.

Tak więc zapis

```
for( ; ; ){
    ....
}
```

Jest nieskończoną pętlą. Inny typ nieskończonej pętli to oczywiście:

```
while(1){
    ....
}
```

Przykład

Przyjrzyjmy się pętli *for* w programie

```
#include <iostream.h>
main()
{
    int i ,                // licznik
        ile ;             // liczba pasażerów

    cout << "Stewardzie, ile leci pasażerów ? " ;
    cin >> ile ;
```

```

for(i = 1 ; i <= ile ; i = i + 1)
{
    cout << "Pasazer nr " << i
        << " prosze zapiac pasy ! \n" ;
}
cout << "Skoro wszyscy juz zapieli, to ladujemy. " ;
}

```

Jeśli w trakcie wykonywania programu steward odpowie, że leci 4 pasażerów to

na ekranie pojawi się:

```

Stewardzie, ile leci pasazerów ? 4
Pasazer nr 1 prosze zapiac pasy !
Pasazer nr 2 prosze zapiac pasy !
Pasazer nr 3 prosze zapiac pasy !
Pasazer nr 4 prosze zapiac pasy !
Skoro wszyscy juz zapieli, to ladujemy.

```

2.6 Instrukcja switch

Switch ^{†)} - jak sama nazwa sugeruje - służy do podejmowania wielowariantowych decyzji. Przyjrzyjmy się przykładowemu fragmentowi programu.

```

int ktory ;
// .....
cout << "Kapitanie, ktory podzespól sprawdzic ? \n"
    << "nr 10 - Silnik \nnr 35 - Stery \nnr 28 - radar\n"
    << "Podaj kapitanie numer : " ;
cin >> ktory ;
switch(ktory)
{
    case 10 :
        cout << "sprawdzamy silnik \n" ;
        break ;

    case 28 :
        cout << "sprawdzamy radar \n" ;
        break ;

    case 35 :
        cout << "sprawdzamy stery \n" ;
        break ;

    default :
        cout << "Zazadales nr " << ktory
            << " - nie znam takiego ! " ;
        break ;
}

```

†) switch - ang. przełącznik (czytaj: „slicz”)



W wypadku, gdy kapitan (czyli Ty!) odpowie, że numer 35, to na ekranie będzie następujący tekst:

```
Kapitanie, który podzespół sprawdzić ?
nr 10 - Silnik
nr 35 - Stery
nr 28 - radar
Podaj kapitanie numer : 35
sprawdzamy stery
```

Jeśli jednak zażąda podzespołu nr 77 to na ekranie zobaczy:

```
Kapitanie, który podzespół sprawdzić ?
nr 10 - Silnik
nr 35 - Stery
nr 28 - radar
Podaj kapitanie numer : 77
Zazadales nr 77 - nie znam takiego !
```

Oto jak taka instrukcja switch działa: Obliczane jest wyrażenie umieszczone w nawiasie przy słowie switch

```
switch(wyrażenie)
{
    case wart1:
        instrA ;
        break ;

    case wart2 :
        instrB ;
        break ;

    default :
        instrC ;
        break ;
}
```

Jeśli jego wartość odpowiada którejś z wartości podanej w jednej z etykiet case^{†)}, wówczas wykonywane są instrukcje począwszy od tej etykiety. Wykonywanie ich kończy się po napotkaniu instrukcji break^{††)}. Powoduje to wyskok z instrukcji switch – czyli jakby wyjście poza jej dolną klamrę.

Jeśli wartość wyrażenia nie zgadza się z żadną z wartości podanych przy etykietach case, wówczas wykonują się instrukcje umieszczone po etykietce default^{†††)}. U nas etykieta ta znajduje się na końcu instrukcji switch, jednak może być w dowolnym miejscu, nawet na samym jej początku. Co więcej, etykiety default może nie być wcale. Jeśli wartość wyrażenia nie zgadza się z żadną z wartości przy etykietach case, a etykiety default nie ma wcale, wówczas opuszcza się instrukcję switch nie wykonując niczego.

†) case – ang. wypadek, sytuacja (czytaj: „kejs”)

††) break – ang. przerwij (czytaj: „brejk”)

†††) default – ang. domniemanie (czytaj: „difolt”)

Instrukcji następujących po etykiecie case nie musi kończyć instrukcja break. Jeśli jej nie umieścimy, to zaczną się wykonywać instrukcje umieszczone pod następną etykietą case. Konkretniej: w naszym ostatnim programie brak instrukcji break w case 10 spowodowałoby, że po wykonywaniu instrukcji dla case 10 nastąpiłoby wykonywanie instrukcji z case 28.

Nie jest to nieudolność języka C++. Czasem się to przydaje. Lepiej przecież, gdy programista sam może zdecydować czy przerwać (break) wykonywanie danych instrukcji, czy też kontynuować. Czasem więc celowo nie umieszczamy instrukcji break.

Np.

```
switch(nr)
{
    case 3 : cout << "*" ;
    case 2 : cout << "-" ;
    case 1 : cout << "!" ;
            break ;
}
```

Zależnie od wartości zmiennej nr możliwe są następujące wydruki na ekran:

```
dla nr = 3      * - !
dla nr = 2      - !
dla nr = 1      !
```

dla innego n nic się nie wydrukuje

2.7 Instrukcja break

Zapoznaliśmy się powyżej działaniem instrukcji break - polegającym na przerwaniu wykonywania instrukcji switch. Jest jeszcze inne, choć podobne działanie break w stosunku do instrukcji pętli: for, while, do...while. Instrukcja ta powoduje natychmiastowe przerwanie wykonywania tych pętli.

Jeśli mamy do czynienia z kilkoma pętlami – zagnieżdżonymi jedna wewnątrz drugiej, to instrukcja break powoduje przerwanie tylko tej pętli, w której bezpośrednio tkwi. Jest to więc jakby przerwanie z wyjściem tylko o jeden poziom wyżej.

Oto jak instrukcja break przerwie pętlę while:

```
int i = 7 ;
while(1)
{
    cout << "Petla, i = " << i << "\n" ;
    i = i - 1 ;
    if(i < 5){
        cout << "Przerywamy !" ;
        break ;
    }
}
```




Wykonanie tego fragmentu programu spowoduje wypisanie na ekranie

```
Petla, i = 7
Petla, i = 6
Petla, i = 5
Przerywamy !
```

A oto przykład z zagnieżdżonymi pętlami.

```
int i, m ;
int dlugosc_linii = 3 ;
for(i=0 ; i < 4 ; i = i + 1)
{
    for(m = 0 ; m < 10 ; m = m + 1)           // ❶
    {
        cout << "*" ;
        if(m > dlugosc_linii)break ;          // tu wyskok
                                                // z for (m...)
    }
    cout << "\nKontynuujemy zewnetrzna petle"
        << " for dla i = "
        << i << "\n" ;
}
```



Wykonanie tego fragmentu objawi się na ekranie jako:

```
*****
Kontynuujemy zewnetrzna petle for dla i = 0
*****
Kontynuujemy zewnetrzna petle for dla i = 1
*****
Kontynuujemy zewnetrzna petle for dla i = 2
*****
Kontynuujemy zewnetrzna petle for dla i = 3
```

To instrukcja break sprawiła, że nie było 10 gwiazdek w rzędzie, (jakby to wynikało z zapisu pętli w linii ❶). Za pomocą instrukcji break przerywana została ta pętla, w której break tkwiło **bezpośrednio**.

2.8 Instrukcja goto

W zasadzie na tym moglibyśmy skończyć omawianie instrukcji sterujących, gdyby nie jeszcze jedna, wstydliva instrukcja goto.^{†)} Ma ona formę:

```
goto etykieta ;
```

Po napotkaniu takiej instrukcji wykonywanie programu przenosi się do miejsca, gdzie jest dana etykieta.

†) go to - ang. idź do (czytaj: „goł tu”)

Powiedzmy jasno: używanie instrukcji `goto` zdradza, że się jest złym programistą. To dlatego, że instrukcji tej zawsze da się unikać. Program (nad-) używający instrukcji `goto` jest dla programisty nieczytelny, a z kolei dla kompilatora stanowi to przeszkodę w eleganckim skompilowaniu.

Z instrukcja `goto` wiąże się zawsze etykieta, do której należy przeskoczyć. Etykieta jest to nazwa, po której następuje dwukropek.

W języku C++ nie można sobie skoczyć z dowolnego punktu programu w dowolne inne. Etykieta, do której przeskakujemy, musi leżeć w obowiązującym w danej chwili tzw. zakresie ważności. (O tym pomówimy na stronie 42). Oto przykład użycia `goto`.

```
cout << "Cos piszemy \n" ;
goto aaa ; // stad przeskok
cout << "Tego nie wypiszemy " ;
aaa: // w to miejsce
cout << "Piszemy " ;
```

Ten fragment objawi się na ekranie jako:

```
Cos piszemy
Piszemy
```

Przypominam, że to, iż w naszym przykładzie etykietę wysunąłem bliżej lewego marginesu, nie ma żadnego znaczenia dla kompilatora. Jemu jest to wszystko jedno. Nam jednak nie. Dla nas chyba lepiej, by etykieta bardziej rzucała się w oczy, łatwiej ją odszukać w tekście programu.



Mimo tej niesławy są sytuacje, gdy instrukcja `goto` się przydaje

Na przykład dla natychmiastowego opuszczenia wielokrotnie zagnieżdżonych pętli. Instrukcją `break` przerwać możemy przecież tylko tę najbardziej zagnieżdżoną pętlę. Dzięki instrukcji `goto` możemy w wyjątkowych wypadkach od razu wyskoczyć na zewnątrz. Na zewnątrz – oznacza tu – na zewnątrz tych zagnieżdżonych pętli. (Zawsze jednak tylko w ramach tego bloku programu, w którym znana jest etykieta).

Oto przykład:

```
int m, i, k ;
while(m < 500)
{
    while(i < 20)
    {
        for(k = 16 ; k < 100 ; k = k+4 )
        {
            // .....
            // tu wyskoczmy !
            if(blad_operacji) goto berlin ; // ❶
        }
    }
}
```

```
berlin : // etykieta, ②
cout << "Po opuszczeniu wszystkich petli " ;
```

Jeśli w jakiś sposób w trakcie pracy tych pętli zmienna `blad_operacji` przybierze wartość niezerową, wówczas nastąpi wyskok ① z pętli, i wykonywane będą instrukcje począwszy od etykiety `berlin` ②

2.9 Instrukcja `continue`

Instrukcja `continue` przydaje się wewnątrz pętli `for`, `while`, `do...while`. Powoduje ona zaniechanie realizacji instrukcji będących treścią pętli, jednak (w przeciwieństwie do instrukcji `break`) sama pętla nie zostaje przerwana. `Continue` przerywa tylko ten obieg pętli i zaczyna następny, kontynuując pracę pętli. Oto przykład :

```
int k ;
for(k = 0 ; k < 12 ; k = k + 1)
{
    cout << "A" ;
    if(k > 1) continue ;
    cout << "b" << endl ;
}
```



W rezultacie wykonania tego fragmentu programu na ekranie pojawi się:

```
Ab
Ab
AAAAAAAAAAAA
```

Innymi słowy napotkanie instrukcji `continue` odpowiada jakby takiemu użyciu instrukcji `goto`

```
for(...)
{
    ...
    continue ;           // goto sam_koniec
    ...
sam_koniec:
}
```

czyli skokowi do etykiety stojącej przed zamykającą pętlę klamrą. W rezultacie komputer „pomyśli”, że już wykonał treść pętli, i przystąpi do wykonywania następnego obiegu.

Identycznie zachowa się wobec pętli `while`

```
while(warunek)
{
    ...
    continue ;           // goto sam_koniec ;
    ...
sam_koniec :
}
```

Czy też pętli do...while

```
do
{
    ...
    continue ;           // goto sam_koniec ;
    ...
sam_koniec :
}while(warunek) ;
```

2.10 Klamry w instrukcjach sterujących

Pamiętasz, mówiłem kiedyś, że język C++ jest językiem o dowolnym formacie. Wynika z tego także, iż klamry { } w naszych instrukcjach sterujących możemy stawiać w różnych miejscach. Oto kilka wariantów

```
while(i < 4) {           ❶
    ...
}

while(i < 4)             ❷
{
    ...
}

while(i < 4)             ❸
{
    ...
}
```

Wszystkie trzy sposoby są jednakowo dobre. Namawiam jednak do przyjęcia jednego standardu.

Dlaczego to takie ważne? Otóż jednym z najczęstszych błędów jest zapomnienie o zamknięciu klamry. Gdy stosujemy zapis ❷ i ❸ to wyraźnie widzimy, które klamry należą do siebie. W sposobie ❶ tego nie widać, ale za to jest on o jedną linijkę krótszy. Osobiście stosuję zapis ❶ lub ❷. W tej książce używać będę zapisu ❷.

Dlaczego nie przeszkadza mi wspomniana wada zapisu ❶ ?

Z dwóch powodów:

- 1) W moim edytorze jest komenda, która pozwala mi odszukać drugi nawias: pokazuję na nawias lewy, a edytor odszukuje mi odpowiadający mu prawy. To samo w wypadku klamer. Jeśli nawet pogubię się z tymi klamrami w długim programie, to dzięki tej opcji łatwo znaleźć błąd. (Sprawdź czy i w Twoim edytorze jest podobna komenda).
- 2) Mam sposób, który prawie całkowicie pozwala mi uniknąć błędu.

Dawniej robiłem mianowicie tak:

Pisałem np: if(warunek), potem otwierałem klamrę i długo pisałem wszystkie instrukcje, po czym klamrę uroczyście zamykałem – o ile tylko jeszcze pamię-

tałem, że mam jakąś klamrę zamknąć. Łatwo o tym zapomnieć, szczególnie wtedy, gdy we wnętrzu są zagnieżdżone inne instrukcje z klamrami.

Teraz robię tak:



Piszę: `if` (warunek), otwieram klamrę, przechodzę dwie linijki niżej i zamykam od razu klamrę, po czym wracam linijkę wyżej i dopiero przystępuję do pisania instrukcji.

Sposób jest naprawdę dobry. Od czasu, gdy go stosuję nawet kilkakrotnie zagnieżdżanie instrukcji `while`, `for`, `do` – nie jest mi straszne. Spróbuj.



Mimo wszystko zapewne czasem pogubisz się w stawianiu klamer. Radzę Ci: spróbuj to zrobić świadomie po to, by się przekonać, jak na to zareaguje Twój kompilator. Albowiem jego komunikat o błędzie wcale nie musi mówić o braku klamry.

W działającym programie usuń lub dodaj jedną z zamykających klamer i spróbuj to skompilować. Komunikat o błędzie może być w stylu „zła deklaracja funkcji” albo coś w tym rodzaju.

Dobrze to zapamiętaj, bo gdy potem otrzymasz taki sam komunikat – będziesz już wiedział, że przyczyny można szukać również w nawiasach klamrowych.

3 Typy

3.1 Deklaracje typów

Każda nazwa w C++ zanim zostanie użyta, musi zostać zadeklarowana.

Deklarujemy mianowicie, że opisuje ona obiekt jakiegoś typu: całkowitego, zmiennoprzecinkowego, itd. – (np. `int`, `float`). To informuje kompilator, jak ma w przypadku napotkania tej nazwy postępować.

Wyjaśnijmy to na przykładzie. Załóżmy, że kompilator napotkał w naszym programie wyrażenie

```
a + b
```

Jest to dodawanie, zatem trzeba uruchomić specjalny podprogram zajmujący się dodawaniem. (Taki podprogram nazywa się też czasem operatorem dodawania.)

W związku z tym, że w komputerze liczby całkowite przechowywane są inaczej niż liczby zmiennoprzecinkowe – zatem operator dodawania musi inaczej postępować w stosunku do liczb całkowitych, a inaczej w stosunku do liczb zmiennoprzecinkowych.

Napotykać ów zapis – kompilator musi więc dokładnie wiedzieć czy symbol `a` oznacza u nas liczbę całkowitą, czy zmiennoprzecinkową. Skąd to będzie wiedział? Właśnie z deklaracji. Deklarując wcześniej zmienną `a` jako typu `integer`, czyli pisząc

```
int a ;
```

powiedzieliśmy kompilatorowi tak:

Jakbyś napotkał nazwę `a` to wiedź, że oznacza ona zmienną typu `int`.

Tutaj deklaracja jest równocześnie definicją. Oznacza to, że nie tylko, iż informujemy kompilator o tym, co oznacza nazwa `a`, lecz także żądamy w tym miejscu, by zarezerwował w pamięci obszar na tę zmienną – czyli powołał ją do życia.

Definicja mówi więc kompilatorowi:

A teraz zarezerwuj mi w pamięci miejsce na obiekt typu całkowitego o nazwie a

Nie zawsze jednak chodzi o powołanie do życia.

Może być przecież sytuacja, gdy zmienna ta już gdzieś w programie istnieje, a tu chcemy tylko poinformować kompilator o jej typie.

```
extern int a ;
```

Słowo `extern` (ang. zewnętrzny) informuje kompilator, obiekt typu `int` o nazwie `a` już gdzieś istnieje, na przykład na zewnątrz pliku, którym zajmuje się właśnie kompilator. To „na zewnątrz” może oznaczać też jakąś funkcję biblioteczną, którą dołączymy dopiero na etapie linkowania. W trakcie kompilacji naszego pliku, kompilator musi już jednak wiedzieć jakiego typu jest ta „zewnętrzna” zmienna `a`.

Powtórzmy więc różnicę między deklaracją a definicją

Deklaracja – informuje kompilator, że dana nazwa reprezentuje obiekt jakiegoś typu, ale nie rezerwuje dla niego miejsca w pamięci.

Definicja zaś – dodatkowo rezerwuje miejsce. Definicja jest miejscem, gdzie powołuje się obiekt do życia.

Oczywiście definicja jest przy okazji zawsze także deklaracją, bo przecież jeśli rezerwuje miejsce w pamięci, to musi ona kompilatorowi wyjaśnić na co rezerwuje.

Deklarować obiekt można w tekście programu wielokrotnie. Definiować go (powoływać go do życia) można tylko raz.

Studium języków obcych

Różnicę między deklaracją a definicją łatwo zrozumieć i zapamiętać tłumacząc sobie dosłownie te słowa.



Deklaracja :

od łacińskiego *clarus*: jasny, zrozumiały. Zresztą po polsku też mówi się: „klarować coś komuś”. *De-klarować* to jakby: *wy-jaśniać*. Deklaracja nazwy *N* jest więc tylko wyjaśnieniem kompilatorowi co dana nazwa oznacza.



Definicja :

pochodzi od łacińskiego słowa *finis* – koniec, granica. Definiować – to jakby zakresła granicę. W naszym wypadku tę granicę wykreśla się wokół komórek pamięci, które są przydzielane w ten sposób obiektowi. Mówi się: *ta, tamta i jeszcze ta komórka – stają się od tej pory obiektem o danej nazwie N*. W ten sposób odbyły się narodziny obiektu o nazwie *N*.

Oto przykłady definicji i deklaracji:

```
int liczba ;           // definicja + deklaracja
extern int licznik,    // deklaracja (tylko ! )
```

3.2 Systematyka typów z języka C++

W deklaracji określa się, że dany obiekt jest jakiegoś typu. Jakie mamy do dyspozycji typy? Jest ich dużo. Wprowadźmy więc pewien porządek.

Typy z języka C++ można podzielić dwojako:

Pierwszy podział to podział na:

- ❖ typy fundamentalne
- ❖ typy pochodne, które są jakby wariacjami na temat typów fundamentalnych

Drugi podział to na:

- ❖ typy wbudowane (ang. build-in) – czyli takie, w które język C++ jest wyposażony,
- ❖ typy zdefiniowane przez użytkownika – czyli typy, które możesz sobie wymyślić samemu. Ta cecha jest chyba jednym z najlepszych pomysłów w języku C++

Zajmiemy się teraz typami wbudowanymi. Natomiast typom definiowanym przez użytkownika poświęcona jest dalsza część książki.

3.3 Typy fundamentalne

Są identyczne jak w klasycznym C. Oto ich lista:



Typy reprezentujące liczby całkowite

short int	inaczej:	short
int		
long int	inaczej:	long

oraz tak zwany typ wyliczeniowy enum, o którym porozmawiamy niebawem. (Str. 52).



Typ reprezentujący obiekty zadeklarowane jako znaki alfanumeryczne

char



Wszystkie powyższe typy mogą być w dwóch wariantach – ze znakiem i bez znaku. Do wybrania wariantu posługujemy się modyfikatorem signed lub unsigned^{†)}

np.

†) signed, unsigned – ang. ze znakiem, bez znaku (czytaj: „sajned”, „ansajned”)


```
signed int
unsigned int
```

Wyposażenie typu w znak sprawia, że może on reprezentować liczbę ujemną i dodatnią. Typ bez znaku reprezentuje liczbę dodatnią.

Przez domniemanie przyjmuje się, że zapis

```
int a ;
```

oznacza, że chodzi nam o typ

```
signed int a ;
```

czyli typ ze znakiem.

Natomiast w wypadku typu char sprawa nie jest tak prosta. To, czy przez domniemanie będziemy mieli signed czy unsigned - zależy od typu kompilatora czy komputera. Mówimy krótko: zależy to od implementacji.



Typy reprezentujące liczby zmiennoprzecinkowe

```
float
double
long double
```

umożliwiają pracę na liczbach rzeczywistych z różną dokładnością.^{†)}



Zastanawiasz się zapewne po co są aż trzy typy reprezentujące liczby całkowite oraz trzy typy reprezentujące liczby zmiennoprzecinkowe.

Chodzi o to, by można było lepiej wykorzystać możliwości danego typu komputera. Zależnie od tego, ile dany komputer przydziela komórek pamięci na zapis danej liczby – otrzymujemy mniejszą lub większą precyzję obliczeń.

Przyjrzyjmy się jak to załatwiane jest na różnych komputerach.

typ	Komputer	
	IBM PC/AT	VAX
short	2 bajty	2 bajty
int	2 bajty	4 bajty
long	4 bajty	4 bajty
float	4 bajty	4 bajty
double	8 bajtów	8 bajtów
long double	10 bajtów	8 bajtów

Za lepszą dokładność płaci się dłuższym czasem obliczeń, dlatego zapewnienie programiście aż 3 typów dla liczb całkowitych daje możliwość wyboru między dokładnością, a szybkością obliczeń.

^{†)} double - (czytaj: „dabl”) ang. podwójny. Zapewne od: double precision - podwójna dokładność.

Jak widać z zestawienia – sposób przechowywania liczby może zależeć od typu komputera. O tym, jak zapisuje dany typ Twój komputer, będziesz się mógł przekonać stosując operator `sizeof` (rozmiar). Operator ten przedstawimy niebawem (str. 69).

3.3.1 Definiowanie obiektów „w biegu”.

W naszych dotychczasowych programach już kilkakrotnie spotkaliśmy się z definicjami zmiennych. W niektórych językach programowania definicje obiektów powinny nastąpić przed wykonywanymi instrukcjami. Tak też jest w klasycznym C.

W C++ zasada ta nie obowiązuje. Obiekt można zdefiniować „w biegu”, „w locie” [ang: on flight], między dwoma instrukcjami – wtedy, gdy uznamy, że jest on nam właśnie potrzebny.

Oto przykład:

```
#include <iostream.h>
main()
{
    float dlugosc_fali ;                               // ❶
    // ..
    cout << "Podaj współczynnik załamania : " ;
    float współczynnik ;                               // ❷
    cin >> współczynnik ;
    cout << " Zrozumiałem, współczynnik ma być : "
        << współczynnik ;
    // .... dalsze obliczenia
}
```

W tym przykładzie widzimy, że przed instrukcjami wykonywanymi w funkcji `main` jest definicja obiektu typu `float` ❶. Jest to definicja w starym, klasycznym stylu.

Tymczasem w trakcie pisania programu dochodzimy do wniosku, że potrzebujemy jeszcze jednego obiektu `float` – na współczynnik załamania. Możemy wrócić do ❶ i tam dopisać następną definicję, ale możemy też zrobić to tu, gdzie sobie o zmiennej przypomnieliśmy ❷, lub gdzie wynikła konieczność jej istnienia. Robimy więc tę definicję w biegu, nie przerywając normalnego toku pisania programu. Ten sposób jest nawet logiczniejszy.

Co prawda, w naszym przykładzie wartość wczytywaliśmy z klawiatury, jednak w innych wypadkach – często w linijce, w której wynikła konieczność istnienia obiektu, już dokładnie wiemy, jaką wartość powinien on od razu zawierać. Możemy więc nie tylko zdefiniować obiekt, ale od razu (w tej samej instrukcji), nadać mu wartość. Takie postępowanie daje szybszy program (bo to mniej pracy niż wariant z powtórным wracaniem do zmiennej, by jej nadawać wartość).

3.4 Stałe dosłowne

W programach często posługujemy się stałymi. Mogą to być liczby, mogą to być znaki (litera) albo ciągi znaków (z angielska: *string*). Stałych tych używamy na przykład, by wstawić je do jakichś zmiennych

```
x = 10.52 ;
```

lub wtedy, gdy występują one w wyrażeniach arytmetycznych

```
i = i + 5 ; // powiększenie obiektu i o stałą 5
```

albo wtedy, gdy chcemy coś z nimi porównać

```
if (m > 12)
```

Zwracam uwagę, że w tym miejscu chodzi nam o stałe dosłowne czyli o sam zapis liczby, a nie o jakiś obiekt, który ma akurat taką wartość. Zapoznamy się tu ze sposobami zapisywania takich stałych.

3.4.1 Stałe będące liczbami całkowitymi

Stałe takie zapisujemy tak, jak do tego przywykliśmy w szkole

```
17 -33 0 1000 itd.
```

Jeśli natomiast zapis stałej zaczniemy od cyfry 0 (zero), to kompilator zrozumie, że zastosowaliśmy zapis liczby w systemie ósemkowym (oktalnym).

```
010 - czyli dziesiętkowo 8
```

```
014 - czyli dziesiętkowo 8+4 = 12
```

```
091 - błąd, w zapisie ósemkowym cyfra 9 jest nielegalna
```

Jeżeli stała zaczyna się od 0x (zero i x) to kompilator uzna, że w stosunku do stałej zastosowaliśmy zapis szesnastkowy (heksadecymalny) (heXadecymalny).

```
0x10 - czyli dziesiętkowo 1*16 + 0 = 16
```

```
0xa1 - czyli dziesiętkowo 10*16 + 1 = 161
```

```
0xff - czyli dziesiętkowo 15*16 + 15 = 255
```

Nie muszę chyba przypominać, że występujące w zapisie znaki a, b, c, d, e, f oznaczają odpowiednio 10, 11, 12, 13, 14, 15 w zapisie dziesiętkowym. W zapisie można się posługiwać zarówno wielkimi literami X, A, B, C, D, E, F, jak i małymi.

Stałe całkowite traktuje się tak, jak typ `int`, chyba, że reprezentują tak wielkie liczby, które nie zmieściłyby się w `int`. Wówczas stała taka jest typu `long`.

Można świadomie zmienić typ nawet niewielkiej stałej – z typu `int` na typ `long`. Robi się to przez dopisanie na końcu liczby: litera `L` (lub `l`)

```
0L 200L
```

Mimo, że liczby te wystarczająco dobrze mieszczą się w typie `int` - dopisana na końcu litera `L` sprawia, że są typu `long`. Osobiście zawsze używam tu wielkiej litery `L`, gdyż mała za bardzo przypomina jedynekę.

Jeśli chcemy by dana stała miała typ `unsigned`, to sprawi to dopisanie na końcu litery `u`

Przypisek u może wystąpić razem z przypiskiem L

50uL

wówczas oznacza to, że dana stała ma być typu `unsigned long`.



Oto przykład zapisu tych stałych w programie:

```
#include <iostream.h>
main()
{
    int i      ;           // definicja obiektu
    int k, n, m, j ;

    i = 5 ;
    k = i + 010 ;          // czyli 5 + 8

    cout << "k= " << k << endl ;

    m = 100 ;
    n = 0x100 ;
    j = 0100 ;

    cout << "m+n+j= " << (m+n+j) << endl ;

    cout << "Wypisujemy : " << 0x22 << " "
        << 022 << " " << 22 << endl ;
}
```

W wyniku wykonania na ekranie pojawi się

```
k= 13
m+n+j= 420
Wypisujemy : 34 18 22
```

Zauważ, że zastosowanie któregoś z zapisów (dziesiętkowego, oktalnego, hexadecymalnego) jest tylko jednym ze sposobów powiedzenia o jaką liczbę nam chodzi. Komputer i tak przetłumaczy to sobie na swój własny sposób (binarny). To tak, jakbyśmy rachmistrzowi powiedzieli czasem trzy, czasem drei, czasem three.

3.4.2 Stałe reprezentujące liczby zmiennoprzecinkowe

Stałe takie zapisać można na dwa sposoby. Pierwszy to normalny zapis liczby z kropką dziesiętną.

12.3 3.1416 -1000.3 -12.

Drugi zapis jest nazywany notacją naukową (scientific notation). W zapisie tym występuje litera e, po której następuje wykładnik potęgi o podstawie 10. A zatem:

8e2	oznacza	8	$\cdot 10^2$	czyli	800
10.4e8	oznacza	10.4	$\cdot 10^8$	czyli	1 040 000 000
5.2e-3	oznacza	5.2	$\cdot 10^{-3}$	czyli	0.0052

Stałe takie traktuje się tak, jakby były typu double

Oto przykład użycia takich stałych:

```
#include <iostream.h>
main()
{
    float pole, promien ;

    promien = 1.7 ;
    pole = promien * promien * 3.14 ;
    cout << "\nPole kola o promieniu "
          << promien << " wynosi " << pole ;

    promien = 4.1e2 ;
    pole = promien * promien * 3.14 ;
    cout << "\nPole kola o promieniu "
          << promien << " wynosi " << pole ;
}
```

W wyniku wykonania tego programu na ekranie pojawi się

```
Pole kola o promieniu 1.7 wynosi 9.0746
Pole kola o promieniu 410 wynosi 527834
```

3.4.3 Stałe znakowe

Stałe znakowe są to stałe reprezentujące na przykład znaki alfanumeryczne. Zapisuje się je ujmując dany znak w dwa apostrofy

```
'a' - oznacza literę a
'7' - oznacza cyfrę 7 (cyfrę, nie liczbę)
```

Oto przykład:

```
char znak ;
znak = 'A' ;
```

Oczywiście wiesz na pewno, że komputer nie potrafi przechowywać w swojej pamięci żadnej litery 'A'. Może jednak przechowywać liczby. Dlatego wszystkie litery alfabetu i znaki specjalne zostały po prostu ponumerowane i to ten numer (kod) danego znaku jest przechowywany w pamięci.

Są różne sposoby numerowania (kodowania) znaków. Jednym z najbardziej popularnych jest chyba kod ASCII.^{†)} Z tabelą kodów ASCII, czyli tym, jakimi liczbami reprezentowane są jakie znaki – spotkałeś się już na pewno przy programowaniu w innych znanych Ci językach programowania.

†) (czytaj „aski”) – jest to skrót od: American Standard of Code Interchanged Information

Są jednak takie znaki, których nie da się wprost umieścić między apostrofami. Służą one do sterowania wypisywaniem tekstu – np. przejście do nowej strony, tabulator, znak nowej linii. Pomagamy sobie wtedy za pomocą kreski ukośnej, tzw. ukośnika [ang. backslash]^{†)}. Obok niego stawiamy umowną literę, która przypomina znaczenie danego znaku.

'\b' -	cofacz	(ang. Backspace)
'\f' -	nowa strona	(ang. Form feed)
'\n' -	nowa linia	(ang. New line)
'\r' -	powrót karetki	(ang. carriage Return)
'\t' -	tabulator poziomy	(ang. Tabulator)
'\v' -	tabulator pionowy	(ang. Vertical tabulator)
'\a' -	sygnał dzwinkowy	(Alarm)

Mimo, że widzimy kilka znaków, zapis reprezentuje jeden znak. Czytamy to tak:

Dwa najbardziej zewnętrzne apostrofy mówią nam, że mamy do czynienia ze znakiem. W środku zaś czytamy \ – bekslesz: ach, będzie to coś niezwykłego. Potem następuje np. litera f. Skoro coś niezwykłego, to patrzymy co na liście niezwykłości oznacza litera f – jest to skrót od form feed (nowa strona). Proste.

Ponieważ pomagaliśmy sobie stosując znaki takie jak apostrofy, kreska ukośna, więc jak zakodować sam znak apostrofu? Za pomocą trzech apostrofów?

```
char c = ' ' ; // błąd
```

Nie. kompilator podejrzewałby błąd, dlatego musimy dodać kreskę ukośną

```
char c = '\ ' ; // apostrof
```

Zapis ten rozumiemy tak: dwa zewnętrzne apostrofy – czyli wewnątrz jest znak. Potem bekslesz czyli „uwaga!”, a potem apostrof. Bekslesz jest w tym wypadku ostrzeżeniem, bo znak apostrofu jest już raz w tej konstrukcji używany w innym znaczeniu (ogranicznik).

Poniżej widać, że podobnie radzimy sobie w wypadku bekslesza, cudzysłowu i w kilku innych wypadkach.

```
'\\' - bekslesz
'\ ' - apostrof
'\ "' - cudzysłów
'\0' - NULL, znak o kodzie 0
'\?' - pytajnik
```

Wśród znaków (już niealfanumerycznych – bo nie da się ich zapisać) jest jeszcze jeden bardzo wyjątkowy. Jest to znak o kodzie 0 zwany znakiem NULL. Na liście widzimy sposób jego zapisu.

Można także stałe znakowe zapisywać bezpośrednio – podając między apostrofami liczbowy kod znaku, zamiast samego znaku. Kod znaku musi być liczbą w zapisie ósemkowym lub szesnastkowym.

†) (czytaj: „bekslesz”)

Np. ponieważ w kodzie ASCII litera a reprezentowana jest przez liczbę 97 dlatego poniższe zapisy są równoważne:

'a'	- to samo co ósemkowo	\0141
'a'	- to samo co szesnastkowo	0x61

3.4.4 Stałe tekstowe, albo po prostu stringi

W językach programowania bardzo często posługujemy się stałymi tekstowymi będącymi ciągami znaków. Zwane są one czasem napisami, czasem łańcuchami znaków.

Spotkaliśmy się już z takimi stałymi:

"Witamy na pokładzie"

W języku angielskim taka stała tekstowa nazywa się to krótko: string.

W całej tej książce pierwotnie posługiwałem się nazwą „ciąg znaków”. Nazwa ta dziesiątki razy odmieniana była przez wszystkie przypadki, co nie zawsze było zręczne. Wreszcie zrezygnowałem. Nie znam bowiem programisty, który by na to „coś” mówił inaczej jak: *string*.

A zatem:

String, czyli stała tekstowa, to ciąg znaków ujęty w cudzysłów.

Oto przykłady:

```
"taki string"
"Pozar na pokładzie"
"Alarm 3 stopnia"
```

Ponieważ string jest ciągiem znaków, więc obowiązują podobne zasady, jak opisane przy stałych znakowych: Jeśli chcemy w tekście (stringu) zastosować znak nowej linii, to wystarczy napisać \n w żądanym miejscu.

```
"Pozar \n na pokładzie"
```

Zdziwiłeś się dlaczego nie ma teraz dwóch apostrofów po obu stronach znaku \n? Nic w tym dziwnego – nie ma także apostrofów obok liter: p o z itd.

Stringi są w pamięci przechowywane jako ciąg liter, a na samym końcu tego ciągu dodawany jest znak o kodzie 0, czyli znak NULL. Tak kompilator oznacza sobie koniec stringu.

Ogranicznikiem stringu są znaki cudzysłowu "...". Ponieważ cudzysłów ma takie szczególne znaczenie dla stringu, dlatego nie można już go użyć dodatkowo wewnątrz stringu. W wypadku stałych znakowych problem ten mieliśmy z apostrofami. Do pomocy mamy jednak identyczny chwyt ze znakiem bekslesz:

```
cout << "Lecimy promem \"Columbia\" nad Oceanem Spokojnym";
```

co na ekranie pojawi się jako

```
Lecimy promem "Columbia" nad Oceanem Spokojnym
```



Mówiliśmy kiedyś, że w języku C++ w prawie każdym miejscu instrukcji można przerwać pisanie, przejść do następnej linii i kontynuować instrukcję.

To słowo „prawie” dotyczy między innymi pisania stringów. Tutaj nie można przerwać pisania. Jeśli kompilator zobaczył w linii cudzysłów otwierający string, to musi w tej samej linii znaleźć cudzysłów zamykający.

Co zrobić jeśli string jest tak długi, że nie mieści się w jednej linii?

Jest na to sposób. Spójrz poniżej: w kilku liniach zapisaliśmy tu string, który kompilator traktuje jako jedną całość.

```
"Caly ten tek"
"st jest traktowa"
"ny jako jeden dlu"           "gi string"
```

Jak widać w ostatniej linii – nawet w tej samej linii można zamknąć cudzysłów, a potem bezpośrednio go otworzyć – i kompilator uzna to za jeden string. (Zauważ, że nie ma żadnych przecinków).

Zapamiętaj:

Bezpośrednio przylegające do siebie stringi, kompilator łączy w jeden.

Wynika stąd też, że zamiast pisać

```
cout << "Moj drogi Kapitanie, który "
      << "podzespól sprawdzić ? " ;
```

można zapisać

```
cout << "Moj drogi Kapitanie, który "
      "podzespól sprawdzić ? " ;
```

oczywiście dlatego, że są to stringi, które bezpośrednio mogą do siebie przylegać (bo akurat nie wstawiliśmy między nimi żadnego wypisywania na ekran wartości jakiejś zmiennej).

3.5 Typy pochodne

Są to jakby wariacje na temat typów podstawowych (fundamentalnych), o których mówiliśmy poprzednio.

Są to takie typy, jak na przykład tablica czy wskaźnik.

Możemy mieć kilka „luźnych” obiektów typu `int`, ale możemy je powiązać w tablice obiektów typu `int`.

Tablica obiektów typu `int` jest typem pochodnym od typu `int`. Nie musisz się jednak tą całą systematyką przejmować, tak jak mechanik nie musi myśleć czy jego tokarka jest typem pochodnym od właśnie, czego?

Typy pochodne oznacza się stosując nazwę typu, od którego pochodzą, i operator deklaracji typu pochodnego. Jest to prostsze niż się wydaje.


```
int a ;           // obiekt typu int
int b[10] ;       // tablica obiektów typu int (10 -elementowa)
```

Co to jest tablica tłumaczyć chyba nie trzeba – taki typ obiektu istnieje nawet w języku BASIC czy FORTRAN. Tablicom poświęcimy specjalny rozdział.

Teraz wymienię jeszcze inne operatory do tworzenia obiektów typów pochodnych. Jednak nie przerażaj się. Wszystkie staną się jasne w najbliższych rozdziałach.

Oto lista operatorów, które umożliwiają tworzenie obiektów typów pochodnych:

```
[ ] - tablica obiektów danego typu,
*   - wskaźnik do pokazywania na obiekty danego typu,
()  - funkcja zwracająca wartość danego typu,
&   - referencja (przezvisko) obiektu danego typu.
```

Nie mówiliśmy jeszcze o tych typach, będzie o nich mowa w odpowiednim czasie.

Tutaj wyjaśnimy więc krótko, że:

- ❖ Tablica – to inaczej macierz, albo wektor obiektów danego typu.
- ❖ Wskaźnik – to obiekt, w którym można umieścić adres jakiegoś innego obiektu w pamięci.
- ❖ Funkcja – czyli podprogram. Jest to zapewne znane Ci z innych języków programowania.
- ❖ Referencja – to jakby przezvisko jakiegoś obiektu. Dzięki referencji na tę samą zmienną można mówić używając jej drugiej nazwy.

A oto przykłady typów fundamentalnych:

```
int a ;           // def. obiektu typu int
short int b ;     // def. obiektu typu short
float x ;         // def. obiektu typu float
```

dalej nie ma co pisać, bo jest to prymitywne. Oto przykłady typów pochodnych:

```
int t[10] ;       // tablica 10 elementów typu int
float *p ;        // wskaźnik mogący pokazać na jakiś
                  // obiekt typu float
char func() ;     // funkcja zwracająca (jako rezultat
                  // wykonania) obiekt typu char
```

Bardzo zachęcam Cię, drogi czytelniku, abyś teraz oswoił się z takimi deklaracjami, a w przyszłości nauczył się je odczytywać. Da Ci to ogromną swobodę w poruszaniu się po królestwie C++. Jeśli są ludzie, którzy nie lubią C i C++, to dlatego, że stają bezradni, gdy zobaczą takie deklaracje.

Do ćwiczeń w odczytywaniu deklaracji jeszcze wielokrotnie powrócimy.

3.5.1 Typ void

W deklaracjach typów pochodnych może się pojawić słowo `void`. [ang. próżny] Słowo to stoi w miejscu, gdzie normalnie stawia się nazwę typu. I tak:

```
void *p ;
```

- tutaj oznacza to, że `p` jest wskaźnikiem do pokazywania na obiekt nieznanego typu. (O tym, do czego taki wskaźnik może się przydać, powiemy sobie w rozdziale o wskaźnikach.)

```
void funkcja() ;
```

- deklaracja ta mówi, że funkcja nie będzie zwracać żadnej wartości.

3.6 Zakres ważności nazwy obiektu, a czas życia obiektu

Wiemy już jak zdefiniować lub zadeklarować obiekt jakiegoś typu. Przykładowo dla obiektu typu `int` robi się to instrukcją

```
int m ;
```

Zajmijmy się teraz zakresem ważności nazwy tak zdefiniowanego obiektu i czasem jego życia.

Czas życia obiektu

- ❖ to okres od momentu, gdy zostaje on zdefiniowany, (definicja przydziela mu miejsce w pamięci) – do momentu, gdy przestaje on istnieć, (a jego miejsce w pamięci zostaje zwolnione).

Zakres ważności nazwy obiektu

- ❖ to ta część programu, w której nazwa znana jest kompilatorowi.

Jaka jest różnica między tymi pojęciami?

Taka, że w jakimś momencie obiekt może istnieć, ale nie być dostępny. To dlatego, że np. znajdujemy się chwilowo poza zakresem ważności jego nazwy. Zależnie od tego, jak zdefiniujemy obiekt, zakres ważności jego nazwy może być czworakiego rodzaju.

3.6.1 Zakres: lokalny

Zakres ważności jest lokalny, gdy świadomie ograniczamy go do kilku linii programu. Pisząc program możemy w dowolnym momencie za pomocą dwóch klamer

```
{
```

```
...
```

```
}
```

utworzyć tzw. blok. Zdefiniowane w nim nazwy mają zakres ważności ograniczony tylko do tego bloku. Po prostu poza tym blokiem nazwy te nie są znane.

```
#include <iostream.h>
main()
{
    {
        int x ;
        ...
    }
    ...
}
```

// tu robimy jakieś obliczenia
// ← **otwieramy lokalny blok**
// definiujemy jakieś zmienne
// pracujemy na tych zmiennych
// ← **zamykamy lokalny blok**
// poza blokiem lokalne obiekty są już nieznane

Nazwa zmiennej *x* jest znana od momentu, gdy ją zdefiniowaliśmy do linii, gdzie jest klamra *}* kończąca jej lokalny blok.

3.6.2 Zakres: blok funkcji

Zakres ważności ograniczony do bloku funkcji ma etykieta. Znaczy to, że jest ona znana w całej funkcji, nawet w tych liniach funkcji, które ją poprzedzają.

Uwaga. Z faktu, że etykieta ma zakres ważności funkcji wynika prosty wniosek:

Nie można instrukcją `goto` przeskoczyć z wnętrza jednej funkcji do wnętrza innej.

Wszystkie etykiety danej funkcji są już poza tą funkcją nieznane. Z zewnątrz tej funkcji nie można więc do nich skoczyć.

3.6.3 Zakres: obszar pliku

Na razie nasze krótkie programy mieściły się zwykle w jednym pliku dyskowym. W przyszłości będziemy pisać dłuższe, które dla wygody rozmieścimy w kilku plikach.

Jeśli w jednym z nich, na zewnątrz jakiegokolwiek bloku (także bloku funkcji) zadeklarujemy jakąś nazwę, to mówimy wówczas, że **taka nazwa jest globalna**. Ma ona zakres ważności pliku.

Oto przykład:

```
float fff ; // nazwa fff jest globalna
main()
{
    //...
}
```

Jednakże taka nazwa nie jest od razu automatycznie znana w innych plikach. Jej zakres ważności ogranicza się tylko do tego pliku, w którym ją zdefiniowaliśmy. I to w dodatku jedynie od miejsca deklaracji, do końca pliku.

3.6.4 Zakres: obszar klasy

To na razie tajemnica. O tym szczegółowo porozmawiamy w rozdziale poświęconym klasom.

3.7 Zasłanianie nazw

Możemy zadeklarować nazwę lokalną, identyczną jak istniejąca nazwa globalna. Nowo zdefiniowana zmienna zasłania wtedy, w danym lokalnym zakresie, zmienną globalną. Jeśli w tym lokalnym zakresie odwołamy się do danej nazwy, to kompilator uzna to za odniesienie się do zmiennej lokalnej. Spójrzmy na prosty przykład:

```
int k = 33 ; // ❶ zmienna globalna (obiekt typu int)
main()
{
    cout << "Jestem w main , k =" << k << "\n" ; // ❷
    { ////////////////////////////////// // ❸
        int k = 10 ; // zmienna lokalna ❹
        cout << " po lokalnej definicji k ="
            << k << endl ; // ❺
    } ////////////////////////////////// // ❻
    cout << "Poza blokiem k =" << k << endl ; // ❼
}
```

 Wykonanie tego fragmentu programu spowoduje pojawienie się na ekranie:

```
Jestem w main, k=33 ❷
po lokalnej definicji k =10 ❺
Poza blokiem k =33 ❼
```

Komentarz

- ❶ Definicja zmiennej globalnej *k* istniejąca gdzieś w programie, poza jakąkolwiek funkcją (także poza funkcją *main*).
- ❷ Odwołanie się do obiektu *k*. Jeszcze nie nastąpiła definicja lokalna, więc kompilator uznaje, że chodzi nam o obiekt *k* globalny.
- ❸ Otwieramy lokalny blok.
- ❹ Definiujemy obiekt lokalny o nazwie *k*.
- ❺ Lokalna nazwa *k* zasłoniła nazwę *k* globalną. Na ekranie zostaje więc wypisana wartość zmiennej lokalnej.
- ❻ Zamknięcie lokalnego bloku. Obiekt lokalny *k* przestaje istnieć, a jego nazwa przestaje być ważna. Skończyło się życie obiektu, skończył się także zakres ważności jego nazwy.
- ❼ Odwołanie się do nazwy *k* jest teraz rozumiane przez kompilator jako odwołanie się do globalnego obiektu *k*.



Mimo wszystko istnieje jednak możliwość odniesienia się do zasłoniętej nazwy globalnej.

Posłużymy nam do tego tzw. operator zakresu :: (dwa dwukropki). Oto przykład:

```
#include <iostream.h>

int k = 33 ;           // ❶ zmienna globalna (obiekt typu int)
/*****
main()
{
    cout << "Jestem w main , k =" << k << "\n" ;
    {
        int k = 10 ;           // zmienna lokalna ❷
        cout << "po lokalnej definicji k ="
            << k                // ❸
            << "\nale obiekt globalny k ="
            << ::k ;            // ❹
    }
    cout << "\nPoza blokiem k =" << k << endl ;
}
*****/
```

Wykonanie objawi się na ekranie jako:

```
Jestem w main , k =33
po lokalnej definicji k =10
ale obiekt globalny k =33
Poza blokiem k =33
```

Uwagi

- ❶ Definicja obiektu globalnego.
 - ❷ Definicja obiektu lokalnego.
 - ❸ Odwołanie się od obiektu lokalnego.
 - ❹ Odwołanie się wewnątrz lokalnego bloku do zasłoniętego obiektu globalnego. Sprawia to zapis z operatorem zakresu :: k
- Ten chwyt możliwy jest tylko w stosunku do zasłoniętego obiektu globalnego:

Jeśli nazwa lokalna zasłania inną nazwę lokalną, wówczas nie da się do niej dotrzeć takim operatorem zakresu.

3.8 Modyfikator const

Czasem chcielibyśmy w programie posłużyć się obiektem (np. typu int), którego zawartości nawet przez nieuwagę nie chcielibyśmy zmieniać. Obiekt tego typu, to tak zwany *obiekt stały*.

Mówiliśmy już o stałych dosłownych. Były to po prostu liczby, które napisane były w tekście programu. Tutaj nie chodzi o liczby, ale o obiekty, które mają

w sobie jakąś wartość. Paradoksem byłoby powiedzieć: chodzi o zmienne, które w programie mają się nie zmieniać.

Przykładem może być choćby program na liczenie pola koła, objętości kuli i czegoś jeszcze. Wielokrotnie w takim programie potrzebować będziemy liczby π . W tym celu zdefiniujemy sobie obiekt typu float i nadamy mu wartość odpowiadającą liczbie π .

```
float pi = 3.14 ;
```

Jeśli jednak chcemy mieć pewność, że nigdy, nawet przez nieuwagę nie zmienimy wartości naszej liczby π , wówczas taką definicję poprzedzamy słowem (modyfikatorem) const. Mówimy modyfikator, bo modyfikuje on zwykłą definicję tak, że teraz jest to definicja obiektu stałego.

```
const float pi = 3.14 ;
```

Zauważmy, że równocześnie inicjalizujemy tutaj nasze π wartością 3.14 - musimy to zrobić właśnie przy definicji. Później - przepadło! Od tej pory już nie można podstawić do obiektu const żadnej wartości. (Nawet takiej samej!)

```
const float pi = 3.14 ;
```

```
pi = 200 ;           //   !!!   (błąd)
pi = 3.14 ;          //   !!!   (błąd)
```

Wszelkie próby przypisania jakiegokolwiek wartości do obiektu π będą uznawane za błąd. Tutaj po raz pierwszy pojawia się nam różnica między inicjalizacją a przypisaniem.

Inicjalizacją nazywać będziemy nadanie obiektowi wartości w momencie jego narodzin.

Przypisaniem nazywać będziemy podstawienie do niego wartości w jakimkolwiek późniejszym momencie.

Oto przykłady inicjalizacji:

```
int a = 7 ;
const int cztery = 4 ;
```

Oto przykłady przypisania:

```
a = 100 ;
x = 25.5 ;
r = 30 * 7.5 ;
cztery = 4 ;           //   BŁĄD - jeśli był to obiekt const   !
```

Zapamiętaj

Obiekty const można inicjalizować, ale nie można do nich nic przypisać.

Słowa, słowa, słowa

Osobiście mam wstręt do takich „mądrych” słów jak: modyfikator, bo gdy kilka takich słów spotka się obok siebie w jednym zdaniu – trudno je zrozumieć. Dlatego słowa takie jak `const` nazywam sobie po prostu: przydomek. Nasz obiekt `pi` ma przydomek `const` - jest więc obiektem stałym.

Jeszcze jedna uwaga językowa: Mówimy „inicjaLIZacja”, a nie „inicjacja”. Jest ogromna różnica między tymi słowami. Jeśli będziesz uparcie mówił „inicjacja”, to zajrzyj sobie kiedyś do encyklopedii i sprawdź co to słowo znaczy. Trochę się pośmiejesz, a potem już zawsze będziesz mówił tylko: „inicjalizacja”

3.8.1 Pojedynek: `const` contra `#define`

Jest to paragraf dla programistów klasycznego C. Jeśli nie programowałeś w języku C, to opuść ten paragraf i przejdź do następnego.



Jeżeli programowałeś w języku C, to zapewne pamiętasz, że w klasycznym C stałe najczęściej definiowaliśmy sobie za pomocą dyrektywy preprocesora. Np.

```
#define PI 3.14
```

Ta forma w C++ jest także dopuszczalna. Pokażemy jednak dlaczego jest gorsza. Działanie dyrektywy `#define` jest mniej więcej takie, jakbyśmy - bezpośrednio po zakończeniu pisania programu - wydali edytorowi polecenie zastąpienia każdego słowa "PI" słowem "3.14". Natychmiast po tym przystępujemy do kompilacji.

Oto co straciliśmy:

- ❖ Nazwa `PI` jest kompilatorowi zupełnie nieznana. Nigdy się nawet nie domyśli, że w ogóle istniała. W programie jest tylko kilkakrotnie użyta liczba 3.14. Kompilator nie skojarzy, że chodzi o tę samą liczbę, chociaż człowiek od razu by się tu domyślił. Nie zawsze jednak sprawa jest tak oczywista.

```
#define LICZBA_SILNIKOW 4
```

Kompilator nie zgadnie, które z występujących w programie liczb 4 są określeniem liczby silników, a które są liczbą pór roku, liczbą nóg konia itd.

- ❖ W związku z tym, że nazwa `PI` jest kompilatorowi nieznana, dlatego kompilator nie potrafi sprawdzić czy dana nazwa została użyta w ramach jej zakresu ważności. Nie ma tu przecież żadnego zakresu ważności. Są tylko luźno porzucane liczby 3.14 (albo liczby 4).

Stała określona przy pomocy dyrektywy procesora `#define` jest znana od liniiki wystąpienia dyrektywy `#define` do liniiki `#undef` lub – gdy takiej nie ma – do końca pliku. Nie ma to jednak nic wspólnego z zakresem ważności. To tylko jakby obszar, na przestrzeni którego wykonujemy edytorem operacji zamiany znaków `PI` na znaki `3.14`

Czy dużo straciliśmy?

Raczej tak. Straciliśmy możliwość świadomego wyboru zasięgu nazwy. Nazwa może być przecież znana w jednej funkcji, a nieznana w innej. Kompilator nie może teraz nas ostrzec wypadku, gdybyśmy popełnili błąd.

Posłużenie się `#define` w naszym wypadku jest jakby zamianą nazwy na liczbę (stałą dosłowną).

Natomiast zdefiniowanie obiektu jako `const` sprawia, że powstaje nam w pamięci normalny obiekt (np. typu `float`, lub `int`). Dodatkowo obiekt ten ma nalepkę: „Nie Zmieniać Pod Żadnym Pozorem!”

Skoro jest to obiekt, to można poznać jego adres, pokazać na niego wskaźnikiem, itd. Gdybyśmy posłużyli się dyrektywą `#define`, to mielibyśmy w programie do czynienia z kilkoma stałymi dosłownymi. Sama liczba 3.14 nie ma adresu, nie można więc posługiwać się wobec niej wskaźnikiem.

Ostatni z argumentów, który chcę przedstawić, dotyczy posługiwania się programami uruchomieniowymi, czyli z angielska – debuggerami. Program taki pozwala na pracę krokową naszego programu, a w dodatku sprawdzenie, co – w danym momencie – tkwi w jakimś obiekcie naszego programu. Robi się to podając po prostu nazwę danego obiektu. Może Ci się wydać śmieszne pytanie debuggera, co w danym momencie tkwi w stałym obiekcie `PI`, jednak jeśli masz stałą `liczba_silników`, będącą jedną z wielu stałych w tym programie, to często się zdarza, że chciałoby się zapytać co tam właściwie jest.

Rozważmy poniższe dwa warianty. (Użycie małych lub wielkich liter w notacji nazw wynika tylko z tradycji).

```
#define ROZDZIELCZOSC 8192
#define KANALOW_W_BLOKU 128
#define CZYNNIK (ROZDZIELCZOSC / KANALOW_W_BLOKU)
#define DLUG_BUF (CZYNNIK*16*CZYNNIK)
```

W takiej sytuacji może się okazać konieczne upewnienie się ile właściwie wynosi `DLUG_BUF`. Przy tym sposobie definiowania stałej, jest to niemożliwe. Debugger odpowie, że nic mu nie wiadomo o nazwie `DLUG_BUF` (Podobnie jak nic nie wie o nazwach `ROZDZIELCZOSC`, `KANALOW_W_BLOKU`, `CZYNNIK`).

Jeśli jednak zastosujemy sposób obiektami `const`

```
const int rozdzielczosc = 8192 ;
const int kanalow_w_bloku = 128 ;
const int czynnik = (rozdzielczosc / kanalow_w_bloku) ;
const int dlug_buf = (czynnik * 4 * czynnik) ;
```

to zapytanie debuggera o to, co się kryje pod nazwą `dlug_buf` jest zupełnie legalne i w rezultacie otrzymamy odpowiedź: 16384

3.9 Obiekty register

register to jeszcze jeden typ przydomka (modyfikatora), który może zostać dodany w definicji obiektu. W tym wypadku można ten przydomek zastosować

do tzw. zmiennej automatycznej typu całkowitego. (Por. paragraf o zmiennych automatycznych – str. 97).

```
register int i ;
```

Dopisując ten przydomek dajemy kompilatorowi do zrozumienia, że bardzo zależy nam na szybkim dostępie do tego obiektu (bo np. zaraz będziemy używać takiego obiektu tysiące razy). Kompilator może uwzględnić naszą sugestię i przechowywać ten obiekt w rejestrze, czyli specjalnej komórce, do której ma bardzo szybki, niemal natychmiastowy dostęp.

Nie ma jednak gwarancji na to, że tak będzie w istocie. Niektóre kompilatory nie są aż tak sprytnie. Jak powiedziałem, jest to tylko pobożne życzenie, które kompilator może spełnić lub nie. Większość znanych mi kompilatorów, takie sugestie bierze pod uwagę i program wykonuje się nieco szybciej.

Jeśli deklarujemy zmienną jako `register`, to nie możemy starać się uzyskać jej adresu. Rejestr to nie jest kawałek pamięci, więc nie adresuje się go w zwykły sposób. Jeśli więc mimo wszystko spróbujemy dowiadywać się o ten adres, kompilator umieści ten obiekt w zwykłej pamięci, czyli tam, gdzie może nam określić (i podać) jego adres.

3.10 Modyfikator `volatile`

Jest to przydomek, który mówi kompilatorowi, że ma być ostrożny w kontaktach z danym obiektem.

```
volatile int m ;
```

Volatile ^{†)} – znaczy po angielsku: ulotny. Słowo to ostrzega, że obiekt może się w jakiś niezauważalny dla kompilatora sposób zmieniać. Kompilator więc nie powinien upraszczać sobie sprawy, tylko za każdym razem, gdy odwołamy się do tego obiektu – kompilator ma rzeczywiście zwrócić się do przydzielonych temu obiektowi komórek pamięci.

Zapytasz: —A czy kiedykolwiek bywa inaczej?

Tak. Dla naszego dobra tak. Wyjaśnijmy to. Otóż założmy, że deklarujemy zmienną określającą słowo stanu jakiegoś zewnętrznego urządzenia. Na przykład urządzenia mierzącego temperaturę oleju w silnikach.

```
int stan_miernika ;
```

To słowo może się zmieniać samo z siebie (bez wiedzy kompilatora), bo przypuśćmy, że do komputera dochodzą przewody z zewnętrznych układów pomiaru temperatury.

Domyślasz się już pewnie, że ten specjalny obiekt został utworzony w tej części pamięci komputera, która reprezentuje układ sprzęgający (interface) komputer

†) (czytaj: „woletail“)

z miernikiem temperatury. Załóżmy, że ten układ sprzęgający obsługiwany jest przez taki fragment programu

```
int a = 5,
    b = 6 ;

volatile int temperatura ;

cout << "Bieżąca temperatura = "
      << temperatura << endl ;           // ❶
a = b + 8 ;                               // ❷
b = 0 ;                                   // ❸
cout << "Bieżąca temperatura = "
      << temperatura << endl ;           // ❹
```

Jest ryzyko, że kompilator mógłby „pomyśleć” sobie tak: Pobrałem z komórki temperatura jej wartość i wypisałem ją na ekranie ❶.

Potem zajmowałem się jakimiś niezwiązanymi z temperaturą zmiennymi a oraz b, (❷, ❸) a teraz znowu mam wypisać na ekran zmienną temperatura. ❹ Zaraz, zaraz, nie muszę tracić czasu na odczytywanie jej z pamięci, miałem ją przecież gdzieś tu zapisaną na boku (czytaj: w rejestrze). Skoro więc jej od tamtej pory nie zmieniałem, to po prostu wypiszę tę wartość.

Kompilator wypisuje, a tu wybuch! Przez ten czas bowiem prawdziwa wartość obiektu temperatura zmieniła się bez wiedzy kompilatora i przekroczyła wartość krytyczną.

Słowo `volatile` przestrzega kompilator przed takim właśnie sprytem. Za każdym razem musi on rzeczywiście sięgnąć do komórki temperatura, a nie polegać na tym, co sobie zapisał „na boku”.

Modyfikator `volatile` (czyli: ulotny) oznacza, że obiekt tak określony może się zmieniać w sposób rzeczywiście ulotny, wy-
mykający się czasem spod kontroli kompilatora.

3.11 Instrukcja typedef

Instrukcja `typedef` pozwala na nadanie dodatkowej nazwy już istniejącemu typowi.

Przykładowo instrukcja

```
typedef int cena ;
```

sprawia, że możliwa staje się taka deklaracja

```
cena x ;           // co odpowiada: int x ;
cena a, b, c ;     // co odpowiada: int a, b, c ;
```

Po co robić takie sztuczki? Dlaczego nie napisać po prostu `int`?

Otóż taka możliwość jest bardzo przydatna. Wyobraź sobie program, w którym wielokrotnie występują zmienne typu `int`. Niektóre z nich mają określać cenę więc zastosowaliśmy tę instrukcję `typedef`. Pewnego dnia decydujemy, że dokładność liczb całkowitych nas nie zadowala - i chcielibyśmy by ceny repre-

zentowane były typem float. Co wtedy robimy? Odszukujemy w programie tę instrukcję typedef i zamieniamy ją na taką

```
typedef float cena ;
```

Tym sposobem wszystkie miejsca w programie gdzie używamy nazwy typu – np. deklarujemy obiekty typu cena – za jednym zamachem zmieniają się we właściwy sposób. Teraz

```
cena x ;           // odpowiada: float x ;
cena a, b, c ;     // odpowiada: float a, b, c ;
```

Jak widać jest to bardzo wygodne.

Typ, który określamy w instrukcji typedef, nie musi być wcale typem fundamentalnym. Równie dobrze może być to typ pochodny. Oto kilka przykładów:

```
typedef int  * wskaznik_do_int ;
typedef char * napis ;

wskaznik_do_int w1 ;   // czyli: int * w1 ;
napis komunikat ;     // czyli: char * komunikat ;
```

Poniższa instrukcja definiuje więcej nazw typów

```
typedef int calk, * wskc, natur ;
```

co umożliwia takie konstrukcje:

```
calk a ;           // czyli: int a ;
wskc w ;           // czyli: int * w ;
natur n ;          // czyli: int n ;
```

Zwróć uwagę jak umieszczona jest gwiazdka wskaźnika.

To zastosowanie instrukcji typedef radzę zapamiętać. Jeśli w przyszłości będziesz musiał posługiwać się skomplikowanymi wskaźnikami, to ta instrukcja zapewni Ci, że zapis Twoich programów będzie mimo wszystko czytelny.

Należy pamiętać, że instrukcja typedef nie wprowadza nowego typu, a jedynie synonim do typu już istniejącego.

Instrukcją typedef nie możemy redefiniować nazwy, która już istnieje w bieżącym zakresie ważności. Konkretnie: jeśli mamy już deklarowaną nazwę – np. calk określającą funkcję wykonującą całkowanie – to nie możemy użyć instrukcji

```
typedef int calk ;
```

bo nazwa calk jest już zajęta.

3.12 Typy wyliczeniowe enum

To bardzo ciekawa rzecz. Jest to osobny typ dla liczb całkowitych, a przydaje się on w wielu sytuacjach.

Często zdarza się, że w obiekcie typu całkowitego chcemy przechować nie tyle liczbę, co raczej pewien rodzaj informacji. Oczywiście musimy uczynić to wpisując tam liczbę, ale liczba ta ma dla nas szczególne znaczenie. Wtedy warto skorzystać z typu wyliczeniowego.

Jak definiuje się taki typ wyliczeniowy? Pokażemy to od razu na przykładzie. Chcemy za pomocą liczb określać jakieś działanie układu pomiarowego. Chcemy mieć jakąś zmienną o nazwie `co_robic`. Do niej będziemy wstawiali liczbę określającą żadaną akcję. Oto jak te akcje sobie ponumerujemy:

0	start_pomiaru
1	odczyt_pomiaru
54	zmiana_probki
55	zniszczenie_probki

Typ wyliczeniowy definiuje się według schematu

```
enum nazwa_typu {lista_wyliczeniowa} ;
```

Co w naszym wypadku wyglądać może tak:

```
enum akcja {
    start_pomiaru = 0,
    odczyt_pomiaru = 1,
    zmiana_probki = 54,
    zniszczenie_probki = 55 } ;
```

Zdefiniowaliśmy nowy typ o nazwie `akcja`. A oto definicja zmiennej tego typu:

```
akcja co_robic ;
```

Oznacza to, że `co_robic` jest zmienną, do której można podstawić tylko jedną z określonych na liście wyliczeniowej `akcja` wartości.

To znaczy legalne są takie operacje

```
co_robic = zmiana_probki ;
co_robic = start_pomiaru ;
```

a nielegalne są operacje

```
co_robic = 1 ;
co_robic = 4 ;
```

Nic, co nie jest na liście wyliczeniowej tego typu wyliczeniowego, nie może zostać podstawione do zmiennej tego typu `akcja`.

Na liście wyliczeniowej zauważamy liczby. Mimo jednak, że odczyt pomiaru jest – jak widzimy – reprezentowany przez liczbę 1, to tej liczby nie mogliśmy wstawić do zmiennej typu `akcja`. Tylko to, co jest na liście.

To bardzo ważna cecha. Dzięki temu nawet przez nieuwagę nie wpisujemy do zmiennej `co_robic` czegoś innego. Nawet gdyby to coś przypadkowo pasowało – jako wartość liczbową.

Lista wyliczeniowa

Reprezentacja liczbowa elementów listy może być przez nas wybierana wtedy, gdy definiujemy dany typ wyliczeniowy. Nawiasem mówiąc gdybyśmy w naszym przykładzie nie napisali tych liczb 0, 1 -to takie właśnie liczby byłyby podstawione tam przez domniemanie.

Oto przykład innego typu wyliczeniowego, gdzie reprezentacje liczbowe są inne:

```
enum operacja_dyskowa { czytaj_blok,  
                        pisz_blok,  
                        przeskocz_blok = 5,  
                        przeskocz_znacznik } ;
```

kolejne pozycje na tej liście mają następujące reprezentacje liczbowe

- ❖ `czytaj_blok` : 0 – bo jeśli nie określiliśmy inaczej, to wyliczanie zaczyna się od 0
- ❖ `pisz_blok` : 1 – znowu nie było określenia, więc z wyliczanki wynika, że będzie to liczba następna, czyli 1
- ❖ `przeskocz_blok` : 5 – tu widzimy wyraźne życzenie, by była to liczba 5
- ❖ `przeskocz_znacznik` : 6 – znowu nie było życzeń, więc kompilator bierze następną liczbę, czyli 6.

Te reprezentacje liczbowe nie muszą koniecznie się różnić. Mogą być na przykład dwa elementy na liście o nazwie `przewin_tasme` oraz `rozladuj_tasme`, które będą miały tę samą reprezentację. Oczywiście robimy to celowo, gdy chcemy umożliwić nadanie dwu nazw tej samej akcji.

To, jakie są reprezentacje liczbowe – nie musi nas wcale obchodzić. Są to jakieś wartości. W podprogramie, który odpowiada za pracę z dyskiem, żadaną akcję porównujemy nie z liczbami, tylko znowu z elementami tej listy.



Dla wtajemniczonych

Typy wyliczeniowe naprawdę bardzo się przydają. W mojej praktyce chyba najczęściej przy wysyłaniu argumentów do funkcji. Funkcja może spodziewać się argumentu typu wyliczeniowego (np. `operacja_dyskowa`) i kompilator będzie pilnował, by tylko argument tego typu został tam wysłany. Jeśli się pomylimy i do funkcji wyślemy coś innego (np. dowolną liczbę `int` lub inny typ wyliczeniowy np. `kolor`) – wówczas kompilator od razu znajdzie nam ten błąd.

Zwróciłeś może uwagę, że do tej pory nasze programy były bardzo prymitywne. To między innymi dlatego, że milcząco założyłem, iż wiesz co oznaczają symbole:

+ - * < >

Tak jednak dalej nie można. O tych i innych operatorach musimy porozmawiać teraz bliżej.

Operatorów jest wiele rodzajów. Ich opanowanie nie wymaga jednak większego wysiłku, bo przeważnie określają one podstawowe operacje arytmetyczne i logiczne, znane nam przecież ze szkoły. Przystąpmy zatem do rzeczy.

4.1 Operatory arytmetyczne

Operatory:

+ dodawania,
- odejmowania,
* mnożenia,
/ i dzielenia

nie wymagają chyba żadnych wyjaśnień. Oto przykłady wyrażeń, w których występują te operatory:

a = b + c ;	// dodawanie
a = b - c ;	// odejmowanie
a = b * c ;	// mnożenie
a = b / c ;	// dzielenie

Operatory te wykonują działania na dwóch obiektach i dlatego nazywamy je *dwuargumentowymi*. Po prostu dodają do siebie dwie liczby, albo dwie zmienne jakiegoś typu. Te mianowicie, które stoją po obu stronach symbolu operatora.

Zatem dodawanie

$$a + 7$$

działa tu na dwóch argumentach:

- 1) obiekcie *a* (np. zmiennej),
- 2) na liczbie 7 (stałej dosłownej)

Praktyczna uwaga:

Zapis Twoich programów będzie dla Ciebie i innych czytelniejszy, gdy przyjmiesz zasadę, by każdy operator po obu stronach miał spację. Na dowód porównaj dwa identyczne wyrażenia:

$$(a+b+0.32)/c-7.1*(12.4+x)+75.3$$

$$(a + b + 0.32) / c - 7.1 * (12.4 + x) + 75.3$$

Kompilatorowi jest tu wszystko jedno. Ty jednak czasem pomyśl też o sobie.

4.1.1 Operator % czyli modulo

Także dwuargumentowym operatorem jest operator dzielenia modulo % (symbol procentu). Jest to inaczej mówiąc operator, dzięki któremu otrzymujemy resztę z dzielenia.

Wyrażenie

$$10 \% 3$$

ma więc wartość 1, gdyż taka jest reszta z dzielenia 10 przez 3. Na przykład po wykonaniu takiego fragmentu programu:

```
int x = 8,
    n = 5 ;

cout << "wynik = " << (x % n) << endl ;
```

 na ekranie pojawi się

wynik = 3

a to dlatego, że 8 dzielone przez 5 daje resztę z dzielenia równą 3.

Operator ten może się przydać często w takich sytuacjach jak poniżej:

```
#include <iostream.h>
main()
{
    int i ;
    for(i = 0 ; i < 64 ; i = i + 1)
    {
        if( i % 8) // ❶
        { // ❷
            cout << "\t" ; // wypis tabulatora
        }
        else
        { // ❸
            cout << "\n" ; // przejście do nowej linii
        }
    }
}
```

```

    }
    cout << i ;
}

```

// 4

W rezultacie wykonania tego programu

na ekranie pojawią się liczby w 8 kolumnach.

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Uwagi

- 1 To miejsce, gdzie działa nasz operator modulo `%`. W zależności czy wynik jego działania jest zerowy czy niezerowy, podejmowana jest później odpowiednia akcja. Wartość wyrażenia $(i \% 8)$ jest niezerowa na przykład w sytuacjach:

1 % 8 2 % 8 3 % 8 ... 7 % 8

natomiast wynik jest zerowy na przykład w sytuacjach:

0 % 8 8 % 8 16 % 8 24 % 8

- 2 Tę akcję podejmuje się, gdy wartość wyrażenia warunkowego jest niezerowa. Jest to wypisanie na ekranie tabulatora.
- 3 Tę akcję podejmuje się, gdy wartość wyrażenia warunkowego jest zerowa. Jest to wypisanie na ekranie znaku nowej linii. Czyli po prostu przejście do nowej linii.
- 4 To jest miejsce, gdzie odbywa się wypisanie liczby na ekranie.



Priorytet omawianych operatorów jest taki sam, jak do tego przywykliśmy w matematyce. Czyli zapis

$$a + b \% c * d - f$$

oznacza to samo co

$$a + ((b \% c) * d) - f$$

Innymi słowy mnożenie i dzielenie wykonywane jest przed dodawaniem lub odejmowaniem.

4.1.2 Jednoargumentowe operatory + i -

Plus i minus mogą też wystąpić jako operatory jednoargumentowe. Nic w tym dziwnego, to także znamy ze szkoły.

Oto przykład:

```
+12.7
-x
-(a*b)
```

Jednoargumentowy operator + właściwie nie robi nic, natomiast jednoargumentowy operator - zamienia wartość danego wyrażenia na liczbę przeciwną.

```
int i = 5 ;
cout << "oto dwa wydruki: " << i << " oraz " << -i ;
cout << "\nA teraz : " << -(-(i+1)) ;
```

W rezultacie wykonania tego fragmentu na ekranie pojawi się

```
oto dwa wydruki: 5 oraz -5
A teraz : -6
```

Zwracam uwagę, że nie ma tu żadnego odejmowania, tylko tworzenie liczby przeciwnej w stosunku do wyrażenia, które zostało poddane tej operacji.

Operatory te są jednoargumentowe, bo działają na tylko jednym argumencie (tym stojącym bezpośrednio po prawej stronie znaku).

4.1.3 Operatory inkrementacji i dekrementacji

Inaczej mówiąc: operatory zwiększenia o jeden i zmniejszenia o jeden.

W pętlach bardzo często wykonywaliśmy działania w rodzaju

```
i = i + 1 ;           // zwiększenie o 1
k = k - 1 ;           // zmniejszenie o 1
```

Zwiększenie o 1 (inkrementacja) lub zmniejszenie o 1 (dekrementacja) jest działaniem tak często spotykanym w programowaniu, że w języku C++ mamy dla wygody specjalne operatory. Oto one:

```
i++ ;               // czyli to samo co: i = i + 1
k-- ;               // czyli to samo co: k = k - 1
```

Dygresja:

Teraz możesz już rozwikłać zagadkę dlaczego C++ nazywa się właśnie C++. Otóż, gdy w czasach archaicznych język B rozwinął się w język C, żartowano jak będzie się nazywał jego następca. Proroctwa mówiły, że pewnie: Język D. Proroctwa były jak widać chybione, bo następca C nazywa się C++, co należy rozumieć jako „lepsza wersja C”

Operatory inkrementacji ++ i dekrementacji -- są jednoargumentowe. Oba mogą mieć dwie formy:

- przedrostkową (prefix) czyli wtedy, gdy operator stoi z lewej strony argumentu, np: ++a, --p (czyli **przed** argumentem)
- końcówkową (postfix), czyli wtedy, gdy operator stoi po prawej stronie argumentu np.: a++, p-- (po prostu **po** argumentem)

Jest w tym pewna, bardzo sprytna różnica. Rozważmy to na przykładzie operatora inkrementacji (zwiększania)

- ❖ Jeśli operator inkrementacji (zwiększania) stoi *przed* zmienną, to najpierw jest ona zwiększana o 1, następnie ta zwiększona wartość staje się wartością wyrażenia.
- ❖ W wypadku, gdy operator inkrementacji stoi *za* zmienną, to najpierw brana jest stara wartość tej zmiennej i ona staje się wartością wyrażenia, a następnie - jakby na pożegnanie pracy z obiektem - zwiększany jest on o 1. Zwiększenie to nie wpłynęło więc na wartość samego wyrażenia.

Może brzmi to bardzo zawile, jest jednak bardzo proste. Zobaczmy to na przykładzie:

```
#include <iostream.h>
main()
{
    int  a = 5,
        b = 5,
        c = 5,
        d = 5 ;

    cout << "A oto wartosc poszczegolnych wyrazen\n"
          << "(nie mylic ze zmiennymi)\n" ;

    cout << "++a = " << ++a << endl
          << "b++ = " << b++ << endl
          << "--c = " << --c << endl
          << "d-- = " << d-- << endl ;

    // teraz sprawdzamy co jest obecnie w zmiennych

    cout << "Po obliczeniu tych wyrazen, same "
          << "zmienne maja wartosci\n"
          << "a = " << a << endl
          << "b = " << b << endl
          << "c = " << c << endl
          << "d = " << d << endl ;
}
```

W rezultacie zobaczymy na ekranie:

```
A oto wartosc poszczegolnych wyrazen
(nie mylic ze zmiennymi)
++a = 6
b++ = 5
--c = 4
d-- = 5
```

Po obliczeniu tych wyrazen, same zmienne maja wartosci

```

a = 6
b = 6
c = 4
d = 4

```

Operator inkrementacji stojący **przed** argumentem nazywa się często operatorem *preinkrementacji*, Natomiast stojący **za** argumentem nazywa się operatorem *postinkrementacji*.

Podobnie w wypadku operatorów zmniejszania mówimy o operatorach *predekrementacji* i *postdekrementacji*.

4.1.4 Operator przypisania =

To jest chyba najbardziej oczywiste. Do tej pory wielokrotnie posługiwaliśmy się tym operatorem

```
m = 34.88 ;
```

Powoduje on, że do obiektu stojącego po jego lewej stronie przypisana (podstawiona) zostaje wartość wyrażenia stojącego po prawej. Zatem w zmiennej *m* znajdzie się liczba 34.88

Jest to operator dwuargumentowy, gdyż pracuje na dwóch argumentach stojących po jego obu stronach.

Dobrze wiedzieć i pamiętać, że każde przypisanie samo w sobie jest także wyrażeniem mającym taką wartość, jaka jest przypisywana.

Zatem wartość wyrażenia

```
(x = 2)
```

jako całości jest także 2. Zauważ

```
int a, x = 4 ;
cout << "Wart. wyrażenia przypisania : " << (a = x) ;
```

W rezultacie na ekranie pojawi się napis:

```
Wart. wyrażenia przypisania : 4
```

Bowiem wyrażenie $(a=x)$ nie tylko, że wykonuje podstawienie, ale jeszcze samo ma wartość równą temu, co podstawia.

Może się zdarzyć, że po obu stronach operatora przypisania stać będą argumenty różnego typu. Nastąpi wówczas niejawną zamiana typu wartości przypisywanej na typ zgadzający się typem tego, co stoi po lewej stronie. Przykładowo:

```
int a ;
a = 3.14 ;
cout << a ;
```

nastąpi tu zamiana (mówimy *konwersja*) typu zmiennoprzecinkowego na typ *int*. Nastąpi ona niejawnie, bez ostrzeżeń. Po prostu zostanie wzięta pod uwagę tylko część całkowita liczby 3.14 — czyli 3 — i to też zobaczymy na ekranie.

O konwersjach będziemy jeszcze mówić w osobnym rozdziale (str. 399).



Warto wspomnieć, że istnieją jeszcze inne operatory przypisania specyficzne dla języków C i C++. Można się bez nich obejść, najlepszy dowód, że obchodzi się bez nich matematyka. Z drugiej strony jednak bardzo ułatwiają życie. O tych innych operatorach przypisania będziemy mówić w jednym z dalszych paragrafów.

4.2 Operatory logiczne

Po operatorach arytmetycznych pora na operatory logiczne. Jest ich kilka rodzajów.

4.2.1 Operatory relacji

Operator

<	mniejszy niż...
<=	mniejszy lub równy...
>	większy niż...
>=	większy lub równy...

są operatorami relacji, w wyniku których otrzymujemy odpowiedź typu: prawda lub fałsz.

Używanie tych operatorów nie przysparza żadnych kłopotów. Oto przykład:

```
if(a > 5)
{
    cout << " a jednak wieksze od 5! ;"
}
```

Następnymi operatorami tego typu są operatory

==	jest równy...
!=	jest różny od...

Zauważyć należy, iż operator == składa się z dwóch stojących obok siebie znaków '='. (Nie ma tam w środku spacji).

Jest bardzo częstym błędem omyłkowe postawienie tylko jednego – zamiast dwóch znaków ==. Rezultat takiej pomyłki możemy prześledzić na następującym przykładzie

```
int a = 5, b = 100 ;
cout << "Dane sa: a= " << a << " b= " << b ;
```

// tu następuje fatalna linijka

```
if(a = b)
    cout << "\nZachodzi równosc \n" ;
else
    cout << "\nNie zachodzi równosc \n" ;
```

```
cout << "Sprawdzam ze: a= " << a << " b= " << b ;
```

// ❶
// ❷

// ❸



Po wykonaniu tego fragmentu programu na ekranie ujrzymy

```
Dane sa: a= 5 b= 100
Zachodzi równosc
Sprawdzam ze: a= 100 b= 100
```



Dlaczego tak się stało ?

Otóż w instrukcji `if` ❶ zamiast porównania (operator `==`) zapisaliśmy przypisanie (operator `=`). Wiemy już z poprzednich paragrafów, że przypisanie to nie tylko przypisanie, ale w dodatku samo wyrażenie przypisania ma wartość równą wartości będącej przedmiotem przypisania – czyli w naszym wypadku zapis

```
if(a = b)...
```

odpowiada zapisowi

```
if(100)...
```

100 jest różne od zera, czyli przez instrukcję `if` traktowane jest jako wynik „prawda” i tym samym wykonywana jest instrukcja ❷. O tym, że zamiast porównania nastąpiło przypisanie (podstawienie) przekonuje nas rezultat wypisany na ekran instrukcją ❸.

Zniszczyliśmy sobie przez nieuwagę wartość zmiennej `a`. Z drugiej jednak strony nie został popełniony żaden błąd składniowy. Po prostu zamiast działania

```
if(a == b)...
```

wykonałoby

```
a = b ;
if(a)...
```

Zatem w naszym przykładzie zrobiliśmy nie to, co chcieliśmy.

Jednak są sytuacje, gdzie chcemy w instrukcji `if` takiego właśnie przypisania, a nie porównania. Chcemy bowiem zyskać na czasie wykonania stosując zamiast dwóch instrukcji jedną.

Jest to składniowo poprawne, jednak niektóre troskliwe kompilatory ostrzegają programistę jeśli przy sprawdzaniu warunku `if` znajdą tam operację przypisania. Tak na wszelki wypadek.

4.2.2 Operatory sumy logicznej `||` i iloczynu logicznego `&&`

Operatory te realizują

<code> </code>	- sumę logiczną	- czyli operację logiczną LUB (alternatywa)
<code>&&</code>	- iloczyn logiczny	- czyli operację I (koniunkcja)

Na przykład:

```
int k = 2 ;
if( (k == 10) || (k == 2) )           // alternatywa
{
    cout << "Hurra ! k jest równe 2 lub 10 ! " ;
}
```

co czytamy: jeśli k równe jest 10 *lub* k równe jest 2 to wtedy...

Przykład na koniunkcję:

```
int m = 22 , k = 77 ;
if( (m > 10) && (k > 0) )           // koniunkcja
{
    cout << "Hurra ! m jest większe od 10 "
          << "i równocześnie k jest "
          << "większe od zera ! \n" ;
}
```



Przypominam, że obliczanie wartości wyrażenia logicznego odbywa się w ten sposób, że wynik „prawda” daje rezultat 1, a wynik „fałsz” daje rezultat 0.

Wyrażenia logiczne tego typu obliczane są od lewej do prawej. Dobrze pamiętać, że kompilator oblicza wartość wyrażenia dotąd, dopóki na pewno nie wie jaki będzie wynik. Oznacza to, że w wyrażeniu

```
(a == 0) && (m == 5) && (x > 32)
```

kompilator obliczał będzie od lewej do prawej, a jeśli pierwszy czynnik koniunkcji nie będzie prawdziwy, dalsze obliczanie zostanie przerwane. Nie ma bowiem dalszej potrzeby: – co by tam dalej nie zostało obliczone, i tak koniunkcja nie może być już prawdziwa. Kompilator oszczędza sobie więc pracy.

Wydawać by się mogło, że nie musimy o tym wszystkim wiedzieć. A jednak przydaje się to. Wyobraź sobie, że chciałeś być taki sprytny i upiec parę pieczeni na jednym ogniu:

```
int i = 6 , d = 4 ;
if( (i > 0) && (d++) )
{
    cout << "warunek spełniony !" ;
}
```

Nie dość, że wykonujesz operacje logiczne, to jeszcze chciałbyś, by przy okazji pracy na obiekcie d – zwiększyć go o 1.

Otóż jest to pułapka, bo jeśli pierwszy człon koniunkcji nie będzie prawdziwy, to kompilator uzna już, że nie opłaca się zajmować dalszymi. W ten sposób nie dojdzie do wykonania wyrażenia d++ na co może tak liczyliśmy.

Podobnie jest w wypadku alternatywy. Jeśli w wyrażeniu

```
if( i || (k > 4) )
{
```

```

    // .....
}

```

pierwszy człon alternatywy (zmienna `i`) jest różny od 0 (czyli „prawda”), to dalsze obliczenia nie są już konieczne. Już w tym momencie wiadomo, że alternatywa ta jest spełniona.

Operator negacji: !

Operator negacji jest operatorem jednoargumentowym. Jego argument stoi po prawej jego stronie. Operator ten ma postać `!` (wykrzyknik)

```
!i
```

Powyższe wyrażenie ma wtedy wartość „prawda”, gdy `i` jest równe zero.

Oto przykłady jak używamy takiego operatora:

```

int i = 0 ;
int gotowe ;

if(!i)
    cout << "Uwaga, bo i jest równe zero\n" ;

gotowe = 0 ;
if(! gotowe){
    cout << jeszcze nie gotowe !
}

```

Operatory bitowe

Mówiliśmy o operatorach arytmetycznych, operatorach logicznych, czas teraz na operatory, które są charakterystyczne dla tego specyficznego sposobu przechowywania informacji w komputerze – jakim jest zapis binarny.

Jak powszechnie wiadomo – w komputerze informacje (liczby i znaki) z kodowane są w poszczególnych komórkach pamięci za pomocą różnych kombinacji zer i jedynek. Te elementarne jednostki informacji nazywane są bitami (bit – ang. kawałek).

Komputer nie odnosi się do każdego z takich bitów osobno. Grupuje je w większe jednostki zwane słowami. Słowo jest jednostką informacji przetwarzaną przez komputer. Zależnie od komputera, różne mogą być rozmiary takich słów. Przykładowo: słowo w komputerze klasy IBM PC/AT składa się z kombinacji szesnastu bitów. Czyli z szesnastu zer lub jedynek.

Komputer przetwarza całe słowa, w których zwykle zapisane są liczby. Nie wszystko jednak w komputerze musi oznaczać liczby. To tak, jak w kokpicie samolotu oprócz informacji liczbowych o wysokości, prędkości, wznoszeniu itd, mamy też informacje logiczne: podwozie schowane lub nie. Oświetlenie samolotu włączone lub nie.

W komputerze informacje takie można zebrać i umieścić razem na poszczególnych bitach jednego słowa pamięci. Do pracy na tak umieszczonej informacji służą nam właśnie operatory bitowe.

Oto lista operatorów bitowych:

<<	przesunięcie w lewo
>>	przesunięcie w prawo
&	bitowy iloczyn logiczny (bitowa koniunkcja)
	bitowa suma logiczna (bitowa alternatywa)
^	bitowa różnica symetryczna (bitowe exclusive OR)
~	bitowa negacja

W nazwach tych operatorów przewija się słowo „bitowy”, „bitowa”. Dlatego w przykładach ilustrujących zastosowanie tych operatorów będziemy pokazywali jak działają one na poszczególne bity. Zakładam, że mamy do czynienia z komputerem, gdzie obiekt typu `int` kodowany jest na 16 bitach (takie są chociażby komputery klasy IBM PC/AT).



Dygresja:

operator << (wypisujący na ekran)
oraz operator >> (wczytujący z klawiatury)

Symbole operatorów przesunięcia w lewo i w prawo przypominają nam poznane wcześniej operatory, którymi posługujemy się do wypisywania na ekran i wczytywania z klawiatury.

```
cout << "podaj liczbę : ";
cin >> x ;
```

Zgadza się, to są rzeczywiście te same symbole. Przez bibliotekę wejścia/wyjścia zostały one tylko wypożyczone. (Nawet takie rzeczy da się robić w C++ !) Nie ma jednak ryzyka nieporozumień. Operatory przesunięcia są rzeczywiście operatorami przesunięcia w sytuacjach, gdy po obu stronach symbolu << lub >> stoją argumenty typu całkowitego.

Jeśli argumentem z lewej strony jest `cin` lub `cout`, to operatory te oznaczają przesyłanie informacji z klawiatury, lub na ekran. Ta pożyczka nastąpiła dlatego, że wygląd operatorów << i >> dobrze sugeruje akcję, o którą chodzi. W rozdziale o tzw. przeładowaniu operatorów dowiesz się jak łatwo samemu dla swoich własnych celów robić takie pożyczki.

4.3.1 Przesunięcie w lewo <<

Jest to dwuargumentowy operator pracujący na argumentach (operandach) typu całkowitego

```
zmienna << ile_miejsc
```

Bierze on wzór bitów zapisany w danej zmiennej, przesuwając go o żadaną ilość miejsc w lewo i jako rezultat zwraca ten nowy wzór. Bity z prawego brzegu słowa uzupełnione zostają zerami. Bity z lewego brzegu zostają zgubione.

Przykładowo na skutek takiej instrukcji

```
int a = 0x40f2 ;
int w ;
```



```
w = a << 3 ;
```

następuje przesunięcie o trzy miejsca w lewo. Oto jak wyglądają poszczególne obiekty. Dla łatwiejszej orientacji poszczególne bity słowa zgrupowałem w czwórki.

```
a    0100 0000 1111 0010
w    0000 0111 1001 0000
```

Sam obiekt *a* nie został zmieniony. Posłużył on tylko jako wartość początkowa. Rezultat został złożony w zmiennej *w*.

Często chodzi nam o to, by przesunąć bity danej zmiennej, a rezultat ma się znaleźć z powrotem w tej zmiennej. To nic trudnego:

```
a = a << 3 ;
```

(Niebawem poznamy jeszcze lepszy sposób na to: operator `<<=`).

4.3.2 Przesunięcie w prawo >>

Jest to dwuargumentowy operator pracujący na argumentach (operandach) typu całkowitego

```
zmienna >> ile_miejsc
```

Bierze on wzór bitów zapisany w danej zmiennej, przesuwając go o żadaną ilość miejsc w prawo i jako rezultat zwraca ten nowy wzór. Bity z prawego brzegu wychodzące poza zakres słowa są gubione.

Jest tu jednak coś, co odróżnia go od opisanego wcześniej kolegi: zachowanie przy uzupełnianiu bitów z lewej strony.

Otóż:



jeśli nasz operator pracuje na danej

- `unsigned` (bez znaku)

lub

- `signed` (ze znakiem), **ale** dana zmienna mieści w sobie akurat liczbę nieujemną

- wówczas bity z lewego brzegu są uzupełniane zerami. To jest gwarantowane.

```
unsigned int d = 0x0ff0 ;
unsigned int r ;
```

```
r = d >> 3 ;
```

Oto jak wtedy wygląda rozkład bitów:

```
d    0000 1111 1111 0000
r    0000 0001 1111 1110
```



Jednak jeśli pracuje on na danej typu `signed` (ze znakiem) i jest tam liczba ujemna, to rezultat może zależeć od typu komputera, na którym pracujemy. Może nastąpić uzupełnianie brakujących z lewej strony bitów zerami, a może jedynekami. To – jako się rzekło – zależy już od typu komputera.

```
signed int d = 0xff00 ;
signed int r ;
```

```
r = d >> 3 ;
```

oto jak wtedy wygląda rozkład bitów – pokazujemy tu dwa warianty:

```
d   1111 1111 0000 0000
r   0001 1111 1110 0000
r'  1111 1111 1110 0000
```

Kompilatorem Borland C++ na komputerze IBM PC/AT uzyskuje się ten drugi wariant oznaczony tu jako r'.

4.3.3 Bitowe operatory sumy, iloczynu, negacji, różnicy symetrycznej

Te dwuargumentowe operatory także działają na argumentach całkowitych. Ich działanie zilustrujemy poniżej

```
int m = 0x0f0f ;
int k = 0x0ff0 ;
```

```
int a, b, c, d ;
```

```
a = m & k ;           // iloczyn bitowy
b = m | k ;           // suma bitowa
c = ~m ;              // negacja bitów
d = m ^ k ;           // różnica symetryczna (XOR) bitów
```

A oto jak wyglądają poszczególne obiekty. Dane wejściowe i wartości wyrażeń:

```
m   0000 1111 0000 1111
k   0000 1111 1111 0000
```

m & k	0000 1111 0000 0000	bitowa koniunkcja
m k	0000 1111 1111 1111	bitowa alternatywa
~m	1111 0000 1111 0000	bitowa negacja
m ^ k	0000 0000 1111 1111	bitowe exclusive or

W zasadzie sprawa jest jasna i nie wymaga komentarza. Podkreślić należy jednak jaka jest:

4.4 Różnica między operatorami logicznymi, a operatorami bitowymi

Czyli między operatorami

```
&&      oraz      &
||      oraz      |
```

Otóż pamiętajmy, że wynikiem działania zwykłego operatora logicznego (np. koniunkcja logiczna) jest wynik „prawda” lub „fałsz”, czyli wynikiem jest słowo z zapisaną tam wartością 1 lub 0.

W wypadku operacji logicznych np. `m && k`

kompilatora nie interesuje rozkład bitów w zmiennej `m` czy w zmiennej `k`. Sprawdza on tylko czy jest tam wartość równa zero, czy różna od zera. To samo z drugim argumentem. Wreszcie na tych dwóch wartościach typu „prawda” lub „fałsz” dokonuje koniunkcji. Wynikiem jest 1 lub 0 (czyli „prawda” lub „fałsz”).

Natomiast operatory bitowe np. `m & k`

zaglądają do wnętrza danego słowa. Na poszczególnych pojedynczych bitach tych słów dokonują operacji koniunkcji. Czyli biorą pierwszy bit z jednej zmiennej i pierwszy bit z drugiej zmiennej i na tych bitach wykonują operacji koniunkcji. Rezultatem jest 0 lub 1 i ten rezultat wstawiają do pierwszego bitu zmiennej wynikowej. Następnie to samo z drugim bitem i wszystkimi dalszymi. W rezultacie więc otrzymujemy wynik – będący słowem o specyficznym układzie bitów.

Taki wynikowy układ bitów można interpretować jako liczbę, dlatego te bitowe operatory przypominają operatory arytmetyczne.

W naszym ostatnim przykładzie wyrażenie `m & k` można zinterpretować jako liczbę

3840

Możesz się o tym przekonać wypisując wartość wyrażenia:

```
cout << (m & k) ;
```

4.5 Pozostałe operatory przypisania

Z paragrafem tym czekałem do tej pory mimo, że jest banalnie prosty.

Poznaliśmy już wcześniej operator przypisania (podstawienia) =

W zasadzie może on nam wystarczyć, jednak dla wygody mamy jeszcze do dyspozycji następujące operatory:

<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>
<code>%=</code>	<code>>>=</code>	<code><<=</code>	<code>&=</code>
<code> =</code>	<code>^=</code>		

Ich działanie jest bardzo proste – pokażemy to na przykładzie. Kto zrozumie zasadę w pierwszej linijce, nie musi się trudzić z zapamiętywaniem, bo analogie narzucają się same. I tak:

<code>i += 2</code>	oznacza	<code>i = i + 2</code>
<code>i -= 2</code>	oznacza	<code>i = i - 2</code>
<code>i *= 2</code>	oznacza	<code>i = i * 2</code>
<code>i /= 2</code>	oznacza	<code>i = i / 2</code>
<code>i %= 2</code>	oznacza	<code>i = i % 2</code>
<code>i >>= 2</code>	oznacza	<code>i = i >> 2</code>
<code>i <<= 2</code>	oznacza	<code>i = i << 2</code>
<code>i &= 2</code>	oznacza	<code>i = i & 2</code>

$$\begin{array}{l} i \mid = 2 \\ i \wedge = 2 \end{array}$$

oznacza
oznacza

$$\begin{array}{l} i = i \mid 2 \\ i = i \wedge 2 \end{array}$$

Analogia ta nie jest jednak zupełna: jeśli i jest wyrażeniem, to w naszym nowym zapisie jest ono obliczane tylko jednokrotnie (w starym - dwa razy). Może to mieć znaczenie, jeśli wyrażenie to ma jakieś działanie uboczne (np. inkrementacja).

Zwracam też uwagę, że te operatory wyglądają tak, iż znak równości następuje jako drugi. Jeśli się pomyśli i napiszesz odwrotnie

$$i = -2 ; \quad // \text{ czyli } i = -2 ;$$

to zrobisz postawienie liczby -2 do zmiennej i .

4.6 Wyrażenie warunkowe

Powtarzam: *wyrażenie*, a nie instrukcja. Jest to wyrażenie, które obliczane jest czasem na taką wartość, a czasem na inną. Oto jego forma:

(warunek) ? wartość1 : wartość2

Przykładowo

$(i > 4) ? 15 : 10$

wyrażenie to (jako całość) w zależności od spełnienia lub niespełnienia warunku $(i > 4)$ przyjmuje różną wartość.

Jeśli warunek jest spełniony — to wartość wyrażenia wynosi 15,

jeśli zaś nie spełniony — wartość tego całego wyrażenia wynosi 10

Jest to bardzo wygodna konstrukcja, bo pozwala zapakować ją do wnętrza innych wyrażeń

np.

$c = (x > y) ? 5 : 12 ;$

do zmiennej c zostanie podstawiona liczba 5, jeśli rzeczywiście x jest większe od y , a liczba 12 jeśli x nie jest większe od y .

Oto inny prosty przykład:

```
#include <iostream.h>
main()
{
    int a ;
    cout << "Musisz odpowiedzieć TAK lub NIE \n"
         << "jesli TAK, to napisz 1 \n"
         << "jesli NIE to napisz 0 \n"
         << " Rozumiesz ? Odpowiedz : " ;

    cin >> a ;

    cout << "Odpowiedziałes : "
         << ( a? "TAK" : "NIE" )
         << " prawda ? " ;
}
```

// ②

// ①



W wyniku wykonania tego programu na ekranie zobaczymy (Jeśli odpowiedzieliśmy 1)

```
Musisz odpowiedzieć TAK lub NIE
jesli TAK, to napisz 1
jesli NIE to napisz 0
Rozumiesz ? Odpowiedz : 1
Odpowiedziałes : TAK prawda ?
```



Komentarz

- ❶ Tutaj tkwi to wyrażenie warunkowe. W zależności od tego, czy zmienna *a* jest zerowa czy niezerowa, wyrażenie to jako całość ma wartość

"TAK"	- gdy <i>a</i> jest różne od 0
"NIE"	- gdy <i>a</i> jest równe 0

Zauważ, że w naszym wypadku wartość wyrażenia nie jest tu wcale liczbą, tylko albo takim, albo innym stringiem (jeśli wolisz: stałą tekstową).

- ❷ Gdybyś tutaj zamiast odpowiedzieć 1 odpowiedział 33, to nie ma problemu – sprawdzany jest przecież warunek czy *a* jest równe, czy różne od zera. Rezultat będzie taki sam jakbyś odpowiedział: 1.



Może na pierwszy rzut oka omówione wyrażenie warunkowe wydaje się trochę mało czytelne. Jednak zobaczysz, że wkrótce bardzo je polubisz, gdyż zaoszczędzi Ci pisania.

4.7 Operator sizeof

Operator `sizeof`^{†)} jest to wygodny operator, który pozwala nam rozpoznać zachowania kompilatora i komputera, z którymi przyszło nam pracować.

Jest to ważne z dwóch powodów:

- ❖ Te same typy obiektów (np. zmiennych) mogą mieć w różnych implementacjach różne wielkości. Przykładowo: o tym, jaką przestrzeń rezerwuje kompilator na danej maszynie dla obiektu typu `int` dowiedzieć się możemy używając właśnie tego operatora.
- ❖ C++ pozwala użytkownikowi na wymyślanie sobie własnych typów obiektów. (Będziemy o tym mówić w późniejszych rozdziałach). Często ważne jest, by wiedzieć ile pamięci maszyny zajmuje obiekt takiego nowo wymyślonego typu.

Operator ten stosuje się w ten sposób

†) `size of [something]` = ang. rozmiar [czegoś] (czytaj: „sajz of“)

```
sizeof(nazwa_typu)
albo
sizeof(nazwa_obiektu)
```

w rezultacie otrzymujemy rozmiar obiektu danego typu podany w bajtach.
Oto przykład zastosowania:

```
#include <iostream.h>
main()
{
    int mm ;

    cout << "Godzina prawdy. W tym komputerze "
          << "poszczególne typy\n"
          << "maja nastepujace rozmiary w bajtach: \n" ;

    cout << "typ char : \t" << sizeof(char) << endl ;
    cout << "typ int : \t" << sizeof(int) << endl ;
    cout << "typ short: \t" << sizeof(short) << endl ;
    cout << "typ long : \t" << sizeof(long) << endl ;
    cout << "typ float : \t" << sizeof(float) << endl ;
    cout << "typ double : \t" << sizeof(double) << endl ;
    cout << "typ long double \t: " << sizeof(long double)
          << endl ;

    cout << "Nasz obiekt lokalny mm ma rozmiar : "
          << sizeof(mm) << endl ;
}
```

W rezultacie wykonania tego programu na komputerze klasy IBM PC/AT
na ekranie ujrzymy tekst:

```
Godzina prawdy. W tym komputerze poszczególne typy
maja nastepujace rozmiary w bajtach:
typ char :      1
typ int :       2
typ short:      2
typ long :      4
typ float :     4
typ double :    8
typ long double : 10
Nasz obiekt lokalny mm ma rozmiar : 2
```

4.8 Operator rzutowania

Poznamy teraz operator rzutowania, albo inaczej - jawnego przekształcania typu. Jest to operator jednoargumentowy. Działa on w ten sposób, że bierze obiekt jakiegoś typu i jako rezultat zwraca obiekt innego typu. Innymi słowy może np. przekształcić typ float na int, typ char na double itp.

Operator ten może mieć dwie formy

```
(nazwa_typu) obiekt
```

względnie

```
nazwa_typu(obiekt)
```

O tym, kiedy której formy użyć, porozmawiamy w przyszłości (str. 413), na razie lepiej jako zasadę przyjąć stosowanie formy pierwszej.

Zobaczmy to na przykładzie

```
int a = 0xffff ;
char b ;

b = (char) a ;
```

Ponieważ obiekt typu `char` nie pomieści całej informacji zawartej w obiekcie typu `int`, dlatego w rezultacie w obiekcie `b` znajdzie się następująca wartość:

```
0xff
```

Takie przypisanie obiektu typu `int` do obiektu typu `char` musi spowodować utratę 8 najbardziej znaczących bitów (czyli bardziej znaczącego bajtu).

Gdybyśmy nie zastosowali operatora rzutowania, to konwersja taka także by nastąpiła. Po co więc tu operator?

Odpowiedź jest prosta: Dobry kompilator widząc, że następuje ryzyko utraty pewnej części informacji powinien nas ostrzec. Ostrzeżenie takie powinno pojawić się w trakcie kompilacji programu.

Jeśli natomiast jawnie zastosujemy w tym miejscu operator rzutowania, to kompilator napotykając go uzna, że widocznie wiemy co robimy. Ostrzeżenia wtedy nie będzie.

Tak naprawdę, to ten operator będzie nam służył do bardziej wyszukanych konwersji. Pomówimy o tym jeszcze w następnych rozdziałach.

Ogólnie mówiąc: **unikajmy nadużywania tego operatora, chyba że naprawdę wiemy, co chcemy zrobić.**

4.9 Operator: przecinek

Jeśli kilka wyrażeń stoi obok siebie oddzielone przecinkiem, to ta całość jest także wyrażeniem, którego wartością jest wartość wyrażenia będącego najbardziej z prawej.

Zatem wartością wyrażenia

```
(2 + 4, a * 4, 3 < 6, 77 + 2)
```

jest 79. Poszczególne wyrażenia obliczane są od lewej do prawej.

4.10 Priorytety operatorów

Na zakończenie spójrzmy na zestawione w tabeli operatory. Do tej pory unikaliśmy rozmów o priorytecie różnych operatorów. Nadmieniałem tylko, że mnożenie ma pierwszeństwo przed dodawaniem.

Zestawiamy więc teraz operatory w tabeli, w której na samej górze są operatory o najwyższym prioritycie. Pod względem prioritytów operatory dzielą się na 17 grup.

Nie przeraż się jeśli w tabeli zobaczysz operatory, których nie rozumiesz. Nie mówiliśmy jeszcze o wszystkich, mimo to taką tabelę dobrze jest oglądać w całości.

Operatory

Priorytet	Symbol	Nazwa	Zastosowanie	Łączność
17	::	określenie zakresu	nazwa_klasy::składnik	L
	::	nazwa globalna	::nazwa_globalna	P
16	->	wybranie składnika	wskaźnik->składnik	L
	[]	element tablicy	wskaźnik[wrażenie]	
	()	wywołanie funkcji	funkcja(lista_argumentów)	L
	()	nawias w wyrażeniach	(wrażenie)	
15	sizeof	rozmiar obiektu	sizeof(wrażenie)	P
	sizeof	rozmiar typu	sizeof(typ)	
	++	post inkrementacja	lwartość ++	
	++	pre inkrementacja	++ lwartość	
	--	post dekrementacja	lwartość --	
	--	pre dekrementacja	-- lwartość	
	~	dopełnienie do 2	~ wyrażenie	
	!	negacja	! wyrażenie	
	-	jednoargumentowy minus	- wyrażenie	
	+	jednoargumentowy plus	+ wyrażenie	
	&	adres czegoś	& lwartość	
	*	odniesienie się wskaźnikiem	* wyrażenie	
	new	kreuj (rezerwuj)	new typ	
	delete	zlikwiduj (anuluj rezerwację)	delete wskaźnik	
	delete[]	zlikwiduj wektor	delete [] wskaźnik	
	()	rzutowanie (konwersja typu)	(typ)wrażenie	
14	.	Wybór składnika wskaźnikiem:	obiett.*wsk_do_skladn.	L
	->*	- i nazwą obiektu - i wskaźnikiem do obiektu	wsk ->*wsk_do_skladn.	
13	*	mnożenie	wyraż * wyraż	L
	/	dzielenie	wyraż / wyraż	
	%	modulo (reszta)	wyraż % wyraż	L
12	+	dodaj (plus)	wyraż + wyraż	
	-	odejmij (minus)	wyraż - wyraż	L

11	<< >>	przesunięcie w lewo przesunięcie w prawo	lwartość << wyraż lwartość >> wyraż	L
10	< <= > >=	mniejsze niż mniejsze lub równe większe od większe lub równe	wyraż < wyraż wyraż <= wyraż wyraż > wyraż wyraż >= wyraż	L
9	== !=	równe nie równe	wyraż == wyraż wyraż != wyraż	L
8	&	iloczyn bitowy	wyraż & wyraż	L
7	^	bitowa różnica symetryczna	wyraż ^ wyraż	L
6		bitowa suma	wyraż wyraż	L
5	&&	koniunkcja	wyraż && wyraż	L
4		alternatywa	wyraż wyraż	L
3	?:	arytmetyczne if	wyraż ? wyraż : wyraż	L
2	= *= /= %= += -= <<= >>= &= = ^=	zwykle przypisanie mnóż i przypisz dziel i przypisz modulo i przypisz dodaj i przypisz odejmij i przypisz przesuń w lewo i przypisz przesuń w prawo i przypisz koniunkcja i przypisanie alternatywa i przypisanie exclusive or i przypisanie	lwartość = wyraż lwartość *= wyraż lwartość /= wyraż lwartość %= wyraż lwartość += wyraż lwartość -= wyraż lwartość <<= wyraż lwartość >>= wyraż lwartość &= wyraż lwartość = wyraż lwartość ^= wyraż	P
1	,	przecinek	wyraż , wyraż	L

Chciałbym Cię też uspokoić po raz drugi. Nie musisz wcale uczyć się tych priorytetów. Bez tego można sobie doskonale dać radę posługując się nawiasami.

Zapamiętaj tylko że:



1) Mnożenie, dzielenie, dodawanie, odejmowanie mają takie same priorytety, jak to pamiętamy ze szkoły. Wiedząc o tym, nie będziesz musiał używać nawiasów w tak banalnych sytuacjach.



2) Skomplikowane wyrażenia logiczne lepiej zaopatrywać w nawiasy. Nie tylko dlatego, że `&&` jest mocniejsze niż `||`. Także dlatego, że wyrażenia takie stają się czytelniejsze.

Inaczej będziesz produkował zapisy w rodzaju

```
i < b && s || a == n
```

nad którymi zawsze trzeba się zastanowić, a ryzyko popełnienia błędu jest kolosalne.



3) Zapamiętaj też, że nawiasy okrągłe `()` – oznaczające wywołanie funkcji oraz klamry `[]` – odniesienie się od elementu tablicy – mają bardzo wysoki priorytet, w szczególności wyższy niż tajemniczy jednoargumentowy operator `*` (czyli odniesienie się do obiektu pokazywanego przez wskaźnik).

4.11 Łączność operatorów

W ostatniej kolumnie tabeli widzimy litery L i P określające lewostronną lub prawostronną łączność operatora. Co to oznacza, pokażemy na przykładach.

I tak operator `!` jest prawostronnie łączny – co oznacza, że działa na argumentie stojącym po jego prawej stronie

```
!x
```

W wypadku operatorów dwuargumentowych łączność określa w jaki sposób grupowane jest wykonywanie wyrażenia.



Lewostronna łączność operatora `+` oznacza, że wyrażenie

```
a + b + c + d + e
```

odpowiada wyrażeniu

```
((((a + b) + c) + d) + e)
```



Prawostronna łączność operatora dwuargumentowego `=` oznacza, że wyrażeniu

```
a = b = c = d = e
```

odpowiada wyrażenie

```
(a = (b = (c = (d = e))))
```



Jedną z najsympatyczniejszych cech nowoczesnych języków programowania jest to, że można w nich posługiwać się podprogramami. Podprogram to jakby mały program we właściwym programie. Dzięki podprogramom możemy jakby definiować swoje własne „instrukcje”:

Jeśli napiszemy sobie podprogram realizujący operację liczenia pola koła na podstawie zadanego promienia – to tak, jakbyśmy język programowania wyposażyli w nową instrukcję umiejącą właśnie to obliczać. Od tej pory – ile razy w programie potrzebujemy obliczyć pole koła – wywołujemy nasz podprogram, a jest to tak proste, jak napisanie zwykłej instrukcji.

Podprogram, który jako rezultat zwraca jakąś wartość, nazywamy po prostu funkcją. W języku C++ wszystkie podprogramy nazywane są funkcjami. Funkcję wywołuje się przez podanie jej nazwy i umieszczonych w nawiasie argumentów.

Oto przykład programu zawierającego funkcję:

Nazywa się ona kukulka, a jej zadaniem jest kukać żadaną ilość razy. To, ile razy – jest parametrem wysyłanym do funkcji.

```
#include <iostream.h>
int kukulka(int ile); // ❶
/*****
main()
{
    int m = 20 ;
    cout << "Zaczynamy" << endl ;

    m = kukulka(5) ; // ❷
    cout << "\nNa koniec m = " << m ; // ❸
}
/*****
int kukulka(int ile) // ❹
{ // ❺
```

```

int i ;
    for(i = 0 ; i < ile ; i++)
    {
        cout << "Ku-ku ! " ;
    }
    return 77 ;
}
// 6
// 7

```

Wykonanie tego programu objawi się na ekranie jako:

```

Zaczynamy
Ku-ku ! Ku-ku ! Ku-ku ! Ku-ku ! Ku-ku !
Na koniec m = 77

```

Przyjrzyjmy się temu programowi

- 1 Funkcja ma swoją nazwę, która ją identyfikuje. Jak pamiętamy z poprzednich rozdziałów, wszelkie nazwy - przed pierwszym odwołaniem się do nich - muszą zostać zadeklarowane. Wymagana jest więc także deklaracja nazwy funkcji. W tym miejscu programu widzimy deklarację nazwy funkcji. Deklaracja ta mówi kompilatorowi:
kukulka jest funkcją wywoływaną z argumentem typu `int`, a zwracającą jako rezultat wartość typu `int`.

Powtórzmy:

Przed odwołaniem się do nazwy wymagana jest jej deklaracja. Deklaracja, ale niekoniecznie od razu definicja. Sama funkcja może być zdefiniowana później, nawet w zupełnie innym pliku. Zdefiniować funkcję, to znaczy po prostu napisać jej treść.

Definicja funkcji znajduje się w 4

- 2 Wywołanie funkcji to po prostu napisanie jej nazwy łącznie z nawiasem, gdzie znajduje się argument przesyłany do funkcji. Ponieważ spodziewamy się, że funkcja zwróci jakąś wartość, dlatego widzimy przypisanie tej wartości do zmiennej `m`.
- 3 Na dowód, że funkcja rzeczywiście zwróciła jakąś wartość, i że nastąpiło przypisanie tej wartości do obiektu `m` - wypisujemy go w tym miejscu na ekran.
- 4 Tu się zaczyna definicja funkcji. Definicja ta zawiera tzw. ciało funkcji, czyli wszystkie instrukcje wykonywane w ramach tej funkcji. Dwie klamry 5 i 7 określają ten obszar ciała funkcji, czyli po prostu jej treść.
- 6 Jest to moment, gdy funkcja kończy swoją pracę i wraca do miejsca skąd została wywołana [return = ang. powrót].^{†)} Obok słowa `return` widzimy wartość, którą zdecydowaliśmy zwrócić jako rezultat wykonania tej funkcji.



Zatrzymajmy się trochę przy deklaracjach funkcji. Oto kilka przykładów:

†) (czytaj: „rytern“)

```
float kwadrat(int bok);
void fun(int stopien, char znaczek, int nachylenie);
int przypadek(void);
char znak_x();
void pin(...);
```

Zauważ, że na końcu każdej z przedstawionych deklaracji jest średnik. Przeczytajmy teraz te deklaracje.

- `kwadrat` jest funkcją (wywoływaną z jednym argumentem typu `int`), która w rezultacie zwraca wartość typu `float`.
- `fun` jest funkcją (wywoływaną z 3 argumentami typu `int`, `char`, `int`), która nie zwraca żadnej wartości. Słowo `void` [ang. – próżny] służy tu właśnie do wyraźnego zaznaczenia tego faktu. Po takiej deklaracji – jeśli kompilator zobaczy, że przez zapomnienie próbujemy uzyskać z tej funkcji jakąś wartość – ostrzeże nas, sygnalizując błąd.
- `przypadek` to funkcja, która wywoływana jest bez żadnego argumentu, a która zwraca wartość typu `int`.
- `znak_x` to funkcja, która wywoływana jest bez żadnych argumentów, a w rezultacie zwraca wartość typu `char`.
- `pin` jest funkcją, którą wywołuje się z bliżej nieokreślonymi (teraz jeszcze) argumentami, a która nie zwraca żadnej wartości.

Uwaga dla programistów C:

Jest tu zmiana w stosunku do klasycznego C.

Pusty nawias w deklaracji funkcji np. `f()` ; oznacza

- ❖ – w języku C – dowolną liczbę argumentów czyli to samo co `f(...)`
- ❖ – w języku C++ – brak jakichkolwiek argumentów, czyli to samo co `f(void)`

Jeśli masz przyzwyczajenia z klasycznego C, to zapamiętaj, że odtąd

Deklaracja `f()`; – oznacza `f(void)`;



Nazwy argumentów umieszczone w nawiasach przedstawionych deklaracji są nieistotne dla kompilatora i można je pominąć. Powtarzam: nazwy, a nie typy argumentów.

Dlatego deklarację funkcji

```
void fun(int stopien, char znaczek, int nachylenie);
```

można napisać także jako

```
void fun(int, char, int);
```

To dlatego, że w deklaracji powiadamiamy kompilator o liczbie i typie argumentów. Ich nazwy nie są w tym momencie istotne. (To będzie ważne w definicji).

Masz więc dwa sposoby deklarowania funkcji. Namawiam Cię jednak do stosowania pierwszego sposobu – tego z nazwami argumentów. Przemawiają za nim dwa względy praktyczne:

- ❖ jest on czytelniejszy dla programisty, przypomina bowiem lepiej czym zajmuje się funkcja,
- ❖ łatwiej taką deklarację napisać. Mimo, że jest dłuższa. Po prostu pracując w edytorze przenosi się we właściwe miejsce (u nas – na górę programu) pierwszą linijkę definicji funkcji i stawia się na końcu tej linii średnik.



Zauważ, że w naszym ostatnim programie definicję funkcji oddzieliłem za pomocą linii komentarza składającą się z rzędu gwiazdek. Radzę Ci robić podobnie. Bardzo to polepszy czytelność Twoich programów. Jeśli masz plik, w którym mieści się trzydzieści definicji funkcji, to łatwiej się w nim orientować. Już na pierwszy rzut oka widać, gdzie się jedna funkcja kończy, a zaczyna druga.

5.1 Zwracanie rezultatu przez funkcję

W naszym przykładowym programie funkcja wywoływana była z argumentem i zwracała jakąś wartość. Przyjrzymy się teraz bliżej temu mechanizmowi przekazywania.

Oto przykład programu liczącego potęg danej liczby:

```
#include <iostream.h>
long potega(int stopien, long liczba) ;
/*****/
main()
{
    int pocz, koniec ;

    cout << "Program na obliczanie poteg liczb"
         << "całkowitych\n"
         << "z zadanego przedziału \n"
         << "Podaj początek przedziału : ";
    cin >> pocz ;

    cout << "\nPodaj koniec przedziału : " ;
    cin >> koniec ;

    // pętla drukująca wyniki z danego przedziału
    for(int i = pocz ; i <= koniec ; i++)
    {
        cout << i
             << " do kwadratu = "
             << potega(2, i)
             << " a do szóstianu = "
             // wywołanie funkcji
    }
```

```

        << potega(3, i) // wywołanie funkcji
        << endl ;
    }
}
/*****/
long potega(int stopien, long liczba)
{
    long wynik = liczba ;
    for(int i = 1 ; i < stopien ; i++)
    {
        wynik = wynik * liczba ;
        // zwiększyć można zapisać to samo jako :
        //      wynik *= liczba ;
    }
    return wynik ; // ❶
}
/*****/

```

Jeśli na pytania programu odpowiedz np. 10 oraz 14 to
na ekranie pojawi się:

```

Program na obliczanie potęg liczb całkowitych
z zadanego przedziału
Podaj początek przedziału : 10
Podaj koniec przedziału : 14
10 do kwadratu = 100 a do szescianu = 1000
11 do kwadratu = 121 a do szescianu = 1331
12 do kwadratu = 144 a do szescianu = 1728
13 do kwadratu = 169 a do szescianu = 2197
14 do kwadratu = 196 a do szescianu = 2744

```

- ❶ – Najpierw zwróćmy uwagę, jak odbywa się zwracanie wartości funkcji. Wspominaliśmy już, że robimy to przez instrukcję `return`. Stawia się po prostu przy niej żadaną wartość. U nas to wygląda w ten sposób:

```
return wynik ;
```

mogą być też inne warianty:

```
return (wynik + 6) ;
return 12.33 ;

```

Jeśli stoi tam wyrażenie (np. `wynik + 6`), to najpierw obliczana jest jego wartość, a następnie dopiero rezultat jest „przedmiotem” zwrotu.

Jest jeszcze coś zaskakującego. Otóż jeśli deklaracja funkcji jest taka:

```
long potega(int stopien, long liczba) ;
```

to znaczy, że funkcja ma zwracać jako rezultat wartość typu `long` (pamiętamy, że jest to jeden z typów liczb całkowitych). Tymczasem koło słowa `return` stoi liczba zmiennoprzecinkowa 12.33

Co wtedy? Czy jest to błąd?

Nie zawsze. Nastąpi bowiem próba niejawnego zamiany (konwersji) typu. W naszym wypadku będzie to konwersja typu zmiennoprzecinkowego na typ `long`. Kompilator domyśli się jak to zrobić i w rezultacie funkcja zwróci wartość 12.

Nie zawsze jednak taka konwersja może się odbyć.

Jak bowiem zamienić tzw. wskaźnik na liczbę zmiennoprzecinkową? Kompilator wtedy nam nie podaruje i w trakcie kompilacji oznajmi błąd.



Zapytasz pewnie: „A właściwie co to znaczy, że funkcja zwraca jakąś wartość? Wiemy już jak to się robi, ale co to znaczy?!“

To bardzo ważne pytanie.

Odpowiedź jest bardzo prosta, ale dobrze ją sobie wyraźnie uświadomić. Znaczący to, że wyrażenie będące wywołaniem tej funkcji ma samo w sobie jakąś wartość. W naszym wypadku wyrażenie

`( potega(2, 2) )`

ma samo w sobie wartość 4. Niezależnie od tego, że jest to wywołanie funkcji. Skoro więc takie wywołanie jest wyrażeniem mającym jakąś wartość, to można go użyć w innych, większych wyrażeniach. Przykładowo

`7 + 1.5 + potega(2, 2) + 100`

odpowiada u nas wyrażeniu

`7 + 1.5 + 4 + 100`



Jeśli funkcja jest zadeklarowana jako zwracająca typ `void` – czyli nie zwracająca niczego – a my, przez nieuwagę, użyjemy ją w takim wyrażeniu, to kompilator ostrzeże nas, że popełniamy błąd. To jedna z wielu zalet obowiązkowych deklaracji funkcji.

Także, gdybyśmy wewnątrz definicji takiej funkcji obok słowa `return` postawili coś oprócz średnika

`return 6 ; // błąd, gdy f-cja zwraca void`

to kompilator wykryje błąd. Mieliśmy bowiem nic nie zwracać, a zwracamy?

Także odwrotnie: jeśli zadeklarowaliśmy, że funkcja ma coś zwracać, a przy słowie `return` stoi sam średnik, kompilator uzna to za nasz błąd. Obiecałeś, że coś tu będzie, a nie ma? Pewnie o czymś zapomniałeś!

5.2 Stos

Może słyszałeś o czymś takim w komputerze, co się nazywa stos. Jeśli nie, to wyobraź sobie taki obrazek: Wszystkie swoje książki trzymasz w biblioteczce. Gdy którąś potrzebujesz, to idziesz do biblioteczki wyjmujesz i czytasz, potem wkładasz z powrotem. Wiem, wiem jestem naiwny. Prawda jest taka, że najpotrzebniejsze książki leżą na Twoim biurku w postaci mniejszego lub większego stosu. Zdjęcie książki z takiego stosu jest szybsze niż przechadzka do biblioteczki. Tak samo postępuje komputer. Ma także swój podręczny stos, na którym trzyma pewne dane. Koniec obrazka.

Jeśli w obrębie funkcji definiujemy jakieś zmienne, to są one przechowywane najczęściej właśnie na stosie – czyli w tej podręcznej pamięci. Stos ma wiele ciekawych właściwości – znają je przede wszystkim Ci, którzy programują w assemblerze. Z niektórymi właściwościami stosu zapoznamy się w następnych paragrafach.

5.3 Przesyłanie argumentów do funkcji przez wartość

Zajmijmy się teraz sposobem przesyłania argumentów do funkcji. Najpierw sprawa nazewnictwa. Załóżmy, że mamy funkcję

```
void alarm(int stopien, int wyjscie)
{
    cout << "Alarm " << stopien
          << "stopnia"
          << " skierowac sie do wyjscia nr "
          << wyjscie << endl ;
}
```

Założmy też, że w programie wywołujemy tę funkcję tak

```
int a, m ;
// ...
alarm(1, 10) ;
alarm(a, m) ;
```

Nazewnictwo jest takie: nazwy

```
stopien , wyjscie
```

-które widzimy w pierwszej linijce definicji funkcji są to tzw. *argumenty formalne* funkcji. Czasem zwane parametrami formalnymi. Ważne jest tu słowo: formalne.

To natomiast, co pojawia się w nawiasie w momencie wywoływania tej funkcji – czyli w naszym wypadku

```
1, 10, a, m
```

to tak zwane *argumenty (parametry) aktualne*.

Czyli takie argumenty, z którymi aktualnie funkcja ma wykonać pracę. Często będę na to mówić prościej: *argumenty wywołania funkcji* – bo z tymi argumentami funkcję wywołujemy.

Dla oswojenia się podajmy obrazek z życia: sklep to jakby funkcja.

```
void sklep(int klient);
```

W sklepie obsługuje się klientów. W sklepie mówią o nas – „muszę obsłużyć klienta”. Klient jest argumentem formalnym funkcji sklep.

Jednak do sklepu przychodzą jacyś konkretni ludzie. Gdy do sklepu wchodzi Claudia, to ona jest argumentem aktualnym tego sklepu. To dla niej w tym momencie pracuje sklep. W sklepie nikt nie nazywa jej inaczej jak tylko (bardzo) *formalnie*: klient(-ka). Nawet nie znają jej nazwiska, jednak *aktualnie* klientką jest

Claudia. Gdy Claudia wyjdzie ze sklepu, a za chwilę wejdzie Sybilla, to ona staje się parametrem (argumentem) aktualnym tego sklepu. Sklep na nią i tak znowu mówi „klientka”, ale nikt nie twierdzi, że to ta sama osoba.

Taka jest więc różnica między argumentami aktualnymi a formalnymi.

Argumenty formalne to jest to, jak na parametry mówi sobie w środku funkcja,

natomiast

argumenty aktualne to to, co aktualnie stosujemy w konkretnym wywołaniu funkcji.



Argumenty przesłane do funkcji – tak, jak to w naszym przykładzie – są tylko kopiami. Jakikolwiek działanie na nich nie dotyczy oryginału.

Oto dowód:

```
void zwieksz(int formalny)
{
    formalny += 1000 ;           // zwiększenie liczby o 1000 ❶
    cout << "W funkcji modyfikuje arg formalny\n\t"
         << " i teraz arg formalny = " << formalny << endl;
}
```

Jak widać, w tej funkcji zwiększa się wartość argumentu formalnego funkcji. Funkcję tę wywołujemy na przykład w takim fragmencie programu.

```
int aktu = 2 ;
cout << "Przed wywołaniem, aktu = " << aktu << endl;
zwieksz(aktu) ;
cout << "Po wywołaniu, aktu = " << aktu << endl;
```



Jeśli wykonamy taki fragment programu to otrzymamy:

```
Przed wywołaniem, aktu = 2
W funkcji modyfikuje arg formalny
    i teraz arg formalny = 1002
Po wywołaniu, aktu = 2
```

Należy uświadomić sobie bardzo ważną rzecz: Do funkcji przesyłamy tylko wartość liczbową zmiennej aktu (parametru aktualnego). Wartość ta służy do inicjalizacji parametru formalnego, czyli zmiennej lokalnej tworzonej przez funkcję na stosie. Jest to więc jakby zrobienie kopii w obrębie funkcji.

Funkcja pracuje na tej kopii. Czyli w naszym przykładzie dodanie 1000 (w miejscu ❶) nie nastąpiło do komórki pamięci, gdzie tkwi aktu, ale do tej zmiennej lokalnej na stosie, gdzie mieści się kopia (o nazwie formalny). Po opuszczeniu funkcji ten fragment stosu jest niszczone, znika więc też kopia, nie ma śladu, że coś tam robiliśmy.

Bardzo pouczająca przypowieść o babci

Jeśli jeszcze to nie jest dla Ciebie, Czytelniku, jasne to rozważmy taki obrazek. My, (program) robimy teściowej (zmienna) zdjęcie (kopia). Zdjęcie to dajemy dziecku (funkcja) by się pobawiło. Dziecko ukochanej babci dorysowuje wąsy (dodanie 1000). Kiedy skończy zabawę zabawki są sprzątane, a śmieci wyrzucane (niszczenie kopii – zdjęcia). Po skończonej zabawie patrzymy na naszą (żywą) teściową: teściowa wąsów nie ma.

5.4 Przesyłanie argumentów przez referencję

Powyżej opisany sposób znany był programistom C. Język C++ przynosi jeszcze inny sposób przesyłania argumentów. Przez referencję. Czyli przez przewisko.

W dalszej części książki powinniśmy mówić: „przez referencję”, jednak dopóki się z tym nie oswoisz – używać będziemy też równolegle terminu: „przez przewisko”

Oto przykład:

```
#include <iostream.h>
void zer( int wart, int &ref); // ❶
/*****/
main()
{
    int a = 44,
        b = 77 ;

    cout << "Przed wywołaniem funkcji: zer \n" ;
    cout << "a = " << a << ", b = " << b << endl ;

    zer(a, b) ; // ❷

    cout << "Po powrocie z funkcji: zer \n" ;
    cout << "a = " << a << ", b = " << b << endl ; // ❸
}
/*****/
void zer( int wart, int &ref)
{
    cout << "\tW funkcji zer przed zerowaniem \n" ;
    cout << "\twart = " << wart << ", ref = "
        << ref << endl ; // ❹

    wart = 0 ;
    ref = 0 ; // ❺

    cout << "\tW funkcji zer po zerowaniu \n" ;
    cout << "\twart = " << wart << ", ref = "
        << ref << endl ; // ❻
}
```



W rezultacie działania tego programu na ekranie pojawi się

Przed wywołaniem funkcji: zer

a = 44, b = 77

W funkcji zer przed zerowaniem

wart = 44, ref = 77

W funkcji zer po zerowaniu

wart = 0, ref = 0

Po powrocie z funkcji: zer

a = 44, b = 0



Komentarz

Patrząc na ekran zauważamy, że funkcja, która chciała wyzerować dwa obiekty a i b wysłane do niej jako argumenty – zaskoczyła nas. Oczywiście obiekt a jest nietknięty. To znamy. Jednak obiekt b ma wartość 0. Dlaczego?

Jeśli chodzi o zmienną o nazwie a – to nic nas tu nie dziwi. Funkcja odebrała ją przez wartość.

- 1 Rzućmy więc okiem na deklarację funkcji zer. Widzimy, że to funkcja, która przyjmuje dwa argumenty. Pierwszy z nich jest przesyłany – tak jak poprzednio – przez wartość. Drugi natomiast jest przesyłany przez referencję. Zauważ znak: &
- 2 W main mamy dwie zmienne, które wysyłamy do funkcji zer. Inaczej mówiąc: wywołujemy funkcję zer z parametrami aktualnymi a,b
- 3 Wewnątrz funkcji zer, na moment przed „egzekucją” wypisujemy jeszcze wartość dwóch parametrów formalnych wart, ref
- 4 Tu następuje jakaś operacja zmieniająca wartość zmiennych wart i ref. W naszym wypadku to wpisanie tam zer.
- 5 Na dowód, że tak się stało w istocie - wypisujemy ich wartość na ekran.
- 6 Kończymy pracę funkcji. Ponieważ funkcja zwraca typ void (po prostu nic nie zwraca) dlatego możemy sobie tu oszczędzić instrukcji return. Gdybyśmy chcieli by ona koniecznie była, to liniijkę wcześniej należałoby napisać

return ;

- 7 Po powrocie z funkcji, będąc już w main wypisujemy na ekranie wartości zmiennych a i b. I tu jest cała niespodzianka. Z treści, która pojawia się na ekranie widać, że ten argument, który funkcja przyjmowała starym sposobem (przez wartość) nie został zmodyfikowany. Natomiast ta zmienna, którą funkcja odebrała przez referencję (przez wisko) została zmodyfikowana.

Dlaczego ?

Otóż w tym wypadku do funkcji, zamiast liczby 77 (wartość zmiennej b), został wysłany adres zmiennej b w pamięci komputera.

Ten adres funkcja sobie odebrała i (na stosie) stworzyła sobie referencję. Czyli powiedziała sobie coś takiego: „Dobrze, zatem komórce pamięci o przysłanym mi adresie nadaję pseudonim (przez wisko) ref.

Podkreślmy jasno: ta sama komórka, na którą w `main` mówiło się `b`, stała się teraz w funkcji `zer` znana pod przezwiskiem `ref`. Są to dwie różne nazwy, ale określają ten sam obiekt.

W ❷ do obiektu o tym przezwisku `ref` wpisano zero. Skoro `ref` było przezwiskiem obiektu `b`, to znaczy, że odbyło się to na obiekcie `b`.

Ponieważ, jak pamiętamy, po zakończeniu działania funkcji likwiduje się śmieci, zobaczymy co zostało zlikwidowane.

- ❖ 1) Będąc na stosie kopia zmiennej `a`. (Która to kopia początkowo miała wartość 44, a potem 0). Pamiętamy, że ten argument odebrany był przez wartość.
- ❖ 2) Drugi argument przesyłany był przez referencję, więc na stosie mieliśmy zanotowany adres tego obiektu, który to obiekt wewnątrz programu przeżywaliśmy `ref`. Ten adres został zlikwidowany. (Jeśli podrzemy kartkę z zapisanym adresem jakiegoś budynku, to mimo wszystko budynek ten nadal stoi. My co prawda tracimy adres tego budynku, ale inni – np. funkcja `main` – mają ten adres u siebie zanotowany.

Wniosek: Przesłanie argumentów funkcji przez referencję pozwala tej funkcji na modyfikowanie zmiennych (nawet lokalnych!) znajdujących się poza tą funkcją.

Programistów klasycznego C opanowała na pewno teraz euforia:

„–Wreszcie jest łatwy sposób modyfikowania argumentów! To, na co nie pozwalało przesyłanie argumentów przez wartość, staje się wreszcie możliwe dzięki przesyłaniu przez referencję!”

Hola, hola! Chciałbym Cię tutaj przestrzec. Na pewno w innych, nawet prymitywnych językach programowania, spotkałeś już taki właśnie sposób przesłania argumentów do funkcji, mimo że tam nie nazywał się on przesłaniem przez referencję.

Dlaczego zatem tak wspaniały język jak C (klasyczny) nie pozwalał na to? Widocznie były powody. Nie, nie chodzi o to, że może dla piszących kompilatory byłoby to trudne w realizacji. Powody są inne. Otóż przesyłanie przez referencję jest prostą drogą do pisania programów bardzo trudnych do późniejszej analizy.

Nasze zmienne w jakimś fragmencie programu zmieniają się bowiem w sposób **niezauważony** na skutek działania jakiegoś innego fragmentu programu (innej funkcji). Niezauważony, bowiem z wywołania funkcji `zer` w środku `main` ❷ nie widać, który argument przesyłany przez wartość, a który przez referencję. Nie ma więc ostrzeżenia: ach, ten argument może być tam modyfikowany!



Ten sposób przesyłania argumentów do funkcji powinien być więc zasadniczo unikany. Są jednak sytuacje, kiedy się bardzo przydaje. To właśnie z powodu takich sytuacji ten sposób przesyłania argumentów został do języka C++ wprowadzony. O tych sytuacjach będziemy jednak mówić dokładniej w dalszych rozdziałach. Tutaj tylko wspomnę, że sposób ten stosuje się do tak dużych obiektów, że przesłanie ich przez wartość (wymagające zrobienia kopii np.

dziesiątków, setek bajtów) powodowałyby znaczące spowolnienie wywoływania takiej funkcji. W wypadku, gdy taka funkcja jest wywoływana bardzo wiele razy, może to być czynnikiem ważnym.



Jeszcze innym sposobem przesłania argumentu może być posłużenie się tzw. wskaźnikiem. Ten sposób omówimy bliżej w rozdziale o wskaźnikach.

5.5 Kiedy deklaracja funkcji nie jest konieczna

Jak wspomnieliśmy – każda nazwa przed odniesieniem się do niej (po prostu użyciem jej) musi zostać zadeklarowana. Dotyczy to też nazw funkcji. Funkcje muszą więc być deklarowane i robi się to w sposób, o którym już mówiliśmy.

Jednakże, jak pamiętamy, każda definicja (funkcji) jest także przy okazji jej deklaracją.

Jeżeli więc w pliku definicja funkcji jest wcześniej (po prostu wyżej) niż linijka z jakimkolwiek wywołaniem tejże funkcji – to nie trzeba osobnej deklaracji tej funkcji.

Jeśli natomiast funkcja nie jest osobno deklarowana, a wywołanie następuje w linijce powyżej definicji tej funkcji – wówczas kompilator zaprotestuje komunikatem o błędzie (-bo nie będzie jeszcze znał nazwy tej funkcji z żadnej deklaracji).

Oto ilustracja obu przypadków:

```

/*****
void funkcja_gorna(void)                                // ❶ definicja która
                                                         // jest też deklaracją
{
    // dla uproszczenia pusta funkcja - czemu nie ?
}
/*****
main()
{
    funkcja_gorna();                                     // ❷
    funkcja_dolna();                                    // ❸
}
/*****
void funkcja_dolna(void)                                // ❹ definicja która też jest
                                                         // deklaracją,
                                                         // ale niestety spóźnioną !
{
    // dla uproszczenia pusta funkcja - czemu nie ?
}
/*****

```

Jeśli spróbujemy skompilować powyższy program otrzymamy komunikat o błędzie kompilacji w linijce ❸. To dlatego, że w tym momencie kompilator nie zapoznał się jeszcze z deklaracją funkcji `funkcja_dolna`, która to deklaracja (łącznie z definicją) znajduje się kilka linijek niżej ❹.

Zupełnie inaczej jest z wywołaniem funkcji `funkcja_gorna`. Kompilując linię ❷ kompilator zna już deklarację funkcji `funkcja_gorna`, bo się na nią natknął powyżej, w linii ❶.

Nie chodzi tu bynajmniej o pozycję funkcji w stosunku do specjalnej funkcji `main`. Nasza funkcja `main` mogłaby być dowolną funkcją wywołującą dwie inne. Błąd byłby ten sam.

Należało zatem pamiętać, że jej definicja następuje po linii wywołania. To, czy dana funkcja jest *przed-* czy *po-* łatwe jest do opanowania w niewielkich programach. W programach większych nie radzę na tym polegać i po prostu deklarować wcześniej wszystkie funkcje.

Moim zdaniem nie ma sensu próbować oszczędzać na deklaracjach. Łatwo je przecież zrobić (- kopiując w edytorze pierwszą linię definicji funkcji i umieszczając ją na górze programu – zaopatrując przy okazji w średnik).

Deklarujmy więc wszystkie funkcje !

Przyjmując taką zasadę uwalniamy się od pamiętania, która funkcja jest powyżej której.

Uwaga dla programistów klasycznego C

Jedną z pierwszych niemiłych rzeczy, która Was spotka, gdy programy w C będziecie przerabiali na C++ będzie właśnie sprawa braku deklaracji funkcji.

W języku C deklaracje takie (zwane tam predefinicjami lub prototypami funkcji) były zalecane, ale nie wymagane. Jeśli ich więc nie zamieszczaliśmy, kompilator robił milczące założenia co do typu argumentów i typu wartości zwracanej przez funkcję.

Kiedy przerabiałem na C++ jeden z moich dużych programów w C, taki na 25 tysięcy linii, – musiałem zrobić deklaracje wszystkich funkcji. Zabrało mi to trochę czasu, ale bardzo szybko się opłaciło, gdyż już w trakcie przeróbki okazało się, że niektóre wywołania funkcji nie zgadzały się dokładnie z definicjami.

Dzięki temu, że kompilator C++ tak krytycznie patrzy na tekst programu, dużo błędów wykrywanych jest już na etapie kompilacji. Stąd też C++ ma opinię takiego języka, w którym programy – jeśli przebrną przez kompilację – działają od razu, bez żmudnego procesu uruchamiania.

5.6 Argumenty domniemane

Do pochopnego przesyłania argumentów przez referencję nie zachęcałem.

Jest za to w C++ inna nowość w argumentach funkcji (w stosunku do C klasycznego) – do której zachęcam. Są to tak zwane argumenty domniemane.

Weźmy taki przykład:

```

/*****
void temperatura(float stopnie, int skala)
{
    cout << "Temperatura komory : " ;

```



```

switch(skala)
{
    case 0 :
        cout << stopnie << " C\n" ;
        break ;

    case 1 :
        cout << (stopnie +273) << " K\n" ;
        break ;

    case 2 :
        cout << cel_to_far(stopnie) << " F\n" ;
        break ;
}

```

Funkcja ta, jak łatwo się zorientować, służy do wydruku informacji o temperaturze. Temperatura jest argumentem przysyłanym do funkcji. Temperatura przysyłana jest zawsze w stopniach Celsjusza, jednak na ekran chcemy tę temperaturę wydrukować czasem w stopniach Celsjusza, czasem w stopniach Kelvina, a czasem w stopniach Fahrenheita.

To, w jakiej skali mamy akurat wypisać bieżącą temperaturę, zależy od drugiego argumentu. Jeśli jest on równy 0 – to w stopniach Celsjusza, jeśli 1 – to w Kelvinach, jeśli równy 2 – to w stopniach Fahrenheita.

Do zamiany stopni Celsjusza na Fahrenheita mamy jakąś funkcję

```
float cel_to_far(float stopnie);
```

To, gdzie ona jest i jak jest zrealizowana – jest teraz nieistotne. Dla uproszczenia założmy, że jest w bibliotece.

Do tej pory nie było jeszcze nic nowego. Zobaczmy jak z naszej funkcji temperatura korzystamy. Oto przykładowe wywołania:

```

// ...
temperatura(52.5, 0);
temperatura(20.1, 2);
temperatura(52.5, 0);
temperatura(100, 0);
temperatura(52.5, 0);
temperatura(60, 2);

```

Jak dotąd także wszystko jest po staremu.

A teraz zastanówmy się: w naszym programie setki razy drukujemy temperaturę w skali Celsjusza, natomiast bardzo rzadko w innych skalach. Jednakże za każdym razem, gdy chcemy wydrukować w skali Celsjusza, musimy jako drugi argument wysyłać to nieszczęsne zero.

A przecież, gdy pytam laboranta o temperaturę wrzenia wody i nie dodaję w jakiej skali to on *domniemywa*, że chodzi o skalę Celsjusza. Czy kompilator nie mógłby się też inteligentnie domyślić? Mógłby. Po to są właśnie argumenty domniemane.

Wystarczy w naszym wypadku deklarację funkcji napisać tak:

```
void temperatura(float stopnie, int skala = 0 );
```


sprawi to, że jeśli w naszym programie wywołamy funkcję np. tak

```
temperatura(66.3);
```

Czyli z jednym (tylko pierwszym) argumentem, kompilator domniema, że drugi argument jest równy 0, tak jak mu to w deklaracji przykazaliśmy.

Podkreślam: – o tym, że argument jest domniemany, informujemy kompilator raz, w deklaracji funkcji. Jeśli definicja jest później, to w definicji już się tego nie powtarza.

Od tej pory wolno nam tę funkcję wywoływać także z jednym argumentem. Stary sposób z dwoma argumentami też jest dopuszczalny, wtedy kompilator nie musi nic domniemywać, po prostu robi to, co kazaliśmy w wywołaniu funkcji.

Zatem poniższe sposoby wywołania są legalne

```
temperatura(100.3); // domniemywa się: skala = 0
// poniżej nic się nie musi domniemywać, bo jest napisane jasno
temperatura(36.6, 0); // chcesz w Celsjuszach, bo 0
temperatura(50.3, 1); // chcesz w Kelwinach, bo 1
temperatura(159.8, 2); // chcesz w Fahrenheitach, bo 2
```

Jeśli chcemy, by funkcja miała kilka argumentów domniemanych, to argumenty takie muszą być na końcu listy

```
int multi(int x, float m, int a = 4,
          float y = 6.55, int k = 10);
```

Ostatnie argumenty (jako domniemane) mogą więc być w niektórych wywołaniach tej funkcji opuszczane.

Oto przykłady wywołań tej funkcji. W komentarzach podaję jakie wartości mają wówczas argumenty *a*, *y*, *k*. Argumentów *x* i *m* nie podaję, bo są to zwykłe, (nie-domniemane) argumenty i sprawa jest jasna: 2 i 3.14

```
multi(2, 3.14); // a = 4, y = 6.55, k = 10
multi(2, 3.14, 7); // a = 7, y = 6.55, k = 10
multi(2, 3.14, 7, 0.3); // a = 7, y = 0.3, k = 10
multi(2, 3.14, 7, 0.3, 5); // a = 7, y = 0.3, k = 5
```

Nie jest możliwe opuszczenie domniemanego argumentu *a* lub *y*, a umieszczenie argumentu *k*. Zatem wywołanie typu

```
multi(2, 3.14, 7, , 5); // !!!
```

jest traktowane jako błąd. Nie trzeba się tu nic uczyć, aby zapamiętać, które kombinacje są dopuszczalne, a które nie. Zasada jest prosta i logiczna:

W języku C++ nie może wystąpić taka sytuacja, że obok siebie stoją dwa przecinki.

Taka sytuacja uważana jest jako błąd literowy. Nie dlatego, żeby kompilator nie potrafił sobie dać rady z zapisem. To dlatego, że gdyby kompilator zezwolił nam na zapisy z dwoma przecinkami – to tym samym znieczuliłby się na wszystkie wypadki, gdzie

rzeczywiście zapomnieliśmy o jakimś argumentcie, albo po prostu niepotrzebnie napisały nam się dwa przecinki.

Kompilator znowu robi to w naszym interesie.

Na zakończenie jeszcze raz podkreślimy: Argumenty domniemane określa się w deklaracji funkcji. W definicji nie. Nawet jeśli w definicji napisalibyśmy te same wartości domniemane, dobry kompilator nie powinien się na taką powtórkę zgodzić.

Ale:

Mówiliśmy kiedyś, że jeśli funkcja jest w programie napisana powyżej jakiegokolwiek jej wywołania, to oddzielna deklaracja tej funkcji nie jest potrzebna. Sama definicja funkcji jest wtedy także jej pierwszą występującą w tym programie deklaracją.

Gdzie wtedy określić argumenty domniemane? Oczywiście w deklaracji - a że jest nią tutaj akurat definicja funkcji - to robimy to tutaj.

Zasada jest taka:

Chodzi o to, żeby kompilator o argumentach domniemanych danej funkcji dowiedział się tylko raz - wtedy, gdy dowiaduje się co to za funkcja (czyli wtedy, gdy dociera do niego deklaracja). Kompilator musi to już wiedzieć w momencie, gdy opracowuje wywołania tej funkcji w programie. Nietrudno się domyślić, że informacje te są mu potrzebne wtedy, by te opuszczone przez nas argumenty uzupełnić.

5.7 Nienazwany argument

Wyobraźmy sobie taką sytuację. Mieliliśmy funkcję wywoływaną z jednym argumentem. Była to na przykład funkcja

```
void ton(int wysokosc) ;
```

powodująca, że komputer zapiszczy jednym z sygnałów ostrzegawczych. Dajmy na to, że parametr o nazwie `wysokosc` służy do wyboru wysokości tonu. (Definicji tej funkcji nie przytaczam, bo jej wygląd zależy od implementacji).

Funkcja ta do tej pory dobrze nam służyła i w naszym programie setki razy wywoływaliliśmy ją w różnych miejscach.

Pewnego dnia jednak zapragnęliśmy ciszy, albo też uznaliśmy, że program, który piszczy za dużo, nie ma wyglądu naukowego, albo... Krótko mówiąc po drastycznym cięciu definicja naszej funkcji wygląda tak:

```
void ton(int wysokosc)
{
    // funkcja jest teraz pusta
}
```

Nie chodzi tu w tym przykładzie o to, że w funkcji nie ma żadnej instrukcji - to nie jest problem. Wolno nam. Chodzi o to, że teraz argument formalny `wysokosc` nie został ani raz użyty. Nie jest to błąd, ale kompilator będzie nas ciągle od tej pory ostrzegał, sugerując że może czegoś zapomnieliśmy. Co robić?

Odpowiedź wydaje się prosta: wyrzucić ten argument z deklaracji i definicji funkcji.

Łatwo powiedzieć. Co się wtedy stanie z tymi setkami wywołań funkcji `ton` w naszym programie? (a może nawet w jeszcze innych, jeśli funkcję `ton` traktowaliśmy jako biblioteczną).

Innymi słowy od tej pory zamiast jednego ostrzeżenia, będziemy mieli setki komunikatów o błędzie nieznaledzenia funkcji `ton` wywoływanej z jednym argumentem typu `int`. Każdy komunikat – od jednego wywołania funkcji `ton` w programie. Tragedia. Miało być lepiej, a jest gorzej.

Jest jednak wyjście: Argument nienazwany.

Otóż zamiast wyrzucać cały argument z definicji funkcji wyrzucamy tylko jego nazwę, a typ argumentu zostaje. Jak poniżej

```
void ton(int)
{
}
```

Jest to znak dla kompilatora, że jest to funkcja wywoływana z jednym argumentem typu `int`, ale tego argumentu w funkcji nie używamy. Niech nam więc oszczędzi ostrzeżeń o tym, że zapomnieliśmy z nim cokolwiek zrobić.

Zwróćmy uwagę, że tego cięcia dokonujemy w definicji, a nie w deklaracji funkcji. Jak bowiem pamiętamy – w samej deklaracji nazwy argumentów formalnych (nie typy, tylko nazwy) mogą istnieć lub nie. Kompilator w deklaracji i tak nazwy te ignoruje (korzysta tylko z typów).

Na koniec zdradzę Ci, drogi czytelniku, że osobiście czasem trudno mi definitywnie wyrzucić nazwę. Nazwa zawsze przypomina mi do czego dany argument miał służyć. Dlatego ujmuję nazwę w znaki komentarza, co dla kompilatora jest równoznaczne, że tam nic nie ma

```
void ton(int /* wysokosc */)
{
}
```

a ja zachowuję wspomnienia.

5.8 Funkcje `inline` (w linii)

Jest to jedna z cech specyficznych dla C++ i nie występuje w klasycznym C, chociaż tam, aby osiągnąć podobny efekt można było sobie jakoś poradzić.^{†)}

Żałujemy, że mamy niewielką funkcję. Niewielką, to znaczy jej definicja jest bardzo krótka, zawiera niewiele instrukcji. Przykładowo:

```
int zao(float liczba)
{
```

†) Stosując tzw. makrodefinicje.

```
return (liczba + 0.5);
```

```
}
```

Jej deklaracja mówi nam: `zao` jest funkcją wywoływaną z jednym argumentem typu `float`, a zwracającą typ `int`.

Ze spojrzenia na ciało tej funkcji – czyli na instrukcje będące jej treścią – widzimy, że jest to funkcja służąca do zaokrąglania liczb rzeczywistych do całkowitych. Do liczby typu `float` dodaje się 0.5, a następnie tę wartość przedstawia się instrukcją `return`. Funkcja ta wie, że ma zwrócić typ `int`, a więc wartość stojąca koło `return` zamieniana jest na typ `int` w najbardziej drastyczny sposób, czyli przez odcięcie części ułamkowej. Zostaje sama wartość całkowita i to właśnie zwraca funkcja.

Jak to się dzieje w przypadkach dla liczb 6.8 i 6.2 pokazuje poniższy zapis:

$$6.8 + 0.5 = 7.3 \quad \longrightarrow \quad 7$$

$$6.2 + 0.5 = 6.7 \quad \longrightarrow \quad 6$$

Żałujemy teraz, że w naszym programie bardzo, bardzo często posługujemy się tą funkcją.

Czytelnicy, którzy mają jakieś doświadczenie z programowaniem w assemblerze, wiedzą, że za wywołanie funkcji trochę się płaci. Musi na poziomie języka maszynowego wystąpić kilka instrukcji, które obsługują to specyficzne przejście w inne miejsce programu. Także po wykonaniu funkcji trzeba trochę posprzątać, więc też jest niewielka praca do zrobienia.

Co prawda w języku C++ koszt wywołania funkcji jest relatywnie niski, jednak jeśli naszą funkcję naprawdę zamierzamy wywoływać tysiące razy, to czas zużyty na wywołanie i powrót może stać się znaczący. Mamy więc wybór: albo posługujemy się tą funkcją (jak w poniższym wyrażeniu)

```
l = zao(m) + zao(n * 16.7); // ❶
```

albo rezygnujemy z niej i zaokrąglamy „na piechotę” wpisując algorytm zaokrąglania w linię wyrażenia.

```
l = (int)(m + 0.5) + (int)((n * 16.7) + 0.5); // ❷
```

Ten drugi zapis wykona się szybciej, ale ostatecznie może się nam nie chcieć w setkach linii z podobnymi wyrażeniami wpisywać ten kod. Co robić?

Jest wspaniałe wyjście kompromisowe. Pozwala ono na

- jasność zapisu – jak w wypadku funkcji (pierwsze wyrażenie) ❶
- szybkość wykonania – jak w wypadku wpisania tego algorytmu zaokrąglania w linię. ❷

Tak: właśnie w linię – czyli po angielsku: *in line*.^{†)} Naszą funkcję definiujemy tak:

```
inline int zao(float liczba)
{
```

†) (czytaj: „yn lajn”)

```

    return (liczba + 0.5) ;
}

```

Różnica żadna z wyjątkiem tego słowa `inline`.

Co ono daje? Otóż teraz, ile razy w programie umieścimy wywołanie funkcji `zao`, kompilator umieści dosłownie jej ciało (treść) w linijce, w której to wywołanie nastąpiło. Nie będzie więc żadnych akcji związanych z wywołaniem i powrotem z tej funkcji. W rezultacie taki kod będzie wykonywał się szybciej.

Wy tłumaczmy to jeszcze prościej: – słowo `inline` sprawia, od tej pory możemy stosować zapis jak w ❶, a komputer i tak sam zamieni sobie to na ❷.

Czy rozmiar programu przez to zmniejszy się czy zwiększy ?

To zależy. Jeśli funkcja, którą zdefiniowaliśmy jest niewielka – taka jak np. nasza – to jej treść może zająć mniej miejsca niż kod generowany związany z obsługą wywołania funkcji i powrotem z niej. Wtedy program może być nieco mniejszy. Mówię tu o przypadku, gdy funkcję `zao` wywołuje się w niewielu miejscach w programie, ale za to miliony razy. W innych wypadkach objętość programu może się zwiększyć. To nic, gdy chodzi o szybkość i łatwość zapisu.

Jednakże pamiętać należy, iż:

Funkcje typu `inline` zostały pomyślane dla naprawdę małych krótkich funkcji i tylko wtedy mają sens. Nie należy ich nadużywać.

Funkcja typu `inline` może zostać skompilowana tak, jak zwykła funkcja

- jeśli dany kompilator nie jest na tyle dobry, że potrafi sobie z nimi radzić. Dlatego słowo `inline` rozumieć należy jako sugestię dla kompilatora. Z tej sugestii może on skorzystać lub nie.

Jeśli nie skorzysta, to zrobi z niej funkcję zwykłą – zwaną czasem żartobliwie `outline` – czyli „poza linią”. Inaczej mówiąc taką, w której ciało jest gdzieś poza linią, w której się ją wywołuje.

Są jeszcze inne sytuacje, gdy funkcja może być skompilowana jako `outline`: Wtedy mianowicie, gdy kompilujemy ją dla pracy z programem uruchomionym tzw. debuggerem. Wówczas kompilator wszystkie nasze funkcje typu `inline` skompiluje jako `outline` dlatego, że tak wygodniej jest pracować debuggerowi. Jeśli jednak potem skompilujemy program jeszcze raz, tak „na czysto”, to kompilator zachowa się tak, jak umie najlepiej.

Umiejscowienie definicji funkcji typu `inline`

Do tej pory wielokrotnie podkreślaliśmy, że kompilator musi znać deklarację funkcji w momencie, gdy napotka pierwsze wywołanie tej funkcji. Po to, by sprawdzić poprawność wywołania. To tyle.

Teraz jednak chodzi o sprawę poważniejszą. Jeśli funkcja jest typu `inline`, to kompilator napotykać w jakiejś linii jej wywołanie, musi w tej linii wstawić właściwe instrukcje. Zatem teraz już sama deklaracja nie wystarczy. Definicja ciała (treść) funkcji musi już być w tym momencie kompilatorowi znana. Kompilator powinien już mieć „na boku” przygotowaną treść tej funkcji i to właśnie włączy do danej linii programu.

Wniosek stąd taki, że wobec tego:

Funkcje typu inline muszą być na samej górze tekstu programu albo nawet w pliku nagłówkowym, gdzie znajdują się deklaracje innych, zwykłych funkcji, a który to plik dołączany jest w czasie kompilacji modułów naszego programu.

A oto przykład programu z naszą funkcją typu inline:

```
#include <iostream.h>
float
    poczatek_x,           // początek układu współrzędnych
    poczatek_y,
    skala_x = 1,          // skala: pozioma i pionowa
    skala_y = 1 ;
/*****/
inline float wspx(float wspolrzedna)           // ❶
{
    return( (wspolrzedna - poczatek_x) * skala_x) ;
}
/*****/
inline float wspy(float wspolrzedna)           // ❷
{
    return( (wspolrzedna - poczatek_y) * skala_y) ;
}
/*****/
main()
{
    float    x1 = 100,          // przykładowy punkt
            y1 = 100 ;

    cout << "Mamy w punkt o współrzędnych \n" ;
    cout << " x = " << wspx(x1)           // ❸
         << " y = " << wspy(y1) << endl ;   // ❹

    // zmieniamy początek układu współrzędnych
    poczatek_x = 20 ;
    poczatek_y = -500 ;

    cout << "Gdy przesuniemy układ współrzędnych tak, \n"
         << "ze początek znajdzie się w punkcie \n"
         << poczatek_x << ", " << poczatek_y
         << "\nto nowe współrzędne punktu \n"
         << "w takim układzie są : "
         << " x = " << wspx(x1)           // ❺
         << " y = " << wspy(y1) << endl ;   // ❻

    // zagęszczamy skalę na osi poziomej
    skala_x = 0.5 ;
    cout << "Gdy dodatkowo zmienimy skalę poziomą tak, "
         << "ze skala_x = " << skala_x
         << "\nto ten sam punkt ma teraz współrzędne : \n"
         << " x = " << wspx(x1)           // ❼
         << " y = " << wspy(y1) << endl ;   // ❽
}
```



W wyniku wykonania tego programu na ekranie pojawi się

Mamy w punkt o współrzędnych

$x = 100$ $y = 100$

Gdy przesuniemy układ współrzędnych tak,

ze początek znajdzie się w punkcie

20, -500

to nowe współrzędne punktu

w takim układzie są : $x = 80$ $y = 600$

Gdy dodatkowo zmienimy skalę poziomą tak, że $\text{skala}_x=0.5$

to ten sam punkt ma teraz współrzędne :

$x = 40$ $y = 600$



Komentarz

Jeśli pierwsze Twoje wrażenie jest takie, że powyższy program więcej gada niż robi, to masz rację. A co w zasadzie robi? Przelicza współrzędne z jednego układu odniesienia na drugi.

Nie jest to takie nic – operacje robi się najczęściej przy postugiwaniu grafiką na ekranie. Ekran ma rozdzielczość np. 600 na 800 punktów, a my chcemy tam narysować coś, co u nas w programie ma rozmiary 100 na 100 i ma to w dodatku zająć cały ekran. Trzeba wówczas przeskalować współrzędne.

W tym rozdziale istotne jest dla nas to, że takich przeskalowań przy skompielowanym rysunku dokonuje się tysiące razy. Tu właśnie dochodzimy do sytuacji, gdy opłaca się użyć funkcji typu `inline`, dla przyspieszenia działania programu.

Jeśli nie jest dla Ciebie jasne na czym polega to przeskalowanie, to nie zaprzątaj sobie teraz tym głowy. Tutaj ważne jest bowiem, że zdefiniowaliśmy dwie funkcje typu `inline`: funkcję `wspx` i funkcję `wspy` (do przeliczania współrzędnej poziomej oraz pionowej). Te definicje widzisz w ❷ i ❸. Jak widać funkcje te korzystają także z niektórych zmiennych globalnych — zdefiniowanych w ❶.

Oczywiście definicje tych funkcji są – zgodnie z zasadą – powyżej miejsc, gdzie są po raz pierwszy wywoływane.

W rezultacie za każdym razem, gdy w programie wywołujemy skalowanie funkcji – u nas w ❹, ❺, ❻, ❼, ❽, ❾ wówczas odbywa się to w sposób maksymalnie szybki – mechanizmem `inline`.

5.9 Przypomnienie o zakresie ważności nazw deklarowanych wewnątrz funkcji

- ❖ Zakres ważności nazw deklarowanych w obrębie funkcji ogranicza się tylko do bloku tej funkcji. Nie można więc spoza funkcji za pomocą danej nazwy próbować dotrzeć do zmiennej będącej w obrębie funkcji.
- ❖ Nazwą deklarowaną w obrębie funkcji jest też etykieta. Nie można więc do niej skoczyć instrukcją `goto` spoza tej funkcji.

- ❖ Skoro etykieta jest lokalna dla funkcji, dlatego w dwóch różnych funkcjach mogą istnieć bezkonfliktowo identyczne etykiety.

5.10 Wybór zakresu ważności nazwy i czasu życia obiektu

Przez sposób, w jaki definiujemy obiekt, można decydować o zakresie ważności jego nazwy i o czasie jego życia.

Poniżej omówimy kilka możliwych sposobów definiowania obiektów.

5.10.1 Obiekty globalne

Obiekt zadeklarowany na zewnątrz wszystkich funkcji ma zasięg globalny. Znaczy to, że jest dostępny wewnątrz wszystkich funkcji znajdujących się w tym pliku. Z jednym zastrzeżeniem: jest znany dopiero od liniiki, w której nastąpiła jego deklaracja, w dół, do końca programu.

Oczywiście praktyka jest taka, że deklaracje umieszcza się na samym początku pliku, dzięki czemu obiekt jest dostępny wszystkim funkcjom z tego pliku.

```
#include <iostream.h>

int liczba ; // ❶
void fff(void) ;
/*****/
main()
{
    int i ;
    liczba = 10 ; // ❷
    i = 4 ;
    cout << "Wartosci: liczba = " << liczba
        << " i = " << i ;
    fff() ; // ❸
}
/*****/
void fff(void) { // ❹
    int x ;
    x = 5 ;
    liczba -- ; // ❺
    // i = 4 ; // blad ! ❻
    cout << " sumka = " << (x + liczba) ;
}
/*****/
```



W rezultacie wykonania tego programu na ekranie zobaczymy

Wartosci: liczba = 10 i = 4 sumka = 14



- ❶ Definicja obiektu globalnego o nazwie `liczba`.
- ❷ Przykład użycia zmiennej globalnej w funkcji `main`.
- ❸ Wywołanie funkcji `fff`.
- ❹ W funkcji `fff` możemy zdefiniować obiekt lokalny o nazwie `x` i go używać.
- ❺ Wewnątrz funkcji `fff` możemy także używać globalnego obiektu `liczba`. Jest przecież globalnie dostępny.
- ❻ Karygodny błąd. Obiekt `i` nie jest obiektem lokalnym tej funkcji. Niepoprawne jest takie odwoływanie się do obiektów lokalnych innych funkcji, bowiem zakres ważności nazwy `i` nie rozciąga się na funkcję `fff`. Tutaj nazwa `i` nie jest znana, więc kompilator zaprotestuje.

5.10.2 Obiekty automatyczne

W naszym przykładzie zmienne lokalne `i` oraz `x` są to tak zwane *zmiennne automatyczne*. Definiujemy je, a one – w momencie gdy kończymy blok, w którym powołałszy je do życia – automatycznie przestają istnieć. To dlatego, że obiekty automatyczne komputer przechowuje właśnie na stosie.

Jeśli po raz drugi wejdziemy do danego bloku (np. przy powtórnym wywołaniu funkcji `fff`) to zmienne takie zostaną powołane do życia po raz drugi. Nie ma żadnej gwarancji, że znajdą się akurat w tych samych miejscach w pamięci co poprzednio. Opuszczając blok – znowu zostaną zlikwidowane.

Wynikają z tego dwa wnioski:

- ❖ – skoro obiekt ten przestaje istnieć, to nie możemy liczyć na to, że przy ponownym wywołaniu tej funkcji zastaniemy go tam z wartością, którą miał na moment przed unicestwieniem.
Przy ponownym wywołaniu tejże funkcji obiekt taki zostanie zdefiniowany na nowo, bardzo możliwe, że w zupełnie innym miejscu pamięci (stosu)
- ❖ – skoro obiekt ten przestaje istnieć, to nie ma sensu by funkcja zwracała jego adres.^{†)} Adres po opuszczeniu takiej funkcji opisuje komórkę, która już nie należy do dawnego właściciela.

Sytuację tę można porównać do takiego obrazka. Właśnie się wyprowadzamy z dotychczasowego mieszkania, a na schodach dajemy jeszcze komuś nasz stary adres. Tymczasem pod tym starym adresem już nas nikt, nawet za chwilę, nie znajdzie.

Dokładnie to samo dotyczy zwracania referencji (przezwoiska) zmiennej lokalnej.

†) O zwracaniu rezultatu będącego adresem, porozmawiamy w rozdziale o wskaźnikach.



Inna sprawa. Pamiętać należy, że zmienne automatyczne nie są zerowane w chwili definicji (czyli w chwili powoływania do życia). Jeśli ich sami nie zainicjalizowaliśmy jakąś wartością, to w tych zmiennych automatycznych tkwią początkowo śmieci.

Dlatego:

Ze zmiennych automatycznych nie należy odczytywać wartości - zanim najpierw coś sensownego do nich nie zapiszemy.

Jeśli mimo tych ostrzeżeń coś stamtąd przeczytasz - będą to zupełnie przypadkowe wartości.

Wy tłumaczenie: zmienne automatyczne przechowywane są na stosie. Przydziela się im tam wymagany dla danego obiektu obszar - i nic więcej. Nie inicjalizuje się tego obszaru - (wpisując tam np. zero).

Natomiast:

Zmienne globalne - te są zakładane w normalnym obszarze pamięci ^{†)}. Ten obszar przed uruchomieniem programu jest zerowany, zatem zmienna globalna, jeśli jej nie inicjalizowaliśmy specjalnie - ma wartość 0.

Z obiektami automatycznymi łączy się słowo kluczowe `auto` stawiane przed definicją obiektu wewnątrz jakiegoś bloku (np. bloku funkcji lub bloku lokalnego). Jest ono jednak rzadko używane, gdyż obiekty tak definiowane są automatyczne przez domniemanie. Zatem jeśli np. w bloku funkcji definiujemy obiekt

```
auto int m ;
```

to jest to równoważne definicji

```
int m ;
```

5.10.3 Obiekty lokalne statyczne

Powiedzieliśmy, że zmienne lokalne dla jakiejś funkcji powoływane są do życia w momencie ich definicji, a gdy kończy się wykonywanie tej funkcji przestają istnieć. Wywołanie ponowne tej funkcji powoduje ponowne utworzenie takiej zmiennej, jej wartość stanowią śmieci, więc takiej zmiennej powinniśmy znowu nadać jakąś wartość początkową.

Czasem taki proceder nas zadawała, czasem jednak chciałoby się, by zmienna lokalna dla danej funkcji nie ginęła bez śladu, tylko przy ponownym wejściu do tej funkcji miała taką wartość, jak przy ostatnim opuszczaniu tej funkcji.

Żałujemy, że mamy napisać funkcję, która nic nie robi tylko pisze ile razy ją do tej pory wywołaliśmy. Musi mieć więc w środku jakiś licznik o czasie życia

^{†)} czyli: przypominając obrazek o stosie - nie na biurku, ale jakby w bibliotece

rozsciągającym się na cały czas wykonywania programu. Czyli taki czas życia, jaki mają zmienne globalne. Z drugiej jednak strony chcemy, żeby była znana tylko lokalnie przez tę funkcję. Gdybyśmy tego ostatniego warunku nie postawili – wystarczyłoby nam użyć zmiennej globalnej.

Oto ilustracja. Sprawę rozwiązujemy tu na dwa sposoby.

```
#include <iostream.h>
/*----- deklaracje funkcji-----*/
void czerwona(void) ; // ❶
void biala(void) ;
/*-----*/
main()
{
    czerwona(); // ❷
    czerwona();
    biala();
    czerwona();
    biala();
}
/*****/
void czerwona(void)
{
    static int ktory_raz ; // ❸
    ktory_raz ++ ;
    cout << "Funkcja czerwona wywolana " << ktory_raz
    << " raz\n" ;
}
/*****/
void biala(void)
{
    static int ktory_raz = 100 ; // ❹
    ktory_raz = ktory_raz + 1 ; // ❺
    cout << "Funkcja biala wywolana " << ktory_raz // ❻
    << " raz\n" ;
}
/*****/
```

Po wykonaniu programu na ekranie pojawi się:

```
Funkcja czerwona wywolana 1 raz
Funkcja czerwona wywolana 2 raz
Funkcja biala wywolana 101 raz
Funkcja czerwona wywolana 3 raz
Funkcja biala wywolana 102 raz
```

Przyjrzyjmy się poszczególnym punktom programu

- ❶ Funkcje przed pierwszym swym wywołaniem powinny być już znane kompilatorowi. Dlatego na górze programu umieściliśmy deklaracje tych funkcji. Dzięki temu kompilator wie jaka jest liczba i typ argumentów wysyłanych do funkcji, a także jaki typ jest zwracany jako rezultat wykonania tej funkcji. Od tej pory kompilator może już nas sprawdzać czy się nie pomyliliśmy w linijkach wywołania tych funkcji.

- ② W funkcji `main` widzimy serię wywołań funkcji `czerwona` i `biala`.
- ③ W funkcji `czerwona` istnieje definicja zmiennej (obiektu) typu `int`, o nazwie `ktory_raz`. Zauważ słowo kluczowe `static`.
- ④ W funkcji `biala` także istnieje definicja takiego samego obiektu. Obiekt ma taką samą nazwę jak w funkcji `czerwona`. Nie przeszkadza to nam jednak, gdyż są to obiekty lokalne, czyli zakres ważności ich nazw jest lokalny. Każda ma zakres ważności ograniczony do funkcji, w której została zdefiniowana. Zauważyłeś przydomek `static`. To on sprawia, że mimo iż obiekt jest lokalny – nie przestaje on istnieć w momencie, gdy kończy się jego zakres ważności. Słowo to nazywa się też modyfikatorem, gdyż zwykłą definicję obiektu lokalnego modyfikuje tak, iż obiekt staje się „nieśmiertelny”.
- ⑤ Zwracam uwagę raz jeszcze – kończy się zakres ważności nazwy obiektu, ale obiekt nie ginie. Jakby zapada w stan hibernacji. Kiedy ponownie wywołana zostaje funkcja, obiekt budzi się z zimowego snu i ma taką wartość, z jaką go ostatnio zostawiliśmy. Na dowód tego w rezultacie wykonania na nim zwiększenia o jedną i wypisaniu wartości na ekran widzimy, że rzeczywiście mamy coś w rodzaju licznika.
- ⑥ Definicja obiektu statycznego w funkcji `czerwona` nie zawiera inicjalizacji, czyli podstawienia wartości początkowej. Obiekty statyczne – jako obiekty przypominające czasem życia obiekty globalne – nie są przechowywane na stosie, lecz w normalnej pamięci. Wynika stąd fakt, że obiekty takie – (w przeciwieństwie do obiektów na stosie) – bezpośrednio po definicji nie zawierają żadnych „śmieci”, tylko mają w sobie wartość zerową. Jeśli ta wartość nam odpowiada, to nie musimy jeszcze raz jawnie wstawiać tam tego zera. W przypadku definicji ④ ta wartość nam jednak nie odpowiadała, więc wstawiliśmy tam liczbę 100. Dzięki temu nasz licznik zaczął liczyć od wartości początkowej 100

Czy tego samego efektu nie moglibyśmy uzyskać za pomocą zmiennych globalnych? Tego samego nie, bowiem nie może być dwóch obiektów globalnych o identycznej nazwie. Musielibyśmy więc zdefiniować dwie zmienne globalne o różniących się nazwach np.

```
int ktory_raz_cz ;
int ktory_raz_bi = 100 ;
```

Używanie zbyt wielu zmiennych globalnych nie jest eleganckim rozwiązaniem i zdradza zły styl programowania. Raz, że trzeba uważać, by wymyślić nazwę do tej pory jeszcze nie istniejącą, a dwa, że obiekt globalny jest dostępny dla każdego – co zwiększa ryzyko błędu.

Co by było gdybyśmy z naszego programu usunęli słowa `static` ?

Obie zmienne `ktory_raz` staną się wówczas obiektami automatycznymi. Co to oznacza łatwo zobaczyć na ekranie po wykonaniu takiej wersji programu

```
Funkcja czerwona wywolana 1259 raz
Funkcja czerwona wywolana 1259 raz
Funkcja biala wywolana 101 raz
Funkcja czerwona wywolana 1259 raz
Funkcja biala wywolana 101 raz
```

Obiekty straciły nieśmiertelność. Ginią w momencie, gdy kończy się ich zakres ważności, a po powtórnych narodzinach, nie pamiętają niczego czym były do tej pory. Widać to wyraźnie na tekście wypisywanym z funkcji `biała`. Zmienna jest zakładana za każdym razem od nowa i za każdym razem nadawana jest wartość 100. W rezultacie zwiększenia o 1 na ekranie pojawia się za każdym razem liczba 101.

Z obiektem z funkcji `czerwona` jest o wiele gorzej. Także i on staje się automatyczny i jako taki zakładany jest na stosie. Ponieważ jednak nie nadajemy mu żadnej wartości początkowej, więc są w nim wstępnie „śmieci”. Te śmieci zwiększamy o 1 i wypisujemy na ekranie. Stąd taka zdumiewająca liczba. (Na Twoim komputerze będzie to na pewno jakaś inna liczba, mimo wszystko jednak, pozostałość po poprzedniej treści stosu).

Podsumowanie:



Jeśli chcemy, by jakaś zmienna (ogólniej obiekt) definiowana w obrębie funkcji nie była po zakończeniu pracy funkcji niszczona i zachowywała swoją wartość do „następnego razu”, to musimy ją zdefiniować jako statyczną.

Jak pamiętamy, definicja taka wygląda identycznie jak definicja zmiennej automatycznej, z tą różnicą, że przed definicją stoi słowo `static`

```
static float m = 1.7 ;
```

Taka zmienna nie jest już lokowana na stosie (jak zmienne automatyczne), tylko w tym obszarze pamięci, gdzie zmienne globalne, zatem nie ma w niej „śmieci” tylko jest inicjalizowana zerem. Chyba, że tak jak w naszym wypadku, zażądamy inicjalizacją inną wartością (1.7)

Gdy do funkcji wejdziemy po raz pierwszy – taka właśnie wartość czekała będzie tam na nas. Gdy po zakończeniu funkcji i ewentualnych modyfikacjach tej zmiennej – opuścimy funkcję, obiekt ten stanie się dla nikogo niedostępny. Jeśli jednak ponownie funkcja ta zostanie wywołana, zastaniemy tę statyczną zmienną z taką wartością, z jaką ją ostatni raz zostawiliśmy. (Inicjalizacja wartością 1.7 odbywa się tylko jeden jedyny raz, potem zmienna statyczna pamięta już swoją ostatnią wartość).



Zapamiętaj:

Obiekty globalne

– są wstępnie inicjalizowane zerami

Obiekty lokalne:

- automatyczne – zakładane są na stosie, a tam nic nie jest wstępnie inicjalizowanie, zatem obiekty te wstępnie zawierają „śmieci”,
- statyczne – ponieważ mają pożyć dłużej, zakładane są w tym obszarze pamięci co obiekty globalne – a więc są wstępnie inicjalizowane zerami.

5.11 Funkcje w programie składającym się z kilku plików

Dopóki nasz program jest nieduży nie ma problemu: całość może się zmieścić w jednym pliku dyskowym. Ten plik kompilujemy i linkujemy jeśli chcemy otrzymać program w wersji gotowej do uruchomienia.

Przychodzi jednak taki moment, że program rozrasta się tak bardzo, że kompilacja trwa długo. Jakakolwiek minimalna poprawka wymaga znowu tej długiej kompilacji. Wtedy musisz podjąć decyzję, że odąd program nie będzie już w jednym pliku – dzielisz go na dwa lub więcej. Jak to zrobić?

Program napisany w jednym pliku można podzielić na dwa pliki tylko w miejscu między definicjami funkcji. Nie można więc dzielić tak, że w pierwszym pliku A będzie funkcja pierwsza, druga i pół trzeciej, a w drugim pliku B reszta trzeciej, czwarta i piąta. Funkcja trzecia musi być albo cała w pliku A, albo cała w pliku B.

O czym jeszcze musimy pamiętać? :

Przede wszystkim o tym, że po to, by funkcje z pliku B miały dostęp do jakichkolwiek zmiennych globalnych z pliku A – trzeba w pliku B umieścić deklaracje tych zmiennych.

Deklaracje – a nie definicje. Definicje (czyli rezerwacja na nie miejsca) odbyły się już w pliku A. W pliku B chcemy mieć tylko do nich dostęp, czyli móc się do nich odwoływać. Aby móc się odwołać do jakichś nazw muszą one zostać zadeklarowane. Do tego, jak wiemy, w przypadku obiektów takich jak na przykład zmienne – służą deklaracje wykonywane za pomocą słowa `extern`.

Jeśli więc w pliku A mamy następujące zmienne globalne

```
int n ; // to wszystko są definicje
float x ;
char z ;
```

to aby móc z nazw `x`, `n` z korzystać w pliku B musimy tam zamieścić deklaracje.

```
extern int n ; // deklaracje !
extern float x ;
extern char z ;
```

Deklaracje te są potrzebne kompilatorowi wtedy, gdy będzie kompilował plik B. Mówią mu one mniej więcej coś takiego:

- Jeśli w pliku B napotkasz nazwę `n`, to na razie deklaruję, że oznacza ona obiekt typu `int`. Gdzie się ten obiekt znajduje? Nie mówię teraz, bo może nawet na zewnątrz (`extern`) tego pliku. (Ale to nic pewnego).
- Jeśli w pliku B napotkasz nazwę `x`, to wiem, że oznacza ona obiekt typu `float`. Także nie określam gdzie dokładnie on jest.
- Jeśli w pliku B napotkasz nazwę `z`... dalej Ci czytelniku oszczędzę...

Dopiero na etapie linkowania (łączenia) tych plików ze sobą, kod z pliku B „dowie się” gdzie to w pamięci naprawdę będą obiekty n , x , z .

Jeśli chcemy, żeby z pliku B można było wywołać jakąś funkcję z pliku A – to także musimy umieścić w pliku B jej deklarację. W wypadku deklaracji nazwy funkcji nie potrzeba już słowa `extern` – jest ono przyjmowane jako domniemane.

W sumie więc zanim w pliku B pojawią się jego funkcje, najpierw muszą wystąpić deklaracje zmiennych i funkcji z pliku A. Nie wszystkich – tych, które z pliku B będą używane.

Czasem jest wygodne umieścić te wszystkie deklaracje w osobnym pliku, – tak zwanym pliku nagłówkowym, który po prostu bezpośrednio przed procesem kompilacji jest włączany do pliku B (a może i dalszych).

To automatyczne wstawianie do pliku wykonuje za nas specjalna dyrektywa

```
#include "naglowek.h"
```

Jest to tzw. dyrektywa preprocesora^{†)}, która bezpośrednio przed rozpoczęciem pracy kompilatora wstawia do pliku, inny plik o danej nazwie (tutaj plik: `naglowek.h`) znajdujący się w bieżącym katalogu (bieżącym – bo nazwa jest ujęta w cudzysłów). Rozszerzenie `.h` jest zwyczajowym rozszerzeniem dawanych plikom nagłówkowym (`.h` to skrót od angielskiego `header` – nagłówek)

Co prawda o tych sprawach dopiero będziemy mówić, jednak chyba już zdążyłeś się oswoić z linijką

```
#include <iostream.h>
```

która towarzyszy naszym programom od dłuższego czasu, a jest niczym innym jak wstawieniem do naszych programów – pliku z deklaracjami zmiennych i funkcji z biblioteki wejścia/wyjścia.

Oto przykład programu, który podzielony został na dwa pliki:

Wszystkie deklaracje zebrano w osobnym pliku nagłówkowym o nazwie `nagl.h`. Plik ten jest włączany do obu plików programu.

Oto treść pliku `afryka.C` :

```
#include <iostream.h>
#include "nagl.h"
int ile_murzynow = 9 ;
main()
{
    cout << "Poczatek programu\n" ;
    funkcja_francuska();
    funkcja_niemiecka();
    cout << "Koniec programu \n" ;
}
```

†) Będziemy o tym mówili w następnym rozdziale.


```

/*****
void funkcja_egipska()
{
    cout << "Jestem w Kairze !----- \n" ;
    cout << "Na swiecie jest " << ile_murzynow
        << " murzynow, oraz " << ile_europejczykow
        << " europejczykow \n" ;
}

/*****
void funkcja_kenijska()
{
    cout << "Jestem w Nairobi ! ----- \n" ;
    cout << "Na swiecie jest " << ile_murzynow
        << " murzynow, oraz " << ile_europejczykow
        << " europejczyków \n" ;
}

*****/

```

A oto plik europa.c :

```

#include <iostream.h>
#include "nagl.h"
int ile_europejczykow = 8 ;
/*****/
void funkcja_francuska()
{
    cout << "Jestem w Paryżu ! ***** \n" ;

    cout << "Na swiecie jest " << ile_murzynow
        << " murzynow, oraz "
        << ile_europejczykow << " europejczyków \n" ;

    funkcja_egipska() ;
}

/*****/
void funkcja_niemiecka(void)
{
    cout << "Jestem w Berlinie ! ***** \n" ;

    cout << "Na swiecie jest " << ile_murzynow
        << " murzynow, oraz "
        << ile_europejczykow << " europejczykow \n" ;
    funkcja_kenijska();
}

/*****/

```

A tak wygląda zawartość pliku nagl.h

```

extern int ile_murzynow ;
extern int ile_europejczykow ;

void funkcja_egipska() ;
void funkcja_kenijska() ;
void funkcja_francuska() ;
void funkcja_niemiecka() ;

```


Po skompilowaniu plików `europa.c` i `afryka.c` oraz po połączeniu (zlinkowaniu) ich ze sobą otrzymamy gotowy program. Program ten po uruchomieniu spowoduje, że

na ekranie pojawi się następujący tekst

```
Początek programu
Jestem w Paryżu ! *****
Na świecie jest 9 murzynów, oraz 8 europejczyków
Jestem w Kairze !-----
Na świecie jest 9 murzynów, oraz 8 europejczyków
Jestem w Berlinie ! *****
Na świecie jest 9 murzynów, oraz 8 europejczyków
Jestem w Nairobi ! -----
Na świecie jest 9 murzynów, oraz 8 europejczyków
Koniec programu
```

Istota programu polegała na pokazaniu, że dzięki deklaracjom funkcje z jednego pliku programu mogą być wywoływane przez funkcje z innego pliku składającego się na program.

Także zmienne globalne z jednego pliku mogły być używane w innym pliku (mówimy module programu). Wszystko dlatego, że do obu plików `europa.c` i `afryka.c` wstawiliśmy plik nagłówkowy `nagl.h` zawierający deklaracje funkcji i zmiennych.

-Mam Cię! - zawołałeś już pewnie triumfalnie - Jak można włączać plik, w którym jest linijka:

```
extern int ile_murzynow ;
```

do pliku `afryka.c`, w którym zaraz jest definicja

```
int ile_murzynow = 9 ;
```

Najpierw mówimy, że deklarujemy, iż obiekt `int` o nazwie `ile_murzynow` jest gdzieś indziej (external – znaczy po angielsku: zewnętrzny), a zaraz potem definiujemy zmienną `int ile_murzynow` właśnie w tym pliku! To oszustwo!

Brawo, brawo. To znaczy, że jesteś czujny. Oto moja obrona. Nie jest to oszustwem pod warunkiem, że nie bierzemy na serio słowa `extern`. To słowo nie oznacza, że obiekt jest zdefiniowany koniecznie na zewnątrz. Przypomnij sobie niedawne moje słowa:

... Gdzie się ten obiekt znajduje? Nie mówię teraz, bo może nawet na zewnątrz (`extern`) tego pliku. (Ale to nic pewnego)...

Otoż słowo `extern` znaczy tu tylko to, że **deklarujemy** obiekty, ale ich nie definiujemy. Przypomnij sobie też, że mówiliśmy, iż obiekt może mieć dowolną liczbę deklaracji, ale tylko jedną definicję.

Zatem poprawna jest taka sekwencja w programie:

```
extern int k ;          // deklaracja
extern int k ;          // deklaracja
int k ;                 // definicja - tylko jedna !
extern int k ;          // deklaracja
extern int k ;          // deklaracja
```

Natomiast nie jest poprawna taka sekwencja:

```
extern int k ;           // deklaracja
extern int k ;           // deklaracja, druga, nie szkodzi
int k ;                  // definicja !
int k ;                  // definicja - powtórzona to błąd!
```

Jeśli jednak trawi Cię uczciwość i chciałbyś dosłownie rozumieć słowo `extern`, to czujesz chyba teraz dlaczego zrezygnowano z takiej dosłowności – po to, by nie budować dla każdego modułu programu osobnego pliku nagłówkowego. Jeden plik nagłówkowy może pomieścić te same deklaracje i być wstawianym do różnych modułów programu. W naszym wypadku pliki `europa.c` i `afryka.c` pracują z tym samym plikiem nagłówkowym `nagl.h`.

A jeśli i to Cię nie przekonuje i od słowa `extern` żadasz dosłowności, to zrób teraz dwa pliki nagłówkowe – osobny dla `afryki`, a osobny dla `europy`.
Za karę !

Zagadka

Jak sądzisz, czy poprawny jest taki zapis:

```
extern int m = 4 ;
```

Jest w tym pewna sprzeczność, bo z jednej strony używamy słowa `extern` – charakterystycznego dla deklaracji, a z drugiej dokonujemy inicjalizacji liczbą 4 – co jest charakterystyczne dla definicji.

Oto odpowiedź:

Kompilator widząc taki zapis uzna, że jest to **definicja**, czyli tak samo jakby był to zapis

```
int m = 4 ;
```

5.11.1 Nazwy statyczne globalne

Jeśli w deklaracji nazwy globalnej postawimy przydomek `static` to oznacza to, że nie życzymy sobie, by ta nazwa była znana w innych plikach (modułach) składających się na nasz program. Podkreślam, że chodzi tu o nazwy, które są deklarowane globalnie (czyli na zewnątrz wszystkich funkcji).

Za sprawą tego przydomka nazwa jest nadal globalna, ale może być znana tylko w tym jednym pliku. Dotyczy to nie tylko nazw obiektów, ale także nazw funkcji. Wyznam, że nie wiem dlaczego służy do tego ten sam przydomek `static`, skoro chodzi tu o zupełnie inne znaczenie niż poprzednio. Sądzę, że musiały tu być jakieś „względny historyczne”.

Powodem by globalną nazwę funkcji czy obiektu zaopatrzyć w ten przydomek `static` (czyli tutaj jakby: ściśle tajne) jest najczęściej chęć by inne moduły (pliki) programu, które danej funkcji czy zmiennej nigdy nie mają używać – nie musiały dbać o unikalność nazw.

Przykładowo: w większym zespole piszemy program, a ja pisząc mój fragment wymyślam jakieś nazwy potrzebne mi w moim module. Niektóre z tych nazw będą ważne i ustalane ze wszystkimi kolegami – np. funkcje, które oni mają z mojego modułu wywoływać. Jednak oprócz takich nazw będą też nazwy moich globalnych zmiennych pomocniczych czy nazwy pomocniczych funkcji.

Teoretycznie powinienem wszystkich kolegów o tych nazwach także informować – by nie wymyślili przez przypadek funkcji o identycznej nazwie.

Po co ten kłopot! Wystarczy, że przed tymi globalnymi nazwami umieszczę przydomek `static`. Sprawi to, że owe nazwy nie będą wówczas znane w modułach innych niż mój. Od tej pory nie ma więc obawy o kolizję nazw.

Sposób ten bardzo przydaje się w wypadku pisania bibliotek.

5.12 Funkcje biblioteczne

Programować w C czy C++ to w zasadzie żadna sztuka, wystarczy opanować tych kilkanaście instrukcji, wiedzieć jakie są operatory i jeszcze parę drobnych rzeczy. Po krótkim czasie nie będziesz musiał w trakcie programowania zaglądać do podręcznika. Jest jednak coś, co zawsze leżeć będzie na Twoim biurku: opis funkcji bibliotecznych.

Funkcje biblioteczne nie są częścią języka C i C++. Są to po prostu funkcje, które ktoś tam napisał, a okazały się tak dobre i tak często przydatne, że zrobiono z nich standardową bibliotekę. Biblioteka ta stała się tak popularna wśród programistów, że każdy producent kompilatora C++ musi ją także dostarczyć. Muszą być w niej funkcje, które wykonują pewne standardowe usługi i w dodatku poszczególne funkcje muszą nazywać się tak samo, jak analogiczne funkcje w innych wersjach kompilatora.

Przykładowo:

Kiedys ktoś napisał funkcję, która zamienia litery w taki sposób, że jeśli wyślemy jej jako argument wielką literę 'W' to w odpowiedzi jako rezultat dostaniemy małą literę 'w'. Czyli funkcja z wielkich liter robi małe. Natomiast małym literom nie szkodzi. Zwraca je jako małe. Nie szkodzi też cyfrom, znakom specjalnym itd. Oto realizacja takiej funkcji. Nazywamy ją `tolower` (ang. – do niższej).

Dygresja :

Nazwa pochodzi od angielskiego określenia liter małych jako „lower case” (niższa kasetka) – wielkie litery nazywają się „upper case” (wyższa kasetka). Określenia te pochodzą od pracy zecerów dawniej składających teksty w drukarni ręcznie. Litery takie znajdowały się w dwóch kasetkach z przegródkami. Kasetka z małymi literami, jako częściej używanymi, leżała bezpośrednio przed zecerem, natomiast po litery wielkie musiał on sięgać wyżej.

A oto jak wygląda nasza funkcja:

```

/*****
char tolower(char znak)
{
    int roznica = 'a' - 'A' ;

    if( (znak >= 'A') && (znak <= 'Z') )
        return (znak - roznica) ;
    else
        return znak ;
}
*****/

```

Jak ta funkcja działa? Przysyłamy do niej kod znaku (jego reprezentację liczbową). Następuje sprawdzenie czy jest to znak z przedziału A - Z. Robimy to przez sprawdzenie czy kod ASCII przysłanego znaku jest większy lub równy kodowi znaku 'A' i równocześnie mniejszy lub równy kodowi znaku 'Z'.

Jeśli tak, to do kodu znaku dodajemy liczbę *roznica*. Skąd wiemy, że trzeba dodać właśnie tyle? Z tablicy kodów znaków ASCII. Z tablicy takiej łatwo znajdziemy, że

A - 65	a - 97
B - 66	b - 98
C - 67	c - 99
.....	
Z - 90	z - 122

Nie jest to przypadkiem, że różnica między kodem litery wielkiej, a kodem odpowiadającej jej litery małej jest zawsze stała. Dzięki temu, aby ze znaku ASCII reprezentującego literę wielką zrobić znak reprezentujący literę małą, wystarczy dodać do jego kodu tę właśnie różnicę)

32 // czyli: 97 - 65

We wszystkich innych powszechnie stosowanych kodach powinna również być zachowana zasada, że kolejne litery alfabetu reprezentowane są kolejnymi kodami liczbowymi.

Jeśli natomiast przysłany do funkcji znak nie jest z interesującego nas przedziału, to go prosto zwracamy, bez zmian.

Ktoś kiedyś napisał taką funkcję. Okazała się tak przydatna, że teraz jest w bibliotece standardowej.



Załóżmy jednak, że interesuje Cię ewentualna zamiana liter małych na wielkie. Co zatem robimy? Nie, nie zabieramy się do pisania! Bierzemy do ręki opis funkcji bibliotecznych dostępnych w naszym kompilatorze.

W opisie tym spisane są wszystkie dostępne standardowe funkcje biblioteczne. Nie musisz ich znać, nie musisz ich nawet używać. Prawie wszystko możesz przecież sobie napisać samemu. Będzie to jednak wyważanie otwartych drzwi.

Uwaga dla programistów C

Jeśli znasz standardowe funkcje biblioteczne z klasycznego C, to znajdziesz je wszystkie także i w C++, więc nie musisz się niczego uczyć.

W opisie funkcji bibliotecznych, funkcje takie są czasami uszeregowane alfabetycznie co do swojej nazwy, czasami podzielone na rozdziały według tego, do czego służą. Odnalezienie właściwej nie jest rzeczą trudną. W naszym wypadku łatwo dojdziemy, że obok funkcji `tolower` jest funkcja `toupper`, która zamienia literę małą na wielką. Tego właśnie szukaliśmy.

Jednak powiedzmy jasno: w opisie nie znajdziemy tekstu (ciała) funkcji, takiego jak to na przykład napisaliśmy dla funkcji `tolower`. Jest tam tylko opisane co ta funkcja robi, z jakimi argumentami należy ją wywoływać i jaki typ funkcja ta

zwraca. Natomiast nie mówi się jak jest to zrealizowane. Tym lepiej, bo komuś, kto chce posłużyć się daną funkcją, nie jest potrzebna znajomość wszystkich sprytnych sztuczek, którymi posłużył się piszący tę funkcję.

Dodatkowo przy opisie danej funkcji bibliotecznej znajdziemy też informację, gdzie znajduje się deklaracja tej funkcji. Po co ta informacja?

W zasadzie mówiliśmy już o tym kilkakrotnie. Pamiętasz zapewne — mówiliśmy, iż każda nazwa w języku C++ musi być zadeklarowana zanim zostanie użyta. W naszym wypadku `toupper` to też taka nazwa. Jest to nazwa funkcji — musi więc znaleźć się w programie deklaracja tej funkcji. Abyśmy nie musieli pisać jej sami (przepisując z opisu biblioteki) została ona już wpisana do pliku nagłówkowego o nazwie `ctype.h`.

Wystarczy więc wstawić ten plik nagłówkowy do tekstu naszego programu za pomocą dyrektywy preprocesora

```
#include <ctype.h>
```

i sprawa jest załatwiona. Jeśli byśmy o tym zapomnieli to kompilator C++ nam tego nie podaruje i upomni się o nią.

(Kompilator klasycznego języka C by nam to podarował, ale ja osobiście taką pobłażliwość kompilatora C okupiłem już kilkoma godzinami poszukiwań błędów w programie. Dlatego wolę teraz ten bardziej krytyczny język C++).

Jak widać z funkcjami bibliotecznymi sprawa jest prosta: najpierw szukamy funkcji w opisie^{†)}, potem wstawiamy do programu plik nagłówkowy z jej deklaracją, a następnie posługujemy się tą funkcją w programie. Tak, jakby to była nasza własna funkcja.

Funkcja jest jednak zdefiniowana w pliku bibliotecznym. Plik ten trzeba dołączyć w czasie linkowania (czyli łączenia). Jest to zwykle bardzo proste, ale tu niestety nie mogę Ci nic pomóc. Jak to zrobić — znajdziesz w opisie swojego kompilatora.

Najczęściej jednak jest to tak zintegrowane z kompilatorem, iż nie trzeba wydawać specjalnych rozkazów.

Co chciałbym, abyś z tego paragrafu zapamiętał:

To mianowicie, że opis (leksykon) funkcji bibliotecznych będzie Twoim najlepszym przyjacielem. Trochę tak, jak „ściąga” na kolokwium.

Osobiście zawsze, jak tylko przyjdzie mi pracować na nowym typie komputera, łączywie rzucam się na opisy funkcji bibliotecznych. Znajdują się tam bardzo różne ciekawe funkcje. Oprócz funkcji naprawdę standardowych takich jak np. obliczenie x do potęgi $17/23$, znaleźć tam można też takie, które posłużą nam do narysowania na błękitnym ekranie białego kółka wypełnionego deseniem z czerwonych serduszek.



†) zwanym po angielsku: Reference Manual

6 Preprocesor

Preprocesor to jakby przednia straż kompilatora. Zanim kompilator przystępuje do akcji, tekst programu jest przeglądany przez preprocesor. Zwykle preprocesor wkracza do pracy sam, bez jakiejkolwiek inicjatywy z naszej strony. W zasadzie byłby dla nas niezauważalny i nieistotny, gdyby nie kilka przysług, które może nam oddać.

O tym, co ma zrobić dla nas preprocesor, decydujemy za pomocą tak zwanych *dyrektyw preprocesora*. Dyrektywy takie są umieszczane przez nas w stosownych miejscach programu, a ich znakiem szczególnym jest to, że pierwszym nie-białym znakiem takiej linii jest znak #. Przed tym znakiem mogą być w linijsie tylko spacje i tabulatory.

Z niektórymi takimi dyrektywami już się oswoiliśmy. Na przykład wielokrotnie występowała już w naszych programach dyrektywa

```
#include <iostream.h>
```

Dyrektywy są żądaniami, by preprocesor wykonał jakąś akcję. Preprocesor podejmuje ją zwykle na nasze wyraźne żądanie. Jest jednak akcja, która zostanie wykonana przez niego samoczynnie. O tym mówi poniższy paragraf.

6.1 Na pomoc rodakom

Jeśli Twój komputer ma na klawiaturze znaki:

\ ^ [] { } !

to możesz opuścić ten paragraf i zacząć czytać następny. Mówić tu będziemy o tym, jak preprocesor pomaga programistom, którzy takich znaków na klawiaturach nie mają.



Zatem, jeśli nie masz na klawiaturze tych znaków, to po pierwsze bardzo Ci współczuję i zastanawiam się jakim cudem dobrnąłeś w tej książce aż tutaj. Wiemy jak bardzo ważne i częste są te znaki w programowaniu w C++. Niestety, w niektórych krajach komputery mają w miejscu tych znaków swoje znaki narodowe. Przykładem niech będzie Dania, gdzie w tych miejscach są znaki typu:

Æ Å å

Pewnie nikt by się biednymi Duńczykami nie przejął, gdyby nie to, że twórca języka C++ Bjarne Stroustrup pochodzi z Danii.^{†)} Także w innych krajach mogą być na klawiaturach jakieś inne znaki zamiast tych, o które nam chodzi. Co wtedy?

Wyjście jest takie: Nieobecne na klawiaturze znaki można w programie uzyskać zastępując je tak zwanymi *sekwencjami trzyznakowymi*. Są one tak dobrane, że odlegle przypominają znak, który mają symulować. Np. sekwencja ?? (oznacza to samo co znak [

Oto zebrane sekwencje i ich odpowiedniki:

??=	#	??(	[
??/	\	??)	]
??'	^	??<	{
??!		??>	}

Prosty program wygląda z użyciem tych symboli tak:

```

??=include <iostream.h>
main()
??<
    int i ;
    char tab??(30??) ;

    for( i = 0 ; i < 30 ; i ++ )
    ??<
        tabl??(i??) = 'a' + i ;
        cout << "zaladowanie do elementu " << i
            << " wartosci " << tabl??(i??) << endl ;
    ??>
    cout << "Poszukiwanie litery k lub m ??/n" ;

    for(i = 0 ; i < 30 ; i ++ )
    ??<
        if((tabl??(i??)=='k') ??!??! (tabl??(i??)=='m')
        ??<
            cout << "Znak " << tabl??(i??)
                << "jest w elemencie " << i << endl ;
        ??>
    ??>
??>

```

†) Mieszka w USA i pracuje w AT&T.

Jeśli powyższy program wydaje Ci się mało czytelny, to nie ma rady - musisz pomyśleć o zmianie klawiatury Twojego komputera.

6.2 Dyrektywa #define

Ta dyrektywa preprocesora ma postać:

```
#define wyraz ciąg znaków zastępujących go
```

Dyrektywa ta[†] powoduje, że w kompilowanym pliku każde następne wystąpienie słowa *wyraz* będzie zastępowane wyszczególnionym ciągiem znaków. Oczywiście nie musi być to koniecznie wyraz. Musi być to jednak grupa znaków bez białego znaku w środku. To natomiast, co jest ciągiem znaków zastępujących, może mieć w środku białe znaki.

Na przykład poniższy fragment

```
#define CZTERY 4
// ...
float tablica[CZTERY] ;
i = CZTERY + 2 * CZTERY ;
funkcja(CZTERY- 0.3) ;
cout << "Mowilem ci to CZTERY razy " ;
```

zostanie przez preprocesor automatycznie zamieniony na

```
// ...
float tablica[4] ;
i = 4 + 2 * 4 ;
funkcja(4 - 0.3) ;
cout << "Mowilem ci to CZTERY razy " ;
```

Zwyczajowo wyrazy zastępowane pisze się wielkimi literami po to, by w tekście programu przypomnieć sobie, że nie chodzi tu o nazwę obiektu, lecz o działanie dyrektywy `define`. Powtarzam – jest to tylko zwyczaj. Równie dobrze można pisać litery małe.



Zauważ, że dyrektywa `define` nie penetruje wnętrza stringów (czyli tzw. stałych tekstowych). Jest to oczywiście zupełnie logiczne. Wewnątrz stringów bowiem, zupełnie przypadkowo mogą się pojawić identyczne wyrazy jak ten zastępowany, ale w zupełnie innych znaczeniach i kontekstach. Lepiej więc, że `define` nie ma do nich dostępu.

Oto dalsze przykłady:

```
#define MAX_LICZ_PASAZER      250
#define LICZB_STEWARD        8
#define PASAZ_NA_STEWD (MAX_LICZ_PASAZER/ LICZB_STEWARD)
```

[†]) (czytaj: „defajn”)

Jak widać następne dyrektywy define mogą korzystać z właśnie zdefiniowanych powyżej nazw.

Dyrektywą define można definiować zastępowanie dowolnego ciągu znaków. Przykładowo – po takiej dyrektywie

```
#define ZEGAREK      (while(!zajety) czas(); )
```

można w programie zastosować zapis

```
funkcja_a();
    i = 15 * czynnik ;
    ZEGAREK ;
    x = 15 * log(17);
```

co odpowiada zapisowi

```
funkcja_a();
    i = 15 * czynnik ;
    (while(!zajety) czas(); ) ;
    x = 15 * log(17);
```

Czyli dyrektywą define możemy sobie oszczędzić trochę pracy przy pisaniu długich, a często występujących instrukcji. Zauważ, że dyrektywa define nie ma na końcu średnika. To dlatego, że nie jest to normalna instrukcja programu, lecz dyrektywa dla preprocesora. Jeśli przez zapomnienie postawimy średnik na końcu, to zostanie on uznany jako jeden ze znaków zastępujących. Czyli w wypadku naszej definicji

```
#define CZTERY 4 ;
```

ten sam fragment wyglądałby następująco:

```
// ...
float tablica[4 ;] ;
i = 4 ; + 2 * 4 ; ;
funkcja(4; - 0.3) ;
cout << "Mowilem ci to CZTERY razy " ;
```

Oczywiście na skutek wstawienia tego średnika, znalazł się on w zupełnie nieodpowiednich dla siebie miejscach – wywołując w trakcie kompilacji komunikaty o błędach. Taki błąd jednak wykrywa się natychmiast, więc nie jest to groźne.

Bardzo długie dyrektywy

Może się zdarzyć, że ciąg znaków zastępujących jest długi i z tego powodu trudno umieścić dyrektywę w jednej linii. Nie ma problemu: gdy uznamy, że linia powinna się skończyć - umieszczamy na jej końcu znak \ (bekslesz) i kontynuujemy pisanie dyrektywy w linii następnej. Tym sposobem dyrektywa może się ciągnąć przez wiele linijek.

```
#define HMI Hahn-Meitner Institut\
fuer Kernforschung\
W.Berlin 39, Glienickestr 100
```

Zwracam uwagę, że chwyt ten stosować trzeba tylko wobec dyrektywy preprocesora. Zwykłą instrukcję programu można przecież swobodnie umieszczać w kilku liniach. To znamy od dawna.

Definiowanie stałych

W języku C klasycznym dyrektywa ta tradycyjnie używana była do definiowania stałych. W języku C++ oprócz tego sposobu mamy inny, lepszy: mianowicie obiekty typu `const`.

Oto porównanie. Stary styl:

```
#define ROZDZIELCZOSC 8192
long widmo[ROZDZIELCZOSC] ;
```

Nowy styl:

```
const int rozdzielczosc = 8192 ;
long widmo[rozdzielczosc] ;
```

O tym, żeby nie używać `define` jako sposobu deklarowania stałych mówiliśmy już przy okazji definiowania obiektów typu `const`. (str. 47)

W skrócie to można streścić tak:

Stosując definiowanie stałych jako obiekty typu `const` dajemy kompilatorowi większe szanse wykrycia naszych ewentualnych błędów.

Innym poważnym zastosowaniem dyrektywy `define` było definiowanie tak zwanych *makrodefinicji*. To także nie wytrzymało próby czasu, o czym porozmawiamy za chwilę.

Zastosowaniem, które próbę czasu wytrzymało, jest między innymi definiowanie symboli dla kompilacji warunkowej. Także o tym będziemy mówili niebawem.



Na koniec jeszcze jedna uwaga. Dużo się napracowałem by wpoić Ci zasadę, że definicja to jest moment, gdy rezerwuje się dla jakiegoś obiektu miejsce w pamięci – czyli inaczej, jest to miejsce w programie, gdzie dany obiekt się rodzi. Tymczasem tutaj słowo `define` znaczy coś zupełnie innego. Tutaj tylko określamy, że jak napotka się jeden ciąg znaków, to należy go zamienić na inny ciąg znaków. Żaden obiekt tutaj nie powstaje. Zapytasz: gdzie tu konsekwencja?

Nie ma żadnej konsekwencji i przepraszam Cię za to. Na usprawiedliwienie dodam, że tamto mówiłem o kompilatorze. W tym rozdziale rozmawiamy o jego służącym – preprocesorze, który przychodzi jeszcze zanim zjawia się prawdziwy kompilator. Nie wymagamy więc konsekwencji od jego głupszego służącego. Jak widać nieco innym językiem mówi się do preprocesora – czyli: proszę o pobłażliwość w liniach zaczynających się od znaku `#`.

6.3 Dyrektywa #undef

Jeśli dyrektywą `define` określiliśmy jakąś nazwę, to owo polecenie (nie chce mi przejść przez gardło – ta: *definicja*) obowiązuje od momentu wystąpienia tej linii w programie, a ważne jest do końca pliku.

Czasem jednak może nam zależeć by preprocesor zapomniał o poleceniu wydanym mu dyrektywą `define`. W tym celu wystarczy użyć dyrektywy^{†)}

```
#undef wyraz
```

Począwszy od tego miejsca w programie, przetwarzanie dalszych linii będzie się odbywało tak, jakbyśmy poprzedniej dyrektywy

```
#define wyraz .....
```

nie wydawali. Uczciwie muszę przyznać, że jeszcze nigdy tej dyrektywy nie używałem.

6.4 Makrodefinicje

Podobnie jak w języku C, tak i tu w C++, dyrektywa `define` może służyć do tworzenia makrodefinicji.

Rozważmy taki przypadek:

```
#define KWADR(a) ((a) * (a))
```

Jak to działa? Otóż przed przystąpieniem do właściwej kompilacji – preprocesor (czyli straż przednia kompilatora) – zamienia w tekście programu wszelkie wystąpienia wyrażenia

```
KWADR(parametr)
```

na wyrażenie

```
((parametr) * (parametr))
```

Po takiej zamianie przystępuje się do właściwej kompilacji. Inaczej mówiąc następujące linijki programu

```
a = KWADR(c) + KWADR(x) ;
cout << KWADR(m+5.4) ;
```

zamienia się automatycznie na linijki

```
a = ((c) * (c)) + ((x) * (x)) ;
cout << ((m + 5.4) * (m + 5.4)) ;
```

Do czego może służyć taka makrodefinicja? Na przykład do tego, by zamiast stosować w linii skomplikowany zapis – uprościć go sobie.

^{†)} To skrót od ang. `undefine` (czytaj: „andef”)

W makrodefinicji może być też więcej parametrów

```
#define OBJECT(a,b,c) ( (a) * (b) * (c) )
```

Przypominam, że nie może być spacji ani – ogólniej – białych znaków w wyrażeniu (słowie) `OBJECT(a,b,c)`. Biały znak bowiem kończy określenie makrodefinicji, a zaczyna określenie tego, czym ma ona być zastąpiona. Czyli jakby otwiera jakby ciało tej „funkcji”. (Naprawdę nazywamy to **rozwinieciem makrodefinicji**).

Inline contra makrodefinicja

Paragraf ten piszę między innymi dlatego, by obrzydzić Ci ochotę do używania makrodefinicji. Myślę, że nie przyjdzie mi to trudno.

Na pewno, drogi Czytelniku, pomyślałeś już o funkcjach typu `inline`, o których rozmawialiśmy w poprzednim rozdziale. Zawołasz pewnie: „To przecież to samo!” Nie, nie to samo. Prawie to samo, ale są różnice. To z powodu tych różnic szykuje czarną propagandę.



Najważniejsza różnica to to, że makrodefinicja jest jakby tępym narzędziem, **mechanicznym zamienianiem jednego stringu na drugi**. Nie ma tu sprawdzania typów parametrów (jakby argumentów), nie ma sprawdzania zakresu ważności użytych nazw. Kompilator nie może nas ostrzec czy popełniliśmy jakiś błąd. Dlatego makrodefinicji nie radzę używać. Lepiej zastosować funkcje typu `inline`, która nam to wszystko zagwarantuje.



Drugi powód to **nieoczekiwane efekty uboczne**.

Rozważmy taki przypadek. Mamy naszą makrodefinicję

```
#define KWADR(a) ( (a) * (a) )
```

i zastosujemy ją w takim wyrażeniu

```
int x = 4, p ;
p = KWADR(x++) ;
cout << "p = " << p << ", x teraz = " << x ;
```

W rezultacie wykonania tego fragmentu otrzymamy na ekranie

```
p = 20, x teraz = 6
```

Niezauważenie dla nas `x` zostało inkrementowane dwukrotnie. Wyniku spodziewaliśmy się także innego – przecież $(4 * 4) = 16$. Zatem dlaczego? To proste. Łatwo to zrozumieć, gdy rozpiszemy sobie wyrażenie, w którym nastąpiła makrodefinicja. Wygląda ono wtedy tak:

```
p = ( (x++) * (x++) ) ;
```

Jak widać inkrementacja została wykonana dwukrotnie mimo, że zamierzaliśmy zrobić to jednokrotnie. Chciałbym zawołać teraz triumfalnie: „–A nie mówiłem?! Nie używajmy makrodefinicji wcale!“. Tak jednak nie zawołam, gdyż są

Sytuacje kiedy makrodefinicja przydaje się^{†)}

Zapytasz: „-Mimo, że nie sprawdza nam typu argumentów?” Nie tylko *mimo*, ale wręcz właśnie dlatego. Są sytuacje, gdy chcemy oszukać kompilator. Na przykład wtedy, gdy nie chcemy, by kompilator sprawdzał nam typ argumentów. Klasycznym przykładem jest tu makrodefinicja

```
#define MAX(a,b)    ( ((a) > (b)) ? (a) : (b) )
```

możemy z niej korzystać niezależnie czy porównujemy ze sobą dwie liczby czy dwa adresy, czy też znaki. Typ argumentów nie jest bowiem sprawdzany. Gdybyśmy jednak chcieli napisać to samo jako funkcję typu `inline`, to należałoby dokładnie określić typ argumentów. Wymagałoby to zapewne napisania kilku wersji takiej funkcji, zależnie od typu porównywanych argumentów.

Jeśli więc zdecydujemy się na posługiwanie się tą makrodefinicją, pamiętajmy, że tak samo jak poprzednia, może być ona źródłem wspomnianych efektów ubocznych.

Nawiasy

O jeszcze jednej rzeczy muszę wspomnieć: Czy zauważyłeś jak gęsto rozwinięcia makrodefinicji zaopatrywałem w nawiasy? Do tego stopnia, że wręcz trudno to było czytelne. Nie bez powodu. Weźmy taką makrodefinicję

```
#define WYR(a,b,c)  a * b + c           // ryzykowne !
```

Jeśli użyjemy tej makrodefinicji w taki sposób:

```
y = WYR(2, 1 + 6.5, 0) * 1000 ;
```

to w efekcie działania preprocesora linijka ta zamieni się na taką:

```
y = 2 * 1 + 6.5 + 0 * 1000 ;
```

Czyli zamiast obliczyć

```
(2 * (1+6.5) + 0) * 1000
```

obliczymy

```
(2 * 1) + 6.5 + (0 * 1000)
```

Aby się przed tym ustrzec, należy w naszej definicji zastosować nawiasy. Poprawnie powinna wyglądać tak:

```
#define WYR(a,b,c)  ( (a) * (b) + (c) )
```

^{†)} W najnowszych wersjach języka C++ nawet ta sytuacja odpada. Zamiast makrodefinicją lepiej posłużyć się narzędziem zwanym *szablonem funkcji*. O szablonach napisałem książkę p.t. „Pasja C++”. Zajrzyj do niej po przeczytaniu „Symfonii”.

6.5 Dyrektywy kompilacji warunkowej

Zdarza się, że chcielibyśmy, by pewne linijki programu pojawiały się w nim tylko wtedy, gdy tego zażądamy. Na przykład na etapie uruchamiania programu przydają się linijki wypisujące na ekranie wyniki pośrednie.

```
x = i * czynnik[r] + szereg(x0);
cout << "teraz x = " << x << endl ;    // pomocniczy wydruk
m = funkcja(x) ;
```

Potem jednak, gdy program działa poprawnie wydruki takie nie są już potrzebne. Teoretycznie można by więc je usunąć, no ale kto wie, czy kiedyś jeszcze przy robieniu jakiś modyfikacji nie przydadzą się.

Wyjściem jest oczywiście chwilowe ujęcie ich w znaki komentarza. Wyjście to nie jest dobre. Jeśli bowiem w programie mamy dużo takich wydruków kontrolnych rozsianych po różnych miejscach, to dużo się napracujemy ujmując je w komentarze.

Drugi powód jest jeszcze ważniejszy – jak pamiętasz komentarzy typu `/*...*/` nie można zagnieżdżać. Jeśli chcemy instrukcję, (albo kilka instrukcji) ująć w taki komentarz, a komentarz typu `/*...*/` już w tym fragmencie jakoś jest używany, to jesteśmy bezsilni. Kompilator, który nie pozwala na zagnieżdżanie komentarzy – próbę taką uzna za błąd.

Jest inny sposób. Wyjście to nazywa się *kompilacja warunkowa*. Polega to na tym, iż, w zależności od spełnienia pewnych warunków, określone linie programu są kompilowane lub nie. Realizujemy to za pomocą dyrektyw preprocesora. To on przygotowuje kompilatorowi materiał i to on określone linie programu może odrzucić z procesu kompilacji. Kompilacja odbędzie się tak, jakby te linijki nigdy w programie nie istniały.

Jak się to robi? Bardzo prosto. Otóż obszar kompilacji warunkowej ograniczamy linijkami będącymi odpowiednimi dyrektywami preprocesora.

Dyrektywa `#if`

```
#if warunek
// linie kompilowane warunkowo
#endif
```

Jak widać przypomina to w pewnym sensie znaną nam instrukcję `if`. *Warunek* jest stałe wyrażenie. Rozumiem to jako wyrażenie, w którym każdy element jest stały, a jego wartość jest już znana w czasie, gdy preprocesor pracuje nad tą linijką. Innymi słowy nie może tam wystąpić żaden z obiektów (np. zmiennych) występujących w programie. *Warunek* jest wtedy spełniony, gdy wyrażenie warunkowe jako całość ma wartość różną od zera („prawda“). Oto przykłady:

```
#define RODZAJ 2
// ...
cout << "To jest kompilowane zawsze " << endl ;    // ❶
#if (RODZAJ == 1)
// ...
cout << "To jest kompilowane, gdy "
    "warunek jest spełniony " << endl ;    // ❷
```

```
#endif
cout <<"To znowu jest kompilowane zawsze "<<endl;    // ❸
```

Rzut oka wystarczy by stwierdzić, że warunek nie jest tu spełniony i linijka ❷ nie zostanie skompilowana. Po linijce ❶ nastąpi bezpośrednio linijka ❸.

Naszą dyrektywę

```
#endif
```

możemy też zapisać jako

```
#endif (RODZAJ == 1)
```

Postawiony tu warunek nic nie oznacza, a jednak bardzo się przydaje. W sytuacji, gdy parę `#if`ów zagnieżdżonych jest jeden w drugim, bardzo ułatwia to rozeznanie, w którym miejscu kończy się jaki obszar.

Słyszałem jednak, że bywają kompilatory, które (wbrew regułom) nie tolerują powtórzenia takiego warunku. Można sobie jednak w tym wypadku radzić stawiając go po znaku komentarza.

```
#endif // (RODZAJ == 1)
```

Inne dyrektywy dla kompilacji warunkowej

```
#if warunek
    // instrukcje 1
#else
    // instrukcje 2
#endif
```

Jest dokładnie tak, jak się domyślasz: zależnie od spełnienia warunku albo do kompilacji wchodzi instrukcje 1, albo instrukcje 2.

Najczęściej taką wielowariantową kompilację warunkową stosujemy, gdy nasz program ma mieć wiele wariantów. Trzon jest ten sam, ale niektóre części są wymieniane zależnie od bieżącej wersji.

Na przykład: piszemy program sterujący wysuwaniem podwozia w samolocie. W różnych typach samolotu jest różny typ podwozia, jednak wszędzie zasada jest podobna, więc tylko niektóre funkcje trzeba wymienić na inne. Oczywiście można skopiować stary program i zmodyfikować robiąc osobny program dla jednego typu, osobny dla drugiego. Nie zawsze się to jednak opłaca. Jeśli bowiem pracując nad jedną wersją tego programu wpadniemy na jakiś dobry pomysł dotyczący tej części, która jest identyczna dla obu programów, to zmianę trzeba będzie wprowadzać dwukrotnie – raz w starym, a raz w nowym programie. Moja praktyka mówi mi, że takich (niekoniecznie od razu genialnych) przeróbek jest zwykle wiele. Opłaca się korzystać z dobrodziejstw kompilacji warunkowej.

Oto taki fragment:

```
// najpierw definiujemy sobie dla wygody takie symbole
#define PODWOZIE_707    1
#define PODWOZIE_747    2
#define PODWOZIE_DC11   3
#define PODWOZIE_LIL     4
```

```
// tutaj definiujemy z którym typem mamy do czynienia
// w tym konkretnym wypadku
#define TYP_PODWOZIA PODWOZIE_747
//-----
int wystaw_kola(){
    #if (TYP_PODWOZIA == PODWOZIE_707)
        cout << "Tak jest kapitanie wystawiam 707\n" ;
    #elif (TYP_PODWOZIA == PODWOZIE_747)
        cout << "Tak jest kapitanie wystawiam 747\n" ;
    #elif (TYP_PODWOZIA == PODWOZIE_DC11)
        cout << "Tak jest kapitanie wystawiam DC11\n" ;
    #elif (TYP_PODWOZIA == PODWOZIE_LIL)
        cout << "Tak jest kapitanie wystawiam LIL\n" ;
    #else
        cout << "To nigdy sie nie powinno zdarzyc \n" ;
        #error "zle zdefiniowana wersja programu !!!!!"
    #endif TYP_PODWOZIA
    return 1 ;
}
```

Ponieważ widać, że wersja programu zdefiniowana jest na typ podwozia `PODWOZIE_747`, zatem powyższy fragment zostanie kompilatorowi przedstawiony jako

```
int wystaw_kola()
{
    cout << "Tak jest kapitanie wystawiam 747\n" ;
    return 1 ;
}
```

W przykładzie tym zobaczyłeś nową dyrektywę

```
#elif
```

co jest jakby skrótem od `else if` i jest konstrukcją analogiczną do takiej właśnie kombinacji instrukcji.

Wybacz mi tę dyrektywę `#error`, o której do tej pory jeszcze nie wspomniałem. Zdecydowałem się jednak wstawić ją tutaj na wypadek, gdybyś kiedyś do tego miejsca książki sięgnął szukając wzorca na zastosowanie kompilacji warunkowej różnych wersji swojego programu.

A swoją drogą to założę się, że domyślasz się jak działa dyrektywa `#error`. Jeśli nie, to i tak pomówimy o tym niebawem.

Warunek `#ifdef`, `#ifndef`

Dyrektywa kompilacji warunkowej

```
#ifdef nazwa
    // ... instrukcje
#endif
```

// rozumieć jako: if defined

nie sprawdza warunku, tylko sprawdza czy dana nazwa została zdefiniowana. To znaczy czy wystąpiła dyrektywa


```
#define nazwa ...
```

i nie unieważniliśmy jej dyrektywą

```
#undef nazwa
```

Jeśli tak, czyli nazwa jest preprocesorowi znana, to działanie jest takie, jakby chodziło tu o warunek i był on spełniony.

Analogicznie dyrektywa

```
#ifndef nazwa2
```

// rozumieć jako: if NOT defined

```
    // .... instrukcje
```

```
#endif
```

sprawdza czy nazwa została zdefiniowana i obowiązuje. Jeśli *nie* obowiązuje to tak, jakby w kompilacji warunkowej – warunek był spełniony.

Dyrektywy kompilacji warunkowej mogą być zagnieżdżane. Oto przykład:

```
#if (WERSJA == 1)
    #if (SZYBKOSC == 1)
        // instrukcje (1)
    #else
        // instrukcje (2)
    #endif
#else
    #if (SZYBKOSC == 1)
        // instrukcje (3)
    #else
        // instrukcje (4)
    #endif
#endif WERSJA
```

Jeśli w trakcie kompilowania tego fragmentu będą już w mocy następujące dyrektywy

```
#define WERSJA 7
#define SZYBKOSC 1
```

to w rezultacie w skład naszego skompilowanego programu wejdą *instrukcje (3)*.

6.6 Dyrektywa #error

Dyrektywa ta ma formę

```
#error string
```

a powoduje, że po napotkaniu jej kompilacja zostaje przerwana i wypisywany jest komunikat błędzie, którego częścią jest informacja umieszczona przez nas jako *string*

Oto kiedy się to przydaje:

```

#if (WERSJA == 1)
    // .....
#elif (WERSJA ==2)
    // .....
#else
    #error "Musi byc albo wersja 1 albo wersja 2!"
#endif

```

Jeśli zapomnieliśmy zdefiniować symbol WERSJA, albo jeśli zdefiniowaliśmy go tak, że ma on taką wartość, iż żaden ze sprawdzanych warunków nie jest spełniony, wówczas w preprocesor natknie się na naszą dyrektywę #error. Dyrektywa ta spowoduje przerwanie dalszej pracy i wypisanie w komunikacie o błędzie proponowanego tekstu:

```

Error directive: "Musi byc albo wersja 1 albo wersja 2!"
in function ...

```

6.7 Dyrektywa #line

Ma ona postać

```
#line stała "nazwa_pliku"
```

Proponowana nazwa_pliku nie jest obowiązkowa. Dyrektywa ta służy do oszukiwania kompilatora. Jeśli nakażemy mu

```
#line 128 "fikcja.c"
```

to od tej pory kompilator uzna, że jest to linijka 128 programu – mimo, że naprawdę jest to linijka numer 10 (na przykład). Dodatkowo kompilator będzie myślał, że kompiluje plik o nazwie "fikcja.c" Właściwą nazwę w tym momencie zapomina.

Dajmy przykład. Mamy plik TST.C z programem, o którym wiemy, że ma błędy w linijce 10-tej i 20-tej. (Wiemy to – bo na użytek tego przykładu specjalnie je tam popełniliśmy). Gdy kompilujemy taki program, to otrzymujemy informacje mniej więcej takiej treści:

```
Plik TST.C ma błędy w linijce 10 i 20
```

Następuje teraz bliższe opisanie tych błędów. Robimy jednak taki eksperyment: Na początku programu dopisujemy jeszcze jedną linijką takiej treści

```
#line 500 "fikcja.c"
```

Dopisanie linijki spowoduje oczywiście, że wszystkie dalsze przesuną się o jedną linię w dół. Błędne są więc teraz linie 11 i 21. Spróbujemy teraz skompilować ten sam program. Otrzymamy znowu komunikat o błędzie, ale tym razem takiej mniej więcej treści:

```
Plik FIKCJA.C ma błędy w linijce 511 i 521
```

6.8 Wstawianie treści innych plików w tekst kompilowanego własnie pliku

Dyrektywy preprocesora `#include` ^{†)}

```
#include <nazwa_pliku_A>
#include "nazwa_pliku_B"
```

powodują, że w tekst kompilowanego własnie programu, w miejsce, w którym znajdują się takie dyrektywy, zostaje wstawiona treść innego pliku o wyszczególnionej nazwie. Zupełnie tak, jakbyśmy w tym miejscu w programie, będąc jeszcze w edytorze, sprowadzili w to miejsce treść rzeczonego pliku.

Dlaczego tak więc po prostu nie zrobić ?

- ❖ Pierwsza odpowiedź brzmi: z lenistwa.
- ❖ Po drugie: jeśli sprowadzaniem plikiem jest plik nagłówkowy przy ewentualnych zmianach jakiejś deklaracji – wystarczy korekta w jednym tylko pliku.
- ❖ Po trzecie: – gdybyśmy chcieli do naszego programu włączyć wszystkie pliki nagłówkowe z deklaracjami funkcji bibliotecznych, to nasze pliki programowe rozrastałyby się ogromnie. Lepiej więc plik nagłówkowy wypożyczyć na chwilę, na samą okoliczność kompilacji.

A teraz o różnicy między przedstawionymi formami tej dyrektywy

Jeśli przy nazwie pliku użyliśmy cudzysłowu, to plik, który nakazujemy włączyć będzie najpierw poszukiwany w bieżącym katalogu, a jeśli tam nie zostanie znaleziony, to będzie szukany tak, jakby zamiast znaków cudzysłowu użyte były tam znaki `< >`.

Jeśli zaś przy nazwie plików są znaki `< >` to plik będzie poszukiwany w standardowym miejscu, gdzie znajdują się pliki zwykle włączane (np. biblioteczne). Miejsca poszukiwania plików włączanych tą dyrektywą są jednak zależne od implementacji, więc należy się zawsze upewnić jak w takim wypadku postępuje nasz kompilator.

Reguła jednak jest przeważnie taka :

używając cudzysłowu włącza się pliki, które sami piszemy, natomiast znaki `< >` stosuje się włączając np. pliki nagłówkowe bibliotek. (Są one zwykle zgromadzone w jakimś specjalnym katalogu).

Dyrektywy `#include` mogą się zagnieżdżać. To znaczy, że gdy tą dyrektywą włączamy jakiś inny plik, to w nim może być dyrektywa `#include` włączająca jakiś jeszcze inny plik. Poziomów zagnieżdżenia może być wiele.

Jak zagwarantować sobie jednokrotne włączanie danego pliku ?

Może się zdarzyć, że dyrektywą `#include` włączamy do programu pliki A, B, C, X. Tymczasem w pliku B jest – o czym nie wiemy lub nie pamiętamy –

†) include – ang. = wstawiać, włączać, wcielać. (Czytaj: „inklud”)

dyrektywa `#include` włączająca plik X. W związku z tym do kompilacji programu użyte zostaną wchodzące teraz w skład naszego programu pliki: A, B, X, C, X. Może to spowodować problemy, gdy w pliku X są fragmenty, które nigdy nie powinny pojawiać się w kompilacji dwukrotnie. (Np. definicje zmiennych lub definicje funkcji).

Jak się przed tym ustrzec? Jest prosty sposób: Użycie kompilacji warunkowej. Plik X powinien wyglądać na przykład tak:

```
#ifndef PLIK_X
#define PLIK_X

    // zwykła treść pliku

#endif
```

Jeśli ten plik zostanie włączony do kompilacji choć raz – zdefiniowana zostanie nazwa `PLIK_X`. Ewentualna powtórna próba włączenia tego pliku odbędzie się, gdy ta nazwa już jest zdefiniowana – czyli na mocy kompilacji warunkowej (dyrektywa `#ifndef`) nic z tego pliku po raz drugi kompilowane nie będzie.

6.9 Sklejacz czyli operator

To bardzo ciekawy operator. Jego działanie jest takie, że dokleja on jeden z argumentów (parametrów) makrodefinicji do stojącego po jego lewej stronie słowa.

Najlepiej zobaczyć to na przykładzie:

```
#define ST(rodzaj)      statecznik ## rodzaj

int ST(poziomy) ;
int ST(pionowy) ;
```

w rezultacie działania sklejacza taki fragment zostanie przetłumaczony przez preprocesor jako

```
int statecznikpoziomy ;
int statecznikpionowy ;
```

Zauważ, że spacje po jednej i drugiej stronie znaków `##` zostały usunięte i nastąpiło rzeczywiście doklejenie do słowa `statecznik`, w rezultacie czego powstał jeden wyraz.

```
#define BOE(typ,co) boeing_ ## typ ## _ ## co ## _cena
```

w wypadku zapisu

```
cout << BOE(747, skrzydlo) ;
```

Jest on zastąpiony przez

```
cout << boeing_747_skrzydlo_cena ;
```

Jak widać dzięki temu operatorowi zaoszczędzić można pisania.

6.10 Dyrektywa pusta

Jest to dyrektywa składająca się z samego znaku

#

Taka dyrektywa nie ma żadnego działania. Jest przez preprocesor po prostu ignorowana.

6.11 Dyrektywy zależne od implementacji

Dyrektywy takie zaczynają się od słowa `pragma`, po którym następuje komenda charakterystyczna dla danego konkretnego kompilatora (preprocesora)

#`pragma` *komenda*

Dzięki temu wprowadzona zostaje możliwość używania dyrektyw właściwych dla danego kompilatora, zatem szczegółowego opisu tych dyrektyw trzeba szukać w opisie kompilatora, którym się posługujemy.

Jeśli *komenda* jest danemu konkretnemu typowi kompilatora nieznana, napotkawszy ją w programie – ignoruje ją.

6.12 Nazwy predefiniowane

W trakcie pracy preprocesora oprócz nazw, które sami zdefiniowaliśmy, są jeszcze nazwy^{†)}, które definiuje dla siebie sam preprocesor. Oto one:

__LINE__

- ta nazwa kryje w sobie numer linijki pliku, nad którą preprocesor właśnie pracuje. Łatwo się domyślić, że jeśli kompilator wykrywa błąd i wypisuje komunikat o błędzie w linijce nr... to posługuje się właśnie tą nazwą.

__NAME__

- pod tą nazwą zapamiętana jest nazwa właśnie kompilowanego pliku

__DATE__

- pod tą nazwą zdefiniowany jest ciąg znaków odpowiadający dacie w momencie kompilacji. Mogą być dwie formy

†) Uwaga, w nazwach występują stojące obok siebie dwa znaki podkreślenia, które tu, w druku książki wyglądają niestety jak jedna długa kreska. Naprawdę jest więc __ a nie _

"Mar 19 1995"

"Mar 3 1995"

zależnie od tego, czy numer dnia jest jedno- czy dwucyfrowy.

__TIME__

- ta nazwa kryje w sobie aktualny czas w momencie translacji. Ma on postać ciągu znaków (stringu) "hh:mm:ss". Na przykład

"15:45:08"

O istnieniu tych predefiniowanych nazw można się przekonać kompilując plik, w którym znajdują się następujące linijki.

```
cout << "Kompilacja tego pliku " << __FILE__ ;
cout << "\n (linijka " << __LINE__
<< ") \n zaczęła się : " << __DATE__
<< " o godzinie : " << __TIME__ << endl;
```

Po uruchomieniu takiego fragmentu programu na ekranie zostanie wypisany tekst

```
Kompilacja tego pliku T.C
(linijka 10)
zaczęła się : Jun 04 1992 o godzinie : 00:45:50
```

Na wszelki wypadek podkreślę jeden fakt: Jeśli za 10 minut uruchomisz jeszcze raz ten sam program, to informacja o czasie będzie identyczna. __TIME__ i __DATE__ zawierają bowiem informacje, które dotyczą momentu czasowego samej kompilacji. W kompilacji zostają one tam zamrożone na zawsze.

Jeśli zaś chodzi Ci o wypisanie na ekranie bieżącej daty, godziny i minuty, to realizuje się to za pomocą standardowych funkcji bibliotecznych, takich jak time(), localtime() itd. Ich deklaracje zebrane są w pliku nagłówkowym time.h

Dodatkowo zdefiniowana jest jeszcze nazwa

__cplusplus

Jeśli kompilujesz kompilatorem C++ to zwykle może on na nasze życzenie zachowywać się tak, jakby był kompilatorem klasycznego C. Wówczas ta nazwa nie jest zdefiniowana.

Czasem się to przydaje. Najczęściej używałem tej możliwości w okresie przejściowym, kiedy nieśmiało przerabiałem moje programy z C na C++. Nie byłem wówczas pewny i chciałem zawsze mieć możliwość wycofania się. Dlatego używałem wówczas kompilacji warunkowej, gdzie warunkiem było zdefiniowanie lub niezdefiniowanie tej nazwy.

Do czego mogą się przydać takie predefiniowane nazwy

Nie przydają się często, ale rozważmy taki przypadek

Program nasz składa się z wielu plików, nad którymi pracują różni programiści dokonując ciągłych ulepszeń. Dajmy na to, że w skład programu wchodzi moduł obsługujący radar. Dostajemy go od kolegi już w postaci binarnej, gotowej do

zlinkowania z naszymi modułami. Może się czasem okazać przydatne, by w gotowym programie wiedzieć z którą wersją części „radarowej” mamy do czynienia.

Jak to zrobić? Na przykład wymagając od kolegi by umieścił w jego pliku funkcję `wersja_radaru()`, która na ekranie wypisze tekst informujący o numerze wersji. Przy okazji modyfikacji swojego modułu nasz kolega powinien zawsze zmienić treść tego tekstu – zatem na ekranie pojawiał się będzie nowy opis wersji. Niestety nasz kolega jest niechluj i mimo dokonania modyfikacji programu – często nie uaktualnia tekstu informującego o wersji.

Co robić? Jest wyjście. Wersję poznać możemy np. po nazwie pliku, w którym dane funkcje („radarowe”) są umieszczone, a także po momencie kompilacji tego pliku.

Wystarczy, by kolega piszący ten moduł, umieścił w nim funkcję

```
void wersja()
{
    cout << cessa: " << __FILE__ << " "
        << __DATE__ << " " << __TIME__ << endl ;
}
```

Dzięki temu ile razy tę funkcję wywołamy, na ekranie pojawi się

```
cessa: crs71.c MAY 10 1992 15:03:47
```

Ważne, że od tej pory ta informacja nie jest zależna od dobrej woli kolegi piszącego ten plik. Bez względu na jego niechlujstwo zawsze mamy ślad w postaci prawdziwej nazwy jego pliku i tego, kiedy on ten plik kompilował.



Jeśli masz do czynienia z grupą zmiennych (ogólniej mówiąc - obiektów), to możesz z nich zrobić tablicę. Tablica to ciąg obiektów tego samego typu, które zajmują ciągły obszar w pamięci.

Korzyść z tego jest taka, że zamiast nazywania każdej ze zmiennych osobno, wystarczy powiedzieć: odnoszę się do n-tego elementu tablicy.

Tablice są typem pochodnym. Znaczy to po prostu, że bierze się jakiś typ – dajmy na to `int`, i z elementów takiego typu buduje się tablicę. Jest to wtedy tablica elementów typu `int`.

Jeśli chcemy mieć 20 zmiennych typu `int`, to można z nich zbudować tablicę

```
int a[20] ;
```

Ta definicja rezerwuje w pamięci miejsce dla 20 liczb typu `int`.

Rozmiar tak definiowanej tablicy musi być stałą, znaną już w trakcie kompilacji. Kompilator bowiem musi już wtedy wiedzieć ile miejsca ma zarezerwować na daną tablicę.

Rozmiar ten nie może być więc na przykład ustalony dopiero w trakcie pracy programu.

```
cout << "Jaki chcesz rozmiar tablicy ? " ;  
int rrr ;  
cin >> rrr ;  
  
int a[rrr] ; // Błąd !!!
```

Jest to błąd, bo wartość `rrr` nie jest jeszcze w czasie kompilacji znana. Znana jest dopiero w trakcie pracy programu.

(Jeśli rzeczywiście zachodzi konieczność takiej deklaracji – powinienes pomyśleć o tzw. dynamicznej alokacji tablicy ~ str. 186)

A oto inne przykłady – będą to definicje różnych tablic. Definicje - czyli miejsca ich narodzin w programie. Definicja jest, jak wiadomo, także deklaracją, czyli

poinformowaniem kompilatora o typie danego obiektu. Stąd dla poniższych definicji podaję w komentarzu jak czyta się deklarację takiej tablicy:

```
char zdanie[80] ;           // zdanie jest tablicą
                             // 80 elementów typu char

float numer[9] ;           // numer jest tablicą
                             // 9 elementów typu float

unsigned long kanal[8192] ; // kanal jest tablicą 8192
                             // elementów typu unsigned long

int *wskaz[20] ;           // wskaz jest tablicą 20 elementów
                             // będących wskaźnikami (adresami)
                             // jakichś obiektów typu int
```

O adresach jeszcze dokładniej nie mówiliśmy, ale ten ostatni przykład przytoczyłem po to, by pokazać z jak różnych typów można zbudować tablice.

Tablice można tworzyć z:

- typów fundamentalnych (z wyjątkiem `void`),
- typów wyliczeniowych (`enum`),
- wskaźników,
- innych tablic;

a także (o czym dowiesz się w przyszłości):

- z obiektów typu zdefiniowanego przez użytkownika (czyli klasy),
- ze wskaźników do pokazywania na składniki klasy.

7.1 Elementy tablicy

Na początku zajmiemy się prostymi tablicami tworzonymi z typów fundamentalnych.

Jeśli zdefiniujemy sobie taką tablicę:

```
int t[4] ;
```

to jest to tablica czterech elementów typu `int`. Poszczególne elementy tej tablicy to:

```
t[0]      t[1]      t[2]      t[3]
```



Jak widzisz

Numeracja elementów tablicy zaczyna się od zera. Jest to bardzo ważne i to trzeba zapamiętać.

W początkowym okresie będziesz się zapewne często mylił. Tym bardziej, że w niektórych językach programowania numeracja elementów tablicy zaczyna się od 1. Jeśli masz takie przyzwyczajenie, to będzie Ci trudniej.

Prawdę mówiąc nie ma specjalnej tragedii jeśli zapomnisz coś wpisać do elementu zerowego. Twoja strata. Gorzej, jeśli zapomnisz, że ostatnim, czwartym elementem naszej tablicy jest element `t[3]`, a nie element `t[4]`. Numerujemy przecież od zera! Element `t[4]` nie istnieje. Próba wpisania czegoś do elementu `t[4]` nie będzie sygnalizowana jako błąd, gdyż w językach C oraz C++ nie jest to sprawdzane.^{†)}

Wpisanie czegoś do nieistniejącego elementu `t[4]` spowoduje zniszczenie w obszarze pamięci czegoś, co następuje bezpośrednio za tablicą. Co dokładnie jest niszczone – zależy to od implementacji. Dla zobrazowania powiem tylko, że zdarza się, iż przy takich definicjach:

```
int t[4] ;
int z ;
```

Podczas próby zapisu czegoś do nieistniejącego elementu `t[4]`

```
t[4] = 15 ;
```

zniszczona zostanie treść zmiennej `z`, bo akurat została umieszczona w pamięci bezpośrednio za tablicą `t`. Ponieważ typ zmiennej `z` zgadza się akurat z typem elementów tablicy `t`, dlatego możliwe jest też, że w zmiennej `z` znajdzie się liczba 15.

Powtarzam jednak: to, co powiedziałem, zależy od implementacji. Ważny jest tutaj tylko pewnik: coś mimowolnie w pamięci zniszczyliśmy.

Należy zatem pamiętać, że tablica `n`-elementowa ma elementy o indeksach od 0 do `n-1` (a nie do `n!`).

Tablicę można zappełnić treścią - na przykład za pomocą zwykłej operacji przypisania.

```
#include <iostream.h>
main()
{
    int t[4] ;
    for( int i = 0 ; i < 4 ; i++)
        t[i] = 100 * i ;
    cout << "Wydruk tresci tablicy : \n" ;
    for(i = 0 ; i < 4 ; i++)
    {
        cout << "Element nr : " << i
            << " ma wartosc " << t[i] << endl ;
    }
}
```

†) Nie traktuj tego jako wady. Jeśli Ci to nie odpowiada, to w przyszłości będziesz potrafił to zmienić – definiując nowy rodzaj tablic – takich, które to sprawdzają. Ale zapłacisz za to czasem dostępu do elementu takiej tablicy. Tablica, która tego nie sprawdza – jest szybsza.



W rezultacie wykonania tego programu na ekranie pojawi się

```

Wydruk tresci tablicy :
Element nr : 0 ma wartosc 0
Element nr : 1 ma wartosc 100
Element nr : 2 ma wartosc 200
Element nr : 3 ma wartosc 300

```

Zauważ, iż zakres obu pętli `for` jest taki, że `i` przebiega od 0 do 3. To właśnie z powodów, o których wspominaliśmy.

Jeśli jesteś ciekaw, to spróbuj w drugiej pętli `for` – tej od wypisywania elementów na ekran – zmienić `i < 4` na `i <= 4`. Spowoduje to wypisanie na ekran piątego, nieistniejącego elementu o indeksie `t[4]`. Pobranie wartości z tego obszaru pamięci bezpośrednio za tablicą nie spowoduje katastrofy, jedynie wartość będzie bezsensowna. Tragedia byłaby dopiero wtedy, gdybyśmy chcieli coś w to miejsce wpisać.

Uwaga praktyczna:

Tu chciałbym Ci coś poradzić. Zwykle elementy tablicy czyta się lub wypisuje za pomocą pętli, na przykład pętli `for`. W naszym przypadku zapisaliśmy to tak:

```
for(i = 0 ; i < 4 ; i++) ... itd
```

czyli inaczej

```
for(i = 0 ; i < rozmiar ; i++) ...
```

Mogliśmy równie dobrze napisać tak:

```
for(i = 0 ; i <= 3 ; i++) ...
```

czyli inaczej

```
for(i = 0 ; i <= rozmiar-1 ; i++) ...
```

Podstawowa różnica jest tu, jak widać, w użyciu mocnej lub słabej nierówności. Jednak oba zapisy sprawiają, że pracujemy na elementach o indeksach 0-3, czyli oba zapisy są równoważne.

Otóż radzę Ci byś zdecydował się na jeden typ zapisu i nigdy nie używał drugiego. Unikniesz wtedy omyłkowego adresowania elementu `t[rozmiar]` (który, jak wiemy, nie istnieje).

Osobiście preferuję ten pierwszy zapis (z silną nierównością), bo nie muszę odejmować jedynki od rozmiaru, a poza tym zapis jest krótszy.

7.2 Inicjalizacja tablic

Innym sposobem nadania wartości elementom tablicy jest inicjalizacja – nadanie wartości początkowych w momencie definicji tablicy (czyli w momencie jej narodzin).

Pamiętasz zapewne jak robiliśmy inicjalizację w wypadku zwykłych typów

```
int liczba = 237 ;
float współczynnik = 0.372 ;
```

W przypadku tablicy trzeba nadać wartość początkową każdemu elementowi. Służy do tego tzw. inicjalizacja zbiorcza (ang. aggregate initialization).

W naszym wypadku wygląda to tak:

```
int t[4] = { 17, 5, 4, 200 } ;
```

Jest to wygodny sposób, bo w jednej linijce załatwia inicjalizację wszystkich czterech elementów. Dzięki temu

t[0]	ma wartość	17
t[1]	ma wartość	5
t[2]	ma wartość	4
t[3]	ma wartość	200

Do znużenia będę przypominał, że element t[4], podobnie jak i element t[107] – nie istnieje. Gdybyś jednak w tym momencie zbiorczej inicjalizacji na liście ujętej klamrami { } umieścił o jedną lub kilka liczb za dużo, to tutaj kompilator zasygnalizuje Ci błąd. W inicjalizacji sprawdza się czy rozmiar tablicy nie jest przypadkiem przekroczony. Tylko przy inicjalizacji. Potem nie. Możliwa jest też taka inicjalizacja naszej tablicy:

```
int t[4] = { 17, 5 } ;
```

Jak widać liczb jest za mało. Inicjalizacja taka spowoduje, że żądane wartości początkowe zostaną nadane tylko dwóm pierwszym elementom. Elementom t[0] i t[1]. **Pozostałe dwa elementy będą inicjalizowane zerami.**

Dla wygody istnieje też taki sposób definiowania i inicjalizacji tablic:

```
int r[] = { 2, 10, 15, 16, 3 } ;
```

Zauważ, że tutaj w kwadratowym nawiasie nie podaliśmy rozmiaru tablicy. Kompilator więc sam sobie przelicza ile to liczb podaliśmy w klamrze i w efekcie rezerwowana jest pamięć na te elementy. W naszym wypadku powstanie tablica pięcioelementowa.

7.3 Przekazywanie tablicy do funkcji

Żałujemy, że mamy tablicę z danymi pomiarowymi. Próbek pomiarowych jest dużo, bo aż 8192

```
long int widmo[8192] ;
```

Chcemy teraz napisać sobie funkcję, która treść każdego elementu tej tablicy pomnoży przez 3. Jak to pomnożyć, oczywiście wiesz – wystarczy napisać pętlę mnożącą przez 3 – po kolei wszystkie elementy tej tablicy od 0 do 8191. To jest proste. Jak jednak przesłać do funkcji tablicę?

Pamiętasz jak mówiliśmy o przesyłaniu argumentów do funkcji? Zwykle przesyła się przez wartość, czyli fotografowany jest każdy argument i jego zdjęcie

(kopia) przesyłana jest do funkcji. Tablicy jednak nie da się przesłać przez wartość. Można tak przesłać pojedyncze jej elementy, ale nie całość. To nie wynika z niedoskonałości języka. Po prostu - czy wyobrażasz sobie funkcję, która w wywołaniu dostaje 8192 argumenty? To tak, jakby fotografować 8192 razy. Samo pojedyncze wywołanie takiej funkcji trwałoby pewien znaczący czas, nie mówiąc już o trudnościach w realizacji takiego przesłania.

Zatem zasada jest taka, że:

tablice przesyła się podając funkcji tylko adres początku tej tablicy.

Jeśli mamy funkcję

```
void funkcja (float ttt[] );
```

która spodziewa się jako argumentu: tablicy liczb typu float, to taką funkcję wywołujemy na przykład tak:

```
float tablica[] = { 7, 8.1, 4, 4.12 };
```

```
funkcja(tablica);           // wywołanie funkcji
```

Przy nazwie tablicy w wywołaniu funkcji nie widzisz żadnych nawiasów kwadratowych.

Zapamiętaj sobie na zawsze (a najlepiej napisz sobie to na kartce i przyklep nad biurkiem), że:

W języku C++ (tak jak i w C):

**NAZWA TABLICY jest równocześnie
ADRESEM ZEROWEGO JEJ ELEMENTU**

Nie żartuję. Rzeczywiście wykuj to zdanie na pamięć, bo bardzo Ci to pomoże w swobodnym poruszaniu się po królestwie C++.

Jest jeszcze coś równie sympatycznego. Mianowicie wyrażenie

```
tablica + 3
```

jest adresem tego miejsca w pamięci, gdzie tkwi element o indeksie 3. Element o indeksie 3 to inaczej element

```
tablica[3]
```

adres takiego elementu to

```
&tablica[3]
```

Znak & (ampersand) jest jednoargumentowym operatorem oznaczającym uzyskiwanie adresu danego obiektu.

Uwaga: nie mylmy tego operatora z dwuargumentowym operatorem & oznaczającym bitowy iloczyn logiczny.

A zatem poniższe dwa zapisy (wyrażenia) są równoważne

```
tablica + 3           &tablica[3]
```

Można je stosować wymiennie, co kto lubi.

Wróćmy jednak do rzeczy. W naszym wywołaniu funkcji napisaliśmy samą nazwę tablicy (bez klamer) więc do funkcji przesyłamy adres. Oto przykład programu, w którym do funkcji jako jeden z argumentów wysyłana jest tablica.

```
#include <iostream.h>
void potrojenie(int ile, long tablica[]);           // ❶
/*****/
main()
{
    const int rozmiar = 8192 ;                     // ❷
    long widmo[rozmiar] ;                          // ❸

    // -----nadanie wartości początkowej
    for(int i = 0 ; i < rozmiar ; i ++ )
    {
        widmo[i] = i ;                             // ❹
        if(i < 6) // pokazanie pierwszych sześciu
            cout << "i= " << i << " ) " << widmo[i]
                << endl ;
    }
    // -----uwaga, wywołujemy funkcję !
    potrojenie(rozmiar, widmo) ;                   // ❺
    cout << "Po wywołaniu funkcji \n" ;
    for(i = 0 ; i < 4 ; i ++ )
    {                                               // ❻
        cout << "i= " << i << " ) " << widmo[i] << endl;
    }
}
/*****/
void potrojenie (int ile, long t[])              // ❼
{
    for(int i = 0 ; i < ile ; i++)
    {
        t[i] *= 3 ;                               // ❽
    }
}
/*****/
```

W wyniku wykonania tego programu na ekranie pojawi się

```
i= 0) 0
i= 1) 1
i= 2) 2
i= 3) 3
i= 4) 4
i= 5) 5
Po wywołaniu funkcji
i= 0) 0
i= 1) 3
i= 2) 6
i= 3) 9
```

- ❶ Deklaracja funkcji, do której jako argument wysyła się tablicę. Zauważ jak deklaruje się argument formalny. Jest to jakby definicja tablicy typu long

o nieznanym rozmiarze. W związku z tym funkcja ta będzie się nadawała do pracy na dowolnej tablicy typu `long`, której elementy zamierza się potroić. Wewnątrz tej funkcji potrzebna jest nam także liczba elementów tablicy, których wartość liczbowa ma ulec potrojeniu. Dlatego wysyłamy sobie tę wartość jako argument. Musimy to zrobić, gdyż na podstawie nazwy tablicy (czyli adresu) nie da się określić ile dana tablica ma elementów.

- ② Rozmiar tablicy wiódmo definiujemy sobie w programie jako obiekt typu `int` z przydomkiem `const` – rozmiaru tej tablicy nie będziemy przecież w trakcie programu zmieniać. Dlaczego słowo `const` jest tu konieczne – okaże się za chwilę.
- ③ Oto definicja tablicy. Jej rozmiar jest określony przez liczbę schowaną w powyższym obiekcie typu `int` z przydomkiem `const`. To `const` sprawia, że kompilator jest pewien stałości tej liczby już na etapie kompilacji. Jak pamiętamy rozmiar tak definiowanej tablicy musi być znany już na etapie kompilacji programu.
- ④ Po definicji tablicy następuje pętla nadająca jej elementom wartości początkowe równe kolejnym liczbom naturalnym. Pierwsze 6 elementów tablicy wypisujemy na ekran.
- ⑤ Wywołanie funkcji. Jako argumenty wysyłamy liczbę elementów, których wartość należy potroić, oraz nazwę tablicy, na której ta operacja potrojenia ma się odbyć. Przypominam, że jeśli wysyłamy do funkcji adres tablicy, to tak, jakbyśmy wysyłali adres zerowego jej elementu.
- ⑥ Po powrocie z funkcji wypisujemy wartości tych elementów na ekranie.

Co się dzieje wewnątrz funkcji:

- ⑦ Odbieramy tam adres początku tablicy i służy on do zbudowania wewnątrz funkcji takiego aparatu obsługi tablicy `t[]`, że odniesienie się do elementu `t[3]` jest dokładnie tym samym, co odniesienie się do elementu `tablica[3]`. Innymi słowy nie pracujemy tu na żadnej kopii tablicy, tylko na oryginale.

Masz rację jeśli myślisz, że mętnie to tłumaczę. Niestety na razie nie mogę powiedzieć całej prawdy. Wszystko stanie się jasne w następnym rozdziale, kiedy to porozmawiamy o wskaźnikach.

- ⑧ W naszej funkcji potrojenie przebiegamy po wszystkich elementach tablicy `t[]` z przedziału 0 - 8191 i mnożymy przez 3. Przypominam że instrukcja

```
tab[i] *= 3 ;
```

to inaczej to samo co

```
tab[i] = tab[i] * 3 ;
```

Do funkcji przysłaliśmy argument określający liczbę elementów tablicy, na których należy przeprowadzić operację potrojenia. Musieliśmy to zrobić dlatego, że jedyne co wewnątrz funkcji wiadomo na temat przysłanej tablicy to to, jaki jest adres jej początku i to, iż jest ona typu `long`. Nic więcej. W szczególności nie wiadomo jaki ma rozmiar ta tablica.



Wysyłanie tylko jednego elementu

Do funkcji, której argumentem formalnym jest typ `int`

```
void fff(int x);
```

można wysłać argument będący jakimś elementem tablicy typu `int`. Nie jest to przesłanie tablicy. Odbывается to identycznie, jak w wypadku zwykłej zmiennej `int`.

```
int m ;
int tabl[100] ;
```

```
    fff(m) ;
```

// wysłanie do funkcji zwykłej zmiennej

```
    fff(tabl[38]) ;
```

// wysłanie do funkcji elementu nr 38

Takie wysłanie jednego elementu odbywa się oczywiście przez wartość.

Łatwo to zapamiętać tak:

Otóż wartością wyrażenia `tabl[38]` jest **liczba** (zapisana w tym elemencie tablicy). Natomiast wartością wyrażenia `tabl` jest **adres** tej tablicy. To dlatego mechanizm przesłania jest inny.



To (prawie) wszystko, co trzeba wiedzieć na temat tablic. Myślę, że nic w tym trudnego, bo z tablicami zetknąłeś się w innych językach programowania. Właściwie więc można by ten rozdział tutaj skończyć, gdyby nie pewien bardzo pożyteczny rodzaj tablic: tablice znakowe.

7.4 Tablice znakowe

Specjalnym rodzajem tablic są tablice do przechowywania znaków (np. liter). Dlatego też tablicom tym przyjrzymy się bliżej.

```
char zdanie[80] ;
```

Ta definicja określa, że `zdanie` jest tablicą 80 elementów będących znakami. W tablicy tej można umieścić tekst, dzięki temu, że każdy z jej elementów nadaje się do przechowywania reprezentacji liczbowej znaków alfanumerycznych. Na przykład – znaków zakodowanych kodem ASCII. Kod taki, jak zapewne wiesz, jest jednym ze sposobów kodowania liter, które w komputerze muszą być przecież przechowywane w postaci liczb. Jest kilka sposobów kodowania liter, my jednak zawsze mówić będziemy o kodzie ASCII.

Teksty w tablicach znakowych zwykło się przechowywać tak, że po ciągu liter (a właściwie ich kodów liczbowych) następuje znak o kodzie 0. Ten znak zwany jest `NULL`. Znak ten jest po to, by oznaczyć gdzie kończy się ciąg liter. Na taki ciąg liter zakończony znakiem `NULL` mówimy *string*.

Naszą tablicę `zdanie` można już w trakcie definicji zainicjalizować

```
char zadanie[80] = { "lot" } ;
```

Zauważ, że odbywa się to przez napisanie tekstu ujętego w cudzysłów.

W rezultacie w poszczególnych komórkach tablicy zdanie znajdują się następujące znaki

l	o	t	NULL							
0	1	2	3	4	5	...	...	...	78	79

Jak widzisz wpisane zostały trzy litery, a za nimi znak o kodzie NULL kończący string. Dalsze elementy tablicy nas nie interesują. Warto jednak pamiętać, że – w myśl omówionych niedawno zasad inicjalizacji zbiorczej tablic – nie wymienione elementy inicjalizuje się do końca zerami.

Znak NULL został automatycznie dopisany po ostatniej literze t dzięki temu, że inicjalizowaliśmy tablicę ciągiem znaków ograniczonym cudzysłowem.

Jest to po naszej myśli, bo w językach C i C++ wszelkie funkcje biblioteczne pracujące na stringach opierają się na założeniu, że koniec stringu oznaczony jest znakiem NULL.

Na przykład:

Jeśli chcemy jakiś string wypisać na ekran, to wywołujemy standardową funkcję biblioteczną (puts – od: put string) i jako argument przekazujemy jej adres początku stringu. Funkcja ta będzie sukcesywnie wypisywała na ekran znaki zawarte w kolejnych komórkach pamięci począwszy od adresu podanego jako początek stringu. Akcja zakończy się dopiero po natknięciu się na komórkę ze znakiem NULL, czyli inaczej mówiąc z zapisanym tam bajtem 0.

Jest też inny sposób inicjalizacji tablicy znaków

```
char zdanie[80] = { 'l', 'o', 't' };
```

Zauważ, że w klamrze pojawiły się pojedyncze litery ograniczone apostrofami. Zapis taki jest nie tylko równoważny trzem instrukcjom:

```
zdanie[0] = 'l' ;
zdanie[1] = 'o' ;
zdanie[2] = 't' ;
```

Ponieważ nie było tu cudzysłowu, więc kompilator nie dokończył tego znakiem NULL umieszczanym poprzednio w elemencie `zdanie[3]`. Zatem sprawa wydaje się ryzykowna – po ciągu liter nie nastąpił znak kończący string.

A jednak przypomnij sobie co mówiłem o inicjalizacji zbiorczej – czyli takiej, gdzie wartości początkowe umieszczone są w klamrze `{ }`. Otóż jeśli wartości początkowych jest mniej niż elementów tablicy, to reszta jest inicjalizowana zerami. Czyli pozostałe elementy tablicy `zdanie`, aż do elementu o indeksie 79, zostaną inicjalizowane zerami. W szczególności w elemencie `zdanie[3]` też znajdzie się zero. A teraz uważaj: znak NULL ma przecież też wartość 0 – zatem nie musimy się martwić, string w tablicy `zdanie` jest poprawnie zakończony.

Pułapka

Nie zawsze jednak życie jest tak piękne. Oto chcieliśmy być sprytniejsi i napisaliśmy taką definicję:

```
char zdanie[] = { 'l', 'o', 't' };
```

Pamiętamy bowiem, że jeśli nie podamy rozmiaru tablicy, to kompilator przeliczy sobie liczbę inicjalizatorów i taką zarezerwuje tablicę. W naszym przykładzie doliczy się trzech elementów. Zostanie więc zarezerwowana tablica 3 elementowa. W poszczególnych jej elementach znajdują się znaki 'l' 'o' 't', ale znaku NULL tam nie będzie. Nie było cudzysłowu więc kompilator nie sądził, że te znaki mają się składać na string. Dopisania reszty zerami nie będzie, bo żadnej reszty nie ma. Tablica ma tylko trzy elementy. Podkreślam jednak – nie jest to błąd. Możliwe, że litery te nie mają być używane jak ciąg znaków, (czyli string), lecz po prostu są to luźne litery, do innych celów. Traktować jako string ich jednak nie można.

Zwróć uwagę na tę deklarację:

```
char zadanie[] = { "lot" } ;
```

Tutaj kompilator obliczając ile elementów ma zarezerwować na tablicę doliczy się trzech liter, ale z faktu, że są one ujęte w cudzysłów wywnioskuje, że mają one być używane jako string, więc zarezerwuje jeszcze jeden dodatkowy element na znak NULL.

Jeśli nie wierzysz to zobacz:

```
#include <iostream.h>
main()
{
    char napis1[] = { "Nocny lot" } ;
    char napis2[] = { 'N', 'o', 'c', 'n', 'y',
                      'l', 'o', 't' } ;

    cout << "rozmiar tablicy pierwszej : "
          << sizeof(napis1) << endl ;

    cout << "rozmiar tablicy drugiej : "
          << sizeof(napis2) << endl ;
}
```

W rezultacie wykonania tego programu na ekranie pojawi się

```
rozmiar tablicy pierwszej : 10
rozmiar tablicy drugiej : 9
```

Często używa się pojęcia **długość stringu** - jest to ilość liter należących do tego stringu. Pamiętajmy, że **rozmiar stringu** jest większy o 1, czyli o znak NULL.

A teraz pytanie kontrolne. Mamy takie dwie definicje tablic. Co o nich sądzisz – czy są poprawne?

```
char t1[1] = { "a" } ;
char t2[1] = { 'a' } ;
```

Nie, nie są. Pierwsza jest błędna. Tablica jest tu jednoelementowa, a chcemy wpisać do niej string złożony z litery 'a' i znaku NULL. Na to potrzeba dwóch elementów tablicy. Zatem błąd.

W drugiej definicji inicjalizacja polega na wpisaniu do tablicy tylko jednej litery, a więc jest poprawna.



Przyglądaliśmy się przed chwilą jak wpisuje się stringi do tablic. Niestety: w podany sposób wpisywać można do tablicy znakowej tekst tylko w czasie inicjalizacji. Potem, próba wpisania do niej czegoś tym sposobem

```
zdanie[80] = "nocny lot";           // Błąd !
zdanie = "nocny lot" ;              // także błąd !
```

jest niepoprawna.

Jak zatem wpisywać string do tablic już istniejących?

Sprawa jest prosta. Trzeba napisać sobie funkcję, która się tym zajmie i podany string, litera po literze umieści w danej tablicy.

Zadanie to jest tak częste, że w standardowej bibliotece jest kilka funkcji realizujących to w różnych wariantach.

Przyjrzyjmy się takiej funkcji. Jako argumenty otrzymuje ona dwie tablice – tę, w której jest string źródłowy i tę, gdzie ma on później zostać skopiowany.

O tym, że tablica może być argumentem funkcji – wiemy już z poprzedniego paragrafu. Naprawdę do funkcji przesyłany jest tylko adres początku tablicy, a nie wszystkie jej elementy (których mogły by być tysiące).

Jak ma wyglądać sam proces kopiowania? Jest to bardzo proste dzięki naszej zapobiegliwości. Kopiuje się znak po znaku elementy z tablicy źródłowej do tablicy docelowej tak długo, aż nie napotka się na znak NULL. Ten znak NULL, także należy przekopiować – jeśli wynik ma być poprawnym stringiem.

Spróbujmy zatem to napisać. Oczywiście posłużymy się pętlą.

```
void strcpy(char cel[], char zrodlo[])           // ❶
{
    for(int i = 0 ; ; i++)                       // ❷
    {
        cel[i] = zrodlo[i];                     // ❸
        if(cel[i] == NULL) break ;              // ❹
    }
}
```

- ❶ Jest to, jak czytamy, funkcja przyjmująca jako argumenty dwie tablice elementów typu char – czyli dwie tablice znakowe. Typem zwracanym jest typ void. Uczciwie przyznam, że mogłem wymyślić to lepiej, ale na razie niech będzie jak jest.
- ❷ Do przebiegnięcia po poszczególnych elementach tablicy służy nam pętla for. Pętla ta zaczyna się od $i = 0$, bo jak wiadomo numeracja elementów tablic zaczyna się od 0. Dalej w instrukcji for widzimy puste miejsce – bo nie wiemy ile elementów tablicy będziemy przepisywać. Puste miejsce oznacza tutaj, że pętla jest nieskończona.
- ❸ To jest właściwa akcja przepisania poszczególnego znaku z tablicy zrodlo do tablicy cel.

- ④ Sprawdzanie czy właśnie przepisany znak nie jest czasem znakiem NULL. Jeśli tak, to pętlę przerywa się instrukcją `break`. Jeśli nie, to wykonujemy następny obieg pętli. Instrukcją `i++` zwiększa się zmienna `i`, którą używamy do indeksowania elementów tablicy, po czym kopiujemy dalej.

Inne sposoby zrobienia tego samego

A oto jak tę samą funkcję można zrealizować używając pętli `do-while`:

```
void strcpy(char cel[], char zrodlo[])
{
    int i = 0 ;
    do{
        cel[i] = zrodlo[i];           // kopiowanie
    }while(cel[i++] != NULL) ;       // sprawdzenie i przesunięcie
}
```

W skrócie można treść tej instrukcji wypowiedzieć tak: Dotąd kopiuj elementy z tablicy do tablicy, dopóki (ang: `while`) – właśnie skopiowany element jest różny od NULL.

Przy okazji wykonuje się przesunięcia indeksu tablicy metodą postinkrementacji. Wyrażenie porównujące

```
cel[i] != NULL
```

jest jeszcze ze starą wartością `i`. Bezpośrednio po tym następuje inkrementacja czyli zwiększenie indeksu o 1 (**post**-inkrementacja).

Pamiętasz może, jak przy okazji omawiania operatora `=` mówiłem, że wartością wyrażenia przypisania (podstawienia) jest wartość będąca przedmiotem przypisania. Inaczej mówiąc wyrażenie

```
(i = 15)
```

nie tylko, że wykonuje przypisanie, ale jeszcze samo jako całość ma wartość 15. Podobnie wyrażenie

```
(cel[i] = zrodlo[i])
```

ma wartość równą kodowi ASCII kopiowanego właśnie znaku. Korzystając z tego możemy napisać naszą funkcję tak:

```
void strcpy(char cel[], char zrodlo[])
{
    int i = 0 ;
    while(cel[i] = zrodlo[i])
    {
        i++ ;
    }
}
```

Przeczytać to można tak: Jeśli wartość wyrażenia kopiowania

```
(cel[i] = zrodlo[i])
```

jest różna od zera (czyli NULL) to inkrementuj indeks `i`. Teraz widzisz dlaczego znak NULL kończący string ma wartość właśnie 0

Sprytne, prawda? Nie mówimy:

- musisz skoczyć po gazetę i potem rozwiąż w niej krzyżówkę, tylko
- jeśli gazeta, którą przyniosłeś ma jakiś tytuł inny niż NULL, to możesz rozwiązać w niej krzyżówkę.

Chcąc sprawdzić jaka jest wartość wyrażenia komputer musi to wyrażenie „obliczyć“, czyli w naszym wypadku wykonać kopiowanie znaku z tablicy `zrodlo` do tablicy `cel`. Za jednym zamachem robimy więc kilka rzeczy.

Dwie uwagi (tytułem przestrogi)



Po pierwsze:

Do wnętrza instrukcji kopiowania nie wstawiłem postinkrementacji indeksu `i`. To dlatego, że w wyrażeniu

```
cel[i] = zrodlo[i++] ;
```

nie mam pewności kiedy naprawdę nastąpiłaby ta postinkrementacja. Jest ryzyko, że na przykład po wyjęciu znaku z tablicy `zrodlo`, a przed wsadzeniem go do tablicy `cel`. Wtedy kopiowanie odbyłoby się na wzór takiej instrukcji:

```
cel[4] = zrodlo[3] ;
```

Jak to jest? Otóż kompilator nie gwarantuje kolejności obliczania zmiennej `i`, dlatego bezpieczniejszą inkrementację zrobić w głębi pętli `while`.



Druga uwaga:

Niektóre bardziej troskliwe kompilatory, gdy zobaczą wyrażenie

```
while(cel[i] = zrodlo[i])...
```

czyli inaczej

```
while(a = b)...
```

nie doceniają naszego sprytu i pomyślą, że prawdopodobnie chodzi nam o porównanie dwóch elementów. Czyli, że mówimy: wykonuj daną pętlę dopóki `a` równa się `b`.

Kompilator taki przypuszcza, że pomyliliśmy porównanie (`==`) z przypisaniem (`=`) czyli, że zamiast dwóch znaków `==` wstawiliśmy tylko jeden `=`. Nie sygnalizuje błędu, jedynie ostrzega nas.

Trzeba jednak przyznać, że najczęściej taki kompilator ma rację. Chwyt, który tutaj zastosowaliśmy nie stosuje się tak często. Za to bardzo często zapomina się przy operacji porównania o podwójnym znaku równości. Dlatego lepiej niech kompilator nam patrzy na ręce.



Wróćmy jednak do naszych baranów.

Tym sposobem napisaliśmy funkcję `strcpy`, która może nam się przydać bardzo często. Przykładowo spójrz na taki fragment programu:

```
//.....
char start[] = { "taki sobie zwykly tekst" } ;
char meta[80] ;

strcpy(meta, start) ;
cout << meta ;
```

Fragment ten wykonuje kopiowanie stringu z tablicy `start` do tablicy `meta`. Następnie wypisuje się na ekran nową zawartość tej tablicy.

Przyjrzyjmy się definicji tablicy `start`. Nie ma w niej rozmiaru, zatem kompilator przeliczy sobie wszystkie znaki w tekście ograniczonym cudzysłowem, następnie doda jeden (na kończący znak NULL) i tyle komórek pamięci zarezerwuje na naszą tablicę.

Z tablicą `meta` jest inaczej. Tutaj nie inicjalizujemy jej, więc kompilator chce znać żądany rozmiar. Podajemy mu np. 80, bo spodziewamy się, że nasze teksty ładowane do tej tablicy nigdy nie będą dłuższe niż 79 znaków.

Ostrożnie !

Co by było, gdybyśmy w definicji tablicy `meta` zamiast [80] podali rozmiar [5] i wykonali program?

Pamiętasz zapewne, że w naszej funkcji `strcpy` kopiowanie odbywa się na oślep. Poszczególne znaki będą brane z tablicy `start` i wpisywane do kolejnych elementów do tablicy `meta`. Tak będzie aż do końca ciągu znaków, czyli do końącego treść tablicy `start` znaku NULL. Jeśli tablica `meta` jest za krótka, bo ma tylko 5 elementów (ostatni to `meta[4]`) – to mimo wszystko dalej będzie odbywało się wpisywanie do nieistniejących elementów

```
meta[5], meta[6], ....
```

i tak dalej dopóki string ze startu się nie skończy.

Wiesz oczywiście co takie wpisywanie do nieistniejących elementów znaczy: Mimowolnie będą niszczone komórki pamięci znajdujące się zaraz za naszą tablicą `meta`. Tragedia. Tym większa tragedia, że nie musisz się zorientować od razu. Zależnie od tego co tam akurat było - błąd może ujawnić się o wiele później. Tym trudniej jest to wykryć.

Czy nie można zabezpieczyć się przed takimi ewentualnościami? Ostatecznie można, ale jest problem: otóż do funkcji `strcpy` przesyłamy tylko adres tablicy. Funkcja z deklaracji argumentów formalnych wie tylko, że ma do czynienia z tablicami znakowymi. Nie wie nic więcej. W szczególności nie wie jaki jest rozmiar tablicy. Nie można wewnątrz funkcji `strcpy` wstawić sobie takiego wyrażenia:

```
sizeof(cel)
```

Jeśli zatem chcemy się przed pomyłkami zabezpieczyć, to musimy przysłać do funkcji argument będący rozmiarem tablicy, albo – bardziej uniwersalnie – argument mówiący o tym, ile maksymalnie znaków życzymy sobie przekopować.

Na przykład życzymy sobie przekopiować tylko 5 znaków. Jeśli string przeznaczony do kopiowania będzie bardzo krótki, np 3 literowy: "ach" to przekopiuje się cały. Jeśli będzie długi: "Niesamowicie długie zdanie", to przekopiuje się tylko początek "Niesa". Na końcu musi się oczywiście znaleźć bajt zerowy (NULL).



Moglibyśmy się teraz rzucić do pisania i wymyślić funkcję `strncpy`. Litera `n` w środku nazwy miałaby nam przypominać, że chodzi po przekopiowanie maksymalnie `n` znaków, (czyli tylko kawałka stringu) bez dodawania na końcu znaku NULL. Moglibyśmy tak zrobić, gdyby nie to, że te wszystkie wspaniałe funkcje dawno zostały napisane i zawarte są w standardowej bibliotece. Aby z nich skorzystać wystarczy włączyć do swojego programu plik nagłówkowy z ich deklaracjami. Plik nagłówkowy tej części standardowej biblioteki, która odpowiada za pracę ze stringami nazywa się `string.h`. Potem, na etapie linkowania dołączana jest (najczęściej automatycznie) ta część biblioteki.

```
#include <iostream.h>
#include <string.h>
main()
{
    char tekst[] = { "Uwaga, tarcza została przepalona !" } ;
    char komunikat[120] ;

    strcpy(komunikat, tekst);
    cout << komunikat << endl ;

    strncpy(komunikat, "1234567890abcdef" , 9 ) ;    // ❶
    cout << komunikat ;

    strcpy(komunikat, "--A ku-ku --!" ) ;
    cout << "\nNa koniec : "
         << komunikat << endl ;
}
```



Wykonanie tego programu objawi się na ekranie jako:

```
Uwaga, tarcza została przepalona !
123456789rcza została przepalona !
Na koniec : --A ku-ku --!
```

- ❶ Zauważ, że źródłem wcale nie musi być tablica definiowana przez nas. Może być to stała tekstowa (`string`) będący stałą dosłowną. Taki `string` przecież także musi być przechowywany gdzieś w pamięci komputera.



Dużo mówiliśmy tu na temat tablic znakowych. Nie dlatego, żeby były one jakieś specjalnie trudne, lecz dlatego, że bardzo często się ich używa. Nie ma tu w zasadzie żadnych cudów – `string` to po prostu znaki ustawione jeden za

drugim, a na ich końcu jest NULL, czyli bajt zerowy. To wszystko. Nazwa tego stringu (nazwa tablicy, gdzie on tkwi) jest równocześnie adresem tego miejsca w pamięci, gdzie string ten się zaczyna. Tam jest więc początek tego stringu. Koniec jest tam, gdzie jest znak NULL.

Jeśli chcemy wysłać string do funkcji, to wysyłamy tam adres jego początku, czyli samą jego nazwę (bez żadnych nawiasów kwadratowych). Dzięki temu funkcja dowiaduje się, gdzie w pamięci zaczyna się ten string. Gdzie się on kończy – funkcja może sprawdzić sobie sama szukając znaku NULL.

Do stringów jeszcze powrócimy w rozdziale o wskaźnikach.

7.5 Tablice wielowymiarowe

Tablice można tworzyć z bardzo różnych typów obiektów. Widzieliśmy już tablice z liczb całkowitych, zmiennoprzecinkowych, tablice znaków.

Mogą być także tablice, których elementami są inne tablice. Nazywamy je tablicami wielowymiarowymi. Oto przykład takiej tablicy:

```
int ddd[4][2] ;
```

Definicję tę, która jest przy okazji deklaracją czytamy tak: ddd jest tablicą 4 elementów, z których każdy jest dwuelementową tablicą liczb typu int.

Zauważ charakterystyczną notację. W innych językach często stosuje się zapis typu [i, j] – u nas jest [i][j]. Można myśleć o tym, jako o zwykłej odmienności notacji. Można też twierdzić, że zapis ddd[i][j] pozwala lepiej pamiętać, że ddd to tablica tablic. Czasem ten punkt widzenia się przydaje. Wkrótce się przekonasz.

Ewentualny błędny zapis ddd[i, j] kompilator zinterpretuje jako ddd[j]. (Przypominam jakie ma działanie operator „,” (przecinek) : wartością wyrażenia złożonego z kilku członów oddzielonych przecinkami jest wartość tego członku, który stoi najbardziej z prawej).

Poszczególne elementy naszej tablicy to:

```
ddd[0][0],    ddd[0][1],    ddd[1][0],    ddd[1][1],  
ddd[2][0],    ddd[2][1],    ddd[3][0],    ddd[3][1]
```

Elementy takie umieszczane są kolejno w pamięci komputera tak, że najszybciej zmienia się najbardziej skrajny prawy indeks.

Zatem poniższa inicjalizacja zbiorcza:

```
int ddd[4][2] = { 10, 20, 30, 40, 50, 60, 70, 80 } ;
```

spowoduje, że elementom tej tablicy zostaną przypisane wartości początkowe tak, jakbyśmy to robili grupą instrukcji:

```
ddd[0][0] = 10 ;  
ddd[0][1] = 20 ;  
ddd[1][0] = 30 ;  
ddd[1][1] = 40 ;  
ddd[2][0] = 50 ;  
ddd[2][1] = 60 ;
```



```
ddd[3][0] = 70 ;
ddd[3][1] = 80 ;
```

Można powiedzieć, że tablica ta ma 4 wiersze i 2 kolumny.

Oto przykład programu:

Założmy, że mamy nasze dane pomiarowe w postaci tablicy 8192 elementów. Takich pomiarów mamy cztery, dla czterech różnych próbek. Zapisane są te dane na dysku w postaci pliku. Uruchamiamy program i wczytujemy te dane do tablicy wielowymiarowej

```
long widmo[4][8192] ;
```

Jak to robimy, to na razie tajemnica, o szczegółach operacji z plikami porozmawiamy w osobnym rozdziale. Teraz tę akcję markujemy jedynie wywołaniem funkcji `wczytaj_dane`.

O co nam chodzi w tym programie: założmy, że między elementem tablicy o indeksie 500, a elementem o indeksie 540 zebrały się dane z interesującej nas reakcji. Chcemy więc zsumować wszystkie liczby zapisane od elementu o indeksie 500 do elementu o indeksie 540. Chcemy to zrobić dla każdego z czterech pomiarów

```
#include <iostream.h>
wczytaj_dane() ;
/*****
main()
{
    long widmo[4][2048] ;
    long suma ;
    int i ;
    // tajemnicza funkcja, która wczyta z dysku cztery
    // zestawy wyników pomiarowych i dane te umieści w tablicy widmo
    wczytaj_dane() ;

    cout << "Jaki przedział widma ma być integrowany ?\n"
          << "podaj dwie liczby : " ;

    int pocz, koniec ;
    cin >> pocz >> koniec ;

    // --- pętla po 4 różnych próbkach
    for(int pomiar = 0 ; pomiar < 4 ; pomiar++) // ❶
    {
        suma = 0 ;
        // --- pętla integrująca dane jednej próbki
        for(i = pocz ; i <= koniec ; i++) // ❷
        {
            suma += widmo[pomiar][i] ; // ❸
        }

        cout << "\nW probce " << pomiar
              << " między kanałami "
              << pocz << " a " << koniec << " jest "
              << suma << " zliczeń" ; // ❹
    }
    // koniec pętli po 4 pomiarach
```

```

}
/*****
wczytaj_dane()
{
    //... wczytywanie danych z dysku
}

```

Po wykonaniu programu (zależnie od odpowiedzi na pytania) możemy na ekranie otrzymać na przykład następujący tekst:

```

Jaki przedział widma ma byc integrowany ?
podaj dwie liczby : 50 75
W probce 0 miedzy kanalami 50 a 75 jest 493 zliczen
W probce 1 miedzy kanalami 50 a 75 jest 392 zliczen
W probce 2 miedzy kanalami 50 a 75 jest 300 zliczen
W probce 3 miedzy kanalami 50 a 75 jest 172 zliczen

```

Uwagi

- 1 Ponieważ w programie mamy wykonać identyczną akcję na 4 różnych próbkach dlatego wygodnie posłużyć się pętlą.
- 2 W danych pomiarowych danej próbki sumujemy kolejne elementy (sąsiednie kanały). O tym, która część widma ma zostać poddana takiej operacji, zdecydowaliśmy odpowiadając na pytania o początek i koniec.
- 3 Jest to moment sumowania elementów tablicy. Jeśli jeszcze nie oswoiłeś się z operatorem += to może wolisz taki, równoważny tamtemu zapis

```
suma = suma + widmo[pomiar][i] ;
```

- 4 To jest moment wypisania wyniku na ekran.



Jak widzisz nie ma w tablicach dwu- i wielowymiarowych niczego nadzwyczajnego ani trudnego. Domyślasz się chyba też jak komputer oblicza sobie, gdzie w pamięci jest dany element tablicy.

Jak już wiemy, w pamięci elementy ustawiane są kolejno tak, że najczęściej zmienia się najbardziej skrajny indeks. Znaczy to, że w naszej tablicy

```
long widmo[4][2048] ;
```

element `widmo[1][3]` leży w stosunku do początku tablicy o tyle elementów dalej

```
(1 * 2048) + 3
```

ogólniej, element `widmo[i][j]` z tablicy o liczbie kolumn 2048 leży o

```
(i * 2048) + j
```

elementów dalej niż początkowy. To bardzo ważny wniosek. Jak widzisz do orientacji w naszej tablicy kompilator potrzebuje znać liczbę jej kolumn, natomiast wcale nie musi używać liczby wierszy.

Ja bym sobie w takich stwierdzeniach nawet nie zawracał głowy wierszami i kolumnami. Można to prościej zapamiętać tak:

Ten wymiar, który w definicji tablicy jest najbardziej z lewej

```
long widmo[4][2048] ;           // u nas to liczba 4
```

nie jest potrzebny kompilatorowi do obliczania położenia (adresu) elementów tablicy.

Zwykle o takich sprawach nie musimy w ogóle wiedzieć, każemy zrobić operację na wybranym elemencie `[i][j]`, a jak sobie to komputer załatwia – to już jego sprawa. Jeśli jednak jest się tego świadomym, to łatwiej zrozumieć jak przekazuje się do funkcji tablice wielowymiarowe.

7.5.1 Przesyłanie tablic wielowymiarowych do funkcji

Ten tytuł może trochę mylić. Tablice jako całość nigdy nie są przekazywane. To między innymi dlatego, że w wypadku takiej tablicy:

```
long widmo[4][8192] ;
```

musielibyśmy do funkcji przesłać $4 * 8192 = \text{ok. 32 tysiące liczb}$ (a każda jest np. 4 bajtowa).

Wiemy już, że tablice przekazuje się w ten sposób, iż przesyłamy do funkcji tylko adres początku.

Jak pamiętamy – nazwa tablicy jest równocześnie adresem jej początku – więc do hipotetycznej funkcji `fun` przesyłamy ją tak:

```
fun(widmo) ;
```

A teraz przyjrzyjmy się jak ją (tablicę) odbieramy w funkcji.

Zanim jednak to pokażemy uświadomijmy sobie co funkcja musi o naszej tablicy wiedzieć

- ❖ po pierwsze – oczywiście typ elementów tej tablicy (czy `float`, czy `char` itd),
- ❖ po drugie – aby funkcja mogła łatwo obliczać sobie, gdzie w pamięci jest określony element tablicy – musi znać liczbę kolumn w tej tablicy. (Na przykład: gdzie w stosunku do początku tablicy jest element `[1][15]`).

Deklaracja takiej funkcji wygląda tak:

```
void fun(long t[][8192]) ;
```

podana jest w niej liczba kolumn tej tablicy, którą funkcja będzie otrzymywać. Oczywiście deklaracja

```
void fun(long t[4][8192]) ;
```

jest tak samo dobra, ale z liczby wierszy (tutaj 4) funkcja nie korzysta.

Więcej wymiarów

Przez analogię łatwo chyba się domyślisz, że w wypadku funkcji otrzymującej tablicę trójwymiarową deklaracja takiej funkcji może wyglądać na przykład tak:

```
void fun_trzy(long m[][20][100] );
```

a czterowymiarowej

```
void fun_cztery(long x[][2][12][365]) ;
```

Tylko lewego skrajnego rozmiaru nie trzeba deklarować, bo i tak nie bierze on udziału w obliczaniu pozycjiżądanego elementu w pamięci.

Cały ten paragraf nie służy po to, by Cię zniechęcić do podawania w deklaracji tego lewego skrajnego wymiaru. Jest po to, byś wiedział dlaczego deklaracja

```
void fff(long m[][][]); // błąd
```

jest błędna, a deklaracja

```
void ggg(long z[]) ;
```

jest poprawna.



Na tym kończy się rozdział o tablicach, ale nie rozstajemy się z tą tematyką, bowiem następny rozdział mówił będzie o wskaźnikach, co mocno jest związane z tematyką tablic.

8 Wskaźniki

Zapytano kiedyś Amerykanów: „-Czy można, żyć bez Coca-Coli?“. Amerykanie odpowiedzieli: „-Można, ale po co?“

Dokładnie to samo można powiedzieć o wskaźnikach. Ich istnienie nie jest konieczne, najlepszy dowód, że jest wiele języków programowania, w których wskaźników nie ma. Z drugiej jednak strony – jeśli języki C i C++ mają opinię tak sprawnych i władnych^{†)}, to jest to głównie za sprawą wskaźników.

Są ludzie, którzy nie lubią C. Moim zdaniem to dlatego, że nie czują się swobodnie w operowaniu wskaźnikami.

Jeśli doskonale znasz język C, a książkę tę czytasz, by poznać C++, to w zasadzie mógłbyś ten rozdział opuścić. Jeśli jednak wskaźniki znasz trochę gorzej, to zachęcam do przeczytania.

8.1 Wskaźniki mogą bardzo ułatwić życie

Do rzeczy: Wyobraź sobie, że masz w ręce bardzo gruby rozkład jazdy. Taki na 600 stron. Ktoś pyta Cię o pociąg z Paryża do Marsylii. Otwierasz więc rozkład jazdy i szukasz przewracając strony. Wreszcie po dłuższym czasie trafiasz na właściwą stronę i przebiegając palcem kolumny cyfr znajdujesz, to czego szukałeś. Dumnym głosem mówisz – 9:26 i zamykasz rozkład z trzaskiem.

Wtedy ten ktoś pyta: „-A następny?“. Bierzesz znowu rozkład jazdy, przewracasz strony, trafiasz na właściwą, potem znowu przebiegasz kolumny i... jest! 13:50

Tylko, że teraz jesteś już sprytniejszy: po odpowiedzi na wszelki wypadek trzymasz palec wskazujący na znalezionej godzinie odjazdu. Jeśli Twój przy-

†) Chciałoby się powiedzieć po angielsku: *powerfull*

jacieli zapyta Cię teraz: „–A następny...?” - błyskawicznie przesuniesz palec o jedną kolumnę w prawo o odczytasz godzinę. Jeśli zapyta Cię „–A o której on jest w Lyonie?” wtedy przesuniesz palec kilka linii w dół.

Podany przykład ilustruje życie ze wskaźnikami i bez. Co było wskaźnikiem w naszym wypadku? Był to Twój palec, ale nie tylko: także wiedza o tym, jak należy go przesunąć w wypadku pytania: „A następny?”

Przetłumaczmy to na język pojęć, którymi operujemy w C++ :

Rozkład jazdy to nic innego jak olbrzymia (wielowymiarowa) tablica liczb. Godzina odjazdu pociągu z Paryża do Marsylii to po prostu element tej tablicy

```
pociąg[Paryz][Marsylia][wyjazd_z_Paryza][0]
```

Ostatni indeks oznacza, o który pociąg w ciągu doby nam chodzi (w ciągu doby mogą bowiem odjeżdżać do Marsylii np. 4 takie pociągi).

Szukanie informacji o tym pociągu to obliczanie miejsca w tablicy, gdzie zapisana jest ta informacja. Pamiętasz zapewne z poprzedniego rozdziału, jak oblicza kompilator pozycję elementu w stosunku do początku tablicy. Dla tamtej dwuwymiarowej tablicy była to zależność

```
(i * 8192) + j
```

gdzie 8192 było liczbą kolumn tablicy. Tutaj, w naszym przykładzie z rozkładem jazdy, mamy dużo więcej wymiarów, więc i obliczanie będzie bardziej skomplikowane. W naszym obliczaniu będą aż 3 mnożenia, a to zajmuje trochę czasu. Gdy wreszcie trafiliśmy w rozkładzie jazdy na właściwe miejsce, to odczytaliśmy je – jest to jakby odczytanie liczby będącej treścią elementu naszej tablicy.

Potem zamknęliśmy rozkład i wtedy nasz przyjaciel wygłosił wspomniane słowa: „A następny?”

Następny pociąg w naszej tablicy to

```
pociąg[Paryz][Marsylia][wyjazd_z_Paryza][1]
```

Od nowa więc zajęliśmy się obliczaniem miejsca w pamięci, gdzie zapisana jest poszukiwana informacja. Znowu natrudziliśmy się wielce powtarzając ten proces, ale wreszcie znaleźliśmy właściwą informację i odczytaliśmy ją.

Wtedy, chcąc oszczędzić sobie dalszej ewentualnej pracy, zaznaczyliśmy sobie palcem to miejsce i dopiero wtedy przymknęliśmy książkę.

Oznacza to – zdefiniowaliśmy wskaźnik (wskazujący palec prawej ręki) i podstawiliśmy do niego wartość początkową – adres odczytanego właśnie elementu tablicy (innymi słowy przystawiliśmy palec wskazujący tak, by pokazywał na właściwą liczbę na właściwej stronie)

Książkę przymknęliśmy, ale tak, że palec mimo wszystko tam tkwi. Kiedy nasz przyjaciel zapytałby nas o następny pociąg, od razu otworzylibyśmy rozkład na właściwej stronie. Błyskawiczny ruch palca w prawo i już wiemy.

Zapytanie o to, kiedy pociąg jest w Lyonie, jest próbą znalezienia elementu

```
pociąg[Paryz][Marsylia][postoj_w_Lyonie][1]
```

Tutaj wystarczy wiedzieć ile stacji jest między Paryżem a Lyonem i o tyle linijek przesunąć palec w dół.

Jaki jest wniosek z tej dykteryjki? Taki, że posługiwanie się wskaźnikiem bardzo przyspiesza operacje na tablicach.

Może być więcej niż jeden wskaźnik pracujący na naszej tablicy. Możemy bowiem innym palcem zaznaczyć sobie informację o pociągach z Marsylii do Avignon. Jeśli nam nie wystarczy palców zorganizujemy sobie zakładki. Zakładki takich możemy mieć dowolnie dużo.

Jeśli po tym ktoś mi powie, że nie lubi wskaźników, to nie uwierzę. Mało rzeczy może tak uprościć życie.

8.2 Definiowanie wskaźników

Wskaźniki nie muszą koniecznie pokazywać na elementy tablicy. Można nimi posłużyć się wobec dowolnej zmiennej, wobec dowolnego obiektu. Zaczniemy od takich przykładów.

Przyjrzyjmy się poniższej definicji

```
int *w ;
```

Jest to definicja wskaźnika mogącego pokazywać na obiekty typu `int`.

Definicję taką odczytujemy jak zwykle od końca. Na widok gwiazdki mówimy: „jest wskaźnikiem do pokazywania na obiekty typu...”

A zatem czytamy na głos:

`w` – jest wskaźnikiem do pokazywania na obiekty typu – `int`.

Innymi słowy, w tym wskaźniku `w` możemy przechowywać adres jakiegoś obiektu typu `int`. Treścią wskaźnika jest informacja o tym, gdzie wskazywany obiekt się znajduje, a nie co w tamtym obiekcie się znajduje.

W definicji widzisz znak `*`, który informuje nas, że mamy tu do czynienia ze wskaźnikiem. Słowo `int` w tej definicji informuje nas, że wskaźnik ten ma służyć do pokazywania na obiekty typu `int`.



Nasz wskaźnik nazywa się `w`. Samo `w`, bez gwiazdki. Gwiazdka zaś, mówi tylko, że to coś o nazwie `w` jest wskaźnikiem. Słowo `int` mówi na jakie obiekty może on pokazywać.

Porównaj to zresztą z definicją tablicy

```
int t[5] ;
```

Tablica nazywa się `t`. Nawiasy `[]` mówią tylko, że to `t` jest tablicą. `int` – że elementów typu `int`.

Mogą być wskaźniki do obiektów różnych typów: np. `char`, `float` itd, (a nawet do obiektów, których typy sami sobie wymyślimy i zdefiniujemy).

```
char * wsk_do_znakow ;  
float * wsk_do_floatow ;
```



Zwróć uwagę, że w definicji jest powiedziane, że wskaźnik pokazuje na obiekty. Referencja (przezvisko) nie jest obiektem, dlatego nie może być do niej wskaźników. Także nie może być wskaźników do pojedynczych bitów jakiegoś słowa – tzw. pól bitowych (patrz str. 323).

W naszej definicji stworzyliśmy sobie wskaźnik, ale nie pokazuje on jeszcze na nic rewelacyjnego. To tak, jakbyśmy na lekcję geografii wystrugali wskaźnik z drewna i położyli go na boku. Mimo, że wskaźnik leży na boku, to przecież jego koniec zawsze na coś pokazuje (celuje), ale z tego pokazywania jeszcze nie można się szczyścić.

Wskaźnik służący do pokazywania na obiekty jednego typu nie nadaje się do pokazywania na obiekty innego typu. ^{†)}

Konkretnie: wskaźnikiem do `int` nie można pokazywać na `float` czy `string`. Próba ustawienia wskaźnika na obiekt innego, niewłaściwego typu wywoła protest kompilatora.

8.3 Praca ze wskaźnikiem

Oto jak można nadać naszemu wskaźnikowi wartość początkową, czyli sprawić, by na coś pokazywał.

Używa się do tego jednoargumentowego operatora `&` (ampersand). To on oblicza adres obiektu, który stoi po prawej stronie przypisania.

```
int *w ;           // definicja wskaźnika do obiektów typu int
int k = 3 ;        // definicja zwykłego obiektu typu int z liczbą 3

w = &k ;           // ustawienie wskaźnika na obiekt k
```

Kiedy wskaźnik już pokazuje na żądane miejsce – możemy odnieść się do tego obiektu, na który pokazuje.

Odnieść się do obiektu – rozumiem jako na przykład: odczytać jego zawartość, czy też coś zapisać do niego. Do takiej operacji służy jednoargumentowy operator odniesienia się `*` (gwiazdka).

„–Znowu gwiazdka?” – zapytasz pewnie – „nie za dużo tego?” Odpowiem jednak: – Jest w tym logika. Otóż zapamiętaj sobie złotą myśl:

W języku C++ definicje obiektów zapisywane są tak, że zapis przypomina sposób, w jaki później dany obiekt występuje w wyrażeniach.

Definicja tablicy zawierała klamry

```
int t[5] ;
```

^{†)} W rozdziale o dziedziczeniu dowiemy się, że od tej reguły jest chwalebny wyjątek.

dlatego, później w wyrażeniach klamry oznaczają pracę na tablicy

```
a = t[1] ;           // odczytanie pierwszego elementu tablicy.
```

Nie inaczej jest w wypadku wskaźnika. Jeśli mamy nasz wskaźnik i już ustawimy go na obiekt *k*, to treść pokazywanego obiektu odczytuje się jednoargumentowym operatorem ***

```
int *w ;              // definicja wskaźnika
int k = 3 ;           // definicja zmiennej

w = &k ;              // ustawienie wskaźnika
cout << "W obiekcie pokazywanym przez "
      "wskaźnik w jest wartosc " << (*w) ;
```

Wykonanie tego fragmentu programu da nam na ekranie

W obiekcie pokazywanym przez wskaźnik *w* jest wartosc 3

Słowem: gwiazdka kieruje nas do obiektu pokazywanego przez wskaźnik. Nasz wskaźnik pokazywał na zmienną *k*. Zatem od tej pory **w* oznacza to samo co *k*. Skoro zawartością *k* była liczba 3, to ta wartość pojawiła się na ekranie.

Wniosek: po ustawieniu wskaźnika obie poniższe instrukcje są równoważne

```
cout << k ;           // wypisz na ekran k
cout << *w ;          // wypisz na ekran k
```

Przyjrzyjmy się teraz takiemu programowi:

```
#include <iostream.h>
main()
{
    int zmienna = 8 , drugi = 4 ;           // ❶
    int *wskaźnik ;                         // ❷

    wskaźnik = &zmienna ;                  // ❸

    // prosty wypis na ekran ;               ❹
    cout << "zmienna = " << zmienna
        << "\n a odczytana przez wskaźnik = "
        << *wskaźnik << endl ;

    zmienna = 10 ;                          // ❺
    cout << "zmienna = " << zmienna
        << "\n a odczytana przez wskaźnik = "
        << *wskaźnik << endl ;

    *wskaźnik = 200 ;                       // ❻
    cout << "zmienna = " << zmienna
        << "\n a odczytana przez wskaźnik = "
        << *wskaźnik << endl ;

    wskaźnik = &drugi ;                     // ❼
    cout << "zmienna = " << zmienna
        << "\n a odczytana przez wskaźnik = "
```

```
<< *wskaznik << endl ;
```

```
}
```

Po wykonaniu tego programu na ekranie pojawi się

```
zmienna = 8
a odczytana przez wskaznik = 8
zmienna = 10
a odczytana przez wskaznik = 10
zmienna = 200
a odczytana przez wskaznik = 200
zmienna = 200
a odczytana przez wskaznik = 4
```

- 1 Tutaj definiujemy dwa najzwyczajniejsze obiekty typu `int`. Od razu inicjalizujemy je wartościami liczbowymi.
- 2 Tutaj definiujemy wskaznik. Definicję czyta się tak: `wskaznik` – jest wskaźnikiem (przepraszam!) do pokazywania na obiekt typu `int`.
- 3 Tutaj sprawiamy, że wskaznik zaczyna pokazywać na coś sensownego. Robimy to za pomocą operatora `&` – uzyskującego adres obiektu `zmienna`. Ten adres jest podstawiony do wskaźnika operatorem przypisania =
W zasadzie powinienem napisać linijkę 2 i 3 razem stosując skrócony zapis, jednak obiecałem sobie nie przerażać Cię. Taki skrócony zapis tych dwóch linijek wygląda tak:

```
int *wskaznik = &zmienna ;
```

- 4 Wielokrotnie w trakcie tego programu wypisujemy na ekranie wartość tego, na co pokazuje wskaznik. W uproszczeniu to po prostu
- ```
cout << *wskaznik ;
```
- 5 Do zmiennej wpisujemy nową wartość. Kolejny wypis na ekran udowadnia, że wskaznik cały czas pokazuje na ten obiekt i oczywiście zauważa zmianę.
  - 6 To jest bardzo ważna linijka. Skoro wyrażenie `*wskaznik` oznacza obiekt, na który pokazuje wskaznik, to zapis

```
*wskaznik = 200 ;
```

oznacza: „do tego, na co pokazuje wskaznik, wpisz liczbę 200”. Ponieważ wskaznik pokazywał na obiekt `zmienna`, więc do obiektu `zmienna` zostaje wpisana liczba 200. Chciałbym, żebyś docenił tę historyczną chwilę: Oto do obiektu można coś wpisać albo używając jego nazwy (`zmienna`), albo wskaźnika, który na ten obiekt pokazuje (`*wskaznik`). Na dowód, że to działa – znowu wypisujemy to na ekran. Widzimy na ekranie dwukrotnie liczbę 200

- 7 Wskaznik nie pokazuje raz na zawsze na ten sam obiekt. Można go łatwo przestawić i od tej pory pokazuje na inny obiekt. Tutaj widzimy ustawienie wskaźnika tak, by pokazywał na obiekt drugi. (Robimy to, jak widać, wstawiając do wskaźnika adres zmiennej o nazwie drugi. Następny wydruk na ekranie pokazuje nam więc już liczby 200 i 4. Liczba dwieście jest treścią zmiennej, natomiast skoro wskaznik pokazuje teraz na obiekt drugi, to wyrażenie `*wskaznik` odnosi się teraz do obiektu drugi.

Pomyślisz pewnie: „Co to za bzdurny program ! Po co do wpisania czy odczytania czegoś z obiektu używać wskaźnika, skoro łatwiej użyć nazwę tego obiektu.” Masz rację, program jest rzeczywiście bzdurny, w zasadzie wskaźniki mogły by w nim w ogóle nie istnieć. Chciałem w nim jednak pokazać, że:

Do danego obiektu można odtąd odnosić się na dwa sposoby: albo przez jego nazwę albo przez zorganizowanie wskaźnika, który na ten obiekt pokaże.



## Mała dygresja o science-fiction

Kiedys tłumaczyłem pracę na wskaźnikach mojemu kilkunastoletniemu przyjacielowi, miłośnikowi fantastyki naukowej. Kiedy doszliśmy do zapisu

$$a = *w ;$$

i próbowałem mu wytłumaczyć, że gwiazdka oznacza... przerwał mi i zdecydowanie powiedział kończąc jakąkolwiek dyskusję:

„To proste: gwiazdka oznacza, że lecimy gwiazdolotem do miejsca, na które pokazuje wskaźnik w i dopiero tamtym obiektem naprawdę się zajmujemy”

Możesz to, drogi Czytelniku uznać za dziecinne, ale takie mnemotechniczne skojarzenia bardzo pomagają w początkowej fazie rozumienia.

Drugim skojarzeniem mojego przyjaciela było, że jednoargumentowy operator & – (znaczek ten po angielsku zwany jest AMPERSAND) zaczyna się na literę a – tak samo jak słowo ADRES.

W sumie miał on więc takie regułki:

|     |   |                                             |
|-----|---|---------------------------------------------|
| * w | – | gwiazdolotem w miejsce, na które pokazuje w |
| & m | – | a, jak adres obiektu m                      |

Przypominam, że „gwiazdolot” występuje tylko w wyrażeniach. Zupełnym wyjątkiem jest gwiazdka w linii definicji wskaźnika. Tam ma ona przypominać sposób w jaki wskaźnika się będzie potem używało.

## 8.4 L-wartość

Operacja odniesienia się do danego obiektu może być po prawej stronie znaku = (przypisania)

$$a = *wskaznik ;$$

ewentualnie po jego lewej stronie

$$*wskaznik = 55 ;$$

Jeśli coś może stać po lewej stronie znaku = (mówiąc mądrzej - po lewej stronie operatora przypisania), to takie „coś” nazywamy l-wartością. Nazwa łatwa do

zapamiętania, bo l – jak lewa strona. Po angielsku nazywa się to l-value<sup>†)</sup>, a mówię o tym dlatego, że jeśli kiedyś napiszesz przez pomyłkę instrukcję

```
10 = i ; // 10 is not a l-value !
```

To kompilator zaprotestuje i powie Ci (po angielsku), że instrukcja ta jest błędna jako, że stojące po lewej stronie 10 nie jest słynną l-value. Czyli, że liczba 10 nie nadaje się do postawienia po lewej stronie. Po prawej tak, ale nie po lewej

```
i = 10 ; // jest poprawne
```

Analogicznie wyrażenie, które może stać po prawej stronie nazywamy r-wartością (od angielskiego right – prawy). Jednak po prawej stronie to stać może już byle kto, więc być r-wartością to żaden honor. L-wartość - to jest coś wyjątkowego.

Oczywiście l-wartość jest także r-wartością. Ale nie odwrotnie.

Dlaczego to takie ważne, że wyrażenie

```
*wskaznik
```

jest l-wartością ?

Dlatego, że mamy wyrażeniem \*wskaznik posługiwać się w takich samych sytuacjach jak wyrażeniem zmienna, które tą l-wartością jest.

Zatem jeśli wskaznik pokazuje na zmienną, to poniższe zapisy są równoważne

```
zmienna = 6 ; *wskaznik = 6 ; // jako l-wartości
```

```
int m ; m = zmienna ; m = *wskaznik ; // jako r-wartości
```

## 8.5 Wskaźniki typu void

Wskaźnik – jak wielokrotnie już mówiliśmy – to adres jakiegoś miejsca w pamięci plus wiedza o tym, jakiego typu obiekt pokazuje się.

Czyli z definicji

```
int *a ;
```

wynika, że a to wskaźnik, którym można przechować adres jakiegoś obiektu, a ta wiedza to pewność, że to obiekt typu int.

Możemy jednak zdefiniować wskaźnik bez tej „wiedzy”. Mówimy wtedy, że jest to wskaźnik typu void.

```
void *w ;
```

†) (czytaj: „el-velju“)



```

int *wi ;
float *wf ;

// tutaj w programie ustawia się te wskaźniki na jakieś obiekty
...
// a teraz przypisanie do wskaźnika typu void (bez konieczności
// rzutowania)
wv = wf ;
wv = wi ;
wv = wf ;

```

Tu nieco wybiegnę w przyszłość:

Jest jeden warunek: nie można temu wskaźnikowi przypisać treści wskaźnika do obiektu z przydomkiem `const`. To oczywiście dlatego, by zapobiec oszustwom. Po takim przypisaniu można by bezkarnie zmienić obiekt, który miał być przecież `const`.

Wobec powyższego sformułujmy wniosek:

Wskaźnik każdego (niestałego) typu można przypisać wskaźnikowi typu `void`

`void` ← ——— dowolny wskaźnik (non-const);

W trakcie takiego przypisywania przekazywany jest adres, a „wiedza” jest zapomniana – i to jest w porządku.

Natomiast nie da się odwrotnie.

To znaczy wskaźnika typu `void` nie można przypisać wskaźnikowi „prawdziwemu”. Trzeba posłużyć się operatorem rzutowania. Dla powyżej zdefiniowanych wskaźników wygląda to tak:

```

wf = (float *) wv ;
wi = (int *) wv ;
wc = (char *) wv ;

```

Jak widać konieczny jest tu operator rzutowania, bo następuje przypisanie

```

float* ←——— void*
int* ←——— void*
char* ←——— void*

```

Reasumując:

Wiedzę o typie obiektu pokazywanego można ewentualnie niezauważenie zapomnieć. Nabyć tej wiedzy niezauważenie nie można. Trzeba to świadomie wyspecyfikować operacją rzutowania.

Uwaga dla programistów C

Jest to jedna z różnic między C++ a ANSI C. W ANSI C było tak, że niezależnie, po której stronie operatora przypisania stał wskaźnik typu `void` nie trzeba było używać rzutowania. Zatem „wiedzę” nabywało się także niezauważenie. Było to jakby wyjście awaryjne dla umożliwienia łatwiejszego zapisu instrukcji rezerwujących pamięć. (Funkcjami w rodzaju `malloc()` – memory allocation). Ponieważ w C++ do tego celu służą dziecinne proste operatory `new` i `delete`, dlatego regułę można zaostriżyć.

## 8.6 Cztery domeny zastosowania wskaźników

Wskaźniki stosuje się w sytuacjach, gdy chodzi nam o:

- ❖ – ulepszenie pracy z tablicami,
- ❖ – funkcje mogące zmieniać wartość przysyłanych do nich argumentów,
- ❖ – dostęp do specjalnych komórek pamięci,
- ❖ – rezerwację obszarów pamięci.

W dalszej części tego rozdziału przyjrzymy się przykładom z tych czterech dziedzin zastosowań.

## 8.7 Zastosowanie wskaźników wobec tablic

### 8.7.1 Ćwiczenia z mechaniki ruchu wskaźnika

Jeśli mamy następujące definicje:

```
int *wskaznik ; // definicja wskaźnika
int tablica[10] ; // definicja tablicy
```

to instrukcja

```
wskaznik = & tablica[n] ; // ustawienie wskaźnika
```

powoduje, że wskaźnik ustawia się na elemencie tablicy o indeksie  $n$ . Wskaźnik nasz jest, jak wiadomo, wskaźnikiem do pokazywania na obiekty typu `int`. Elementy naszej tablicy są właśnie typu `int`, więc wszystko się zgadza. W naszej ostatniej instrukcji po prostu wstawiamy do wskaźnika adres  $n$ -tego elementu tablicy.

Instrukcja

```
wskaznik = & tablica[0] ;
```

jest ustawieniem wskaźnika na zerowym elemencie, czyli na początku tablicy. Ponieważ (już kiedyś mówiliśmy o tym, a Ty obiecałeś powiesić sobie to nad biurkiem) – nazwa tablicy jest równocześnie adresem jej początku, dlatego równie dobrze moglibyśmy tę ostatnią instrukcję napisać tak:

```
wskaznik = tablica ;
```

A teraz uważaj, będzie coś bardzo ciekawego: Jeśli ustawiliśmy wskaźnik na żądany element tablicy, np. tak:

```
wskaznik = &tablica[4] ;
```

to, aby go potem przesunąć tak, by pokazywał na następny element tej tablicy wystarczy do niego dodać 1

```
wskaznik = wskaznik + 1 ;
```

czyli krócej

```
wskaznik++ ;
```

To jest właśnie ta prostota przesunięcia w rozkładzie jazdy palca o jedną kratkę, by odczytać następny pociąg. Aby wskaźnik przesunąć o  $n$  elementów wystarczy instrukcja

```
wskaznik += n ; // inaczej : wskaznik = wskaznik + n ;
```

Jest to bardzo ważny fakt:

Dodanie do wskaźnika jakiejś liczby całkowitej powoduje, że pokazuje on tyleż elementów tablicy dalej. Niezależnie od tego jakie są te elementy.

Nie jest to takie trywialne, bo przecież mogą być tablice typu `float`, których elementy zajmują w pamięci większą przestrzeń niż np. typu `int`. A jednak wskaźnik jest na tyle inteligentny, że wie jak się ma przesunąć, aby przeskoczyć o zadaną liczbę elementów.

Skąd to wie? Ze swojej własnej definicji! Jest przecież wskaźnikiem do pokazywania na obiekty jakiegoś wybranego typu. Wiedząc jakim typem ma się zajmować – zna rozmiar jednego elementu i może przyjąć na to poprawkę.

Zobaczmy teraz na przykładzie, jak sprytnie jest przesuwanie wskaźników

Poniższy program służy do wydruku adresu, na który pokazują wskaźniki. Interesuje nas tutaj tylko adres, na który wskaźnik pokazuje. Chwilowo nie zajmujemy się tym, co pod owym adresem się kryje.

```
#include <iostream.h>
main()
{
 // definicje dwóch tablic ❶
 int ti[6] ; // jedna int
 float tf[6] ; // druga float
 // dwa wskaźniki ❷
 int *wi ; // do pokazywania na obiekty int
 float *wf ; // do pokazywania na obiekty float

 wi = &ti[0] ; // inaczej wi = ti ; ❸
 wf = &tf[0] ; // inaczej wf = tf ; ❹

 cout << "Oto jak przy inkrementacji wskaźników\n "
 << "zmieniają się ukryte w nich adresy :\n" ;
 for(int i = 0 ; i < 6 ; i++, wi++, wf++) // ❺
 {
 cout << "i= " << i
 << ") wi = "
 << (unsigned long) wi // ❻
 << " , wf = "
 << (unsigned long) wf << endl ; // ❼
 }
}
```



## Po uruchomieniu takiego programu na komputerze klasy IBM PC/AT na ekranie pojawi się

```
Oto jak przy inkrementacji wskaźników
zmieniają się ukryte w nich adresy :
i= 0) wi = 1087246316, wf = 1087246292
i= 1) wi = 1087246318, wf = 1087246296
i= 2) wi = 1087246320, wf = 1087246300
i= 3) wi = 1087246322, wf = 1087246304
i= 4) wi = 1087246324, wf = 1087246308
i= 5) wi = 1087246326, wf = 1087246312
```

- ❶ Dwie definicje tablic. Tablica `ti` ma elementy typu `int`. Tablica `tf` ma elementy typu `float`.
- ❷ Definicje wskaźników. Wskaźnik `wi` może pokazywać na obiekty typu `int`, wskaźnik `wf` może pokazywać na obiekty typu `float`.
- ❸ i ❹ Nadanie wartości początkowych wskaźnikom. Po prostu wstawia się do nich adresy obiektu, na który mają pokazywać. Wskaźnik `wi` ma pokazywać na zerowy element tablicy `ti`, natomiast wskaźnik `wf` ma pokazywać na zerowy obiekt tablicy `tf`. Skoro mają pokazywać na początki tych tablic, to równie dobrze można użyć zapisu, który podany jest w komentarzu.
- ❺ Pętla. Nie ma w niej nic niezwykłego poza tym, że po każdym obiegu wykonywane są:

```
wi++; wf++;
```

- ❻ Wypisanie na ekran adresu, na który wskaźnik `wi` pokazuje. Adres to jakaś liczba. To, jak adresowane są komórki pamięci w danym komputerze, zależy od typu komputera. My tutaj przez operację rzutowania chcemy to wypisać tak, jakby to była wartość typu `unsigned long`.
- ❼ Wypisanie tego samego o wskaźniku `wf`

Zauważ, że na ekranie liczby symbolizujące adresy zmieniają się z różnym skokiem mimo, że przecież dodawaliśmy do wskaźnika tylko jedynki. W tym właśnie objawia się inteligencja wskaźnika. Wie on jak naprawdę ma się zmienić po to, by wskazać na kolejny element tablicy.

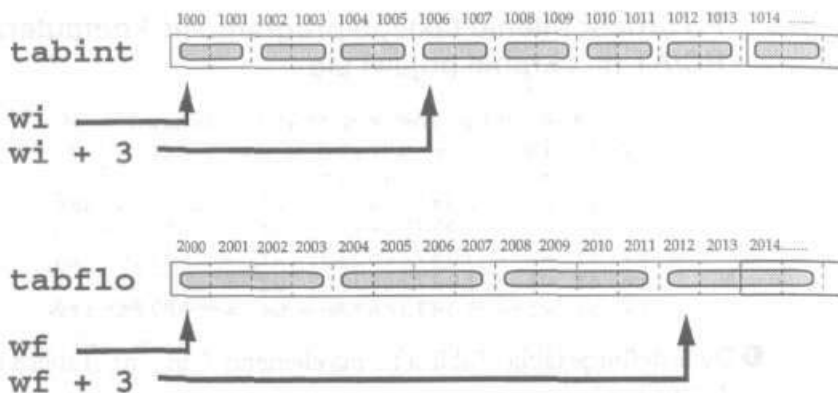
Skąd wskaźnik wie na jaki typ pokazuje? Stąd, że przecież go zdefiniowaliśmy jako: „wskaźnik służący do pokazywania na elementy typu `int`”

*W miejscu wypisywania wskaźnika na ekranie widzisz*

```
cout << (unsigned long) wi ;
```

*przypominam, że operacja ujęta w nawias nazywa się rzutowaniem (casting). Mówiliśmy o niej na str 70. Przypomnę jednak, że oznacza ona mniej więcej coś takiego: „co prawda `wi` jest to wskaźnik, czyli w zasadzie adres, ale przeczytajmy go i zamieńmy na liczbę typu `unsigned long`” (i taką liczbę wydrukujemy na ekranie).*

Przyjrzyjmy się poniższemu obrazkowi. (Liczby nad tablicami oznaczają przykładowe adresy poszczególnych komórek pamięci (bajtów) zajmowanych przez elementy tablic. Ze względów estetycznych wybrałem bardziej okrągłe liczby, a nie te, które pokazał nasz ostatni program).



Elementy tablicy typu float zajmują więcej miejsca w pamięci niż elementy typu int. Jednakże wskaźnik, który jest przeznaczony do pokazywania na dane typu int wie, jak ma się zmieniać przy przechodzeniu do sąsiedniego elementu tablicy. Podobnie wskaźnik przeznaczony do pokazywania na elementy typu float wie, jak zachować się przy przejściu do następnego elementu tablicy float.

Podobnie wyrażenie

$$(wf + 3)$$

daje w sumie adres trzeciego z kolei elementu tablicy za tym, na który właśnie wskazuje wskaźnik **wf**, a wyrażenie

$$(wf - 1)$$

jest adresem elementu o jeden wcześniejszego.<sup>†)</sup>



Wniosek stąd taki:

Wskaźnik to nie tylko adres jakiegoś miejsca w pamięci. To także wiedza o tym, na jaki typ obiektu pokazuje. Wiedza ta wykorzystywana jest przy:

- interpretowaniu tego, na co wskaźnik pokazuje,
- ewentualnym poruszaniu wskaźnika

Przykład:

ad a) Wyobraź sobie bowiem, że wskaźnik pokazuje na jakiś bajt w pamięci<sup>††)</sup>. Jeśli wskaźnik jest typu **int** to wiadomo, że ten bajt i następny należy interpretować jako zakodowaną liczbę typu **int**. Jeśli zaś na ten sam bajt pamięci pokazywałby wskaźnik typu

†) W obu wypadkach sami musimy zadbać o legalność naszych poczynań. Jeśli **wf** pokazuje na początek tablicy, to wyrażenie  $(wf - 1)$  jest jakby adresem nieistniejącego elementu o indeksie: -1 (minus 1). O tych sprawach pomówimy za chwilę przy arytmetyce wskaźników.

††) Szesnastobitowego komputera IBM PC/AT

float, to wiadomo, że ten bajt i trzy następne (razem cztery) należy rozumieć jako zakodowaną liczbę typu float.

ad b) Kontynuując powyższy przykład: W przypadku tablicy typu int, aby przesunąć się do następnego elementu typu int - trzeba przeskoczyć o te dwa bajty. W przypadku tablicy typu float trzeba przeskoczyć rzeczone 4 bajty.

Wskaźniki są tak wspaniale pomyślane, że nie musimy o tych sprawach wiedzieć. Dodanie 1 do wskaźnika przesunęło go o właściwą ilość bajtów i pokazuje on na następny element.

Tak więc pamiętamy odtąd, że wyrażenie

```
(wi++)
```

oznacza przesunięcie wskaźnika do następnego elementu tablicy. Co ciekawsze – jeśli w przyszłości będziemy sobie wymyślali tablice z elementów mających po kilkaset bajtów, to i tak dodanie jedynki do wskaźnika służącego do pokazywania na tę tablicę przesunie poprawnie wskaźnik na następny element.

Dygresja:

*Często mówiłem tutaj: wskaźnik „wie” – nadając wskaźnikowi jakąś osobowość i wolność działania. Oczywiście to nie wskaźnik „wie”, tylko kompilator. Kompilator nasze wyrażenie wi++ zamienia na dodanie odpowiedniej liczby do obecnie tkwiącego w wi adresu. Czyli to kompilator jest tak mądry. Jednak – wybac mi tę słabość - lepiej odpowiada mi interpretacja nadająca pewne cechy osobowości wskaźnikom. Życie w świecie takich wyobrażeń, gdzie wskaźnik to stary znajomy, którego przyzwyczajenia zna się na wylot, uprzyjemnia programowanie.*

## 8.7.2 Użycie wskaźnika w pracy z tablicą

Skoro już wiemy, jaki jest mechanizm przesuwania wskaźników – zobaczmy jak korzystać z tego, na co wskaźnik bieżąco pokazuje.

Zacznijmy od programu. Będzie to program na proste operacje na tablicach, jednak wykonywane będą one za pomocą wskaźników.

```
#include <iostream.h>
main()
{
 int *wi ;
 float *wf ;
 int tabint[10] = { 0,1,2,3,4,5,6,7,8,9 } ; // ❶
 float tabflo[10] ; // ❷
 // ustawienie wskaźnika
 wf = &tabflo[0] ; // ❸

 // załadowanie tablicy float wartościami początkow.
 for(int i = 0 ; i < 10 ; i++)
 {
 *(wf++) = i / 10.0 ; // tablica float ❹
 }

 cout << "Tresc tablic na poczatku\n" ;
```





## Oto ciekawsze szczegóły programu:

- ❶ Definicja tablicy typu `int`. Od razu następuje inicjalizacja.
- ❷ Definicja tablicy typu `float`. Nie inicjalizujemy.
- ❸ Ustawienie wskaźnika na początkowy element tablicy.
- ❹ Po ustawieniu wskaźnika posługujemy się nim w pętli wpisującej do tablicy `tabflo` wartości początkowe. Wpisanie dzieje się za sprawą instrukcji

```
* (wf++) = i / 10.0 ;
```

co oznacza inaczej

```
*wf = i / 10.0 ;
```

```
wf ++ ;
```

Odbywa się więc tutaj wpisanie liczby do elementu tablicy pokazywanego właśnie przez wskaźnik `wf`, a następnie przesunięcie tego wskaźnika na element następny.

- ❺ Jest to pętla wypisująca na ekran treść tablic. Ponieważ chcemy to robić metodą przesuwania wskaźników, dlatego ustawiamy je na początkach tablic. Zauważ, że instrukcje ustawiające wskaźniki są wewnątrz instrukcji `for`. Dokładniej mówiąc w tej części, która jest wykonywana przed pierwszym obiegiem pętli. Zastosowaliśmy tu dla odmiany inny sposób ustawiania wskaźnika

```
wi = tabint ;
```

Sposób ten może nie jest tak od razu jasny, ale (pamiętasz?) umówiliśmy się, że przykleisz sobie nad biurkiem kartkę z napisem:

**Nazwa tablicy jest równocześnie adresem jej zerowego elementu**

To tłumaczy wszystko. Po prostu wyrażenia `(tabint)` oraz `(&tabint[0])` są równoważne.

- ❻ Wypisanie na ekran elementu pokazywanego przez wskaźnik to – jak wiemy – instrukcja typu

```
cout << *wi ;
```

- ❼ Przesunięcie wskaźników do następnego elementu. Mogliśmy to zrobić także zwięźle w poprzedniej linijce operatorem postinkrementacji zapisując

```
cout << *(wi++) ;
```

- ❽ Zamierzamy w wybrane miejsca tablic wpisać nowe wartości. Skoro chcemy użyć do tego celu szybkiego sposobu za pomocą wskaźnika, dlatego musimy najpierw ustawić sobie wskaźnik. W wypadku `tabint` ustawiamy wskaźnik na elemencie `tabint[5]`.

- ❾ Natomiast w wypadku tablicy `tabflo` zacząć się to ma od elementu o indeksie 2. Zastosowaliśmy tu inny zapis, cały czas pamiętając, że przecież

```
wf = tabflo + 2 ;
```

to to samo co:

```
wf = &tabflo[2] ;
```

- ⑩ Wpisanie w te miejsca pamięci, gdzie pokazują wskaźniki. Przy wpisaniu od razu dokonujemy postinkrementacji wskaźników. Wpisujemy, jak widać, do kolejnych czterech elementów tablic. Następne linijki programu to po prostu pokazanie efektów pracy na ekranie.



Widzisz więc, że nie ma nic specjalnie trudnego w posługiwaniu się wskaźnikami. Zapytasz pewnie: „–A właściwie po co to wszystko, skoro da się to zrobić starym sposobem posługując się tablicami?” Masz rację. Mówiłem przecież, że bez wskaźników można żyć. Jest jednak zaleta. Posłużenie się wskaźnikiem da szybszy program. Pamiętasz naszą przypowieść o rozkładzie jazdy? To właśnie przecież robiliśmy tutaj. Na każde pytanie „–A następny?” odpowiadaliśmy instrukcją `wi++` i już byliśmy przy następnym elemencie. Będąc przy elemencie `tabint[5]` jednym skokiem znajdowaliśmy się przy elemencie `tabint[6]`, bez żmudnego liczenia adresu. Liczenie adresu przecież chwilę trwa.

## Nazwa tablicy a wskaźnik

A teraz wróćmy jeszcze do sprawy omawianej w punkcie ⑨. Skoro dwie poniższe instrukcje ustawiające wskaźnik na początku tablicy są równoważne

```
wskaznik = &tablica[0] ;
wskaznik = tablica ;
```

czyli, że wskaźnikowi można przypisać nazwę tablicy, to właściwie zachowuje się ona tak, jak wskaźnik. Istotnie – są ogromne podobieństwa. Zapis

```
tablica + 4
```

oznacza to samo co

```
&tablica[4]
```

Czy zatem zamiast naszej złotej regułki:

Nazwa tablicy jest równocześnie **adresem** jej zerowego elementu  
nie powinniśmy zastąpić regułą:

Nazwa tablicy jest równocześnie **wskaźnikiem** do jej zerowego elementu

Tak, możemy to zrobić pod warunkiem, że będziemy pamiętać, iż to nie zwykły wskaźnik, tylko taki, którego nigdy nie będziemy przesuwac. (czyli: `const`)

Pokażmy tę różnicę jaśniej. Po instrukcji

```
wskaznik = tablica ;
```

`wskaznik` pokazuje na początek tablicy. O ile możemy postąpić tak:

```
wskaznik ++ ;
```

czyli przesunąć wskaźnik tak, by pokazywał na element następny, o tyle operacja

```
tablica ++ ; // błąd !!!
```

jest niedopuszczalna. A zatem nasza złota regułka powinna brzmieć tak:

Nazwa tablicy jest jakby stałym wskaźnikiem do jej zerowego elementu.

Mimo tej bardzo mądrej konkluzji nie radzę napisanej przedtem złotej regułki zmieniać i zastępować nową. Pojęcie „adres” przeraża mniej niż pojęcie „stały wskaźnik”. Przynajmniej na początku.

Oto dalsze konsekwencje. Skoro nazwa tablicy to właściwie wskaźnik, więc poniższe wyrażenia są równoważne

```
tablica[3]
*(tablica+3)
```

Wartością obu jest treść tego elementu tablicy, który ma indeks 3

Nie posądzaj mnie też, że w drugim z tych wyrażen dokonuje się zabronionego przesuwania wskaźnika-nazwy tablicy. Dodanie +3 nie jest przesunięciem, tylko powiedzeniem, że chodzi nam o trzy pozycje dalej niż on teraz pokazuje.

Oto ilustracja: Gdy znajomi pytają mnie, w którym miejscu na długiej ulicy Bülow StraÙe mieszkam, wówczas mówię, że tam, gdzie chińska restauracja, tylko trzy domy dalej. Wskaźnikiem jest tutaj „chińska restauracja”. „Trzy domy dalej” – to operacja, którą przeprowadzamy w myśli, bez rujnowania chińskiej restauracji.

Inną różnicą między wskaźnikiem, a nazwą tablicy jest to, że wskaźnik jest jakimś obiektem w pamięci, więc można ustalić jego własny adres. Nie to, na co ten wskaźnik pokazuje, ale to, gdzie sam się mieści. Robi się to starym sposobem. Jeśli mamy wskaźnik

```
int *wskaznik ; // definicja wskaźnika
```

to jego adres jest wartością wyrażenia

```
&wskaznik
```

Natomiast nie można takiej operacji przeprowadzić w stosunku do nazwy tablicy.



Nazwa nie jest obiektem. Tablica tak, ale nazwa nie. **To też dobrze zapamiętać.** Ja mam na imię Jurek. Sam jestem obiektem (materialnym), natomiast moje imię obiektem nie jest. (Jeśli ktoś na mnie pokaże patykiem, to ten patyk-wskaźnik jest obiektem materialnym).

### 8.7.3

## Arytmetyka wskaźników

Mówiliśmy już, że można dodawać i odejmować liczby całkowite do wskaźników i w ten sposób przesunąć je po wskazywanej tablicy. Jest tu jednak podobna sytuacja jak z odnoszeniem się do nich za pomocą konwencjonalnego – „tablicowego” zapisu: Nie jest sprawdzana legalność takiej operacji. To znaczy:



jeśli tablica ma tylko 10 elementów, a my wskaźnik aktualnie pokazujący na element `tablica[5]` przesuniemy o 80 elementów dalej, to wskaźnik będzie pokazywał na nieistniejący element `tablica[85]`. Możemy to zinterpretować tak: będzie on pokazywał na takie miejsce w pamięci, które zajmowałby element `tablica[85]`, gdybyśmy tylko zdefiniowali byli tak dużą tablicę. Ponieważ jednak tego nie zrobiliśmy – w miejscu tym są zupełnie inne dane. Jeśli mimo wszystko spróbujemy przeczytać to miejsce wskazywane przez wskaźnik, otrzymamy bezsensowny wynik, natomiast próba zapisania tam czegoś – zniszczy istniejącą tam legalnie inną daną.

O błędzie takim nie ostrzeże nas kompilator. Legalność pokazywania wskaźnikiem nie jest bowiem sprawdzana. Na dodatek, niekoniecznie od razu po zniszczeniu, program może wykazać błędne działanie. Dopóki nie korzystamy z tej zniszczonej danej – dotąd wszystko wydaje się w porządku.

Takie błędy są najtrudniejsze do znalezienia, bowiem objawiają się w programie czasem bardzo daleko od miejsca, w którym je spowodowaliśmy.

Oto moja rada:

Jeśli program z zupełnie niewiadomych przyczyn błędnie działa czy zawiesza się, to zastanów się nad wskaźnikami. Najczęściej okaże się, że gdy wpisywałeś coś w miejsce pamięci pokazywane przez wskaźnik – pokazywał on na niewłaściwe miejsce lub (o zgrozo!) w ogóle zapomniałeś nadać wskaźnikowi jakąkolwiek wartość początkową. W rezultacie pokazywał on w przypadkowe miejsce, a Ty coś tam zapisałeś niszcząc coś nieznanego.

## Oprócz tych operacji można też wskaźniki od siebie odejmować

Co z takiej operacji wynika? Zastanówmy się: jeśli mamy wskaźnik `wa`, który pokazuje na piętnasty element tablicy, oraz wskaźnik `wb`, który pokazuje na dziesiąty element tablicy, to jaka jest różnica między tymi wskaźnikami? Czyli jaka jest wartość wyrażenia

$(wb - wa)$

Zdrowy rozsądek podpowiada: 5 elementów. Rzeczywiście wartością tego wyrażenia jest liczba 5.

Odjęcie od siebie dwóch wskaźników pokazujących na różne elementy tej samej tablicy daje w rezultacie liczbę dzielących je elementów tablicy. Liczba ta może być ze znakiem ujemnym lub dodatnim.

Oto prosty przykład:

```
#include <iostream.h>

main()
{
 int tablica[15]; // definicja tablicy
 int *wsk_a, *wsk_b, *wsk_c;

 wsk_a = &tablica[5];
 wsk_b = &tablica[10];
```



```

wsk_c = &tablica[11] ;

cout << " (wsk_b - wsk_a) = " << (wsk_b - wsk_a)
<< "\n(wsk_c - wsk_b) = " << (wsk_c - wsk_b)
<< "\n(wsk_a - wsk_c) = " << (wsk_a - wsk_c)
<< "\n(wsk_c - wsk_a) = " << (wsk_c - wsk_a) ;
}

```



## W rezultacie jako wynik otrzymamy

```

(wsk_b - wsk_a) = 5
(wsk_c - wsk_b) = 1
(wsk_a - wsk_c) = -6
(wsk_c - wsk_a) = 6

```

Czy zauważyłeś, że wielokrotnie podkreślałem, iż muszą to być wskaźniki pokazujące na tę samą tablicę? Dlaczego to takie ważne?

## Przykład z geografii

Na ścianie wisi mapa Europy. Jednym, drewnianym wskaźnikiem pokazujemy na tej mapie na Paryż, a drugim wskaźnikiem na Berlin. Jaka jest różnica tych wskaźników? (odległość między tymi wskaźnikami). Odpowiedź jest na przykład taka: 28 centymetrów, co przy uwzględnieniu podziałki mapy może oznaczać tyle-set kilometrów. Przyznasz, że ta odpowiedź ma sens.

A teraz dodatkowo na drugiej ścianie zawieszamy inną mapę, na przykład mapę nieba. Jednym drewnianym wskaźnikiem pokazujemy na Rzym, a drugim na gwiazdozbiór Oriona. Jak jest różnica tych wskaźników (odległość między tymi wskaźnikami)? To pytanie nie ma sensu. Mapy mają inne skale, wiszą na przypadkowych ścianach. Jeśli nawet weźmiemy taśmę mierniczą i zmierzymy tę odległość, to co ona będzie oznaczać?

Pamiętajmy więc -

Odejmowanie wskaźników daje wynik, który można sensownie zinterpretować tylko wtedy, gdy te wskaźniki pokazują na elementy w tej samej tablicy.

Nie ma sensu mnożenie dwóch wskaźników (nawet pokazujących na tę samą tablicę) – no bo co by to miało oznaczać? Nie ma też sensu dzielenie itd.

Podsumujmy:

Legalnymi operacjami arytmetycznymi na wskaźnikach są:

- 1) dodawanie i odejmowanie od nich liczba naturalnych – daje to przesuwanie wskaźników,
- 2) odejmowanie dwóch wskaźników pokazujących na tę samą tablicę.

## 8.7.4

## Porównywanie wskaźników

Wskaźniki można ze sobą porównywać. Dla porównania dwóch wskaźników posługujemy się operatorami

```

== != < > <= >=

```

Jeśli mamy dwa wskaźniki

```
int *wsk1, *wsk2 ;
```

i zostały one już ustawione tak, że pokazują na jakieś obiekty, to równość tych wskaźników oznacza, że pokazują one na ten sam obiekt.

```
if((wsk1 == wsk2)
 cout<< "oba wskazniki pokazuja na to samo !" ;
```

Jeśli wskaźniki są różne to znaczy, że pokazują na różne obiekty.

```
if (wsk1 != wsk2)
 cout << "wskazniki pokazuja na rozne obiekty" ;
```

Zwracam uwagę, że elementy tablicy to są też obiekty. Element piąty jest innym obiektem niż element szósty. Jeżeli dwa wskaźniki pokazują na jakieś elementy tej samej tablicy, to wskaźnik, który jest mniejszy pokazuje na obiekt o mniejszym indeksie.

```
#include <iostream.h>
main()
{
 int tablica[5] ;
 int *wsk_czer, *wsk_ziel ;
 int i ;

 wsk_czer = &tablica[3] ;
 cout << "Mamy piecioelementowa tablice \n"
 "Wskaznik czerwony pokazuje na "
 "element o indeksie 3\n"
 "Na ktory element ma pokazywac "
 "wskaznik zielony ? (0-4) : " ;
 cin >> i ;

 if(i < 0 || i > 4)
 cout <<
 "\nNie ma takiego elementu w tej tablicy !" ;
 else
 {
 wsk_ziel = &tablica[i] ;
 cout <<
 "\nZ przeprowadzonego porownania wskaznikow\n"
 "czerwonego z zielonym wynika, ze : \n" ;

 // właściwa akcja porównania
 if(wsk_czer > wsk_ziel) {
 cout << "zielony pokazuje na element "
 "blizej poczatku tablicy" ;
 }
 else if(wsk_czer < wsk_ziel){
 cout << "zielony pokazuje na element "
 "o wyzszyim indeksie" ;
 }else{ // czyli: wsk_czer == wsk_ziel
 cout << "zielony i czerwony pokazuja "
 "na to samo\n" ;
 }
 }
}
```

**Po wykonaniu tego programu i przykładowej odpowiedzi 4, na ekranie pojawi się tekst:**

```
Mamy piecioelementowa tablice
Wskaznik czerwony pokazuje na element o indeksie 3
Na ktory element ma pokazywac wskaznik zielony ? (0-4) : 4
Z przeprowadzonego porownania wskaznikow
czerwonego z zielonym wynika, ze :
zielony pokazuje na element blizej konca tablicy
```

W przykładzie nie ma użycia operatorów `<=` i `>=`, ale ich znaczenie jest chyba oczywiste.

Warto znowu przypomnieć, że operacje

`>`   `<`   `>=`   `<=`

mają sens tylko dla wskaźników pokazujących na tę samą tablicę. Powód jest dokładnie taki sam, jak przy arytmetyce wskaźników.

Jeśli tylko wskaźniki są tego samego typu – czyli inaczej: służą do pokazywania na obiekty tego samego typu (`int` albo `char` itd.) – i pokazują one na obiekty nie należące do tej samej tablicy, to mimo powyższego zastrzeżenia wolno nam je porównać. Wolno nam także porównać wskaźniki pokazujące właśnie na zupełnie odosobnione zmienne. Jest tylko kwestia jaki sens ma takie porównanie. A sens jakiś ma! Dowiadujemy się w ten sposób jak w pamięci komputera ulokowane są względem siebie te obiekty. Wynik i jego nasza interpretacja zależy tutaj od konkretnego kompilatora, którym się posługujemy.

Każdy wskaźnik można porównać z adresem 0 zwanym czasem `NULL`. Jest to przydatna właściwość, bo ustawienia wskaźnika na ten adres często służy nam by zaznaczyć, że wskaźnik nie pokazuje na nic sensownego. Wpisujemy tam świadomie 0 :

```
wsk = 0 ; // lub : wsk = NULL ;
```

Potem możemy to ewentualnie łatwo sprawdzić

```
if(wsk == 0)
 cout << "Wskaznik nie pokazuje na nic sensownego !" ;
```

to samo sprawdzenie wskaźnika można wykonać jako

```
if(wsk == NULL) ...
```

albo jeszcze prościej

```
if(! wsk) ...
```

## 8.8 Zastosowanie wskaźników w argumentach funkcji

Mówiliśmy, że są 4 domeny zastosowania wskaźników. Kolejną, którą się teraz zajmiemy, to wskaźniki jako argumenty funkcji. W zasadzie mówiliśmy już o tym trochę w rozdziale o funkcjach. Tutaj rozszerzymy ten temat, ale najpierw

### Przypomnienie

Jeśli mamy funkcję z jednym argumentem

```
int funkcja(int argum);
```

i gdy wywołamy ją tak:

```
int a, x = 5 ;
a = funkcja(x) ;
```

to funkcja ta otrzymuje wówczas do pracy kopię zmiennej  $x$  (a nie oryginał). Jeżeli nawet w obrębie funkcji na tej kopii dokonujemy jakichś zmian, to w momencie opuszczania funkcji jest ona likwidowana. Funkcja więc nie może dokonać zmian zmiennej przysłanej do niej jako argument – przez wartość. (Czy pamiętasz jeszcze tę przypowieść z fotografią teściowej?).

Innymi słowy, gdybyśmy np. chcieli mieć funkcję, która przysłała do niej parametr zwiększony o 130, a jej treść (czyli definicję) napisali tak:

```
void funkcja(int foto)
{
 foto += 130 ; // czyli: foto = foto + 130 ;
}
```

to wywołanie funkcji

```
int m = 10 ;
funkcja(m) ;
```

nie spowoduje jakiegokolwiek zmiany  $m$ .

Co się tu dzieje ?

Zmienna zostaje sfotografowana, a jej fotografia znajdzie się w podręcznym magazynku funkcji (na stosie). W trakcie działania funkcji do tej fotografii zostaje dodana liczba 130, więc obiekt na stosie ma teraz już wartość 140.

Ponieważ funkcja nie ma już więcej nic do roboty, więc kończy się ją uprzątając wszystkie śmieci ze stosu. Wtedy to nasza fotografia przestaje istnieć. Gdy wróciliśmy z funkcji patrzymy na prawdziwą zmienną  $m$  – jest nietknięta. Bawiliśmy się jej kopią, która została zniszczona.

Przypomnieliśmy tutaj mechanizm przesyłania argumentów przez wartość. Tak przesyłane są zwykłe obiekty. (O tablicach mówiliśmy już, że jest inaczej).

Co zatem zrobić, by funkcja mogła obiekt  $m$  zmienić ?

Najprostsze i najbardziej zalecane wyjście jest takie, by funkcja jako swój rezultat zwracała wartość, którą my świadomie wpisujemy do obiektu  $m$ :

Oto definicja takiej funkcji:

```
int fun2(int foto)
{
 foto += 130 ;
 return(foto) ;
}
```

A tak tę funkcję wywołujemy w programie:

```
int m = 10 ;
m = fun2(m) ;
cout << m ;
```

Po wykonaniu tego fragmentu na ekranie pojawi się liczba 140.

Co się tu odbywa: fotografowana jest zmienna `m` i jej wartość (czyli liczba 10) jest umieszczana na stosie jako obiekt typu `int` o nazwie `foto`. Powstał więc lokalny obiekt automatyczny. Następnie do tego obiektu `foto` dodawane jest 130, co powoduje, że w rezultacie w obiekcie `foto` jest teraz wartość 140.

Teraz funkcja się kończy – instrukcją `return`. To, co stoi obok słowa `return`, jest to wyrażenie, którego wartość staje się rezultatem funkcji. W naszym wypadku to wyrażenie składa się tylko z jednej zmiennej. Jego wartość to 140. Ta właśnie liczba zamieniana jest na typ deklarowanego rezultatu zwracanego przez funkcję. U nas deklarowaliśmy, że funkcja zwraca wartość typu `int`, więc konwersja jest w zasadzie niepotrzebna. (Gdybyśmy mieli jednak 140.1 to nastąpiłoby obcięcie do `int` czyli wartość 140). Po konwersji wartość tę zapamiętuje się w jakimś tajemniczym miejscu. Zaczynamy sprzątać śmieci ze stosu. Likwidowany jest więc obiekt `foto`. Wracamy do miejsca w programie skąd wywołaliśmy funkcję. Widzimy tam, że rezultat funkcji ma zostać przypisany obiektowi `m`, w którym nadal tkwi wartość 10. Odszukujemy schowany w bezpiecznym miejscu rezultat funkcji (140) i wstawiamy go do obiektu `m`. W tym momencie odbyła się modyfikacja obiektu `m`.

Sposób jest bardzo dobry dlatego także, iż patrząc na zapis

```
m = fun2(m) ;
```

od razu widzimy, że do zmiennej `m` wpisuje się coś nowego, więc zmiana jego wartości nie jest dla nas niespodzianką. Takie względy są bardzo ważne przy analizie cudzych programów, a nawet swoich własnych, które przy kilkunastu tysiącach linii już trudno nam opanować.

Co jednak zrobić w wypadku, gdy funkcja ma zmienić więcej niż jeden obiekt?

Opisanego wyżej sposobu zastosować się nie da, dlatego, że funkcja za pomocą instrukcji `return` zwraca tylko jedną wartość. Tu właśnie przydają się wskaźniki. Otóż zamiast wysłać do funkcji kopię naszej zmiennej wysyłamy... Pewnie pomyślałeś „oryginał”.

Nie, tego zrobić nie można. Wysyłanie argumentów do funkcji jest jakby napisaniem do niej listu.

Weźmy taki obrazek: W łazience zepsuł się nam kran. Piszemy list do hydraulika. Nie możemy mu w liście przesłać tego kranu. Mamy dwa wyjścia:

- ❖ 1) Możemy mu przesłać fotografię tego zepsutego kranu. Wtedy hydraulik znajdzie u siebie w warsztacie na stosie rupieci taki sam kran (kopię naszego). Jeśli nawet sobie go naprawi, to nasz własny kran i tak będzie zepsuty.
- ❖ 2) Możemy mu w liście jednak przesłać – uwaga, uwaga! : ADRES tego naszego zepsutego kranu – nazwę ulicy numer domu, piętro i to, gdzie w mieszkaniu jest ten kran. Piszemy mu w liście, że ma naprawić kran będący pod takim adresem.

Oto jak to przełożyć na język programowania:

```
#include <iostream.h>
void hydraulik(int *wsk_do_kranu) ; // ❶
/***** */
main()
{
 int kran = -1 ; // ❷

 cout << "Stan techniczny kranu = " << kran << endl ;
 hydraulik(&kran) ; // ❸
 cout <<
 "Po wezwaniu hydraulika stan techniczny kranu = "
 << kran << endl ; // ❹

}
/***** */
void hydraulik(int *wsk_do_kranu) // ❺
{
 *wsk_do_kranu = 100 ; // akcja naprawiania ❻
} // ❼
```



**Po wykonaniu tego programu na ekranie pojawia się:**

```
Stan techniczny kranu = -1
Po wezwaniu hydraulika stan techniczny kranu = 100
```



**Przyjrzyjmy się ciekawszym punktom**

- ❶ Deklaracja funkcji hydraulik. Czytamy ją tak: hydraulik jest funkcją wywołowaną z jednym argumentem będącym wskaźnikiem do obiektu typu int. Funkcja ta zwraca typ void, czyli nic nie zwraca.
- ❷ Definicja obiektu typu int o nazwie kran. Wstawione tam od razu -1 oznacza, że kran jest bardzo zepsuty.
- ❸ Wywołanie funkcji hydraulik, której definicja jest w ❹. Prześledźmy co tu się zdarza. Otóż wzywamy hydraulika podając mu listownie adres zmiennej (operator jednoargumentowy & oznacza jak wiadomo „adres“). Co z tym adresem robi hydraulik?
- ❹ Widzimy, że przysłany adres służy hydraulikowi do inicjalizacji wskaźnika. Tak — boiem hydraulik definiuje sobie (na stosie) wskaźnik do obiektu typu int i daje mu nazwę wsk\_do\_kranu. Odebrany od listonosza adres wstawia właśnie do tego wskaźnika. Odtąd więc jego prywatny (lepiej: lokalny) wskaźnik pokazuje na nasz kran.

- ⑤ Do obiektu pokazywanego przez wskaźnik wstawia się liczbę 100. Wskaźnik pokazuje na nasz kran, więc to do naszego kranu wstawia się tę wartość 100. To jakby symbolizuje naprawę naszego kranu.
- ⑥ Następnie opuszcza się funkcję, likwiduje się śmieci czyli zniszczony zostaje wskaźnik do naszego kranu – hydraulik podarł niepotrzebną mu już kartkę z adresem.
- ⑦ Na dowód, że naprawa została dokonana naprawdę na naszym kranie, wypisujemy jego zawartość na ekran.  
Gdybyśmy za chwilę wywołali funkcję hydraulik podając jej adres innego obiektu typu `int`, to zadziała ona na innym obiekcie. Jak w życiu: hydraulik naprawia jeszcze inny kran

```
int kurek = -10 ;
hydraulik(&kurek) ;
```

czyli wstawi on liczbę sto do obiektu `kurek` (bowiem adres tego obiektu właśnie wysłaliśmy).

Jeśli natomiast mamy całą baterię kurków

```
int bateria[15] ;
```

i chcemy naprawić czwarty kurek z tej baterii (tablicy), to wywołujemy

```
hydraulik(&bateria[4])
```

Chyba Cię ten zapis nie dziwi, oswoiłeś się zapewne z tym, że tak zapisuje się adres elementu tablicy.

Naprawa elementów od 4 do 8 tej baterii może zostać zrealizowana przez

```
for(int i = 4 ; i <= 8 ; i++){
 hydraulik(&bateria[i]);
}
```

## 8.8.1 Jeszcze raz o przesyłaniu tablic do funkcji

Jak pamiętasz, gdy do funkcji wysyła się tablicę jako całość, to wysyłane nie są kopie wszystkich jej elementów, ale po prostu jej adres. To też sprawia, że mając adres funkcja może pracować na oryginalnych jej elementach.

Jeśli jednak wysyłamy do funkcji jeden element tablicy, lub kilka - w każdym razie nie całość – to funkcja traktuje je jak zwykłe obiekty przesłane przez wartość. Jeśli chcemy wysłać je przez adres, to musimy powiedzieć to jasno – tak było właśnie przy wysyłaniu elementów baterii.

Porozmawiajmy teraz jednak o wysyłaniu tablicy jako całości

Odbывało się ono jakby według takiego schematu. Mając funkcję

```
void fun(int tab[]) ;
```

oraz tablicę

```
int tablica[20] ;
```

wywołanie funkcji wyglądało tak:



```
fun(tablica);
```

Przesyłaliśmy do funkcji adres tablicy. (Złota regułka: nazwa tablicy jest adresem jej początku). Natomiast w poprzednim paragrafie zobaczyliśmy funkcję, która jest zdolna przyjąć jako argument – adres jakiegoś obiektu typu `int`.

Oto wywołanie funkcji `hydraulik` z argumentem będącym tablicą (całą):

```
hydraulik(tablica);
```

Jak widzisz zapis jest tu identyczny. Czy zatem `hydraulik` naprawi całą tablicę? Nie. On po prostu tego nie umie. Napisaliśmy funkcję `hydraulik` tak, że naprawia tylko obiekt o przysłanym adresie. Nazwa tablicy jest adresem jej zerowego elementu, więc naprawi on tylko ten zerowy element. Aby mógł naprawiać więcej musielibyśmy nauczyć go przesuwania tego wskaźnika.

Nie to jest tu jednak istotne. Chodzi o to, że tablicę można wysłać do funkcji jako tablicę, a odebrać na dwa sposoby:

- jako tablicę,
- jako wskaźnik.

## 8.8.2 Odbieranie tablicy jako wskaźnika

W rozdziale o tablicach mówiliśmy o tym, jak do funkcji wysłać tablicę. Przypominam, że tablicy nie wysyła się przez kopiowanie wszystkich elementów danej tablicy, bo może być ich potwornie dużo. Wysyła się do funkcji nazwę tej tablicy – która to nazwa, jak wiemy, jest przecież adresem jej początku.

Jednak skoro wysyłamy do funkcji ten adres, to możemy wobec tego odebrać go jako wskaźnik. Oto przykład kilku rodzajów funkcji:

```
#include <iostream.h>
/*****
void funkcja_wska(int *wsk, int rozmiar);
void funkcja_tabl(int tab[], int rozmiar);
void funkcja_wsk2(int *wsk, int rozmiar);
*****/
main()
{
 int tafla[4] = { 5,10,15,20 };
 funkcja_tabl(tafla, 4); // ❶
 funkcja_wska(tafla, 4); // ❷
 funkcja_wsk2(tafla, 4); // ❸
}
/*****
void funkcja_tabl(int tab[], int rozmiar) // ❹
{
 cout << "\nWewnatrz funkcji funkcja_tabl \n" ;
 for (int i = 0 ; i < rozmiar ; i++)
 cout << tab[i] << "\t" ;
}
/*****
void funkcja_wska(int *wsk, int rozmiar) // ❺
{
 cout << "\nWewnatrz funkcji funkcja_wska \n" ;
 for (int i = 0 ; i < rozmiar ; i++)
```



```

 cout << *(wsk++) << "\t" ;
 }
 /*****
void funkcja_wsk2(int *wsk, int rozmiar)
{
 cout << "\nWewnatrz funkcji funkcja_wsk2 \n" ;
 for (int i = 0 ; i < rozmiar ; i++)
 cout << wsk[i] << "\t" ;
}
 *****/

```

**Po wykonaniu tego programu na ekranie otrzymamy**

```

Wewnatrz funkcji funkcja_tabl
5 10 15 20
Wewnatrz funkcji funkcja_wska
5 10 15 20
Wewnatrz funkcji funkcja_wsk2
5 10 15 20

```

### Uwagi:

- 1 Funkcję wywołujemy podając jej nazwę tablicy (czyli adres jej zerowego elementu). W definicji funkcji funkcja\_tabl 4 widzimy, że wysłany jej adres tablicy odebrany jest także jako tablica. Wewnątrz funkcji posługujemy się znanym zapisem „tablicowym”. (Wiem, że oszukuje, ale cierpliwości!)
- 2 Identycznie wyglądające wywołanie. Tym razem chodzi o inną funkcję 5. Wewnątrz tej funkcji przysłany adres służy do inicjalizacji lokalnego wskaźnika wsk. Tak jakbyśmy wykonali taką instrukcję

```
int *wsk = tafla ;
```

Wskaźnikiem tym posługujemy się wewnątrz funkcji. W naszej funkcji zastosowaliśmy wyrażenie

```
*(wsk++)
```

które (przypominam) odpowiada złożeniu

```
*wsk oraz wsk++
```

czyli odczytaj coś, a potem przejdź do następnego elementu.

- 3 Identyczne wywołanie. Tym razem to funkcja funkcja\_wsk2. Jak widać z jej definicji odbiera ona tablicę w ten sam sposób, co funkcja powyżej. Najciekawsze jest to, że potem używa tablicy korzystając z notacji „tablicowej”.

Jakie są wady i zalety tych typów funkcji? Czy lepiej odebrać jako tablicę, czy lepiej jako wskaźnik?

- ❖ Odebranie tablicy jako rzeczywiście tablicy – sprawia, że treść funkcji jest bardziej czytelna. Wskaźniki są genialnym narzędziem do zagmatwania zapisu.
- ❖ Odebranie tablicy jako adresu, którym inicjalizuje się wskaźnik – sprawia, że funkcja pracuje szybciej. Mówiliśmy już, że szybciej dociera się

do sąsiedniego elementu tablicy posługując się wskaźnikiem. (Pamiętasz jeszcze ten przykład z rozkładem jazdy?).

*Aby znaleźć następny element tablicy wystarczy przesunąć tylko wskaźnik o 1 i gotowe. W wypadku tablicy – komputer musi żmudnie obliczyć położenie szukanego elementu tablicy. To trwa.*

- ❖ Tablice wielowymiarowe łatwiej odbiera się w funkcji stosując notację wskaźnikową. Jest to sposób bardziej uniwersalny, bo rozmiary tablicy nie muszą być na stałe zaszyte w funkcji. Wystarczy jeśli funkcja dowiedzie się o nich dopiero w momencie jej wywołania.

## 8.8.3 Argument formalny będący wskaźnikiem do obiektu `const`

Pamiętamy, że tablice do funkcji przysyła się nie tak, że funkcja otrzymuje kopie wszystkich elementów tablicy (np. w liczbie 8155) czyli nie przez wartość, ale tak, że funkcja otrzymuje adres tablicy. W rezultacie więc funkcja pracuje na oryginale tablicy i może dowolnie zmieniać jej elementy.

To bardzo wygodne, gdy funkcja naprawdę powinna zmieniać elementy tablicy – na przykład pomnożyć każdy przez 2.

Może być jednak sytuacja całkiem odwrotna. Czasem tablicę dajemy funkcji tylko po to, by ją sobie poczytała, ale nie chcemy, żeby w niej cokolwiek zmieniała. Jak się przed tym zabezpieczyć?

Tu właśnie przydaje się nam przydomek `const`, który może sprawić, że ze wskaźnika do obiektu zrobimy wskaźnik do stałego obiektu. Taki wskaźnik wskazuje na obiekty, ale nie pozwala na ich modyfikację.

Funkcja definiuje sobie na stosie wskaźnik do obiektu stałego. Otrzymany adres obiektu wstawia właśnie do takiego wskaźnika. Posługując się potem takim wskaźnikiem uniemożliwia sobie samej jakiegokolwiek modyfikację obiektu, na które on pokazuje.

```
#include <iostream.h>
```

```
// deklaracje funkcji ❶
```

```
void pokazywacz(const int *wsk, int ile);
```

```
void zmieniaacz(int *wsk, int ile);
```

```
/******
```

```
main()
```

```
{
 int tablica[4] = { 110,120,130,140};
```

```
 pokazywacz(tablica, 4);
```

```
 zmieniaacz(tablica, 4);
```

```
 pokazywacz(tablica, 4);
```

```
 cout << "Dla potwierdzenia tablica[3] = "
 << tablica[3];
```

```
}
```

```
/******
```

```
void pokazywacz(const int *wsk, int ile)
```

```
{
```

```
 cout << "Działa pokazywacz " << endl;
```

```
// ❷
```

```
// ❸
```

```

for(int i = 0 ; i < ile ; i ++, wsk++)
{
 // *wsk += 22 ; // błąd ! ④
 cout << "element nr " << i << " ma wartosc "
 << *wsk << endl; // ⑤
}
/*****/
void zmieniaacz(int *wsk, int ile) // ⑥
{
 cout << "Dziala zmieniaacz " << endl ;

 for(int i = 0 ; i < ile ; i ++, wsk++)
 {
 *wsk += 500 ; // wolno nam ! ⑦
 cout << "element nr " << i << " ma wartosc "
 << *wsk << endl;
 }
}

```

## Po wykonaniu zobaczymy na ekranie

```

Dziala pokazywacz
element nr 0 ma wartosc 110
element nr 1 ma wartosc 120
element nr 2 ma wartosc 130
element nr 3 ma wartosc 140
Dziala zmieniaacz
element nr 0 ma wartosc 610
element nr 1 ma wartosc 620
element nr 2 ma wartosc 630
element nr 3 ma wartosc 640
Dziala pokazywacz
element nr 0 ma wartosc 610
element nr 1 ma wartosc 620
element nr 2 ma wartosc 630
element nr 3 ma wartosc 640
Dla potwierdzenia tablica[3] = 640

```

## Kilka uwag :

- ① Deklaracje funkcji. Są obowiązkowe, bo wywołania ich następują w programie wcześniej niż kompilator zobaczy ich definicje (są w tekście programu później). Gdyby nie były konieczne - i tak bym je napisał, taką już mam zasadę.
- ② Wywołanie funkcji `pokazywacz`. Wysłałyśmy tam tablicę. Czy nam się to podoba czy nie, funkcja dostaje jej adres i może nawet całą tablicę zniszczyć.
- ③ Oto jak funkcja `pokazywacz` odbiera adres tablicy. Dostaje co prawda adres tablicy - czyli wszystkie uprawnienia, jednak funkcja definiuje wskaźnik z przydomkiem `const` i tam chowa adres tablicy. Równoważne to jest więc instrukcji

```
const int *wsk = tablica ;
```



## 8.9 Zastosowanie wskaźników przy dostępie do konkretnych komórek pamięci

Trzecią domeną zastosowania wskaźników jest bezpośredni dostęp do specjalnie wybranych komórek pamięci. Chodzi tu o dostęp do komórki pamięci bez podawania jakiejkolwiek jej nazwy.

Dajmy na to, że w pamięci jest jakaś komórka o adresie 93952. Jest tam coś zupełnie szczególnego. (Np. komórka ta połączona jest zewnętrznie z miernikiem temperatury). Mamy zadanie wpisania tam jakiejś wartości lub odczytania jej. Komórka ta oczywiście nie ma nazwy.

Jak zatem się do niej odnosimy? Oczywiście za pomocą wskaźnika! Jak to zrobić konkretnie? Najpierw ustawiamy wskaźnik na żadaną komórkę wpisując do niego jej konkretny adres.

```
wsk = 93952 ;
```

Od tej pory posługujemy się już tym wskaźnikiem w znany sposób.

```
cout << "Obecna temperatura << *wsk ;
```

Niestety nie zawsze ustawienie wskaźnika na żądany adres jest tak proste – w różnych typach komputerów istnieją różne sposoby adresowania.

### Dygresja dla wielbicieli IBM PC

W szczególności w komputerach klasy IBM PC jest to nieco skomplikowane. Jednak kompilatory dostarczają zwykle łatwych narzędzi do „zbudowania” konkretnego adresu. W kompilatorze Borland C++ mamy do dyspozycji makrodefinicję o nazwie `MK_FP` – (make far pointer). Użycie tej makrodefinicji rozwiązuje cały problem. Po szczegóły odsyłam do opisu kompilatora.

## 8.10 Rezerwacja obszarów pamięci

Czwartą domeną jest zastosowanie wskaźników przy rezerwowaniu jakichś obszarów pamięci.

Wiąże się z tym operator `new`, który tu właśnie będę reklamował.

*Miłośnikom języka C podpowiem, że operator ten robi to samo, co znana im funkcja biblioteczna `malloc` (memory-allocation). O tej funkcji od dzisiaj należy zapomnieć, gdyż operator `new` robi to lepiej i łatwiej. (Na przykład w IBM PC uniezależnia nas od tak zwanego modelu pamięci).*

O co tutaj chodzi: W trakcie pisania programu nie zawsze wiadomo jak duże będą tablice, którymi chcemy się posługiwać. Powstaje pytanie: czy nie można by zrobić tak, że zaraz po starcie programu mówimy programowi jak wielka na być dana tablica?

Można. Do tego właśnie używa się operatora `new`. Natomiast problem, o którym mówimy nazywa się **dynamiczną alokacją (rezerwacją) tablic**.

Zanim pokażemy jak to zrobić, inny przykład kiedy operator `new` może się przydać:

Opracowujemy program kontroli lotów. Na ekranie obrazowane są samoloty lecące właśnie nad tym obszarem. Jeśli samolot wlatuje na nasze terytorium, pojawia się na brzegu mapy (ekranu) jako mały znaczek. Stopniowo przesuwa się w trakcie lotu, a kiedy opuszcza obszar – znika. Oczywiście te samolotki muszą istnieć już jakoś wcześniej w naszym programie. Tak jak musimy w tekście programu zdefiniować zmienną `x` jeśli mamy się nią kiedyś posługiwać. Musimy sobie więc gdzieś w programie napisać definicje obiektów reprezentujących te samoloty. Tylko ile ich napisać? 5, 10, 20? Na wszelki wypadek z zapasem definiujemy 25.

Niby „na wszelki wypadek”, ale już w tym momencie ograniczyliśmy działanie programu od obsługi 25 samolotów – ale po co? Lepiej by było przecież niczego nie ograniczać.

W tym pomoże nam właśnie operator `new`. Jeśli dostaniemy – już w trakcie pracy programu – komunikat, że samolot wchodzi nad nasze terytorium, to dopiero wtedy zdefiniujemy nowy obiekt. Także nie ma problemu, gdy będą odbywać się pokazy lotnicze i w grę będzie wchodzić dodatkowych 100 obiektów. Będzie trzeba, to się je – już w trakcie pracy programu – zrobi operatorem `new`. Nie ma też problemu jeśli odbędzie się nalot dywanowy – proszę bardzo – nowe 4000 obiektów.

Wszystko to dzięki operatorowi `new` (i jego satelicie – operatorowi `delete` – likwidującemu potem te obiekty).

Inna sytuacja, kiedy mogą się jeszcze przydać `new` i `delete`.

Potrzebujemy wielkiej tablicy. Deklarujemy ją na przykład tak:

```
long tablica[4*8192] ;
```

a tu w trakcie linkowania dostajemy informację, że jest to błąd, ponieważ linker na tak wielkie tablice się nie zgadza. Łączna suma komórek z danymi nie może dla niego przekroczyć np. 64 KB i już. Co robić? Jest odpowiedź: Oszukać go za pomocą dynamicznej rezerwacji tablicy już w trakcie wykonywania programu.

## 8.10.1 Operatory `new` i `delete` albo Oratorium Stworzenie Świata.

Po takiej reklamie pora na przedstawienie. W rolach głównych wystąpią specjalne operatory `new` i `delete`<sup>†)</sup>. Operator `new` zajmuje się kreacją, a `delete` unicestwianiem obiektów.

Do rzeczy: Jeśli mamy zdefiniowany np. taki wskaźnik:

```
char *wsk ;
```

†) `new` – ang: nowy (czytaj: „nju”)  
`delete` – ang: usunąć (czytaj: „dilit”)

to następująca instrukcja:

```
wsk = new char ;
```

powoduje utworzenie nowego obiektu typu `char`. **Nie ma on nazwy**, ale jego adres przekazywany jest wskaźnikowi `wsk`.

Z kolei instrukcja

```
delete wsk ;
```

powoduje likwidację tego obiektu. (Zakładam, że wskaźnik `wsk` nadal pokazywał na ten obiekt).

Inny przykład:

```
float *w ;
w = new float[15] ;
```

Ostatnia instrukcja powoduje utworzenie piętnastoelementowej tablicy typu `float`. Tablica ta oczywiście nie ma nazwy, ale wskaźnik jest informowany o jej adresie. Kasowanie tej tablicy realizujemy instrukcją

```
delete [] w ;
```

## Cechy obiektów stworzonych operatorem `new`

Cztery sprawy są tu bardzo ważne:



- ❖ Obiekty tak utworzone istnieją od momentu, gdy je utworzymy operatorem `new` do momentu, gdy je skasujemy operatorem `delete`. Inaczej mówiąc - **to my decydujemy o czasie ich życia**.
- ❖ Obiekt tak utworzony **nie ma nazwy**. Można nim operować tylko za pomocą wskaźników.
- ❖ Obiektów tych **nie obowiązują** zwykle zasady o zakresie ważności – czyli to, w których miejscach programu są widzialne, a w których niewidzialne (mimo, że istnieją). Jeśli tylko jest w danym momencie dostępny choćby jeden wskaźnik, który na taki obiekt pokazuje, to mamy do tego obiektu dostęp.
- ❖ Tylko statyczne obiekty wstępnie inicjalizowane są zerami (o ile nie określiliśmy inaczej). Natomiast obiekty tworzone operatorem `new` nie są statyczne (wręcz przeciwnie – są dynamiczne!) dlatego **zaraz po utworzeniu tkwią w nich jeszcze śmieci**. Musimy sami zadbać o zapisanie tam sensownych wartości.

Oto przykład ilustrujący prostotę posługiwania się tym operatorem:

```
#include <iostream.h>
char * producent(void) ; // ❶
/*****
main()
{
 char *w1, *w2, *w3, *w4 ; // definicje wskaźników ❷

 // tworzenie obiektów
 w1 = producent() ; // ❸
```



```

w2 = producent();
w3 = producent();
w4 = producent();

*w1 = 'H' ; // 4
*w2 = 'M' ;
*w3 = 'I' ;

cout << "oto 3 znaki :" << *w1 << *w2 << *w3
 << "\noraz smiec w czwartym :" << *w4 // 5
 << endl ;

delete w1 ; // kasowanie obiektow 6
delete w2 ;
delete w3 ;

// *w1 = 'F' ; // byłaby tragedia, bo obiekt
 // już nie istnieje !!! 9
}
/*****/
char * producent(void) // 7
{
 char *w ;
 cout << "Wlasnie produkuje obiekt \n";
 w = new char ; // 8
 return w ;
}

```

## Po wykonaniu programu na ekranie pojawi się

```

Wlasnie produkuje obiekt
Wlasnie produkuje obiekt
Wlasnie produkuje obiekt
Wlasnie produkuje obiekt
oto 3 znaki :HMI
oraz smiec w czwartym :

```

## Uwagi do programu

- 1 Deklaracja funkcji. Czytamy ją tak: producent jest funkcją wywoływaną bez żadnych argumentów, a która jako rezultat zwraca wskaźnik (\*) do obiektu typu char.
- 2 Definicja czterech wskaźników mogących pokazywać na obiekty typu char.
- 3 Wywołanie funkcji producent, w której produkuje się obiekty typu char.
- 7 Oto definicja funkcji producent. Jak widzisz to tutaj, w funkcji tworzymy obiekty. Wcale nie musieliśmy tutaj, wystarczyłoby w miejscu 3 napisać instrukcje

```
w1 = new char ;
```

Jednak zrobiłem to celowo w funkcji - po to, by pokazać, że mimo, iż obiekty są tworzone wewnątrz funkcji, to jednak nie znikają po jej zakończeniu (jak to się zwykle dzieje z obiektami automatycznymi tworzonymi w funkcjach). Dlaczego



go? Otóż zwykle obiekty definiowane wewnątrz funkcji są tworzone na stosie. Po zakończeniu pracy tej funkcji obiekty ze stosu są uprzątane.

- ⑧ Zupełnie inaczej jest z obiektami tworzonymi operatorem `new`. Są one tworzone w obszarze pamięci, który przyznawany jest programowi do swobodnego używania. Obszar ten po angielsku nazywa się „free store” (swobodnie dostępny magazyn) lub heap (zapas). Trudno to dokładnie przetłumaczyć. Będę używał nazwy „zapas pamięci”, bo najlepiej oddaje istotę problemu.

Zatem dzięki operatorowi `new` nasza funkcja właśnie tam, w dostępnym zapasie pamięci definiowała nowy obiekt, a informację o tym, w którym miejscu konkretnie (adres), przysłała jako rezultat funkcji.

Po zakończeniu działania funkcji nowy obiekt istnieje sobie nadal. Będzie istniał aż do momentu, gdy w dowolnym miejscu programu nie skasujemy go operatorem `delete`.

- ④ Praca na nowych obiektach odbywa się tak, jak na zwykłych obiektach pokazywanych przez wskaźniki. Tu widzimy wpisanie czegoś do naszych trzech obiektów pokazywanych przez trzy wskaźniki. O czwartym obiekcie pokazywanym przez wskaźnik `w4` celowo zapominamy.
- ⑤ Wypisujemy na ekran treść trzech obiektów - są tam oczywiście litery HMI. Wypisujemy też czwarty obiekt, do którego nic jeszcze nie wpisaliśmy, więc zawiera śmieci. Na ekranie pojawia się jakiś symbol będący w kodzie ASCII odpowiednikiem tkwiącego tam przypadkowego śmiecia. Jeśli uruchomisz ten program jeszcze raz, to śmieć będzie najprawdopodobniej inny. Tak było w moim wypadku. Jak śmieć to śmieć.
- ⑥ Kasowanie obiektów. Tu właśnie zarezerwowane dla nich miejsce jest oddawane z powrotem do zapasu pamięci. Ewentualne następne wywołania funkcji `producent` mogą ten obszar znowu otrzymać.
- ⑨ Skoro się oddało obszar pamięci z powrotem do zapasu, to go już nie ma. Nasz wskaźnik co prawda nadal pokazuje na to miejsce, ale tam może mieszkać już ktoś inny. Próba zapisania tam czegoś zniszczy tego ewentualnego nowego lokatora. Jak zwykle - tego typu błąd może objawić się o wiele później niż sam akt przestępstwa.



Obiektowi kreowanemu operatorem `new` można nadać wartość już w momencie stworzenia.

```
int * wsk ;
wsk = new int(32);
```

Takie użycie operatora `new` sprawi, że w zapasie pamięci zostanie stworzony obiekt typu `int`, a do niego zostanie od razu wpisana liczba 32.

Można też sprawić, że obiekt stworzony zostanie w zapasie pamięci nie „był gdzie”, ale w określonym miejscu, na które pokazujemy wskaźnikiem. Jeśli mamy już wskaźnik `adr` ustawiony na konkretny adres, to wystarczy instrukcja o następującej składni:

```
wsk = adr new int ;
```

Jak widać – wystarczy bezpośrednio przed operatorem `new` umieścić upodobały adres (zapisany we wskaźniku `adr`).

## 8.10.2 Dynamiczna alokacja tablicy

Za pomocą operatora `new` można tworzyć (kreować) nie tylko pojedyncze obiekty, ale także i tablice.

```
int *tabptr ;
tabptr = new int[rozmiar] ;
```

gdzie *rozmiar* jest wyrażeniem typu `int`. W ten sposób stworzyliśmy nienazwaną tablicę elementów typu `int`. Wynikiem działania operatora `new` jest wskaźnik do początku tej tablicy. Podstawiamy go do naszego wskaźnika `tabptr`. Powyższe dwie linijki można napisać krócej jako:

```
int * tabptr = new int[rozmiar] ;
```

Zauważ, że rozmiar tablicy nie musi być stałą. Przypominam, że przy tradycyjnym sposobie definiowania tablic rozmiar musiałby być stałą znaną już w momencie kompilacji.

```
int tabliczka[15] ;
```

Operator `new` daje nam swobodę. Tablica definiowana jest dynamicznie, w trakcie wykonywania programu.

```
cout << "Ile elementow ma miec tablica ? \n" ;
int rozm ;
cin >> rozm ;
int *tabptr = new int[rozm] ;

//----- praca z tablica -----
*tabptr = 44 ; // wpisanie do zerowego elementu
tabptr[0] = 44 ; // to samo inaczej

*(tabptr+3) = 100 ; // wpisanie do elementu o indeksie 3
tabptr[3] = 100 ; // to samo inaczej
```

Mówiliśmy kiedyś o tym, że zapis wskaźnikowy i tablicowy są w zasadzie wymienne, dlatego możemy do naszej tablicy stosować równie dobrze zapis „tablicowy”. Oba zapisy pokazałem w powyższym przykładzie. Zapis „tablicowy” wydaje mi się łatwiejszy i bardziej naturalny.

Jednak uwaga: jeśli powiedzieliśmy, że wystarczy nam rozmiar 2 – to mamy tablicę tylko dwuelementową. Sami jesteśmy winni jeśli potem pracujemy na nieistniejącym elemencie czwartym i zdarzy się tragedia (strzał na oślep).

Aby zlikwidować tak wykreowaną tablicę, stosujemy operator `delete`.

```
delete [] tabptr ;
```

wynik działania operatora `delete` jest typu `void` (czyli nie zwracany jest żaden typ).

Za pomocą operatora `delete` kasuje się tylko obiekty stworzone operatorem `new`

Próba skasowania czegokolwiek innego może się okazać katastrofalna. Nie będzie jednak nieszczęścia jeśli zastosujemy operator `delete` w stosunku do wskaźnika pokazującego na adres zerowy (NULL) – takie sytuacje komputer sam rozpoznaje, bo żaden obiekt nie może mieć adresu 0.

### Uwaga na pułapkę :

Łatwo się domyślić, że nie należy dwukrotnie kasować obiektu. Chodzi o sytuację, gdy obiekt stworzyliśmy operatorem `new`, potem skasowaliśmy go operatorem `delete`. Obiekt już nie istnieje. Tymczasem przez zapomnienie jeszcze raz bierzemy wskaźnik pokazujący na to miejsce w pamięci i wykonujemy kasowanie. Rezultat będzie niefortunny. Błąd nie musi ujawnić się od razu, dlatego trudniej go wykryć.

Ja radzę sobie w takich sytuacjach tak, że łącznie z kasowaniem obiektu, umieszczam instrukcję ustawiającą wskaźnik na NULL. Jak już wiemy ewentualne użycie przy kasowaniu adresu NULL nie jest katastrofalne.

```
int * wsk ;

wsk = new int ;
*wsk = 15 ;
delete wsk ;
wsk = NULL ;
// ...
delete wsk ; // skoro NULL, to nie będzie tragedii
```

### Druga pułapka przy tworzeniu obiektów operatorem `new`

Powiedzieliśmy, że obiekty tworzone za pomocą operatora `new` nie mają nazw. Pracujemy z nimi tylko za pośrednictwem wskaźników. Jest tu w związku z tym pułapka. Spójrz na ten program

```
#include <iostream.h>
main()
{
 int *cze, *zol ; // def. 2 wskaźników ❶

 cze = new int ; // tworzymy obiekt A ❷
 zol = new int ; // tworzymy obiekt B

 *cze = 100 ; // ładujemy 100 do obiektu A ❸
 *zol = 200 ; // ładujemy 200 do obiektu B

 cout << " Po wpisaniu : Na czerwonym = " << *cze
 << " Na żółtym = " << *zol << endl ;

 cze = zol ; // ← Niefortunna linijka ! ❹

 cout << " Po przelozeniu - Na czerwonym = " << *cze
 << " Na żółtym = " << *zol << endl ;

 *cze = 5 ;
```

```
*zol = 1 ;
```

```
// 5
```

```
cout << " Jakis wpis - Na czerwonym = "<< *cze
 << " Na zolnym = " << * zol << endl ;
```

```
delete zol ;
```

```
// delete cze ;
```

```
// Horror !
```

```
// 6
```

## Po wykonaniu programu na ekranie zobaczymy

Po wpisaniu : Na czerwonym = 100 Na zolnym = 200

Po przelozeniu - Na czerwonym = 200 Na zolnym = 200

Jakis wpis - Na czerwonym = 1 Na zolnym = 1

## Uwagi

- 1 Najpierw definiujemy sobie 2 wskaźniki do obiektów typu int.
- 2 Następnie operatorami new tworzymy dwa (nienazwane) obiekty typu int, a ich adresy wstawiamy do wskaźników: czerwonego cze i żółtego zol. Przypominam, że wartością wyrażenia

(new int)

jest adres nowowytworzonego obiektu.

- 3 Do obiektów pokazywanych przez żółtego i czerwonego ładujemy 100 oraz 200.
- 4 Ta linijka jest istotą naszego przykładu. Sprawia, że wskaźnik czerwony pokazuje odtąd na to samo, na co pokazuje wskaźnik żółty. Adres miejsca, na które pokazywał do tej pory wskaźnik czerwony, zostaje przez nieuwagę zniszczony. Od tej pory obiekt ten staje się dla nas niedostępny.
- 5 Niezależnie, którym operatorem się posługujemy, pracujemy teraz na tym samym obiekcie – pokazywanym pierwotnie przez wskaźnik żółty.
- 6 Nie potrzebujemy już obiektów więc kasujemy je. Najpierw ten pokazywany przez wskaźnik żółty.

Natomiast obiektu A pokazywanego pierwotnie przez wskaźnik czerwony nie można już nigdy skasować. Operator delete żąda bowiem pokazania mu wskaźnikiem, który to obiekt ma skasować. Tymczasem my tę informację straciliśmy w linijce 4. Teraz na ten obiekt nie pokazuje **żaden** wskaźnik.

Gdybyśmy usunęli z tej linijki komentarz, to odbyłoby się tutaj katastrofalne, powtórne kasowanie obiektu już raz skasowanego. (Obiektu B).

Opisaną sytuację można przyrównać do takiego obrazka:

Mały i psotny Jaś uwielbia przekłuwać szpilką napętlione gazem baloniki. Bierze wobec tego z domu dwa sznurki o kolorach żółtym i czerwonym 1, idzie do ulicznego sprzedawcy w parku i kupuje dwa nowe (new) baloniki (obiekty typu int) 2. Sprzedawca i Jaś przywiązują baloniki do sznurków. Baloniki są nierozróżnialne, bo mają ten sam kolor (nie mają nazw), ale Jaś ma do nich dostęp za pomocą sznurków – czerwonego i żółtego.

Jaś mógłby od razu swoje baloniki przekłuć, ale chce jeszcze się nimi chwilę pobawić. Maluje na nich liczby 100 i 200. ③

Potem wpada na nierozważny pomysł: odwiązuje czerwony wskaźnik od balonika z liczbą 100 i dowiązuje go do balonika z liczbą 200. ④ Balonik z liczbą 200 jest teraz na dwóch sznurkach: czerwonym i żółtym.

–A ten balonik z liczbą 100? No cóż, szybuje teraz ku przestworzom. Nieostrożny Jaś stracił go na zawsze. Nie może go już teraz przekłuć. Ostał mu się jeno sznur (czerwony i żółty) z jednym balonikiem.

Ciągnijmy dalej tę opowieść:

Jaś wyjmuje z kieszeni igłę i przekłuwa to, co jest na końcu sznurka żółtego. Balonik z hukiem pęka (delete) ⑤. Uparty lub roztargniony Jaś chce teraz przekłuć to, co jest na końcu sznurka czerwonego. Tylko że drugi sznurek był przywiązany do tego samego balonika (właśnie przekłutego). Jaś dźga igłą w powietrze i w miejscu, gdzie do tej pory był właśnie przekłuty balonik, trafia na oko Małgosi. Horror!



Utrata kontaktu z obiektem uniemożliwia na skasowanie go. Jeśli jest to tylko jeden taki obiekt, to mała strata, niech sobie będzie nieskasowany. Jeśli jednak jest to sytuacja w programie, gdzie tworzy się i kasuje wiele obiektów (np. pętla, lub wielokrotnie wywoływana funkcja), wtedy nieskasowanych obiektów będzie bardzo dużo. Wyczerpie to przyznany nam obszar zapasu pamięci (free store) i już żadnych nowych obiektów nie będziemy mogli w naszym programie tworzyć.

### 8.10.3 Zapas pamięci to nie jest studnia bez dna

Może się okazać, że w pewnym momencie nasz obszar dostępnej pamięci się wyczerpie. Wówczas próba utworzenia nowego obiektu (np. typu float) za pomocą wyrażenia

```
(new float)
```

da nam w rezultacie nie adres do tego obiektu, ale 0 czyli NULL.

Jeśli więc w programie zamierzamy kreować dużo obiektów korzystając z zapasu pamięci – to musimy się spodziewać, że pamięć się w końcu wyczerpie. Trzeba się z tym liczyć i po prostu sprawdzać czy operacja się powiodła.

```
float * wsk ;
wsk = new float[8192] ; // kreacja tablicy o 8192
 // elementach typu float
if (!wsk) // ←czyli if (wsk == NULL) ...
{
 error("pamiec sie wyczerpala") ;
}
```

Jest także inny sposób. W bibliotece standardowej C++ są do dyspozycji funkcje, które pozwalają nam zareagować na brak pamięci w zapasie. Po prostu możemy określić, która z naszych (własnych) funkcji ma się uruchomić w takim awaryjnym wypadku.

Oto przykładowy program:

```
#include <iostream.h>
#include <stdlib.h> // ❶
#include <new.h> // ❷
/*****
void funkcja_alarmowa() ; // ❸
long k ; // ❹
*****/
main()
{
 set_new_handler(funkcja_alarmowa); // ❺

 for(k = 0 ; ; k++)
 {
 new int ; // tworzenie obiektu ❻
 }
/*****
void funkcja_alarmowa()
{
 cout << "\n zabraklo pamieci przy k = "
 << k << " !\n" ;
 exit(1) ; // ❼
}
```

**Po wykonaniu na ekranie pojawi się na przykład taki napis**

zabraklo pamieci przy k = 36972 !

Oczywiście liczba ta zależy od bieżącej sytuacji w pamięci komputera. Czasem mógł nam przydzielić większy zapas pamięci, czasem bardzo mały. Zależać to może na przykład od tego, czy akurat w pamięci rezydują jakieś inne programy.

### Kilka uwag

- ❶ Nowy plik nagłówkowy. Dotyczy on tej części biblioteki standardowej, gdzie jest funkcja `exit` kończąca działanie programu.
- ❷ W tym pliku nagłówkowym jest deklaracja funkcji `set_new_handler`, którą właśnie tu reklamuję.
- ❸ Deklaracja funkcji `funkcja_alarmowa`. Ta funkcja zostanie oddelegowana do zadziałania w momencie, gdy wyczerpie się zapas pamięci (free store).
- ❹ Obiekt typu `long` o nazwie `k` definiuje jako globalny. To po to, by był dostępny także w naszej funkcji `alarmowej`.
- ❺ Tu jest właśnie moment poinformowania kompilatora kogo delegujemy do reakcji na wyczerpanie się pamięci. Inaczej mówimy – jest to instalacja nowego programu obsługi. Jak widać robi się to bardzo prosto – wywołując biblioteczną funkcję `set_new_handler` (ang: ustaw jako nowy program obsługi). W na-

wiasie jako argument jest nazwa funkcji. W jednym z następnych paragrafów dowiesz się, że nazwa funkcji jest jej adresem w pamięci. Zatem to ten adres właśnie podajemy funkcji bibliotecznej.

- ⑥ Nieskończona pętla kreująca (tworząca) obiekty typu `int`. Wyniku operacji `new` nie podstawiamy nigdzie, bo obiekty te naprawdę nie są nam w tym przykładzie potrzebne - chodzi tu tylko o to, by całkowicie wyczerpać zapas pamięci.
- ⑦ W momencie, gdy ten cel osiągniemy (wyczerpanie zapasu), automatycznie zostaje uruchomiona wyznaczona przez nas funkcja `_alarmowa`. Funkcja ta wypisze na ekranie wartość zmiennej globalnej `k` - w ten sposób będziemy wiedzieli przy którym z kolei obiekcie zapas się wyczerpał. (Funkcja ma dostęp do `k`, bo jest ono globalne - zdefiniowane poza wszelkimi funkcjami).

## 8.10.4 Porównanie starych i nowych sposobów

Jakiś czas temu Jerzy Stuhr zaśpiewał w Opolu kabaretową piosenkę „Śpiewać każdy może, trochę lepiej lub gorzej...”, a kiedy skończył wzruszył ramionami i powiedział „Wielkie mecyje zaśpiewać w Opolu...”

To samo zapewne pomyśleli teraz programiści klasycznego C: w zasadzie operatory `new` i `delete` to przecież to samo, co dawne funkcje biblioteczne:

|                       |                     |                                       |
|-----------------------|---------------------|---------------------------------------|
| <code>malloc()</code> | <code>new</code>    | <code>// memory allocation</code>     |
| <code>free()</code>   | <code>delete</code> | <code>// free allocated memory</code> |

Czyli - „wielkie mecyje tworzyć obiekty!” Rzeczywiście. Problem tylko w słowach: „trochę lepiej lub gorzej”. Wspomniane funkcje biblioteczne klasycznego C są nadal dostępne w C++, więc możesz sobie je dalej używać.

### Co jednak przemawia za używaniem `new` i `delete` ?

Bardzo ważna cecha: Otóż jeśli kiedyś zdefiniujemy sobie nasz własny typ obiektu, (a będziemy to robić w następnych rozdziałach) to w momencie stwarzania (kreacji) pojedynczego obiektu danego typu - automatycznie może zostać wywołana nasza specjalna funkcja zwana konstruktorem. (O szczegółach mówić będziemy w osobnym rozdziale, str. 336). Przy zastosowaniu `malloc` tej akcji wywołania konstruktora oczywiście nie będzie.

Natomiast przy kasowaniu obiektu operatorem `delete` automatycznie wykona się inna nasza funkcja zwana destruktor. Tego w klasycznym C nie mieliśmy.

Zapomnij więc o tamtych staromodnych sposobach.

### Dodatkowo:

`malloc` zwraca wskaźnik do `void`, czyli wskaźnik do czegoś nieokreślonego. Natomiast `new` jest bezpieczniejszy, bo jako rezultat zwraca wskaźnik do typu, który właśnie stwarza. Dzięki temu nie możemy omyłkowo stworzyć operatorem `new` np. obiektu `float` a jego adres przypisać wskaźnikowi do `int`. Kompilator od razu nas uratuje przed nieszczęściem sygnalizując błąd.



## Inny powód:

Gdy pracuję na komputerze klasy IBM PC z kompilatorem Borland C++

*to lubię operatory `delete` i `new` za to, że uniezależniają mnie od przyjątego tak zwanego modelu pamięci. (Zainteresowani wiedzą o czym mówię). Przy starym sposobie jeśli model pamięci był „small” to rezerwacje pamięci robiłem funkcjami bibliotecznymi:*

```
malloc(), free()
```

*a jeśli był model „huge” to funkcjami:*

```
farmalloc(), farfree()
```

*Dzięki operatorom `new` – `delete` w ogóle nie muszę myśleć o tych modelach pamięci.*



Poznaliśmy już pobieżnie dziedziny, gdzie wskaźniki mogą się przydać. Porozmawiajmy teraz o samych wskaźnikach.

## 8.11 Stale wskaźniki

Mówiliśmy niedawno o wskaźnikach do obiektów stałych. Są to wskaźniki, które pokazywanego obiektu nie mogą zmieniać. Traktują go jako obiekt stały. Sam obiekt, na który pokazują nie musi być rzeczywiście stały. Ważne jest to, że wskaźnik tak go traktuje.

Są jeszcze inne wskaźniki z przydomkiem `const`. Czy widziałeś kiedyś zwiedzając nieznaną miasto stojący na ulicy wielki plan miasta, a na nim czerwona strzałkę z napisem „TU STOISZ”? Ta strzałka to właśnie stały wskaźnik. Na tej mapie pokazuje ona zawsze w to miejsce. Możesz pokazywać na tej mapie różne obiekty, różnymi wskaźnikami, ale nie tą strzałką. Jest ona dobrze przyklejona w obawie przed dowcipnisiem.

Ta strzałka to właśnie stały wskaźnik. Oto definicja takiego obiektu i ustawienie go obiekt:

```
int zoo ;
int * const wskaz = &zoo ;
```

Stały wskaźnik to taki wskaźnik, który ustawia się raz i już od tej pory nigdy nie można go zmienić. Obrazowo można powiedzieć, że stały wskaźnik to **wskaźnik nieruchomy**. Zamrożony zostaje adres, który w nim jest zapisany.

Pomyślisz pewnie: „–O co ta cała sprawa? – bierzemy zwykły wskaźnik ustawiamy na jakiś obiekt i po prostu nigdy go nie przesuwamy!”. Rzeczywiście, masz rację. Tyle, że za pomocą tego słowa `const` zabezpieczamy się przed ewentualnym nieuważnym przesunięciem wskaźnika. Gdybyśmy go chcieli przesunąć, to kompilator zasygnalizuje błąd. Gdyby nasz kolega z zespołu pracujący nad inną częścią programu i nie znający tej – próbował przez nieu-



wagę wskanik poruszyć – kompilator zaprotestuje dla naszego wspólnego dobra.

Przyjrzyjmy się bliżej powyższej definicji tego wskaźnika. Definicję tę czytamy od nazwy i posuwamy się w lewo:

wskaz jest to stały (const) wskaźnik (\*) pokazujący na obiekty typu int.

Uwaga:

Ponieważ jest to stały wskaźnik, należy już w trakcie definicji inicjalizować go, czyli nadać mu wartość początkową. (Po prostu ustawić go na jakiś adres). Można to zrobić tylko teraz albo nigdy.

Już liniijkę później próba nadania mu jakiegś wartości będzie uznana za pogwałcenie zasady, że wskaźnik jest stały (nieruchomy). Nawet gdybyśmy chcieli wpisać do niego ten sam adres, który już ma. Nie można i koniec !

W naszym przykładzie wskaźnik jest inicjalizowany adresem obiektu zoo.

## 8.12 Stałe wskaźniki, a wskaźniki do stałych

Jaka jest zasadnicza różnica między wskaźnikami stałymi a wskaźnikami do stałych?

- ❖ Stały wskaźnik to taki, który zawsze pokazuje na to samo. Nie można nim poruszyć.
- ❖ Wskaźnik do stałego obiektu to taki wskaźnik, który pokazywany obiekt uznaje za stały. Nie może go więc modyfikować.

Te dwa typy wskaźników można ze sobą ożenić. Mamy wtedy stały (nieruchomy) wskaźnik do stałego (niezmiennego) obiektu. W definicji wystąpi dwa razy słowo const

```
const float * const p ;
```

definicję taką czytamy (znowu od prawej do lewej): p jest stałym (const) wskaźnikiem (\*) pokazującym na obiekt typu float będący stałą (const).

Uściślijmy: będący stałą dla tego wskaźnika. Inny, zwykły wskaźnik pokazujący na ten sam obiekt może go zmieniać.

A oto przykłady użycia: najpierw przykład na stały (nieruchomy) wskaźnik:

```
int a = 5 ,
 b = 100 ;
int *wa ;
int * const st_wsk = &a ;

wa = &a ;
*wa = 1 ;
*st_wsk = 2 ;

// _____ teraz próbujemy ruszyć oba wskaźniki

wa = &b ;
// przestaw wskaźnik by pokazywał na zmienną b
```

```
st_wsk = & b; // błąd - bo to jest nieruchomy wskaźnik !!!
 // jest na zawsze ustawiony na zmienną a
```

A oto przykład ze wskaźnikiem pokazującym na stałą:

```
int x[4] = { 0, 1, 2, 3 } ;
int tmp ;
int *w ; // zwykły wskaźnik
const int * wsk_od_st ; // wskaźnik do obiektu stałego. Nie musi
 // on być od razu ustawiany. Można
 // nim nawet potem poruszać

w = x ; // ustawienie obu wskaźników na początek tablicy
wsk_od_st = x ;

tmp = *w ; // odczytanie zerowego elementu tablicy
tmp = *wsk_od_st ; // jak wyżej

// ————— przesunięcie obu wskaźników na następny element tablicy
w++ ;
wsk_od_st ++ ; // poruszać nim wolno

// ————— będziemy tam wpisywać
*w = 0 ; // wpisanie 0 do elementu x[1]
*wsk_od_st = 0 ; // ← BŁĄD ! Ten wskaźnik traktuje
 // to, na co pokazuje, jako obiekt stały.
 // Za pomocą TEGO wskaźnika obiektu
 // modyfikować nie wolno
```

A oto przykład na stały (nieruchomy) wskaźnik do stałego obiektu:

```
int m = 6,
 n = 4,
 tmp ;
const int * const w = &m ; // ponieważ jest to stały (nieruchomy) wskaźnik to musimy go
 // od razu zainicjalizować adresem na jaki ma pokazywać

tmp = *w ; // odczytanie wartości z obiektu pokazywanego
*w = 15 ; // ← BŁĄD ! - zapisać tam nie możemy. Wskaźnik
 // traktuje przecież swój obiekt jako stały.

w = &n ; // ← BŁĄD ! wskaźnik jest na dodatek nieruchomy,
 // nie można nim pokazać na obiekt inny niż m
```

## 8.13 Strzał na oślep – Wskaźnik zawsze pokazuje na coś

Jedną z pułapek, w którą często się wpada jest zapomnienie nadania wskaźnikowi wartości początkowej. Inaczej mówiąc zapominamy go ustawić, by na coś pokazywał. To jest źle powiedziane, albowiem wskaźnik zawsze pokazuje na coś, nawet jeśli to coś nie jest niczym zamierzonym. Tak samo, jak drewniany wskaźnik do mapy pokazuje na coś nawet wtedy, gdy leży odłożony na boku. Powiedziałbym nawet drastyczniej: celuje na coś, jak leżący na boku pistolet.

Jeśli więc zapomnimy ustawić wskaźnik, to próba odczytania tego miejsca, na jakie pokazuje, da bezsensowne i przypadkowe rezultaty. Gorzej z zapisem. To tak, jakby leżący na boku pistolet nagle wystrzelił. Zapisując coś do takiego (przypadkowo pokazywanego) miejsca niszczymy je, a to jest zwykle fatalne w skutkach, chociaż błąd może objawić się dużo później.

Dla ciekawości podam, że wskaźnik, który jest zdefiniowany jako obiekt statyczny (to znaczy albo jest globalny, albo co prawda lokalny, ale za to z przydomkiem *static*) taki wskaźnik pierwotnie pokazuje na adres zerowy NULL<sup>†</sup>). Z takim wskaźnikiem pokazującym na NULL nie ma specjalnego ryzyka, bo operacje na takim szczególnym adresie komputer łatwo wykryje i ostrzeże nas.

Jednakże wskaźniki, które definiujemy jako obiekty automatyczne pokazują na całkowicie przypadkowe adresy.

*Łatwo sobie ten fakt uprzytomnić. Przypomnijmy: Obiekty automatyczne (a wskaźnik to także obiekt – wszystko jedno czy w komputerze, czy wystrugany z drewna), zatem obiekty automatyczne są przecież tworzone na stosie, a zasada jest taka, że tworzonych na stosie obiektów komputer dla nas nie inicjalizuje. Są tam śmieci, dopóki o inicjalizację nie zatroszczy się sam programista.*

Dobra rada:

Jeśli Twój program z niewiadomych przyczyn zawiesza komputer (IBM PC) lub powoduje tzw. crash programu (VAX) – czyli komputer odmawia dalszej pracy z tym programem wyrzucając nam na ekran – że tak powiem – protokół z sekcji zwłok programu (postmortem dump) to jest ogromne prawdopodobieństwo, że winne są jakieś nieustawione wskaźniki.

Powracając do naszej analogii – to wystrzeliliśmy z leżącego na boku pistoletu. Kula trafiła kogoś na oślep.

Oto jak prosto zrobić taki błąd:

```
void fun()
{
 float a ;
 float *x, *m ; // def wskaźników bez nadania wart początkowej

 m = &a ; // teraz ustawiamy wskaźnik m
 *m = 10.7 ; // poprawne wpisanie liczby do obiektu a
 // ----- wskaźnika x nie ustawiliśmy na nic
 *x = 15.4 ; // ← tragedia !
}
```

Ten wskaźnik x nie był ustawiony na nic sensownego, więc pokazywał na coś przypadkowego. Tutaj więc coś w pamięci komputera niszczymy.

†) Pamiętamy, że obiekty statyczne wstępnie inicjalizowane są zerami chyba, że sami je inicjalizujemy inną wartością

## 3.14 Sposoby ustawiania wskaźników

Skoro wskaźniki przed ich pierwszym użyciem powinny być ustawione - jak to zrobić?

Niektóre sposoby już poznaliśmy. Zbierzmy jednak wszystkie najważniejsze.

- ❖ Wskaźnik można ustawić tak, by pokazywał na jakiś obiekt wstawiając do niego adres wybranego obiektu

```
wsk = & obiekt ;
```

- ❖ Wskaźnik można ustawić również na to samo, na co pokazuje już inny wskaźnik. Jest to zwykła operacja przypisania wskaźników

```
wsk = inny_wskaźnik ;
```

- ❖ Wskaźnik ustawia się na początek jakiejś tablicy podstawiając do niego jej adres. Skoro wiemy, że nazwa tablicy jest równocześnie adresem jej zerowego elementu, zatem w zapisie niepotrzebny jest operator &. Piszemy po prostu

```
wsk = tablica ;
```

- ❖ Wskaźnik może pokazywać także na funkcję. O wskaźnikach do funkcji będziemy mówić za chwilę (str. 206). Tam też dowiemy się, że nazwa funkcji to także jej adres, zatem i tu zbędny jest operator &

```
wskf = funkcja ;
```

- ❖ Operator new zwraca adres właśnie stworzonego nowego obiektu. Taki adres natychmiast wpisujemy do wskaźnika. Od tej pory wskaźnik pokazuje na ten nowy obiekt.

```
float *wsk ;
wsk = new float ;
```

- ❖ Wskaźnik można ustawić też tak, by pokazywał na jakieś konkretne miejsce w pamięci. Na przykład na jakiś adres znany nam z książki, choćby instrukcji obsługi jakiegoś układu sprzęgającego (interface). Tutaj trudniej podać przykład, bo bardzo zależy to od metody adresowania stosowanej w danym typie komputera.

Weźmy jednak sytuację najprostszą – komórki w komputerze numerowane są po prostu od 0 w górę. Jeśli wówczas chcemy pokazać na adres 3007307 to wskaźnik ustawia się po prostu instrukcją:

```
wsk = 3007307 ;
```

*Jeśli nasz komputer to komputer klasy IBM PC, a nasz kompilator to Borland C++, to do takiego ustawiania wskaźnika służy specjalna makrodefinicja MK\_FP. Aby ustawić wskaźnik na takie miejsce w pamięci o współrzędnych: Segment 0xffff, Offset = 0xfa wykonujemy instrukcję*

```
wsk = MK_FP(0xffff, 0xfa) ;
```

*Nie przejmuj się jeśli tego ostatniego nie rozumiesz. To nie jest C++, tylko prywatne sprawy komputera IBM PC.*

- ❖ Jeśli wskaźnik ma pokazywać na ciąg znaków (string) – można go ustawić w ten sposób.

```
wsk = "taki napis" ;
```

Ten sposób dopuszczalny jest tylko dla stringów. Nie jest to *kopiowanie* stringu. Tekst ten (string) istnieje przecież gdzieś w pamięci, (tam złożył go kompilator), a tą instrukcją ustawiamy tylko wskaźnik na to nieznane miejsce. Oczywiście nie można tego sposobu zastosować do tablic liczb.

```
int *wskint = { 1,2,3,4 } ; // błąd !!!
```

## 8.15 Tablice wskaźników

Jak pamiętamy, tablica to ciąg zmiennych tego samego typu zajmujących ciągły obszar w pamięci. Jeśli mogą być tablice zawierające zmienne typu `int`, `float`, `char` itd. – to dlaczego nie miałyby być tablic, których elementami są wskaźniki, czyli adresy różnych miejsc w pamięci. Adresy to w końcu też jakieś liczby. Można je przechowywać w tablicach.

### Tablica wskaźników do float

Oto przykład tablicy do przechowywania pięciu wskaźników. Wszystkie te wskaźniki służą do pokazywania na obiekty typu `float`

```
float *tabwsk[5] ;
```

Przeczytajmy tę definicję: Zaczynamy od nazwy

```
tabwsk
```

(i posuwamy się w prawo, bo operator `[]` jest mocniejszy od operatora `*` – czytamy więc:)

```
[5] - jest tablicą pięcioelementową
```

(teraz w lewo i napotykamy gwiazdkę)

```
* - wskaźników mogących pokazywać na obiekty typu float.
```

Tę samą definicję można by napisać tak:

```
float *(tabwsk[5]) ;
```

Użycie nawiasu okrągłego pokazuje wyraźniej kolejność (sposób) czytania definicji.

### Tablica wskaźników do stringów

A oto definicja innej tablicy wskaźników. Jej elementami są wskaźniki mogące pokazywać na stringi.

```
char *miasta[6] ;
```

Wskaźniki te można ustawić tak, by pokazywały na jakieś stringi.

```
char *nazwy[6] = {
 "Krakow", "Berlin", "Paryz", "Oslo",
 "Los Angeles", "Compostella"
};
```

Elementami tej tablicy nie są – jak można by przypuszczać – stringi z nazwami miast. Są tam adresy tych miejsc w pamięci, gdzie kompilator umieścił sobie te stringi. Podobna konstrukcja z liczbami byłaby błędna:

```
int *wskint[4] = { 10, 11, 12, 13 } ; // !!!!
```

znaczyłoby to bowiem, że chcemy by wskaźnik będący pierwszym elementem tablicy pokazywał na adres 10 itd.

Dlaczego w wypadku stringów błędu nie ma? Na tej samej zasadzie, dla której konstrukcja

```
char *w = {"abcde"} ;
```

jest poprawna. Gdzieś w pamięci kompilator musiał umieścić sobie ten string.<sup>†)</sup> W momencie, gdy dochodzi do definicji i inicjalizacji wskaźnika, podstawia on adres tego stringu do wskaźnika.

Oto krótki program, w którym posługujemy się tablicą wskaźników:

```
#include <iostream.h>
/*****
main()
{
char *stacja[] = {
 "Wansee", "Nikolassee", "Grunewald",
 "Westkreuz", "Charlottenburg",
 "Savigny Platz", "Zoologischer Garten" };
char *www[3] ;
int i ;

for(i = 0 ; i < 7 ; i++){
 cout << "Stacja: "<< stacja[i] << endl ;
}
www[0] = stacja[2] ;
www[1] = stacja[5] ;
www[2] = "Taki tekst" ;

cout << "Oto 3 elementy tablicy : \n"
 << www[0] << ", "
 << www[1] << ", "
 << www[2] << endl ;
}
```

†) Mimo, że nie żądaliśmy tego specjalnie – stringi są przechowywane jako obiekty statyczne.

## 8.16 Wariacje na temat stringów

O stringach (ciągach znaków) mówiliśmy już – jednak obecnie (kiedy już mamy za sobą wskaźniki) do sprawy tej wracamy raz jeszcze. Teraz jednak jesteśmy mądrzejsi. Wiemy na przykład, że jeśli do funkcji wysyłamy string (czyli tablicę znakową), to można w tej funkcji odebrać tę tablicę jako

- rzeczywście tablicę

```
chr tab[]
```

lub

- jako wskaźnik do obiektów typu char

```
char * wsk
```

Posłużenie się wskaźnikiem da w efekcie funkcję, która może wykonywać się szybciej. Z drugiej strony jednak funkcja ze wskaźnikiem jest na pierwszy rzut oka mniej czytelna. Tak to zwykle bywa: coś za coś!

Oto przykład dwóch funkcji:

Obie drukują string przysłany do nich, ale żeby było ciekawiej drukują go tak, że po kolejnych znakach wstawiają pauzy. Zatem tekst tornado wyglądał będzie na ekranie tak:

```
t-o-r-n-a-d-o-
```

Oto program, w którym mamy realizację „tablicową” i „wskaźnikową” takiej funkcji. Funkcje te nazywają się odpowiednio `przedzielacz_tabl` oraz `przedzielacz_wsk`

```
#include <iostream.h>
void przedzielacz_tabl(char tab[]) ;
void przedzielacz_wsk(char *w) ;
/*****/
main()
{
 char ostrzezenie[80] = { "Alarm trzeciego stopnia " } ;

 cout << "\n wersja tablicowa \n" ;
 przedzielacz_tabl(ostrzezenie); // ❶

 cout << "\n wersja wskaznikowa \n" ;
 przedzielacz_wsk(ostrzezenie); // ❶
}
/*****/
void przedzielacz_tabl(char tab[]) // ❷
{
 int i = 0 ;

 while(tab[i])
 {
 cout << tab[i++] << "- " ;
 }
}
/*****/
```

```
void przedzielacz_wsk(char *w) // 3
{
 while(*w)
 {
 cout << *(w++) << "-";
 }
}
/*****
```

## Po wykonaniu programu na ekranie zobaczymy

```
wersja tablicowa
A-l-a-r-m- -t-r-z-e-c-i-e-g-o- -s-t-o-p-n-i-a- -
wersja wskaźnikowa
A-l-a-r-m- -t-r-z-e-c-i-e-g-o- -s-t-o-p-n-i-a- -
```

- 1 Dwa miejsca, gdzie te funkcje się wywołuje. Jak widać sposób przesłania tablicy znakowej jest identyczny w obu wypadkach.

- 2 Realizacja tablicowa funkcji. Nie ma tu nic nadzwyczajnego. Definiujemy lokalny obiekt `i` po to, by mieć indeks do elementów tablicy. Potem następuje pętla `while`.

Najpierw sprawdza się co jest w elemencie `tab[i]`. Jeśli jest to cokolwiek innego niż znak NULL (bajt zerowy) kończący string, to następuje wypisanie tego znaku, a potem kreseczki -. Przy okazji wykonuje się postinkrementacja indeksu. To znaczy już po wydobyciu znaku z tablicy, indeks `i` jest zwiększany o 1.

Zauważ, że w tej funkcji dwukrotnie musi być liczona pozycja (adres) danego elementu tablicy w pamięci.

- 3 Realizacja „wskaźnikowa” jest sprytniejsza. Przysłana do funkcji tablica znakowa (czyli właściwie adres jej początku) służy do inicjalizacji lokalnego wskaźnika. Pokazuje on na początek stringu.

Wewnątrz funkcji mamy znów pętlę `while`. Wykonuje się ona dotąd, dopóki znak wskazywany przez wskaźnik — czyli `(*w)` — jest różny od NULL. Wypis na ekran to znowu wydobyte z pamięci elementu, na który wskaźnik pokazuje. Przy okazji robiona jest postinkrementacja wskaźnika, czyli po spełnieniu swojej roli przesuwa się on na następną literę stringu.

Zauważ, że ani razu nie liczy się tu żmudnie pozycji adresu danej litery w pamięci. Bierzemy po prostu to, na co wskaźnik już pokazuje. Przejście do następnej litery jest tylko przesunięciem wskaźnika na sąsiada.



A oto jak wyglądałaby funkcja kopiująca string z tablicy do tablicy. W rozdziale o tablicach widzieliśmy realizację „tablicową”. Teraz zobaczymy jak to będzie wyglądało w przypadku posłużenia się wskaźnikami.

Oto krótki program:

```
#include <iostream.h>
char * strcpy(char *cel, char *zrodlo) ;
/*****
main()
```



```
{
char poziom[] = { "Poziom szumu w normie" } ;
char komunikat[80] ;

 strcpy(komunikat, poziom) ; // ❶
 cout << poziom << endl ;
 cout << komunikat << endl ;
}
/*****/
char * strcpy(char *cel, char *zrodlo) // ❷
{
 char *poczatek = cel ; // ❸

 while(*(cel++) = *(zrodlo++)); // ❹
 return poczatek ; // ❺
}
/*****/
```



## Po wykonaniu program drukuje

```
Poziom szumu w normie
Poziom szumu w normie
```

aby udowodnić, że istotnie skopiował string z jednej tablicy do drugiej.



## Kilka wyjaśnień

- ❶ Wywołanie funkcji celem skopiowania stringu z tablicy `poziom` do tablicy `komunikat`.
- ❷ Definicja naszej funkcji. Czytamy ją: `strcpy` jest funkcją wywoływaną z dwoma argumentami - pierwszym: wskaźnikiem do tablicy znakowej (`char*`) i - drugim: wskaźnikiem do tablicy znakowej (`char*`). Funkcja ta ma zwracać wskaźnik do tablicy znakowej (`char*`)
- Jak widzimy w ❶ nasza funkcja została wywołana z dwoma argumentami aktualnymi: tablicami `poziom` i `komunikat`. Wysłano do funkcji nazwy tablic - czyli inaczej adresy ich początków.
- Tymczasem funkcja `strcpy` definiuje sobie lokalne wskaźniki `zrodlo` i `cel`. Wskaźniki te są inicjalizowane przysłanymi adresami tablic. Pokazują więc one wstępnie na ich początki.
- ❸ Nie pytaj mnie teraz dlaczego, ale czasem okazuje się przydatne, by taka funkcja zwracała wskaźnik do tej tablicy, do której string zostaje wpisywany. Teraz jeszcze na tę tablicę pokazuje wskaźnik `cel`. Ponieważ jednak zamierzamy nim za chwilę poruszać, dlatego zapamiętujemy go sobie we wskaźniku `poczatek`. Później instrukcją `return` ❺ zwrócimy właśnie tę zapamiętaną wartość.
- ❹ Praca całej funkcji polega na wielokrotnym wykonywaniu wyrażenia

```
*cel = *zrodlo
```

czyli kopiowaniu znaku pokazywanego wskaźnikiem `zrodlo` w miejsce pokazywane wskaźnikiem `cel`.

Przy okazji po robieniu tej akcji każemy oba te wskaźniki przesunąć na następne pozycje (następny element tablicy znakowej). To: „przy okazji” – to właśnie postinkrementacja zaznaczona za pomocą znaków ++ w wyrażeniu

```
{ *(cel++) = *(zrodlo++) }
```

wyrażenie to jest przypisaniem, a więc jako całość ma wartość równą wartości przypisywanej. Czyli w naszym programie najpierw wartością tego wyrażenia będzie kod litery 'P', potem kod litery 'o', potem liter 'z', 'i', 'o' i tak dalej, aż do znaku kończącego każdy string czyli NULL. Wtedy wartość tego wyrażenia będzie zero. Ponieważ wyrażenie to tkwi jako warunek pętli `while`, zatem wtedy właśnie pętla ta zostanie przerwana.

A co jest właściwą treścią pętli? : NIC! Zauważ, że zaraz za warunkiem sprawdzanym przez `while` jest średnik, czyli pętla jest pusta. Mimo to jednak wykonuje dla nas pracę. Jak to się dzieje?

Mówiliśmy już kiedyś o tym, przypomnijmy jednak. Otóż pętla `while` zawsze najpierw sprawdza sobie warunek. Jeśli będzie on spełniony, to ewentualnie wykona treść pętli. Jednakże my jako warunek daliśmy jej skomplikowane wyrażenie, którego wynik (wartość) ma ostatecznie zdecydować czy robić pętlę czy nie. U nas to wyrażenie to jest przypisaniem - kopiowaniem znaku. Wyrażenie to musi zostać więc najpierw obliczone. Wartością przypisania jest kod kopiowanego znaku i on właśnie decyduje czy wykonać obieg pętli czy nie. Wielka decyzja to nie jest - jeśli tylko ten znak jest inny niż NULL, to warunek pętli jest spełniony.

Pętla `while` wie już czy ma wykonać swą właściwą treść czy nie. Treścią pętli jest średnik – czyli instrukcja pusta – ale to nic nie szkodzi, bo kopiowanie odbyło się przecież już w chwili sprawdzenia.



Uwaga:

*Jeśli masz troskliwy kompilator, to w tym miejscu otrzymasz ostrzeżenie, że możliwe, iż wykonujesz tu niepoprawne przypisanie. Świadczy to dobrze o kompilatorze. Spodziewa on się tu najczęściej porównań typu*

```
while(a == b) ...
```

*natomiast u nas jest tylko jeden znak = oznaczający przypisanie. Kompilator na wszelki wypadek ostrzega nas wiedząc, że sytuacje z przypisaniem są tu raczej rzadkością. Powinniśmy jeszcze raz się upewnić czy naprawdę chcemy przypisywać, a nie porównywać. My jednak naprawdę chcemy tu przypisywać, więc na to ostrzeżenie nie reagujemy.*

Zauważ jeszcze jedną ciekawą rzecz. String, jak wiadomo, ma na końcu znak NULL. Poprawne skopiowanie stringu w inne miejsce wymaga oczywiście skopiowania także i tego ważnego znaku. Czy nasza funkcja to robi?

Założmy, że skopiowaliśmy ostatnią literę napisu źródłowego. Dzięki postinkrementacji wskaźnik `zrodlo` przeskoczył na następny znak i pokazuje na znak NULL. Pętla `while` przystępuje do ponownego obliczenia wyrażenia będącego warunkiem. Następuje więc kolejne przypisanie - czyli skopiowanie znaku NULL. Wartością tego wyrażenia przypisania jest teraz NULL, a więc w tym

momencie pętla przerywa się. Jednak znak NULL został już właśnie skopio-  
wany.



Wróćmy do sprawy tego, co zwraca nasza funkcja `strcpy`.

Funkcja zwraca jakąś wartość, ale ostatecznie mogłaby też nic nie zwracać i być typu `void`. Byłoby to bez wpływu na kopiowanie stringu. Zwracana jest jednak wartość typu

```
char *
```

czyli wskaźnik do obiektu typu znakowego. Na przykład do stringu. Po co tak? Jest taka tendencja w większości funkcji bibliotecznych zajmujących się stringami, aby funkcja zwróciła wskaźnik do miejsca, gdzie odbyło się kopiowanie. Co przez to zyskujemy?

Otóż teraz, aby po procesie kopiowania wykonać jakąś operację na tablicy komunikat – (np. wypisanie na ekran) – zamiast pisać 2 instrukcje

```
strcpy(komunikat, poziom);
cout << komunikat ;
```

wystarczy napisać

```
cout << (strcpy(komunikat, poziom));
```

Jest to instrukcja „wypisz”. Co ma zostać wypisane? To ukryte jest w wyrażeniu w nawiasie. Najpierw więc musi zostać obliczone wyrażenie czyli wykonywana jest funkcja `strcpy`. Zwracana przez nią wartość (wskaźnik do tablicy komunikat) jest właśnie wartością wyrażenia. Natomiast `cout` widząc wskaźnik typu `char*` rozumie to jako żądanie wypisania stringu znajdującego się pod wskazywanym adresem. A wszystko dlatego, że funkcja `strcpy` coś zwraca. Sprytne, prawda?

Oczywiście tak jest nie tylko w wypadku wypisywania na ekran. Jeśli np. mamy

```
char pierwszy[80] = { "hurra" }
char drugi [80] ;
char trzeci [80] ;
```

to zapis

```
strcpy(trzeci, strcpy(drugi, pierwszy));
```

odpowiada temu samemu co

```
strcpy(drugi, pierwszy) ;
strcpy(trzeci, drugi);
```

W obu przypadkach najpierw string z tablicy `pierwszy` kopiowany jest do tablicy `drugi`, a następnie z tablicy `drugi` do tablicy `trzeci`. W rezultacie we wszystkich trzech tablicach będzie napis „hurra”.

Mimo, że naszą funkcję wyposażyliśmy w typ zwracany `char*`, możemy ją używać na dwa powyższe sposoby. Nikt bowiem nie każe nam korzystać

z rezultatu zwracanego przez funkcję. Udamy po prostu, że funkcja zwraca void.

## Inne pożyteczne funkcje

Rozważmy teraz następujący przypadek

```
char stara[80] = { "zdanie trzecie" } ;
char nowa[80] ;

cout << (strcpy(nowa, stara+1)) ;
```

Pytanie: Co zostanie wypisane na ekran?

## Czyli inaczej – co będzie treścią tablicy nowa?

Zauważmy, że wysyłamy do funkcji strcpy nie tablicę stara, ale wyrażenie (stara+1). Co jest wartością tego wyrażenia? To proste. Pamiętasz przecież naszą koronną zasadę, że nazwa tablicy jest wskaźnikiem do jej zerowego elementu? A pamiętasz, że dodanie liczby całkowitej do wskaźnika powoduje, że rezultat pokazuje o n elementów dalej? Słowem wyrażenie (stara+1) jest adresem nie zerowego elementu tablicy, tylko pierwszego. Wskaźnik stara pokazywał na literę 'z', a wyrażenie (stara+1) pokazuje na literę 'd'. Ten właśnie adres posyłamy do funkcji strcpy.

Czy to jakoś przeszkadza funkcji strcpy? Skądże! Funkcja ta, niezależnie co jej przyślemy, rozpoczyna kopiowanie od tego miejsca w pamięci aż do napotkania NULL.

W sumie zatem do tablicy nowa zostanie skopiowany string

"danie trzecie"

bowiem pierwszą literę 'z' przeskoczyliśmy.

strcat

Oto funkcja, która dopisze do jednego stringu drugi. Przykładowo jeśli przed operacją mieliśmy dwa stringi:

"Najpierw to "  
"a teraz tamto"

po operacji będziemy mieli

"Najpierw to a teraz tamto"

Takie łączenie nazywa się konkatencją, stąd nazwa funkcji strcat

Znowu prosty program:

```
#include <iostream.h>
char* strcat(char *cel, char *zrodlo) ;
/******/
main()
{
 char co[] = { "urządzeń sterowych" } ;
 char komunikat[80] = { "Alarm : " } ;
```

```

 strcat(komunikat, co) ;
 cout << "po dopisaniu = "
 << komunikat << endl ; // ❶
 cout << (strcat(komunikat, ", o godz 17:12")) ;// ❷
}
/*****
char * strcat(char *cel, char *zrodlo) // ❸
{
 char *poczatek = cel ;

 // przesunięcie napisu na koniec stringu
 while(*(cel++)); // ❹

 // teraz pokazuje o 1 znak za NULL
 cel--; // ❺

 // to już braliśmy przy strcpy
 while(*(cel++) = *(zrodlo++)); // ❻

 return poczatek ;
}

```

## Po wykonaniu tego programu na ekranie pojawi się

```

po dopisaniu = Alarm :urządzen sterowych
Alarm :urządzen sterowych, o godz 17:12

```

## Oto ciekawsze punkty programu:

- ❸ Jest to funkcja przyjmująca jako argumenty dwa wskaźniki do tablic znakowych. Zwraca także wskaźnik do tablicy znakowej. Z poprzednich stron wiemy już dlaczego.
- ❹ Aby dopisać coś do stringu powinniśmy wskaźnik pokazujący na jego początek przesunąć tak, by pokazał na koniec, czyli na kończący string znak NULL. Robimy to właśnie tą instrukcją.
- ❺ W poprzedniej instrukcji znaleźliśmy znak NULL, ale ponieważ instrukcja zawierała postinkrementację, więc wskaźnik `cel` pokazuje teraz na następny znak za znakiem NULL. Jest tam jakiś śmieć. Musimy się więc cofnąć wskaźnikiem tak, by pokazywał na znak NULL.
- ❻ Jest to identyczny proces kopiowania jak w funkcji `strcpy`. Znaki zostają przepisywane tak, że pierwszy z nich zniszczy (zatrze) znak NULL kończący string „Alarm :”. Kolejne znaki będą dopisywane począwszy od tego miejsca. W rezultacie otrzymamy wydłużony string, a na końcu oczywiście znajdzie się znak NULL.
- ❶ Na dowód, że to prawda wypisujemy to na ekranie.
- ❷ Do tablicy `komunikat` możemy dopisać jeszcze dalszy string. Tym razem nie jest on treścią tablicy, ale wstawiliśmy go bezpośrednio jako argument ograniczony cudzysłowami:

```
strcat(komunikat , "abc");
```

Czy można tak? Można – Kompilator bowiem bez naszego specjalnego żądania umieścił ten string gdzieś w pamięci komputera. (Stringi ujęte w cudzysłowy traktowane są tak, jakby były `static` – mają gdzieś swoje określone miejsce w pamięci – nawet jeśli tego miejsca nie znamy). Do funkcji zostanie więc wysłany adres tego specjalnego miejsca w pamięci, gdzie mieści się nasz string.

Nie można jednak zrobić następującej operacji

```
strcat("Uwaga :", "abc") ; // straszny błąd !
```

Oba stringi są wtedy gdzieś w pamięci. Z drugim wszystko jest w porządku, natomiast do tego pierwszego chcemy coś dopisać (ten drugi). Tymczasem na pierwszy string zarezerwowano 7+1 znaków i ani jeden więcej. Dalej teren nie należy już do nas. Dopisywanie coś do tego stringu będzie więc niszczeniem czegoś, co jest bezpośrednio za tym stringiem. To ma zwykle fatalne skutki.

## 8.17 Wskaźniki do funkcji

Jak wiemy wskaźnikiem można pokazywać na różne obiekty. Okazuje się, że logiczne jest także pokazanie na funkcję. To tak, jakbyśmy powiedzieli: a teraz masz wykonać *tę* funkcję.

Ostatecznie wskaźnik zawiera adres, więc czemu nie miałby to być adres tego miejsca w pamięci, gdzie zaczyna się kod będący instrukcjami żądanej funkcji. Oto przykład definicji takiego wskaźnika:

```
int (*wfun)() ;
```

Jak się taka deklarację czyta?

Zaczynamy od nazwy. Następnie poruszamy się (o ile można) w prawo dlatego, że w po prawej stronie mogą stać tylko operatory `()` lub `[]` – a jak wiemy są one najsilniejsze z możliwych.

Potem, gdy już się nie da w prawo (bo napotkaliśmy nawias zamykający), poruszamy się w lewo. Jeśli odczytaliśmy wszystko w obrębie danego nawiasu wychodzimy na zewnątrz niego i znowu zaczynamy w prawo.

A zatem naszą pierwszą definicję wskaźnika przeczytamy tak:

```
wfun
```

w prawo się nie da, bo jest nawias zamykający, więc idziemy w lewo -

```
(*wfun)
```

- **jest wskaźnikiem** – (załatwiliśmy całe wnętrze nawiasu, więc wychodzimy na zewnątrz i poruszamy się w prawo, gdzie stoi bardzo mocny operator wywołania funkcji -

```
(*wfun)()
```

- **do funkcji wywoływanej bez żadnych argumentów** (nawias był pusty) – teraz już w lewo – a zwracającą

```
int (*wfun)()
```

wartość typu `int`

Bardzo ważne były tu nawiasy. Gdybyśmy je opuścili i napisali ostatnią definicję tak:

```
int *wf() ;
```

to byłaby to deklaracja funkcji (a nie wskaźnika do funkcji). Wedle powyższej reguły czytamy ten zapis tak:

```
wf()
```

`wf` jest funkcją wywoływaną bez żadnych argumentów, a zwracającą

```
* wf()
```

wskaźnik do

```
int * wf()
```

typu `int`

A zatem coś zupełnie innego. To dlatego, że nawiasy są silniejsze niż gwiazdka. Zobaczmy czym prędzej jak stosuje się to w praktyce. Oto przykład prostego programu:

```
#include <iostream.h>
int pierwsza();
int druga(); // ❶
/*****/
main()
{
 int i ;
 int (*wskaz_fun)() ; // ❷
 cout << "Na ktora funkcje ma pokazac wskaznik ?\n"
 "pierwsza -\t1 \n czy druga - \t2 \n"
 " napisz numer : " ;
 cin >> i ; // ❸
 switch(i){
 case 1 :
 wskaz_fun = pierwsza ; // ❹
 break ;
 case 2 :
 wskaz_fun = druga ;
 break ;
 default :
 wskaz_fun = NULL ; // ❺
 break ;
 }

 cout << "Wedlug rozkazu ! \n" ;

 if(wskaz_fun) // if not NULL ❻
 {
 for(i = 0 ; i < 3 ; i++){
 (*wskaz_fun)() ; // ❼
 }
 }
}
```

```

}
/*****
int pierwsza()
{
 cout << "funkcja pierwsza ! \n" ;
 return 9 ;
}
*****/
int druga()
{
 cout << "funkcja druga !\n" ;
 return 106 ;
}

```

**Po wykonaniu na ekranie zobaczymy na przykład następujący wydruk**

```

Na ktora funkcje ma pokazac wskaznik ?
pierwsza - 1
czy druga - 2
napisz numer : 2
Wedlug rozkazu !
funkcja druga !
funkcja druga !
funkcja druga !

```

### Kilka słów o tym programie:

- 1 Deklaracje dwóch funkcji. Czytamy: funkcja pierwsza jest funkcją wywoływaną bez żadnych argumentów, która jako rezultat zwraca wartość typu `int`. Funkcja druga – tak samo.
- 2 Definicja wskaźnika mogącego pokazywać na te wyżej zadeklarowane funkcje. To dlatego, że w myśl definicji (czytamy:) `wskaz_fun` jest wskaźnikiem do pokazywania na funkcje wywoływane bez żadnych argumentów, a zwracające jako rezultat wartość typu `int`.
- 3 Pytamy użytkownika, którą funkcję chce wykonywać.
- 4 Zależnie od odpowiedzi (stąd instrukcja `switch`) ustawiamy wskaźnik tak, by pokazywał na żadaną funkcję. W praktyce polega to na wpisaniu do niego adresu danej funkcji.  
I oto natknęliśmy się na inną ważną zasadę:

Nazwa funkcji jest inaczej jej adresem w pamięci

Dzięki temu tak prosto operuje się wskaźnikiem do funkcji. Ustawienie go to po prostu instrukcja

*wskaznik = nazwa\_funkcji ;*

Zauważ, że nie ma tu żadnych nawiasów towarzyszących zwykle nazwie funkcji. Nawiasy takie rozumiemy jako „wywołaj funkcję o tej nazwie”. Tym-



czasem my wcale nie chcemy jeszcze jej wywołać. Na razie tylko o niej mówimy. Umawiamy się ze wskaźnikiem, że ma na tę funkcję pokazywać.

Gdybyśmy zapomnieli i w omawianej linii postawili nawiasy

```
wskaz_fun = pierwsza() ; // błąd !
```

Funkcja zrozumiałaby to jako zachętę do pracy: „Co? mówią o mnie z nawiasami? – to znaczy, że mam ruszyć do akcji! “. Inaczej mówiąc ta linijka oznaczałaby coś takiego: wykonaj funkcję `pierwsza`, a jej rezultat wstaw do wskaźnika `wskaz_fun`.

Ryzyko jednak nie jest takie duże, bo kompilator nas sprawdzi i najprawdopodobniej do tego błędu nie dopuści. Wie on, że funkcja `pierwsza` ma zwrócić wartość typu `int`, a takiej wartości nie można przypisać (podstawić) do `wskaz_fun`, który spodziewa się adresu funkcji. Kompilator widząc tę niezgodność zaprotestuje.



Zapamiętaj: Gdy mówisz komuś o funkcji – to używasz samej nazwy bez nawiasu i argumentów. Nawiasy są operatorem wywołania tej funkcji

- ⑤ Jak wiemy do każdego typu wskaźnika możemy podstawić `NULL`. Nie oznacza to żadnej szczególnej funkcji, ale po prostu wygodnie się potem obecność `NULL`'a sprawdza.
- ⑥ Jeśli nie dostaliśmy od użytkownika żadnej sensownej odpowiedzi – to do wskaźnika wstawiliśmy `NULL`. Oczywiście nie możemy nawet próbować uruchomić funkcji o takim adresie. Dlatego sprawdzamy: jeśli jest we wskaźniku coś innego niż `NULL`, to wtedy uruchomimy tak pokazaną funkcję. Jeśli `NULL` – to przeskakujemy ten fragment.
- ⑦ Nie przerażaj się tą instrukcją. Porównaj dwie instrukcje:

```
pierwsza() ; // czyli to samo co: (pierwsza)() ;
```

oraz

```
(*wskaz_fun)() ;
```

Z dotychczasowych rozmów o wskaźnikach pamiętasz już, że zapis

*\*wskaźnik*

oznacza – „to, na co wskaźnik pokazuje“. W naszym wypadku pokazuje on np. na funkcję `pierwsza`. Dlatego powyższe dwa zapisy są równoważne. Te dwa (w tym wypadku puste) nawiasy to oczywiście sygnał, że chcemy by funkcja wystartowała.



## 17.1 Ćwiczenia z definiowania wskaźników do funkcji

Nie ma nic trudnego we wskaźnikach do funkcji. Jeśli jednak początkujący programiści się ich boją, to powodem jest moim zdaniem zapis. Z tymi gwiazdkami i nawiasami trzeba się oswoić.

Uważam, że swobodę operowania wskaźnikami do funkcji nabyć można tylko wtedy, gdy umie się czytać ich definicje i takie definicje samemu pisać. Dlatego proponuję: skoro już wiemy jak czytać deklaracje (i definicje) wskaźników do funkcji, to

Spróbujmy sami napisać definicję wskaźnika do pokazywania na określony typ funkcji

Wiem, że jest to trochę nudne, ale obiecuję Ci, że jeśli nauczysz się teraz czytania i zapisu definicji wskaźników (także do funkcji) - to będziesz się zawsze czuł pewnie przy programowaniu w C++.



Wyobraźmy sobie, że mamy gdzieś funkcję

```
int muzyka() ;
```

która odgrywa melodyjkę. Chcemy teraz zdefiniować wskaźnik mogący pokazywać na taką funkcję. Wskaźnik ma się nazywać na przykład `www`.

Piszemy więc na środku linijki nazwę `www` i będziemy ją obudowywać dookoła. Mówimy więc `www`

```
www
```

jest wskaźnikiem

```
(*www)
```

służącym do pokazywania na funkcję

```
(*www)()
```

zwracającą wartość typu `int`

```
int (*www)() ;
```

Gotowe! (czyli triumfalny średnik na końcu).

*Jeśli się już taki wskaźnik ma, to na naszą funkcję muzyka ustawia się go choćby taką prostą instrukcją:*

```
www = muzyka ;
```



A teraz inny wskaźnik. Ma się on nadawać do pokazywania na funkcję

```
float dzielenie(int, int);
```

Funkcja ta na przykład dzieli dwie liczby całkowite i zwraca nam rezultat dzielenia. Zbudujmy więc wskaźnik do niej. Niech się on nazywa `ddd`. Zatem mówimy `ddd` i zapisujemy

```
ddd
```

jest wskaźnikiem

```
(*ddd)
```

do funkcji wywoływanej z dwoma argumentami typu `int`

```
(*ddd)(int, int)
```

a zwracającej wartość typu float

```
float (*ddd)(int, int) ;
```

Gotowe!



Nie drzyj jeszcze tej książki! Wiem, że to wszystko jest suche, nudne i formalne, ale mam dla Ciebie teraz prezent. Podam Ci sposób...

Jak nie rozumiejąc niczego, napisać sobie definicję wskaźnika mogącego pokazywać na daną funkcję

Dajmy na to, że chodzi o wskaźnik mogący pokazać na taką funkcję

```
float funkcyjka(int, char);
```

Czyli na funkcję wywoływaną z dwoma argumentami (typu int oraz char), a zwracającą w rezultacie wykonania typ float.

Mój sposób polega na tym, że bierzemy deklarację tej funkcji i w tej deklaracji nazwę funkcji – zastępujemy ujętą w nawias nazwą wskaźnika z gwiazdką z przodu. Czyli dokonujemy takiej zamiany:

`nazwa_funkcji`       $\longrightarrow$       `(*nazwa_wskaźnika)`

W rezultacie otrzymujemy definicję wskaźnika mogącego pokazywać na daną funkcję. Jeśli tak zrobimy z naszą deklaracją, wówczas w rezultacie otrzymujemy zapis

```
float (*nazwa_wskaznika)(int, char);
```

który jest dokładnie tym, o co nam chodziło. Jest to poszukiwana definicja wskaźnika. Żeby się przekonać przeczytamy ten zapis.

|                                                  |                                   |
|--------------------------------------------------|-----------------------------------|
| <code>nazwa_wskaznika</code>                     |                                   |
| <code>(*nazwa_wskaznika)</code>                  | - jest wskaźnikiem                |
| <code>(*nazwa_wskaznika)(...)</code>             | - do funkcji                      |
| <code>(*nazwa_wskaznika)(int, char)</code>       | - wywoływanej z dwoma argumentami |
| <code>float (*nazwa_wskaznika)(int, char)</code> | - a zwracającej typ float         |

Sprytne, prawda? Można to zrobić zupełnie bezmyślnie! No, może prawie bezmyślnie. To dlatego, że potrzebna jest umiejętność przeczytania tego, cośmy zdefiniowali. Tak dla kontroli.

Podam Ci teraz bardzo ważną rzecz - sposób jak czytać skomplikowane deklaracje

Otóż zasada jest taka, że:

- 1) Zaczynamy czytanie od wygłoszenia nazwy, której deklarację czytamy.

- 2) Następnie od tej nazwy posuwamy się w prawo. W prawo dlatego, że tam mogą stać najmocniejsze operatory. Operator wywołania funkcji lub indeksowania tablicy. (Te operatory, jak pamiętamy z tablicy priorytetów, mają jeden za najwyższych priorytetów). To, co napotkamy tam, odczytujemy na głos.
- 3) Jeśli w prawo już nic nie ma, lub natkniemy się na zamykający nawias - wówczas zaczynamy czytanie w lewo. Czytanie w lewo kontynuujemy dotąd, dokąd wszystkiego nie przeczytamy, lub gdy nie natkniemy się na zamykający nas nawias.
- 4) Jeśli napotkamy taki nawias, to wychodzimy na zewnątrz i - będąc już na zewnątrz tego nawiasu znowu zaczynamy w prawo czyli wracamy do punktu 2)

Procedurę przeprowadzamy dopóki nie przeczytamy wszystkiego w tej deklaracji.

## Jak "czytać" ?

Bardzo prosto. Jeśli na naszej drodze napotkamy znaczek

- \* (gwiazdkę) - czytamy: jest wskaźnikiem mogącym pokazywać na...
- (typ1, typ2) - czytamy: jest funkcją wywoływaną z argumentami typ1, typ2 (tu czytamy typy będące w nawiasie), a zwracającą jako rezultat...
- [n] - czytamy: jest n-elementową tablicą

Dygresja:

*Ze względu na fleksję w języku polskim - nie jest to do końca tak eleganckie. Czasem zamiast „jest wskaźnikiem” lepiej by było powiedzieć „będący wskaźnikiem”. Sądzę jednak, że szybko się nauczysz jak to wypowiadać zgrabnie.*

Wypróbujmy ten sposób czytając to, co bezmyślnie wyprodukowaliśmy niedawno, czyli nasz zapis

```
float (*amazonka)(int, char)
```

Jak widzisz teraz samą nazwę zmieniłem, by nas nie sugerowała. Czytamy:

Według punktu 1) mówimy na głos:

amazonka

Według punktu 2) chcemy poruszać się w prawo, ale się nie da, bo od razu napotykaamy ograniczający nas nawias. Zatem przechodzimy do punktu 3)

Według punktu 3) idziemy w lewo i napotykaamy gwiazdkę, co oznacza „jest wskaźnikiem mogącym pokazywać na...”. Mówimy to na głos i wobec tego nasza dotychczasowa wypowiedź to:

amazonka jest wskaźnikiem mogącym pokazywać na

Dalej w lewo się nie da, bo napotykaamy na zamykający nas nawias. Wychodzimy więc na zewnątrz tego nawiasu i zaczynamy znowu posuwać się w prawo. W prawo napotykaamy zapis (int, char), co upoważnia nas do powiedzenia

na głos: „funkcja wywoływana z 2 argumentami (typu `int` i `char`), a zwracająca jako rezultat...”

Tu polonista dostaje zawału serca, a spod kroplówki dochodzi jego jęk o tym, że poprawnie po polsku ma być:

amazonka jest wskaźnikiem mogącym pokazywać na funkcję wywoływaną z dwoma argumentami (typu `int` i `char`), a zwracającą jako rezultat...

W prawo jest już tylko średnik, więc zmieniamy kierunek i czytamy w lewo. Tam jest tylko typ `float`, co czytamy na głos. To już koniec, więc w sumie powiedzieliśmy na głos:

amazonka jest wskaźnikiem mogącym pokazywać na funkcję wywoływaną z dwoma argumentami (typu `int` i `char`), a zwracającą jako rezultat typ `float`.

Jest to dokładnie taki wskaźnik, o jaki nam chodziło. Zatem bezmyślny sposób okazuje się doskonały. O jego geniuszu przekonaliśmy się tylko dzięki temu, że nauczyliśmy się deklaracje czytać.

Oczywiście już chyba rozumiesz, że zastosowałem podstęp, bo przecież - gdy umiesz już czytać deklaracje - sposób nie jest bezmyślny.

Pisanie deklaracji jest tylko trochę trudniejsze niż ich czytanie. Różnica polega na tym, że najpierw mówimy na głos a potem to, co powiedzieliśmy, zapisujemy.

Jednak wydaje mi się, że łatwiej odczytywać niż zapisywać, więc lepiej użyć tego automatycznego sposobu do zapisu, a potem przeczytać by sprawdzić. Gdy nauczysz się już dobrze czytać, czyli zrozumiesz istotę tego czytania, to tym samym będziesz już umiał zapisywać.

Do zagadnienia tego wrócimy jeszcze na stronie 395.



Bardzo rzadko się zdarza by definicje wskaźników do funkcji były bardziej skomplikowane. Tak więc raczej „dla hecy” przytoczę taki wskaźnik – spróbuj go najpierw sam odczytać

```
int (* (*fw)(int, char*))[2] ;
```

Poddajesz się? – No to spróbujmy razem. Zaczynamy od środka, gdzie jest nazwa, a potem w prawo ile się da, potem w lewo ile się da, a jak się już nie da, to wychodzimy na zewnątrz nawiasu i kontynuujemy. Zatem

`fw`

(w prawo się już nie da, więc w lewo)

`*fw`

jest wskaźnikiem (w lewo się już nie da, więc wychodzimy na zewnątrz nawiasu i czytamy z prawej

`(*fw)(int, char *)`

do funkcji wywoływanej z 2 argumentami: typu `int` i typu `char*`, a zwracającej

```
(* (*fw)(int, char *))
```

(dalej w prawo się już nie da, bo jest nawias, próbujemy w lewo, a tam jest \* czyli czytamy:) ...wskaźnik do... (wychodzimy na zewnątrz nawiasu i czytamy z prawej)

```
(* (*fw)(int, char *)) [2]
```

...dwuelementowej tablicy (w prawo się już nie da więc w lewo, a tam stoi tylko int)

```
int (* (*fw)(int, char *)) [2]
```

...obiektów typu int.

W skrócie brzmi to tak: fw jest wskaźnikiem do funkcji wywoływanej z argumentami (int, char), a zwracającej wskaźnik do dwuelementowej tablicy typu int.

W tym przykładzie chodziło mi bardziej o to, byś zobaczył **jak** próbować takie zagadki rozwiązać. Nie przejmuj się też jeśli Ci to nie poszło. W mojej codziennej praktyce nigdy nie wymyślałem tak skomplikowanych wskaźników do funkcji.

## Zastrzeżenia

Należy podkreślić, że

typ wskaźnika do funkcji musi się zgadzać z typem funkcji.

Jeśli mamy np. funkcję

```
char fun(float, int, int) ;
```

to nie możemy na nią pokazać wskaźnikiem

```
int (*wsk)()
```

Kompilator się na to nie zgodzi. Do pokazania na taką funkcję nadaje się tylko wskaźnik

```
char (*www)(float, int, int) ;
```

Dlaczego? Dlaczego wskaźniki nie są uniwersalne? Dlaczego nie jest to po prostu wskaźnik do funkcji i już?!

Tak nie jest – znowu dla naszego dobra. Kompilator musi wiedzieć jaki jest typ wskaźnika po to, by wykryć czy nie pomyliliśmy się przy wysłaniu argumentów. Przy wywołaniu funkcji za pomocą wskaźnika

```
www(3.14, 1, 5);
```

kompilator sprawdza nas czy ta lista argumentów zgadza się z listą, która stoi przy definicji wskaźnika www. A czy ta lista zgadza się z listą oczekiwaną przez wskazywaną funkcję? Musi się zgadzać, bo inaczej kompilator odmówiłby ustawienia tego wskaźnika na tę funkcję. Po prostu nie dałoby się wykonać przypisania

```
wsk = fun ;
```

```
// Błąd – niezgodność typu
// wskaźnika i funkcji
```

www = fun ;

// O.K. - zgadzają się te typy

I jeszcze jedno: Na wskaźnikach do funkcji nie wolno robić operacji arytmetycznych. To oczywiście jest intuicyjnie wyczuwalne. Co bowiem miałyby znaczyć odjęcie od siebie dwóch wskaźników do funkcji? Bezsens.



Tyle do tej pory mówiliśmy o definicjach wskaźników, że mogłeś odnieść wrażenie, że to coś trudnego. Wręcz przeciwnie. Przypominają one poznane wcześniej wskaźniki do zwykłych obiektów. W szczególności trzeba przypomnieć, że tak, jak w przypadku każdego wskaźnika, i ten nie pokazuje na nic, dopóki nie wstawimy do niego adresu funkcji, na którą ma pokazywać.

Podsumujmy naszą dotychczasową wiedzę o wskaźnikach do funkcji:

Podobnie jak w przypadku tablic nazwa funkcji jest równocześnie adresem jej początku (- czyli adresem miejsca w pamięci, gdzie zaczyna się kod odpowiadający instrukcjom tej funkcji).

Tę zasadę także proponuję przykleić sobie nad biurkiem. Przyda się ona jednak trochę rzadziej, bo wskaźnikami do funkcji nie posługujemy się aż tak często.

Jeśli zdefiniowaliśmy sobie wskaźnik

```
int (*wf)() ;
```

to instrukcja

```
wf = muzyka ;
```

sprawia, że od tej pory wskaźnik zaczyna pokazywać na funkcję muzyka. A zatem możemy tę funkcję wywołać teraz za pomocą jej prawdziwej nazwy lub za pomocą wskaźnika. Zauważ, że są dwa sposoby z użyciem wskaźnika, drugi jest wyraźnie czytelniejszy:

```
muzyka() ; // za pomocą nazwy
(*wf)() ; // za pomocą wskaźnika
wf() ; // za pomocą wskaźnika
```

Wyrażenie `(*wf)` oznacza: „skocz do miejsca w pamięci, na które pokazuje wskaźnik – zapewniam, że jest tam funkcja” – a stojące dalej dwa nawiasy mówią: „proszę tę funkcję wykonać”.

Gdyby jednak naprawdę tak miało wyglądać używanie wskaźników do funkcji, to korzyść nie byłaby duża.

Kiedy zatem tak naprawdę wskaźnik do funkcji może się przydać ?

W sytuacjach podobnych do tych, w których najczęściej używa się zwykłych wskaźników:

- ❖ Przy przesyłaniu argumentów do funkcji. Adres innej funkcji można też wysłać jako argument. To tak, jakbyśmy powiedzieli funkcji: tu masz

takie argumenty, a dodatkowo tu jeszcze adres funkcji, którą masz u siebie wykonać.

- ❖ Do tworzenia tablic ze wskaźników do funkcji ; w takiej tablicy mamy jakby listę działań i odtąd możemy mówić: – „a teraz wykonajmy funkcję numer 5”.

O obu tych sprawach powiemy szerzej w następnych paragrafach.

## 8.17.2 Wskaźnik do funkcji jako argument innej funkcji

Wyobraź sobie taką sytuację: piszesz bardzo ogólną funkcję, która służy do jakiejś rozmowy z użytkownikiem. Funkcja zadaje pytanie, na które użytkownik odpowiada „tak” lub „nie”. W trakcie, gdy użytkownik będzie się zastanawiał nad odpowiedzią, funkcja ma coś robić. To, co ma robić, przysyłamy do niej jako argument. Inaczej - nazwę funkcji, która ma być „w międzyczasie” wykonywana – przysyłamy jako argument.

Oto przykład. Korzysta on z dodatkowych funkcji bibliotecznych dostępnych w kompilatorze Borland C++ (wersja 3.1) dla komputera klasy IBM PC. Jeśli posługujesz się innym kompilatorem, to niektóre z tych funkcji mogą mieć inne nazwy – nie są to bowiem funkcje standardowe. Jednak nie o to tu chodzi – mówimy tu tylko o tym, jak do funkcji wysyła się nazwę innej funkcji. Zresztą zobaczmy!

```

/*-----
Program został napisany z wykorzystaniem niektórych
funkcji bibliotecznych charakterystycznych dla
kompilatora
 Borland C++
-----*/

#include <iostream.h> // dla cin, cout
#include <ctype.h> // dla tolower
#include <conio.h> // dla kbhit
#include <dos.h> // dla sound, nosound, delay

int pytanie(char *pyt, void (*wskaźnik_funkcji)()) ; // ②
void muzyczka() ; // ③
void wiatraczek() ;
void kurs() ;
/*****
main()
{
 int i ;
 cout << "Samolot gotowy \n" ;
 while(1)
 {
 i = pytanie("Czy mam juz startowac ?",
 muzyczka) ; // ④
 if(i)
 {
 cout << "Uwaga, startujemy !\n" ;
 break ;
 }
 }
}

```



```

 else
 {
 cout << "nie to czekam...\n " ;
 }
 }
 cout << "Lecimy...\n" ;
 switch(pytanie("Czy dodac gazu ? ",wiatraczek)){// 5
 {
 case 1 :
 cout << "Zrobione !\n" ;
 break ;
 case 0 :
 cout << "Nie zmieniam !\n" ;
 break ;
 }
 pytanie("dobrze sie leci, prawda ? ", kurs); // 6
}
/*****/
int pytanie(char *pyt, void (*wskaznik_funkcji)())
{
 char c ;
 cout << pyt << endl;
 while(1)
 {
 (*wskaznik_funkcji)() ; // 7
 cin >> c ;
 switch(tolower(c))
 {
 case 't' :
 return 1;
 case 'n' :
 return 0 ;
 default :
 cout<< "odpowiedz 't' lub 'n' \n" ;
 break ;
 }
 }
}
/*****/
void muzyczka()
{
 int i ;
 while(!kbhit()) // 8
 {
 for(i=100 ; i < 1200 ; i+=100)
 {
 sound(i) ;
 delay(250);
 }
 nosound();
 }
}
/*****/
void wiatraczek() // 9
{
 char t[] = { ' ', '\\',

```

```

 '-', '/');
int i ;

while(!kbhit())
{
 cout << " " << t[(i++) % 4] << "\r";
 delay(200);
}
}
/*****
void kurs()
{
 int i ;
 while(!kbhit())
 {
 cout << "kurs " << (239 + ((i++) % 4))
 << "... \r";
 delay(200);
 }
}

```

**Trudno dokładnie pokazać to, co zobaczymy na ekranie, jednak w przybliżeniu będzie to wyglądało tak:**

```

Samolot gotowy
Czy mam juz startowac ? T
Uwaga, startujemy !
Lecimy...
Czy dodac gazu ?
\
T
Zrobione !
dobrze sie leci, prawda ?
kurs 240...
N

```

*<- tu ciągle kręci się wiatraczek*

*<- aktualizowane informacje o kursie*

## Przyjrzyjmy się teraz ciekawszym miejscom tego programu

- 1 Powiedziałem, że program jest napisany z użyciem pewnych funkcji bibliotecznych dostępnych dla kompilatora Borland C++. Jak pamiętamy, aby posłużyć się funkcją biblioteczną należy do programu włączyć (wstawić) plik nagłówkowy zawierający deklarację tej funkcji. W naszym wypadku jest to nawet kilka nagłówków. W komentarzach podałem jakie funkcje wymagają danego pliku.
- 2 Deklaracja funkcji. Funkcja pytania jest właśnie tą, która zawiera istotę naszego przykładu. Z deklaracji tej widzimy, że pierwszym argumentem jest string. (Tak prześlemy tekst pytania, które funkcja ma zadać). Natomiast drugi argument służy do przesyłania nazwy funkcji. Jest to wskaźnik do funkcji wywoływanej bez żadnego argumentu i zwracającej rezultat typu void (czyli nic nie zwracającej).
- 3 Deklaracje trzech funkcji. To właśnie na te funkcje będziemy pokazywali wskaźnikiem.

- 4 Istota tego programu. Wywołanie funkcji pytanie. Jako drugi argument wysyłamy jej nazwę funkcji - muzyczka. Zauważ, że nazwa jest bez nawiasów. To dlatego, że jedynie mówimy o funkcji muzyczka, a nie chcemy by właśnie teraz startowała.
- 7 Oto co się dzieje wewnątrz funkcji pytanie. Do funkcji wysłaliśmy nazwę funkcji (np. muzyczka). Tymczasem funkcja definiuje sobie wskaźnik do funkcji i inicjalizuje go (przysyłanym jako argument) adresem funkcji muzyczka. Czyli odpowiada to jakby instrukcji

```
void (*wskaznik_funkcji)() = muzyczka ;
```

aby wywołać teraz funkcję pokazywaną przez ten wskaźnik wystarczy napisać

```
(*wskaznik_funkcji)() ;
```

Wiem co pomyślałeś – że to prawie identyczne jak to powyżej – ależ oczywiście! Według zasady, że składnia instrukcji używającej wskaźnika przypomina składnię w jego definicji.

Dzięki tej wspaniałej zasadzie mniej się musimy uczyć. (A swoją drogą to także i tę zasadę napisz sobie nad biurkiem).

- 8 Jest to jedna z tych funkcji „zabijających czas”. Sama funkcja muzyczka nie ma w sobie nic specjalnego. Zwykła funkcja. Tyle, że jej nazwę ktoś komuś przesyłał. Jediną interesującą rzeczą w tej funkcji jest nieskończona pętla przerywana w momencie, gdy użytkownik tylko dotknie klawiatury. To robi właśnie funkcja biblioteczna kbhit.<sup>†)</sup> Funkcja ta zwraca 0, gdy nikt nie nacisnął niczego na klawiaturze. Jeśli ktoś coś nacisnął, to zwraca 1, ale wcale nie interesuje ją co zostało naciśnięte. To odczyta dopiero instrukcja

```
cin >> c ;
```

w funkcji pytanie.

W funkcji muzyczka obecne są wywołania funkcji bibliotecznych:

- sound – uruchamiająca generator wytwarzający dźwięk o podanej w hercach wysokości,
- nosound – zatrzymująca generator dźwięku,
- delay – funkcja powodująca zwłokę czasową o żądanym czasie trwania (zadany w milisekundach).

W sumie więc nasza funkcja muzyczka wygrywa pożałuj się-Boże melodyjkę. Czas trwania jednego dźwięku: 250 milisekund.

- 9 Funkcja wiatraczek zabawia użytkownika rysując kręcący się wiatraczek. Jest on robiony przez kolejne rysowanie w tym samym miejscu na ekranie znaków

```
| \ / - \ /
```

Te znaki są umieszczone w tablicy znaków. (Zauważ jaki chwyt musiał zostać zastosowany, by można było umieścić tam \ (bekslesz)). Kolejne wypisywanie

---

†) keyboard hit – ang. uderzenie w klawiaturę

na ekran tych czterech znaków jest zrobione za pomocą operatora dzielenia modulo 4. Zmienna i cały czas rośnie, a mimo to indeks tablicy jest zawsze z przedziału 0 - 3

- ⑤ Wywołanie funkcji pytanie z innym zabaviaczem - wiatraczek. To wywołanie jest trochę trudniejsze, bo zapakowane do instrukcji switch. Ciągłe jednak zasada jest ta sama.
- ⑥ Wywołanie funkcji pytanie z zabaviaczem kurs. Nie reagujemy tu wcale na odpowiedź.



Program nie jest napisany elegancko. Przykładowo jeśli wciśniesz literę 't', ale jeszcze nie wciśniesz ENTER to muzyczka, czy wiatraczek zatrzymują się. Oczywiście da się to zrobić lepiej, jednak z tym musimy poczekać do rozdziału o operacjach wejścia i wyjścia. Tam poznamy wszystkie takie sztuczki.

Z tego paragrafu dowiedzieliśmy się, że wskaźnik do funkcji może być argumentem innej funkcji i jest to zupełnie zwyczajna sytuacja. Do tego stopnia, że argument ten może też mieć wartość domniemaną. Gdyby deklaracja naszej funkcji pytanie wyglądała tak:

```
int pytanie(char *pyt, void (*wskaźnik_funkcji)() = muzyczka);
```

to możliwe by było takie wywołanie funkcji:

```
pytanie("tekscik");
```

co odpowiada jak wiadomo

```
pytanie("tekscik", muzyczka);
```

*Drobna uwaga: deklaracja funkcji pytanie powinna być wówczas w programie o jedną liniękę niżej – gdyż kompilator powinien już w tym momencie znać deklarację funkcji muzyczka.*

## 8.17.3 Tablica wskaźników do funkcji

Wiemy już, że w tablicach można przechowywać wskaźniki (czyli adresy) do jakichś obiektów. Można też sporządzić tablicę składającą się ze wskaźników do funkcji.

Oto przykład tablicy wskaźników do funkcji:

```
void (*(twf[5]))();
```

Przeczytajmy tę definicję, jak zwykle zaczynając od środka, czyli od nazwy

twf - twf

[5] - ...jest 5 elementową tablicą

\* - ...wskaźników...

() - ...do funkcji wywoływanej bez żadnych argumentów...

void - ...a zwracając typ void (czyli nic).

Jeśli pamiętamy, że operator `[]` jest o wiele mocniejszy od operatora `*`, to tę samą definicję możemy napisać po prostu tak:

```
void (*twf[5]) () ;
```

Zastanówmy się teraz co naprawdę zdefiniowaliśmy i kiedy może się nam to przydać. Otóż mamy tablicę, w której możemy przechować wskaźniki pokazujące na jakieś wybrane funkcje naszego programu. Jest to jakby lista czynności, które można wykonywać. Możemy załadować taką tablicę, a potem komentować: a teraz proszę wykonać funkcję trzecią, a teraz piątą.

Żeby było jaśniej pokażmy to na przykładzie. W programie tym występują trzy funkcje, na które pokazujemy wskaźnikami z tablicy. Dla ułatwienia stosujemy tu te same funkcje wiatraczek, kurs i muzyczka. Ciał tych funkcji dla skrócenia nie zamieszczam. Są takie same jak w poprzednim paragrafie.

```
#include <iostream.h> // dla cin, cout
#include <ctype.h> // dla tolower
#include <conio.h> // dla kbhit
#include <dos.h> // dla sound, nosound, delay
#include <stdlib.h> // dla exit

void muzyczka() ;
void wiatraczek() ;
void kurs() ;
/*****
main()
{
void (*twf[3])() = { muzyczka, wiatraczek, kurs } ; // ❶
int co ;

while(1)
{
cout << "Menu : "
 << "\t0 - muzyczka\n\t1 - wiatraczek \n\t"
 << "2 - kurs\n\t3 - koniec programu\n\n"
 << "podaj numer zadanej akcji : " ;
cin >> co ; // ❷
switch(co)
{
case 0 :
case 1 :
case 2 :
 (*twf[co])() ; // ❸
 break ;
case 3 :
 exit(1) ;

default:
 break ;
}
}
}
/*****/
void muzyczka()
{ /* ... */ }
```

```

/*****
void wiatraczek()
{
 /* ... */
}
*****/
void kurs()
{
 /* ... */
}

```

 Wygląd ekranu po wykonaniu tego programu zależy oczywiście od wyboru „opcji” naszego menu.

```

Menu :
0 - muzyczka
1 - wiatraczek
2 - kurs
3 - koniec programu

```

podaj numer zadanej akcji : 2

kurs 239...

kurs 240...

.....

```

Menu :
0 - muzyczka
1 - wiatraczek
2 - kurs
3 - koniec programu

```

podaj numer zadanej akcji : 1

\ <- kręci się wiatraczek

```

Menu :
0 - muzyczka
1 - wiatraczek
2 - kurs
3 - koniec programu

```

podaj numer zadanej akcji : 0

<- tutaj gra muzyczka

```

Menu :
0 - muzyczka
1 - wiatraczek
2 - kurs
3 - koniec programu

```

podaj numer zadanej akcji : 3

## Uwagi :

- Definicja 3 elementowej tablicy wskaźników do funkcji (funkcji wywoływanych bez żadnych argumentów i zwracających void). Od razu inicjalizujemy tę tablicę tak, że wskaźniki pokazują na nasze funkcje. Zauważ, że w klamrze są tylko nazwy funkcji - bez nawiasów. Pamiętajmy, że nazwa funkcji jest adresem jej początku.

- ② Menu wypisywane na ekranie daje możliwość wybrania żądanej akcji. W tablicy czekają już wskaźniki pokazujące na różne funkcje. Tutaj tylko prosimy o podanie numeru funkcji, którą mamy uruchomić.
- ③ Skoro wybraliśmy numer funkcji, to pozostaje tylko ją uruchomić. Dzieje się to właśnie tą instrukcją. Znowu zaznaczam, że moment użycia wskaźnika przypomina składnię jego definicji (porównaj z ①).



Jak widać dzięki tablicy wskaźników możemy wydać polecenia: jeśli tak – to wykonaj funkcję nr... Jest to jeden ze sposobów sporządzania menu.

Oczywiście są także inne sposoby sporządzania menu:

```
switch(co_robic)
{
 case 1 :
 muzyczka(); break ;
 case 2 :
 wiatraczek(); break ;
 case 3 :
 wiatraczek(); break ;
 default :
 break ;
}
```

Jednak dzięki tablicy wskaźników można na przykład w trakcie pracy programu zmienić jeden z nich tak, by pokazywał na inną funkcję. Czasem takie operacje są przydatne. Jeśli na przykład mamy w programie funkcję

```
void symfonia() ;
```

a nastąpi gdzieś w programie instrukcja

```
twf[0] = symfonia ;
```

to od tej pory po wybraniu wariantu 0 zamiast funkcji muzyczka będzie się wykonywała funkcja symfonia.

## 18 Argumenty z linii wywołania programu

Spotkałeś na pewno programy, które uruchamia się pisząc obok nazwy programu dodatkowo jakieś opcje. Jest to eleganckie rozwiązanie problemu przesłania parametrów do programu. Rozwiązanie takie pozwala też pisać programy, które wywołuje się tak, jakby były komendami systemu operacyjnego.

Wyobraź sobie, że napisałeś ładny program, który - w momencie, gdy się zaczyna - maluje na ekranie piękną kolorową planszę tytułową.

Pracując nad modyfikacjami takiego programu – wielokrotnie musisz go uruchamiać. Oczywiście za każdym razem najpierw plansza tytułowa. Jeśli uruchamiasz program dziesiątki razy, to w pewnym momencie ta plansza Cię zdenerwuje. Chciałbyś by nie pojawiała się wtedy, gdy tego nie chcesz.

Oczywiście jest rozwiązanie: program startuje i pyta „czy mam pokazać planszę?”. Na odpowiedź „nie” przeskakuje wywołanie funkcji zajmującej się rysowaniem planszy. Jest to rozwiązanie, które nie jest żadnym rozwiązaniem. Uruchamiając teraz kilkadziesiąt razy program musisz kilkadziesiąt razy odpowiadać na pytanie „Czy mam...?” Świąty by nie wytrzymał!

Potrzebna jest możliwość, by już w momencie uruchamiania programu móc przesłać do programu informację o tym, że nie życzymy sobie planszy.

## Jak zatem przesłać do programu parametry ?

Sprawa jest bardzo prosta. Przykładowo jeśli nasz program nazywa się „pelikan”, to wywołujemy go pisząc jego nazwę, a dalej kolejno parametry

```
pelikan param1 77.2
```

w naszym wypadku są to: string: "param1" i liczba 77.2

Wysłać parametry to jeszcze nie wszystko. Program powinien umieć je odebrać. To też nie jest trudne. Aby odebrać tak wysłane parametry, funkcję main musimy zapisać w taki sposób:

```
main(int argc, char *argv[])
{
 // ... normalna treść funkcji main
}
```

Jak widzimy przesłanie parametrów do programu polega na tym, że funkcja main dostaje w prezencie od systemu operacyjnego dwa argumenty. Pierwszy typu int, a drugi nieco bardziej skomplikowany... To, jakie im nadamy nazwy w naszym programie, zależy wyłącznie od nas. Zwyczajowo nazywają się:

**argc** – od ang.: argument counter – licznik argumentów. Mówi nam ile parametrów system operacyjny wysłał do programu. Licznik ten ma co najmniej wartość 1. (Czy chcemy czy nie – system wysyła nam jako parametr nazwę programu, który właśnie uruchamiamy).

**argv** – od ang.: argument vector – tablica argumentów. Jest to wskaźnik do tablicy, w której poszczególnymi elementami są adresy stringów. Te stringi, to właśnie nasze kolejne parametry wywołania programu. Bardziej formalnie – zapis

```
char *argv[]
```

czyta się: argv jest tablicą wskaźników do (ciągów) znaków.



Zamiast długich tłumaczeń proponuję spojrzeć na rysunek poniżej.



Łatwo zauważyć, że poszczególne parametry są zapisane jako ciągi znaków pod następującymi adresami

|          |   |           |
|----------|---|-----------|
| *argv[0] | - | "pelikan" |
| *argv[1] | - | "param1"  |
| *argv[2] | - | "77.2"    |

Poniższy przykład pokazuje jak to zrealizować w programie

```
#include <iostream.h>
#include <stdlib.h>

main(int argc, char * argv[])
{
 cout << "Wydruk parametrow wywolania : \n" ;

 for(int i = 0 ; i < argc ; i++)
 {
 cout << "Parametr nr "<< i
 << " to string: " << argv[i]
 << endl ;
 }

 /*———zamienimy string na liczbe———*/
 float x ;
 x = atof(argv[2]);
 x = x + 4;
 cout << "x = " << x << endl ;
}
```

**Jeśli program ten wywołamy tak:**

pelikan param1 77.2  
to na ekranie zobaczymy

```
Wydruk parametrow wywolania :
Parametr nr 0 to string: pelikan
Parametr nr 1 to string: param1
Parametr nr 2 to string: 77.2
x = 81.2
```



## Pamiętajmy jednak, że:

wszystkie parametry zostają przysłane jako stringi. Czyli parametr nr 2 przysłany zostaje nie jako liczba, ale jako ciąg takich znaków: cyfra 7, cyfra 7, kropka, cyfra 2, NULL.

Jeśli chcemy zamienić taki string na liczbę – możemy posłużyć się jedną z funkcji bibliotecznych. Nazywa się ona `atof` – od ang: Ascii TO Float. Deklaracja tej funkcji jest w pliku nagłówkowym `stdlib.h`.

Widzimy, że w programie zamieniliśmy ten parametr na liczbę i złożyliśmy w zmiennej `x`, na której możemy już przeprowadzać operacje matematyczne.

W rozdziale o operacjach wejścia/wyjścia poznamy o wiele wygodniejsze sposoby odbierania przysłanych argumentów (str. 688).



---

## 9 Przeładowanie nazw funkcji

---

Są sytuacje, gdy nazwa funkcji doskonale określa akcję, którą wykonuje. Jeśli więc Czytelniku zamierzasz swoim funkcjom nadawać nazwy typu `X24a15c()` to możesz w ogóle nie czytać tego rozdziału.

---

### 9.1 Co to znaczy: przeładowanie

W języku angielskim przeładowanie (overloading) jakiegoś słowa oznacza, że ma ono więcej niż jedno znaczenie. Powiedzmy obrazowo: słowo jest przeładowane znaczeniami. Zjawisko to występuje także z nazwami funkcji w języku C++.

Na podstawie swoich doświadczeń z innymi językami programowania przywykłeś zapewne do faktu, iż w programie może być tylko jedna funkcja o danej nazwie. Używając tej nazwy mówiliśmy kompilatorowi o jaką funkcję nam w danym momencie chodzi. Gdybyśmy mieli w programie dwie funkcje o tej samej nazwie, to kompilator nie wiedziałby, którą z nich w danym momencie mamy na myśli, i którą z nich ma uruchomić.

Kompilator C++ jest inteligentniejszy. Wyobraź sobie takie dwie funkcje:

```
void wypisz_na_ekran(int);
void wypisz_na_ekran(char, float, char);
```

Pytanie: Gdybyś to Ty był kompilatorem C++ i napotkał w programie wywołanie

```
wypisz_na_ekran('A', 3.14, 'E');
```

to czy miałbyś jakieś wątpliwości o wywołanie której z dwóch powyższych funkcji chodzi?

Koń, jaki jest – każdy widzi! – mówi stara encyklopedia. Tę zasadę stosuje się także czasem w programowaniu. Otóż jeśli przyjąć zasadę, że funkcję rozpoznaje się nie tylko po jej nazwie, ale także po typie argumentów, to w pewnych

warunkach może istnieć więcej niż jedna funkcja o tej samej nazwie. Byle tylko te dwie funkcje różniły się argumentami.

To zjawisko nazywamy przeładowaniem nazwy funkcji. Uściślijmy:

Przeładowanie nazwy funkcji polega na tym, że w danym zakresie ważności jest więcej niż jedna funkcja o takiej samej nazwie. To, która z nich zostaje w danym wypadku uaktywniona zależy od typu argumentów wywołania jej.

Funkcje takie mają tę samą nazwę, ale muszą się różnić liczbą lub typem poszczególnych argumentów. Znaczy to, że może być np. jedna taka funkcja z jednym argumentem typu `int`. Próba zdefiniowania w tym samym zakresie ważności drugiej takiej funkcji o tej samej nazwie i identycznym zestawie argumentów – czyli tutaj jedynym argumentem `int` – uznana zostanie za błąd.

Dla porządku trzeba dodać, że we wcześniejszych wersjach języka obowiązywało specjalne słowo kluczowe `overload` – ostrzegające kompilator, że zamierzamy daną nazwę funkcji przeładowywać. W nowszych wersjach języka to słowo nie jest już konieczne. Ze względów na zgodność jest jednak tolerowane. Zwykle jednak kompilator ostrzega, że jest ono staromodne.

Oto przykład programu z przeładowanymi funkcjami:

```
#include <iostream.h>
void wypisz(int liczba); // ❶
void wypisz(char znak1, float x, char *tekst); // ❷
void wypisz(int liczba, char znak); // ❸
void wypisz(char znak, int liczba); // ❹
/*****/
main()
{
 wypisz(12345); // ❺
 wypisz(8, 'X');
 wypisz('D', 89.5, " stopni Celsjusza ");
 wypisz('M', 22);
}
/*****/
void wypisz(int liczba)
{
 cout << "Liczba typu int : " << liczba << endl ;
}
/*****/
void wypisz(char znak1, float x, char *tekst)
{
 cout << "Blok " << znak1 << " : "
 << x << tekst << endl ;
}
/*****/
void wypisz(int liczba, char znak)
{
 cout << znak << ") " << liczba << endl ;
}
/*****/
void wypisz(char znak, int liczba)
```

```
{
 cout << liczba << " razy wystapil stan "
 << znak << endl ;
}
```



## Po wykonaniu tego programu na ekranie ujrzymy

```
Liczba typu int : 12345
X) 8
Blok D : 89.5 stopni Celsiusza
22 razy wystapil stan M
```



## Komentarz:

- 1 Program ten nie byłby niczym zajmującym, gdyby nie to, że posługujemy się w nim czterema funkcjami o identycznej nazwie. Widzimy tu deklaracje tych funkcji. Nazwa funkcji `wypisz` jest czterokrotnie przeładowania. Poszczególne funkcje różnią się typem i ilością argumentów.
- 2 Zastrzeżenie o odmiennym typie argumentów dotyczy także kolejności. Mogą być dwie funkcje, które pracują na tym samym zestawie argumentów – porównaj z deklaracją funkcji 3. Nie jest to nic dziwnego. W obu wypadkach chodzi o argumenty `int` oraz `char`, jednak ich inna kolejność sprawia, że funkcje są łatwo rozróżniane. Jest tylko jedna taka funkcja o nazwie `wypisz`, której pierwszy argument ma typ `int` a drugi `char`. Podobnie jest tylko jedna funkcja `wypisz`, której pierwszy argument jest typu `char` a drugi `int`.
- 4 Wywołania funkcji `wypisz`. Kompilator przygląda się argumentom i stąd doбира funkcje, do której one pasują. O tym, że przychodzi mu to łatwo, przekonuje nas to, co otrzymujemy na ekranie w rezultacie wykonania programu. Przedstawiony przykład przekonał Cię chyba jak dziecinnie łatwe jest przeładowywanie nazw funkcji.



Tu chciałbym zrobić zastrzeżenie: co prawda to *nazwa* funkcji jest przeładowana, jednak często będziemy mówić, że to po prostu funkcja jest przeładowana.

Podsumujmy:

Przeładowanie nazwy funkcji polega na nadaniu jej wielu znaczeń. Istnieje bowiem kilka funkcji o identycznej nazwie. To, która „wersja” funkcji jest uruchamiana zależy od kontekstu, w jakim została użyta – czyli od towarzyszących tej nazwie argumentów wywołania.

To tak, jak w życiu. Mamy funkcję „wywołaj”. Nazwa `wywołaj` jest przeładowana znaczeniami. Powiedzenie: „wywołaj” z argumentem „ducha” rozumiane jest inaczej niż powiedzenie: „wywołaj” z argumentem „szefa z zebrania”, a jeszcze co innego znaczy „wywołaj” z argumentem „film kolorowy”. Nie ma jednak nieporozumień, bo kontekst jest jasny.

## Słowa, słowa, słowa - uwaga do wydania czwartego

Pisząc w Niemczech tę książkę nie wiedziałem, że pewien polski autor, przetłumaczył termin *overloaded* – na polski jako "przeciążony". Dziś, wiedząc już o tym – nadal z całą świadomością pozostaje przy "przeładować". W literaturze możesz jednak spotkać także echa tamtego tłumaczenia.

Warto dodać, że na niemiecki przetłumaczono ten termin jako *überladen* (przeładować), a nie *überlasten* (przeciążyć - np. obwód elektryczny). Podobnie na francuski - przetłumaczono jako "recharger" (naładować od nowa). Na włoski ten termin przetłumaczono jako *sovraccaricare* - a słowo to nie oznacza wcale czegoś, co jest ciężkie. (Porównaj: *carica* - ładunek, *caricare* ładować [także broń]). Te fakty potwierdzają słuszność tłumaczenia: "przeładować".

Chodzi przecież o to, by z danej nazwy zdjąć jedno znaczenie i naładować nowe.

Niezależnie jednak od tego wszystkiego drogi Czytelniku - mów jak chcesz, byłeś tylko wiedział o przeładowanie czego i czym - tu chodzi.



## Kiedy przeładowywać ?

Przeładowywać nazwę funkcji tylko dlatego, że wolno – byłoby głupotą. Jak to często bywa – i tego narzędzia należy używać z pewną logiką. W naszym poprzednim przykładzie przeładowaliśmy nazwę funkcji *wypisz* dlatego, że w różnych wariantach wykonywała ona analogiczną akcję na różnych zestawach obiektów. Zawsze chodziło na wypisanie czegoś na ekranie. Te cztery funkcje miały pewną cechę wspólną: wszystkie wypisywały coś na ekranie. Ta wspólna cecha jest najczęściej nazwą danego zestawu funkcji.

(Na przykład liczenie średniej z zadanych różnych obiektów, albo wyławianie wartości maksymalnej, albo sortowanie).

Można by też zapytać odwrotnie: kiedy **nie** przeładowywać nazwy funkcji? Odpowiedź jest prosta: Wtedy, gdy nie potrzebujemy tej samej nazwy dla różnych działań. Nie ma sensu nadawanie tej samej nazwy funkcji, która liczy logarytm, co funkcji wygrywającej melodyjkę.

Problem moim zdaniem nie jest poważny i nie ma ryzyka, że ktoś zechce przeładować wszystkie nazwy funkcji. Najczęściej nazwa funkcji określa istotę jej działania. To znaczy można spotkać nazwę funkcji

```
oblicz_srednia(...)
```

a prawie nigdy nazwę funkcji

```
x12_em4(...)
```

Dyktuje to zmysł praktyczności. Gdy więc w innym miejscu programu zechcemy liczyć średnią dla innych obiektów (np. nie dla liczb całkowitych tylko zespolonych), to wówczas przeładowanie nazwy funkcji nasunie się nam samo. Bez powodu takie skojarzenia i pomysły raczej nam nie grożą.



Myszę, że do tej pory nie udało mi się jeszcze namówić Cię na przeładowywanie nazw funkcji. Nie było to moim zamiarem. Tak naprawdę, to prawdziwe zastosowanie przeładowania nazw funkcji poznamy dopiero później, gdy mówić będziemy o definiowaniu swoich własnych typów.

## 9.2 Bliższe szczegóły przeładowania

Jak już wiemy, przeładowanie oznacza, że są w dwie (lub więcej) funkcje o identycznej nazwie, ale różniące się listą argumentów. Błędem by była próba definicji dwóch funkcji o identycznej nazwie i identycznej liście argumentów.

```
int rysuj(int aaa);
int rysuj(int zmienna); // błąd - taka funkcja już jest
int rysuj(float x); // o.k. - takiej jeszcze nie ma
int rysuj(int n, int m); // o.k. - takiej też jeszcze nie było
```

Zwracam jednak uwagę, że to nie powtórna deklaracja wywoła błąd. Pamiętamy, że nawet bez żadnego przeładowywania – deklaracja (funkcji czy zmiennej) może wystąpić wielokrotnie. To nic nie przeszkadza. To tak, jakbyśmy kompilatorowi przypominali coś wielokrotnie. Coś, o czym on już dawno wie, a na te dalsze powtórzenia nie reaguje, sądząc że mamy sklerozę. O ile możliwe są wielokrotne deklaracje, o tyle definicja może być tylko jedna.

Zatem w wypadku naszych funkcji `rysuj` – kompilator nie zareaguje jeszcze przy powyższych deklaracjach. Zaprotestuje dopiero przy definicjach tych funkcji. Czyli tam, gdzie jest ciało (treść) tych funkcji. Konkretnie wtedy, gdy napotka drugą definicję funkcji o nazwie `rysuj`, a lista argumentów będzie taka, jaką już w innej definicji funkcji `rysuj` kiedyś napotkał.



Przy przeładowywaniu ważna jest tylko odmiennność argumentów. Natomiast typ zwracany przez funkcję nie jest brany pod uwagę.

Zatem niepoprawna jest próba takiego przeładowania:

```
int akcja(int) ;
float akcja(int) ; // błąd !
```

W trakcie kompilacji takie przeładowanie uznane zostanie za błąd.

Można zdefiniować funkcje o identycznej nazwie i takim samym typie argumentów, pod warunkiem, że kolejność argumentów będzie inna. Poniższe funkcje są zatem poprawnym przeładowaniem

```
int fun(int, float) ;
int fun(float, int) ;
```

### A teraz zagadka

Mamy taki zestaw przeładowanych funkcji:

```
void dru(int);
void dru(float);
void dru(int, int);
void dru(int, float);
```

a w programie występuje takie wywołanie funkcji:

```
dru(5, (int) 62.34);
```

Która z tych funkcji zostanie uruchomiona? Odpowiedź jest prosta: Pierwszy argument ma typ `int`. Drugi argument to wyrażenie, w którym zastosowano rzutowanie. Liczba 62.34 zamieniona jest operatorem rzutowania na liczbę typu `int` (czyli wartością wyrażenia jest 62). Wartością wyrażenia jest typ `int`, a zatem zostanie uruchomiona wersja

```
void dru(int, int);
```

Druga zagadka. Czy poprawne jest takie przeładowanie funkcji?

```
void zz(int) ;
void zz(unsigned int);
```

Tak, poprawne! Albowiem typ `unsigned int` oraz typ `int` to różne typy.

## Co zrobić, gdy się nie da przeładować?

Może się tak zdarzyć, że dany zbiór argumentów wystąpił już w definicji funkcji o takiej samej nazwie. Jeśli mimo wszystko potrzebujemy powtórnie właśnie takiego zestawu argumentów, to oczywiście jeśli wystąpią one w identycznej kolejności – kompilator uzna to za błąd. Aby mimo wszystko móc taką nową funkcję zdefiniować należy rozważyć zmianę kolejności argumentów. To zwykle daje pozytywny efekt

```
void przeglad(float, float, char *);
void przeglad(float, char*, float);
```

Jeśli takie rozwiązanie nam nie odpowiada, należy rozważyć dodanie jakiegoś argumentu tak, aby lista stała się unikalna.

## Przeładowanie w wypadku argumentów domniemanych

Wyobraźmy sobie taką sytuację. Mamy takie oto wersje przeładowanej funkcji `fun`:

```
void fun(float) ;
void fun(char *) ;
void fun(int, float = 0) ;
```

A oto wywołania funkcji, które kompilator musi dopasować:

```
fun(3.14); // fun(float);
fun("napis"); // fun(char *);
fun(5); // fun(int, float = 0);
fun(5, 6.5); // fun(int, float);
```

W komentarzu podana jest funkcja, którą wybierze kompilator.

Sprawa wygląda na oczywistą, jednak trzeba uważać. W powyższym zestawie przykładowych wersji funkcji nie może się znaleźć taka deklaracja:

```
void fun(int) ;
```

gdyż to powodowało by dwuznaczność. Wywołanie



```
fun(5);
```

pasuje bowiem jednakowo do obydwu poniższych deklaracji funkcji

```
void fun(int);
void fun(int, float = 0);
```

Nie ma tu żadnej preferencji wynikającej z faktu, iż skoro w wywołaniu jest jeden argument, to zapewne chodzi o funkcję z jednym argumentem. Preferencji takiej nie ma, bo sami z niej zrezygnowaliśmy. Kiedy? Wtedy, gdy zdecydowaliśmy, że drugi argument w deklaracji

```
void fun(int, float = 0);
```

jest domniemany. Kompilator rozumie to w ten sposób, że nadałeś tym samym tej funkcji identyczne prawo jak funkcji z jednym argumentem.

```
void fun(int);
```

Sprawę łatwo zapamiętać w momencie, gdy uświadomimy sobie, że deklaracja jednej funkcji z domniemanymi argumentami (w liczbie  $n$ ) jest jakby równoznaczna  $n+1$  deklaracjom, w których te argumenty w różnej liczbie występują; a więc deklaracja

```
void fun(int, float = 0);
```

(gdzie, jak widać, liczba domniemanych argumentów jest  $n=1$ ) odpowiada takim  $n+1 = 2$  deklaracjom

```
void fun(int, float)
void fun(int);
```

Tu masz odpowiedź na pytanie dlaczego do naszego zestawu funkcji przeładowanych nie da się dołączyć funkcji

```
void fun(int);
```

Po prostu dlatego, że taka deklaracja już tam jest. Co prawda zakamuflowana w deklaracji z argumentem domniemanych, ale od tej pory ten kamuflaż już Cię chyba nie zwiedzie.

## 9.3 Czy przeładowanie nazw funkcji jest techniką obiektowo orientowaną?

Jak widać, dzięki możliwości przeładowania nazwy funkcji mamy sytuację, w której sama maszyna decyduje, którą funkcję zastosować dla danego obiektu. Uwalnia to programistę od myślenia o szczegółach leksykalnych. Mówimy: wykonaj działanie na takim-a-takim obiekcie (liście obiektów), i wtedy uruchomiona zostaje funkcja właściwa dla danego obiektu.

Można by teraz od razu krzyknąć „Wreszcie narzędzie prawdziwie obiektowo orientowane! Hosanna! “.

Problem w tym, że różni ludzie różnie rozumieją granicę, gdzie naprawdę zaczyna się programowanie obiektowo orientowane. O tym jednak, przy innej okazji.

## Zasłona spada

Czas by wyjawic jak to jest zrobione, że nazwa funkcji może być przeładowana, i że program (kompilator) orientuje się według obiektów będących argumentami funkcji. Będziesz pewnie rozczarowany, że to takie prymitywne, jednak sądzę, że musisz to rozczarowanie przeżyć. Wiedząc jak to jest naprawdę – łatwiej zrozumiesz dalszą część rozdziału.

Otóż: tak naprawdę, to te funkcje mają różne nazwy. Ty, co prawda, dałeś dwu funkcjom tę samą nazwę, jednak kompilator zmienia nazwy wszystkim funkcjom Twojego programu. Nie zapomina Twoich nazw, tylko je uzupełnia. Dopusuje do nich po prostu z jakimi argumentami jest ta funkcja.

Pokażmy zasadę tych zmian. Funkcja

```
void akcja(void);
```

otrzymuje *przykładowo* nazwę

```
akcja__Fv
```

F oznacza tu słowo funkcja, litera v oznacza void - pusta lista argumentów. Sposób oznaczania może być zależny od typu kompilatora.

Z kolei funkcja

```
void akcja(int, float);
void akcja(float, int);
```

ma nazwę     akcja\_\_Fif  
ma nazwę     akcja\_\_Ffi

czyli jeśli są argumenty, to nazwy ich typów także doczepiane są do naszej nazwy. Zamiana ta zostaje zrobiona bez naszej wiedzy. Dotyczy ona zarówno definicji i deklaracji funkcji, jak też i wywołań funkcji. Zatem wywołanie

```
akcja(3.14, 100) ;
```

zostaje zastąpione wywołaniem

```
akcja__Ffi(3.14, 100);
```

Czyli w rezultacie w programie są teraz funkcje o zupełnie innych nazwach. To tutaj czar pryska. Nie ma już więcej programu „obektowo orientowanego” – okazuje się, że kompilator zamienił go sobie na zwykły „klasyczny” program.

Rozumiemy teraz dlaczego dwie funkcje o identycznych nazwach muszą mieć inną listę argumentów. To gwarantuje kompilatorowi, że jeśli nawet tron nazwy będzie ten sam, to przynajmniej doczepiony fragment opisujący argumenty – rozróżni te nazwy.

Dodatkowo rozumiemy dlaczego przeładowane funkcje nie mogą się różnić jedynie typem zwracanym: informacja o typie zwracanym nie jest doczepiana do nazwy. Nie da się więc takich funkcji rozróżnić.

## 9.4 Linkowanie z modułami z innych języków

Ważny jest fakt, że opisanej zmianie nazw podlegają wszystkie funkcje. Nie tylko te, które są przeładowane, ale naprawdę wszystkie.<sup>†)</sup>

W zasadzie o tym fakcie można by w ogóle nie myśleć – jest to w końcu prywatna sprawa kompilatora jak on sobie radzi ze swoją pracą. Niestety, tutaj jest pewien kłopot. Otóż jeśli masz program, na który składają się dwa moduły: jeden stary, dobrze chodzący, napisany i skompilowany w klasycznym C, a drugi moduł skompilowany kompilatorem C++, to podczas linkowania tych modułów w jeden program wyniknie problem:

dostaniesz mianowicie komunikat, że linker nie odnajduje niektórych funkcji. Funkcji, o których wiesz na pewno, że tam przecież są!

Podajmy przykład takiej sytuacji. Załóżmy, że w „starym” module programu (tym z klasycznego C) jest funkcja

```
void mapa(int, float);
```

a Ty wywołujesz ją z modułu C++. Aby to było możliwe, musisz ją oczywiście wcześniej w tym module zadeklarować.

```
extern void mapa(int, float);
```

Gdy linkujesz taki program, otrzymujesz komunikat, że funkcja `mapa(int, float)` w ogóle nie istnieje. Dlaczego?

Powód jest bardzo prosty. Otóż w module C++ automatycznie powyższa deklaracja – a także właściwe wywołanie funkcji `mapa` – uległo zmianie nazwy. Deklaracja ta zmieniła się na taką:

```
extern void mapa__Fif(int, float)
```

W trakcie linkowania okazało się, że funkcji o nazwie `mapa__Fif` nie ma zdefiniowanej nigdzie. I słusznie, bo przecież w klasycznym C nie następują żadne zmiany nazwy. Tam funkcja nazywa się po prostu `mapa`.

Tak samo będzie jeśli nasz „stary” moduł jest w napisany assemblerze, Pascalu lub języku innym niż C++. Nie da się takich modułów zlinkować w jeden program.

### Impas ?

Nie! Język C++ nic by nie był wart, gdyby nie pozwalał na łączenie z modułami pochodzącymi z C, assemblera czy innych języków programowania.

Oto wyjście: w module C++ należy zadeklarować funkcję w ten sposób

```
extern "C" void mapa(int, float);
```

†) Jest to zmiana w stosunku do wcześniejszych wersji języka C++

Litera C nie mówi, że to musi być koniecznie funkcja z klasycznego C. Mówi tylko, że **nie jest to** według konwencji C++. Czyli według takiej konwencji jak to jest np. w klasycznym C. Innymi słowy postawienie tam symbolu "C" jest jakby powiedzeniem: –Bardzo proszę nie robić mi żadnych kombinacji z nazwą tej funkcji, albowiem jest to funkcja, która została skompilowana bez modyfikacji nazwy!

Jeśli masz zadeklarować więcej takich funkcji, to możesz je umieścić w środku nawiasu klamrowego

```
extern "C" {
 pierwsza(int);
 druga(float, char*, int);
 // ...
}
```

W środku takiego nawiasu (czyli bloku) może się znaleźć nawet dyrektywa `include`.

```
extern "C" {
 #include "moje_deklar.h"
}
```

Wtedy wszystkie umieszczone we włączanym pliku deklaracje funkcji też traktowane są jak deklaracje funkcji w klasycznym C.

---

## 9.5 Przeładowanie a zakres ważności deklaracji funkcji

Definiując pojęcie „przeładowanie” powiedzieliśmy, że przeładowanie nazwy funkcji następuje wtedy, gdy w danym zakresie ważności jest więcej niż jedna funkcja o takiej samej nazwie. Nie rozwijaliśmy tego zastrzeżenia o identyczności zakresów ważności.

Pamiętamy, że:

*najczęściej funkcje mają zakres ważności pliku, w którym je zdefiniowano. Czyli są znane w pliku od linii ich deklaracji. Jeśli program składa się z kilku plików, to w tych innych plikach funkcja jest nieznana, dopóki nie zostanie tam zadeklarowana. Deklaracja może objąć zakres całego pliku, albo też mieć zakres mniejszy – lokalny.*

Oto przykład. Załóżmy, że mamy następujący plik:

```
#include <iostream.h>
/*****
void dzwiek(int a)
{
 cout << a << " nuty \n" ;
}
/*****
void dzwiek(float h)
{
 cout << "Dzwiek o czestotliwosci : "
 << h << " hercow \n" ;
}
/*****/
```

Zawiera on, jak widać, definicje dwóch funkcji o nazwie `dźwięk`. A oto inny plik, w którym korzystamy z tych funkcji:

```
#include <iostream.h>
extern void dźwięk(int); // deklaracja o zasięgu pliku ❶
/*****
main()
{
 dźwięk(1); // ❷
 { // ❸
 extern void dźwięk(float); // deklaracja lokalna ❹
 dźwięk(2); // ❺
 dźwięk(3.14); // ❻
 } // ❼
 dźwięk(5); // ❽
 dźwięk(6.28); // ❾
}
```

**Po zlinkowaniu tych plików i wykonaniu programu na ekranie zobaczymy tekst**

```
1 nuty
Dźwięk o częstotliwości : 2 herców
Dźwięk o częstotliwości : 3.14 herców
5 nuty
6 nuty
```

Blizsze przyjrzenie się programowi upewnia nas, że żadne przeładowanie nie nastąpiło. Spodziewaliśmy się, że będzie wykonywana ta wersja funkcji `dźwięk`, która jest właściwa argumentom wywołania. Tymczasem tak się nie stało. Dlaczego? Powód jest jeden. Deklaracje tych funkcji nie mają tego samego zakresu ważności. Zamiast przeładowania nastąpiło zastrzeżenie.

**Przyjrzyjmy się ciekawszym punktom programu**

- ❶ Jest deklaracją o zasięgu pliku.
- ❷ Jest wywołaniem funkcji `dźwięk` - jedynej znanej w tym momencie, czyli tej zadeklarowanej powyżej.
- ❸ Otwierany jest jakiś blok lokalny. Może być to tak sztuczne jak u nas, a może być to po prostu wnętrze jakiejś funkcji.
- ❹ W tym lokalnym bloku deklarujemy, że istnieje gdzieś funkcja `dźwięk(float)`. Nazwa `dźwięk` zasłania wszystkie inne możliwe nazwy `dźwięk` z innych zakresów. Stają się niedostępne.
- ❺ Wywołanie funkcji `dźwięk` z argumentem typu `int`. Jediną dostępną funkcją o nazwie `dźwięk` jest teraz funkcja `dźwięk(float)`. Tamta funkcja jest co prawda kompilatorowi znana, ale jej nazwa jest teraz zasłonięta. Zatem kompilator zamienia argument typu `int` na typ `float` i uruchamia tę jedyną możliwą teraz funkcję. Zamiana następuje przy użyciu tzw. konwersji standardowej.
- ❽ Wywołanie funkcji z argumentem typu `float`. Tu nie ma problemu. Takiego argumentu kompilator właśnie oczekiwał.

- 7 Kończy się zakres lokalny. Deklaracja funkcji `dzwiek(float)` zostaje zapomniana i odsłania deklarację `dzwiek(int)`.
- 8 Wywołanie teraz z argumentem typu `int` uruchamia po prostu tę jedyną możliwą teraz funkcję `dzwiek`.
- 9 Natomiast wywołanie z argumentem typu `float`, to znowu dla kompilatora pewien kłopot. Funkcji o takim argumencie on już nie zna (deklarację lokalną przed chwilą zapomniał). Zamienia więc argument typu `float` na typ `int` i uruchamia jedyną możliwą funkcję o nazwie `dzwiek`.  
Sformułujmy wniosek:

Aby mieć rzeczywiście dwie funkcje o tej samej nazwie dostępne równocześnie (przeładowane) obie muszą mieć identyczny zakres ważności.

W naszym przykładzie tak by było, gdyby obie funkcje były zadeklarowane tak, by miały w zakres ważności pliku, czyli gdyby deklaracja 4 była tam gdzie 1. Trzeba jednak zaznaczyć, że klauzula o identyczności zakresu ważności przy przeładowaniu nie jest bynajmniej balastem, z którym trzeba nauczyć się żyć! Wręcz przeciwnie – otwiera nam drogę do lokalnego przeładowywania funkcji. W ramach jednego lokalnego obszaru funkcje mogą się przeładowywać, a nie ma to żadnego skutku wobec świata zewnętrznego, gdzie mogą być inne lokalne obszary, w których dokładnie ta sama nazwa funkcji może być również przeładowana. Jeden lokalny obszar nie wchodzi w kolizję z drugim.

Wiem, brzmi to może trochę zawile. Wszystko jednak stanie się jasne, gdy wkroczymy w krainę lokalnych obszarów jakimi są definicje klas, czyli typów, które możemy definiować sami.

## 9.6 Rozważania o identyczności lub odmienności typów argumentów

Wróćmy na poziom prostych spraw. Powiedzieliśmy, że przeładowanie funkcji jest możliwe wtedy, gdy funkcje różnią się listą argumentów. Funkcje mogą mieć tę samą nazwę – pod warunkiem, że argumenty będą inne.

### 9.6.1 Przeładowanie a typedef i enum

Przypominamy sobie zapewne deklarację `typedef` (str. 50). Wprowadza ona synonim dla jakiegoś istniejącego już typu. Nie tworzy ona nowego typu. Zatem poniższa próba przeładowania zostanie w czasie kompilacji uznana za błąd

```
typedef int calkow ;

void funkcja1(int) ;
void funkcja1(calkow); // !
```

Ponieważ `calkow` jest tylko innym określeniem tego samego typu `int` dlatego mamy tu w gruncie rzeczy do czynienia z funkcjami

```
void funkcja1(int) ;
void funkcja1(int); // !
```

Czyli są to dwie funkcje o tej samej nazwie i identycznej liście argumentów, a to jest błędem.

Zupełnie inna sprawa jest z typami wyliczeniowymi definiowanymi instrukcją `enum` (str. 52). Typ wyliczeniowy jest naprawdę odrębnym typem. Co prawda typ ten także bazuje na liczbach całkowitych, ale to nie ma znaczenia. (Przypominam, że typ `int` oraz `unsigned int`, to także odmienne typy).

Zatem instrukcją `enum` definiujemy nowy typ (inny od typu `int`) i nadajemy mu nazwę. Ta nazwa może być używana przy przeładowaniach funkcji.

```
enum operacja { pisz = 1, czytaj, skocz, przewin } ;

void tasma(int);
void tasma(operacja);
```

Jest to poprawne przeładowanie. Listy argumentów obu funkcji różnią się.

## 9.6.2 Tablica a wskaźnik

Zwróć uwagę na następujące dwie deklaracje funkcji

```
void fff(int tab[]) ;
void fff(int * wsk) ;
```

Obie funkcje pod kątem przeładowania uznawane są za identyczne i dlatego w danym zakresie ważności nie mogą mieć tej samej nazwy. Kompilator uzna to za błąd.

Łatwo to intuicyjnie wyczuć. Wyobraź sobie taki fragment programu:

```
int ta[10]; // definicja tablicy
fff(ta); // wywołanie funkcji
```

Jako argument aktualny w wywołaniu funkcji znajduje się nazwa tablicy, czyli adres jej początku. Gdybyś był kompilatorem, to którą z wyżej zadeklarowanych funkcji powinienś przy takim wywołaniu uruchomić?

Z rozdziału o wskaźnikach wiemy, że tablice i wskaźniki mogą być w zasadzie traktowane wymiennie. Zatem obie deklaracje funkcji jednakowo pasują do wywołania. A to jest błąd. Nie może być żadnej dwuznaczności.

Mówiąc bardziej formalnie: zarówno

```
int tab[]
```

jak i

```
int *wsk
```

mogą mieć te same inicjalizatory (tutaj – argumenty aktualne w wywołaniu funkcji). Zatem na podstawie wyglądu inicjalizatora nie można zdecydować do której wersji on się nadaje. Nadaje się bowiem do obu. Dwuznaczności być nie może i to właśnie powie Ci kompilator w informacji o błędzie.

Podsumujmy:



Typy argumentów różniące się tylko co do oznaczenia:

wskaźnik  $*$ ,

albo: tablica  $[]$

- są uznawane przy przeładowaniu za identyczne.



Następne paragrafy mogą Cię trochę nudzić i może wydadzą się sformalizowane. Nie zniechęcaj się jednak. Jeśli czytasz tę książkę po raz pierwszy, to w pewnym momencie zdecydowanie odradzę Ci czytanie dalszej części tego rozdziału. Tymczasem postaraj się jednak mimo wszystko czytać najbliższych siedem paragrafów (są wyjątkowo krótkie), nawet jeśli nie wszystko wyda Ci się interesujące.

Będziemy tu nadal mówili o tym, kiedy argumenty deklarowanych funkcji pozwalają na przeładowanie, a kiedy nie.

Najmłodszym czytelnikom proponuję takie skojarzenia:

Zbiór wszystkich funkcji o jednej (przeładowanej) nazwie, to jakby menażeria zwierząt – po jednym okazie różnych gatunków. Argument wywołania funkcji to jakby pokarm dla zwierząt. Z kolei funkcje to same zwierzęta – niektóre są roślinożerne, niektóre mięsożerne.

Otóż jeśli chcemy by zwierzęta się nie pogryzły powinniśmy dawać zawsze taki pokarm, który może zjeść tylko jedno z nich. To jest oczywiste.

Niestety są zwierzęta, które mogą zjeść to samo co inne. W następnych paragrafach porozmawiamy właśnie o tym, jak rozpoznawać takie zwierzęta i unikać konfliktów.

Cała trudność w tych paragrafach polega na tym, że wielokrotnie pojawia się w nich słowo „inicjalizator”. Umówmy się, że ile razy ja napiszę „inicjalizator”, czy „argument wywołania funkcji” to Ty sobie myślisz: „pokarm podany zwierzętom”.

### 9.6.3 Pewne szczegóły o tablicach wielowymiarowych

Mówiąc o tablicach wspomnieliśmy o sposobie przesyłania ich do funkcji. Przypominam – tablicy nie przesyła się przez wartość (bo kto by przesyłał do funkcji np. 8192 elementów tej tablicy), ale przesyła się ją przez adres. Nazwa tablicy, jak wiadomo według egipskich papyrusów, jest adresem jej początku. Umieszczając w wywołaniu nazwę tablicy wysyłamy więc do funkcji jej adres.

- ❖ W obrębie funkcji tak otrzymany adres może posłużyć do inicjalizacji wskaźnika, którym będziemy się swobodnie poruszali po elementach tej tablicy.
- ❖ Innym sposobem odebrania przysłanego do funkcji adresu tablicy jest odebranie go jako tablicy.

To wszystko oczywiście przypomnienie, bo mówiliśmy o tym w poprzednim rozdziale, tutaj więc tylko 2 przykłady takich funkcji



```
int tablica[4][2] ;
//...
void funwsk(int *wsk);
void funtab(int t[4][2]);
```

Funkcje te mają inne nazwy, bo jak wiemy z poprzedniego paragrafu - typy  $T^*$  i  $T[]$  są dla przeładowania nierozróżnialne. To dlatego, że oba mają identyczny inicjalizator (czyli tutaj: identyczny możliwy argument wywołania)

```
funwsk(tablica);
funtab(tablica);
```

Z kolei następujące dwie funkcje

```
void chap(int m[2][7]) ;
void chap(int m[3][2]) ;
```

mogą mieć tę samą nazwę funkcji - czyli mogą być przeładowane. To dlatego, że w wypadku wywołania obu funkcji jako argument aktualny musi wystąpić nazwa innej - odpowiedniej dla danego wywołania tablicy. Przykładowo:

```
int kil[2][7] ;
int zagieli[3][2] ;

chap(kil); // wywołanie chap(int m[2][7])
chap(zagieli); // wywołanie chap(int m[3][2])
```

Ponieważ obie funkcje mają jako argument inny typ obiektu, dlatego mogą mieć inną nazwę.

I znowu zapytam: -Gdybyś był kompilatorem, to czy miałbyś tu wątpliwości, którą funkcję przy danym wywołaniu uruchomić? Da się tu bez dwuznaczności określić, która funkcja pasuje do danego argumentu.

Idźmy dalej. Pamiętaj z paragrafu o przesyłaniu do funkcji tablic wielowymiarowych (str. 147), że dla funkcji istotne są rozmiary tablicy, ale rozmiar pierwszy od lewej nie. Dopiero drugi i następne.

```
int tablica[4][5][1]; // pierwszy od lewej, to ten gdzie jest 4
```

To dlatego, że ten pierwszy od lewej rozmiar nie bierze udziału w obliczaniu pozycji elementu danej tablicy w pamięci komputera. (Zwykła arytmetyka). Dlatego więc jeśli w dwóch funkcjach (o tym samym zakresie) argument formalny będący tablicą różniłby się tylko tym wymiarem „pierwszym od lewej”, to takie funkcje nie mogą mieć tej samej nazwy. Na przykład:

```
void abra(int x[5][10][2]) ;
void abra(int y[9][10][2]) ; // błąd, bo identyczna jak powyższa
```

Także funkcje

```
void kadabra(int s[3]) ;
void kadabra(int m[8]) ;
```

Nie mogą mieć tej samej nazwy. Tablice będące ich argumentami formalnymi różnią się jedynie wymiarem pierwszym z lewej. To, że jest to jedyny wymiar — nie ma znaczenia. Przy obliczaniu pozycji elementu w pamięci nie bierze on udziału.

Podsumujmy:

Nie możemy mieć przeładowania funkcji w sytuacji, gdy dwie funkcje różnią się jedynie argumentem tablicowym tak, że tylko ten rozmiar najbardziej z lewej jest inny. Dla kompilatora jest to nierozróżnialne - dlatego przy definicji takich funkcji zaprotestuje.

## 9.6.4 Przeładowanie a referencja

Opiszemy tu kolejną sytuację, gdy przy przeładowaniu argumenty traktowane są jako identyczne. Oto ilustracja:

```
void ggg(int &k);
void ggg(int m); // za mało się różnią
// ...
int m;
 ggg(m);
```

Jak widać, obie funkcje różnią się tylko tym, że jedna przyjmuje argument przez wartość, a druga przez referencję. Jest to błąd. Takie argumenty są pod kątem przeładowania identyczne.

Tu znowu bardzo łatwo wyczuć dlaczego. Zapytam znowu: –Gdybyś to Ty był kompilatorem i zobaczył takie wywołanie funkcji:

```
 ggg(m);
```

to którą funkcję należałoby uruchomić? No właśnie – nie wiadomo którą. Wywołanie to pasuje jednakowo do jednej, jak i do drugiej funkcji. Następuje niedopuszczalna dwuznaczność.

Inaczej mówiąc – oba typy argumentu formalnego mają taki sam typ inicjalizatora. Sam zobacz inicjalizację referencji i obiektu:

```
int m; // obiekt;
int &ref = m; // inicjalizacja referencji
int a = m; // inicjalizacja innego obiektu
```

Z prawej strony znaku '=' stoi w obu wypadkach to samo. Zatem po inicjalizatorze (argumentie wywołania) rozstrzygnąć problemu nie można.

Podsumowanie:

Ponieważ dla dowolnego typu  $T$  – następujące deklaracje argumentów formalnych

$T$   
 $T \ \&$

**mają te same inicjalizatory**, dlatego pod względem przeładowania nie może być funkcji o tej samej nazwie, a różniące się jedynie tym, że jeden argument jest w jednej odbierany przez wartość, a w drugiej przez referencję.

### Dygresja dla najmłodszych:

Okazuje się, że funkcje mające argumenty formalne  $T$  i  $T \ \&$  to dwa gatunki zwierząt mogących zjeść to samo. Wypróbowaliśmy to właśnie – oba zjadają

„inicjalizator” (pokarm) będący obiektem typu T. (To nic, że potem trawia go inaczej). Aby uniknąć awantury, nigdy nie należy trzymać w menażerii dwóch takich zwierząt.

## 9.6.5 Identyczność typów: T, const T, volatile T

Inną sytuacją, gdy nastąpiłaby dwuznaczność jest

```
void f(int);
void f(const int m); // dla przeladowania
void f(volatile int m) ; // nierozróżnialne - błąd !
```

Każda z tych funkcji akceptuje taki sam argument aktualny (inicjalizator). W konsekwencji przy poniższym fragmencie programu:

```
int x = 6 ;
f(x) ;
```

Nie dało by się określić, o którą wersję funkcji chodzi. Wszystkie - jednakowo dobrze pasują do tego wywołania.

Wszystkie otrzymują tak samo argument przez wartość. Różnica między nimi polega tylko na tym, że:

- pierwsza funkcja jest zwykłą funkcją,
- druga - obiecuje, że swojej lokalnej kopii przesłanego argumentu nie będzie zmieniała,
- trzecia - obiecuje nie robić żadnych optymalizacji z lokalną kopią.

Te obietniczki są jednak prywatną sprawą funkcji. Każda z nich może zainicjalizować swój argument formalny tym samym inicjalizatorem.

Nasze wywołanie  $f(x)$  jest jednakowo dobre do wywołania każdej z nich. Następuje wieloznaczność, więc już w trakcie kompilacji wykryty zostanie błąd.

Podsumujmy:

Ponieważ dla dowolnego typu T –

|         |            |
|---------|------------|
| zarówno | T          |
| jak i   | const T    |
| jak i   | volatile T |

inicjalizuje się **identycznym inicjalizatorem**, dlatego tak określone argumenty formalne uznawane są za identyczne.

### Dygresja dla najmłodszych:

Okazuje się, że trzy funkcje mające argumenty formalne typu T, const T i volatile T - to trzy gatunki zwierząt mogących zjeść to samo. Słowa const i volatile określają tu nie pokarm, ale to, co z niego jest już lokalnie w przewodzie pokarmowym bestii. To jednak nie ma żadnego znaczenia – ważne, że przed spożyciem był to pokarm typu T, o który wszystkie trzy zwierzęta pogryzły się nawzajem. Aby uniknąć awantury nie należy nigdy trzymać w menażerii choćby dwóch takich zwierząt.

## 9.6.6 Przeladowanie a typy: T\*, volatile T\*, const T\*

Zagadka. Czy możliwe jest takie przeladowanie funkcji?

```
void radio(float *k);
void radio(const float *k);
void radio(volatile float *k);
```

Jeśli na podstawie poprzedniego paragrafu mówisz że nie, to przegrałeś. Jeśli nie przychodzi Ci do głowy żadna odpowiedź, to spróbuj pomyśleć, jak każdą z tych funkcji się wywołuje. Jeśli wywołanie pasować może do więcej niż jednej funkcji – to takie przeladowanie jest niemożliwe.

Co widzimy w pierwszej funkcji?

- ❖ Jest to deklaracja funkcji `radio`, którą to funkcję wywołuje się podając jako argument – adres obiektu typu `float`. Zatem na przykład tak:

```
float obj = 15.6 ;
radio(&obj);
```

Co widzimy w drugiej funkcji?

- ❖ Jest to deklaracja funkcji `radio` wywoływanej z argumentem będącym adresem obiektu, który jest stały (`const`)

```
const float pi = 3.14 ;
radio(&pi);
```

Zaś deklaracja trzeciej funkcji mówi:

- ❖ Jest to funkcja `radio` wywoływana z argumentem będącym adresem obiektu typu `float` – i to nie zwykłego obiektu, ale takiego, który ma cechę `volatile`.

```
volatile float wulkan ;
radio(&wulkan);
```

We wszystkich trzech wypadkach chodzi o różne typy obiektów. Kompilator napotykać na wywołanie funkcji z udziałem jednego z tych obiektów (jako argumentu wywołania funkcji) – wystarczy, że spojrzy na ten argument i już wie, do której z wersji funkcji `radio` on pasuje - a pasuje tu zawsze tylko do jednej.

Możesz jednak zapytać: – Jak to, przecież w poprzednim paragrafie mieliśmy podobną sytuację, a tam nie wolno było przeladowywać. Jak jest różnica?

Taka, że w poprzednim rozdziale słowa `const` i `volatile` określały lokalny obiekt tworzony w obrębie funkcji. Czyli kopię. Ten obiekt (kopia) może sobie być jaki chce – i tak do jego inicjalizacji nadawał się dowolny typ obiektu `T`. To powodowało wieloznaczność.

Tutaj jest odwrotnie:

słowa `const` oraz `volatile` określają tu typ obiektu, który służy do inicjalizacji – czyli określają argument wywołania funkcji. (A nie jak poprzednio: lokalny obiekt inicjalizowany wewnątrz funkcji). Skoro aż tak precyzyjnie określiliśmy typ argumentu wywołania,

to znaczy, że tylko on może wywołać daną wersję funkcji. Przy tak precyzyjnym określeniu nie ma mowy o wieloznaczności.

Podsumujmy:

Ponieważ dla dowolnego typu  $T$  następujące deklaracje argumentów formalnych:

```
T*
const T*
oraz volatile T*
```

wymagają **każda odmiennego inicjalizatora (odmiennego argumentu wywołania)** – dlatego funkcje różniące się tylko obecnością jednej z wersji wspomnianych deklaracji – mogą być przeładowane.

Zatem funkcje zadeklarowane na początku tego paragrafu są poprawnym przeładowaniem.

### Dygresja dla najmłodszych:

Tu sprawa była inna. Słowa `const` i `volatile` stoją tak, że określają nie to, co lokalnie jest w przewodzie pokarmowym zwierzaka, ale są określeniem pokarmu – jaki ma być *przed* zjedzeniem. Okazuje się, że takie trzy zwierzaki mają ściśle sprecyzowane jaki pokarm zjadają. Jeden zwierzak je tylko befsztyki krwiste, drugi średnio przysmażone, a trzeci bardzo przysmażone. Jeśli znasz osobiście jakieś bestie żywiące się befsztykami, to wiesz, że jeśli taki jada średnio przysmażone, to brzydzi się ociekającymi krwią lub spalonymi na węgiel. Zatem nie ma konfliktu – po wyglądzie befsztyka kompilator łatwo zdecyduje komu go rzucić na pożarcie.

## 9.6.7 Przeładowanie a typy: $T\&$ , `volatile T`, `const T`

Dokładnie ta sama argumentacja, którą omówiliśmy w poprzednim paragrafie dotyczy różnych wersji referencji. Oto ilustracja. Funkcje:

```
void lad(int &m);
void lad(const int &m);
void lad(volatile int &m);
```

są wywoływane – każda z innym rodzajem obiektu

```
const int stala = 12 ;
volatile int elektron = 4 ;
int liczba = 1 ;
```

```
lad(stala) ; // wywoła lad(const int &)
lad(elektron) ; // wywoła lad(volatile int &)
lad(liczba) ; // wywoła lad(int &)
```

albowiem słowa `const` i `volatile` określają typ inicjalizatora (czyli typ argumentu wywołania). Tak precyzyjne określenie inicjalizatora nie prowadzi do wieloznaczności zabójczej dla przeładowania.

Podsumujmy:

Ponieważ dla dowolnego typu T następujące deklaracje argumentów formalnych

```
T &
const T &
volatile T &
```

wymagają każda odmiennego inicjalizatora (odmiennego argumentu wywołania) – dlatego funkcje różniące się tylko obecnością jednej z wersji wspomnianej deklaracji – mogą być przeładowane.

### Dygresja dla najmłodszych:

Tłumaczenie jest takie jak poprzednio z tym, że teraz na sam befszytk zwierzak mówi przezwiskiem. Nadal jednak 'to' ma być albo krwiste albo średnio przy-smażone albo... Są to upodobania tak wykluczające się, że nigdy nie dojdzie do awantury. Jedzące to trzy zwierzaki mogą byćzymane w tej samej menażerii.

## 9.7 Adres funkcji przeładowanej

Jeśli w zwykłym przypadku chcemy się posłużyć wskaźnikiem do funkcji, to musimy go oczywiście w pewnym momencie ustawić tak, by na żadaną funkcję pokazywał. Oto ilustracja:

```
int sposob(float) ;
int sposob(char) ;
//...
int (*wskfun)(char) ;

wsk = sposob ;

// <----- funkcja
// deklaracja wskaźnika mogącego
// pokazywać na powyżej
// oznaczoną funkcję
```

*Jak wiemy nazwa funkcji jest też adresem początku tej funkcji, stąd też przy ostatniej instrukcji nie potrzeba operatora & (adres).*

Co zrobić jeśli funkcja jest (wielokrotnie!) przeładowana? Ta sama nazwa oznacza wówczas różne wersje funkcji, a więc różne adresy.

### Skąd kompilator będzie wiedział, o adres której wersji nam chodzi?

Kompilator patrzy wówczas na lewą stronę przypisania: w naszym wypadku stoi tam wskaźnik do funkcji i to nie byle jakiej funkcji, ale takiej, która jest wywoływana z argumentem typu char. Wszystko jasne! Kompilator, wśród przeładowanych wersji funkcji sposob, odszuka tę wersję, która ma argument typu char i adres tejże funkcji podstawia do wskaźnika.

Jeśli kiedyś zdefiniujemy wskaźnik

```
int (*wsk2)(float) ;
```

to instrukcja przypisania

```
wsk2 = sposob ;
```

(metodą identycznej dedukcji) sprawi, że do wskaźnika wsk2 wstawiony zostanie adres wersji `int sposob(float)`. Sprytne, prawda ?

Zasada jest ciągle ta sama:

w wypadku brania adresu funkcji przeładowanej – spośród wszystkich funkcji o tej samej nazwie (i ma się rozumieć – tym samym zakresie ważności) – do operacji wybierana jest ta wersja funkcji która **dokładnie** pasuje do celu przypisania. Celem w wypadku przypisania jest wyrażenie stojące po lewej stronie znaku równości. To ono zdecydowało, który adres funkcji będzie użyty.

## Wysyłanie do funkcji adresu innej, przeładowanej funkcji

Mogą jednak być inne sytuacje. Na przykład jeśli chcemy do jakiejś funkcji wysłać adres funkcji, która jest przeładowana. Oto przykład programu:

```
#include <iostream.h>
// ---deklaracje funkcji nazwy funkcji o przeładowanej nazwie- przelad
void przelad (int k) ;
void przelad (float x) ;

// ---deklaracje zwykłych funkcji
void pierwsza(void (*adrfun)(int));
void druga(void (*adrfun)(float));
/*****/
main()
{
 pierwsza(przelad); // ❶
 cout << "-----\n" ;
 druga(przelad); // ❷
}
/*****/
void pierwsza(void (*adrfun)(int)) // ❸
{
 cout << "Jestem wewnątrz funkcji PIERWSZA\n"
 "teraz wywołam funkcję której adres przysłano"
 " jako argument\n";
 adrfun(5);
 cout << "PO wywołaniu funkcji\n" ;
}
/*****/
void druga(void (*adrfun)(float))
{
 cout << "Jestem wewnątrz funkcji DRUGA\n"
 "teraz wywołam funkcję której adres przysłano"
 " jako argument\n";
 adrfun(3.14);
 cout << "PO wywołaniu funkcji\n" ;
}
/*****/
void przelad (int k)
{
 cout <<
 "**** Funkcja przelad - wersja: przelad(int) \n"
 " argument k = " << k << endl ;
}
/*****/
```

```
void przelad (float x)
{
 cout <<
 "*** Funkcja przelad - wersja: przelad(float) \n"
 " argument x = " << x << endl ;
}
```

## Po wykonaniu tego programu na ekranie pojawi się

```
Jestem wewnątrz funkcji PIERWSZA
teraz wywołam funkcję której adres przysłano jako argument
*** Funkcja przelad - wersja: przelad(int)
argument k = 5
PO wywołaniu funkcji

Jestem wewnątrz funkcji DRUGA
teraz wywołam funkcję której adres przysłano jako argument
*** Funkcja przelad - wersja: przelad(float)
argument x = 3.14
PO wywołaniu funkcji
```

## Komentarz :

W main widzimy wywołania dwóch różnych funkcji – pierwsza i druga. Mają one jednak ten sam argument będący wskaźnikiem do funkcji. Skąd kompilator wie, o którą wersję przeładowanej funkcji przelad w konkretnym wypadku chodzi? (Czyli adres której funkcji przelad ma wysłać jako argument?)

- 1 Przy wywołaniu funkcji pierwsza kompilator sprawdza jakiego wskaźnika do funkcji się pierwsza spodziewa. Z deklaracji, a także definicji ❸ orientuje się, że skoro celem przypisania jest wskaźnik do funkcji z argumentem typu int – to znaczy, że ma wysłać adres funkcji:

```
void przelad (int k) ;
```

bo to przecież tak, jakby robić przypisanie

```
void (*adrfun) (int) = przelad ;
```

- 2 Przy wywołaniu funkcji druga sprawdza jaki jest cel przypisania wysłanego adresu funkcji. Celem jest wskaźnik do funkcji z argumentem typu float. Acha, to znaczy, że należy wysłać adres tej wersji:

```
void przelad (float x) ;
```

## 9.7.1 Zwrot rezultatu będącego adresem funkcji przeładowanej

Celem przypisania może być nie tylko argument formalny funkcji. Może być też typ wartości zwracanej przez funkcję. To znaczy w instrukcji return stawiamy przeładowaną nazwę funkcji, a kompilator decyduje, którą z wersji wybrać.



Decyduje na podstawie tego, jaki typ zadeklarowany jest jako typ zwracany przez bieżącą funkcję.

Jeśli funkcja ma na przykład zwrócić wskaźnik do takiej funkcji, która jest wywoływana z argumentem typu `int` – to wybrana zostanie wersja

```
void przelad(int);
```

Chciałem od razu napisać przykładową funkcję, która to właśnie robi, ale boję się, że się przerazisz. Zróbmy to więc etapami.

Funkcja będzie się nazywać `zwrot`. Po pierwsze ustalmy jaka to ma być funkcja. To ustalenie pomoże nam w późniejszym pisaniu deklaracji funkcji.

A zatem `zwrot` ma być funkcją wywoływaną bez żadnych argumentów, a zwracającą wskaźnik do takiej funkcji, która:

- - wywoływana jest z argumentem typu `int`
- - zwraca typ `void` (czyli nic)

Czyli ma to być wskaźnik mogący pokazać na choćby taką funkcję:

```
void f(int)
```

Korzystając z tych ustaleń zacznijmy budowanie deklaracji. Zatem:

`zwrot` jest funkcją wywoływaną bez żadnych argumentów...

```
zwrot(void)
```

...a która zwraca wskaźnik...

```
*zwrot(void)
```

...do funkcji...

```
(*zwrot(void)) (...)
```

... wywoływanej z argumentem typu `int`...

```
(*zwrot(void)) (int)
```

...i zwracającej typ `void`.

```
void (*zwrot(void)) (int) ;
```

Uff! Zrobione. Na pociechę powiem, że takie funkcje będziesz pisał bardzo rzadko. A na pewno nie na początku.

Skoro już mamy deklarację, to łatwo zapiszemy definicję funkcji – czyli zdefiniujemy ciało tej funkcji

```
void (*zwrot(void)) (int)
{
 cout << "zwracam wskaźnik do funkcji ! \n" ;
 return przelad ;
}
```

Istota tego przykładu polega na tym, że obok słowa `return` stoi nazwa funkcji – nazwa, która jak nam wiadomo jest przeladowana. Jest więc kilka funkcji o nazwie `przelad`. O tym, którą z wersji wybierze kompilator, decyduje dek-

laracja funkcji. W deklaracji bowiem jest jasno powiedziane, że życzymy sobie, by został zwrócony adres do funkcji, która jest wywoływana z jedynym argumentem typu `int`. To, co sobie zażyczyliśmy otrzymać, jest jakby celem przypisania. Do tego celu kompilator dobierze pasującą wersję funkcji przeład.

Wniosek z tego paragrafu jest taki:

Przy operacjach zwracania przez funkcję `F` adresu funkcji przeładowanej `P` – zwracany zostaje adres tej wersji funkcji przeładowanej, która pasuje do celu przypisania. Celem jest w tym przypadku zadeklarowany typ zwracany przez funkcję `F`

Opisane przypadki nie wyczerpują wszystkich możliwych celów. Dalszymi takimi sytuacjami zajmiemy się później. (Dla wtajemniczonych: Innym możliwym celem może być także inicjalizowany obiekt, a także argument formalny operatora).

---

## 9.8 Kulisy dopasowywania argumentów do funkcji przeładowanych

Powtórzę znowu: wiemy już, że nazwa funkcji może być przeładowana, czyli że może być kilka funkcji o tej nazwie, byle tylko różne wersje tej funkcji różniły się listą argumentów. Kiedy wywołujemy taką funkcję, kompilator patrzy na argumenty wywołania funkcji i - zależnie od ich typu - wybiera tę jedyną funkcję, do której dokładnie pasują.

Tak jest w pierwszym przybliżeniu. Zastanówmy się jednak co się zdarzy, gdy argumenty nie będą takie, że dokładnie pasują do jednej z wersji. Dajmy na to, że pasują prawie zupełnie, gdyby nie jakiś drobny szczegół.

```
void f(float); // mamy taki zestaw
void f(char);
// a wywołujemy
f(4); // mimo, że przecież void f(int)
 // nie istnieje
```

Co w takiej sytuacji ma zrobić kompilator? Oto dwa warianty tego co kompilator sobie pomyśli:

❖ wariant 1):

*-Czy on oszalał? Wywołuje mi funkcję w sposób, który nie pasuje do żadnej z funkcji o tej nazwie. Trudno! Sygnalizuje błąd kompilacji, a on niech się wreszcie nauczy programować!*

❖ wariant 2):

*-No tak, co prawda wywołanie nie pasuje do żadnej z zadeklarowanych funkcji o tej nazwie, ale z tego co widzę, to najbliższe to jest takiej-a-takiej wersji. Wszystkie inne wersje różnią się o wiele bardziej. Programista zrobił to dlatego, że nie umie jeszcze dobrze programować w C++. Może jednak dlatego, że sądzi iż ja, kompilator, jestem na tyle inteligentny, że jed-*

*noznacznie domyśle się, o którą wersję może w tym lekko niepasujący wypadku chodzić.*

Ponieważ kompilator C++ rzeczywiście ma ambicję, dlatego rozumuje według wariantu 2. W naszym przypadku nie znajdując funkcji

```
void f(int);
```

zamieni 4 na 4.0 i uruchomi funkcję

```
void f(float);
```



Dalsza część tego rozdziału poświęcona jest szczegółom rozumowania, na podstawie którego kompilator ustala, która z wersji przeładowanej funkcji pasuje najbardziej do argumentów wywołania.

Jeśli nie potrafi tego ustalić jednoznacznie, bo na przykład są dwa warianty, które mogą być jednakowo prawdopodobne, to dopiero wtedy sygnalizuje błąd. Jeśli zauważy, iż jeden z wariantów pasuje wyraźnie lepiej niż inne, wówczas ten właśnie zostanie wybrany.

Przy pierwszym czytaniu tej książki radzę Ci w tym miejscu przerwać czytanie tego rozdziału i przeskoczyć do następnego. Wydaje mi się bowiem, że lepiej jeśli najpierw będziesz miał ogólny pogląd na język C++, a dopiero potem należy zacząć studiować szczegóły.

Nie pytaj mnie też dlaczego wobec tego nie umieściłem tego rozdziału na samym końcu książki. To dlatego, że chciałem, aby szczegóły dotyczące przeładowania nazw funkcji były w jednym miejscu. Po to, że gdybyś chciał do tego wrócić, to znajdziesz wszystko w jednym miejscu.

Tymczasem skok do rozdziału następnego, mówiącego o klasach.



## 9.9 Etapy dopasowania

Kiedy kompilator napotyka wywołanie przeładowanej funkcji – pracuje nad nim w kilku etapach. Jeśli w rezultacie znajdzie dokładnie jedną z wersji funkcji, która pasuje do wywołania lepiej niż inne, wówczas można powiedzieć, że dopasowanie się udało.

Jeśli znajdzie dwie lub więcej funkcji, które jednakowo zbliżone są do tego, co umieściliśmy w wywołaniu funkcji, to kompilator uzna to za błąd – nie może być żadnej dwuznaczności.

A oto etapy poszukiwania właściwej funkcji. Kompilator, mając przed sobą wywołanie funkcji o przeładowanej nazwie, patrzy na możliwe realizacje tej funkcji i rozważa kolejno:

- 1) dopasowanie dosłowne,
- 2) dopasowanie dosłowne z trywialną konwersją,
- 3) dopasowanie na zasadzie awansowania (z awansem, z promocją),

- 4) dopasowanie z użyciem konwersji standardowych,
- 5) dopasowanie z użyciem konwersji wymyślonych przez programistę,
- 6) dopasowanie do funkcji z wielokropkiem.

Wyjaśnijmy teraz poszczególne etapy.

---

### 9.9.1      Etap 1. Dopasowanie dosłownie

Inaczej mówiąc kompilator sprawdza czy argumenty wywołania pasują dokładnie do argumentów formalnych jednej z wersji przeładowanej funkcji.

Na przykład: w wywołaniu funkcji mamy argument będący tablicą typu `int`

```
int tablica[10] ;
fun(tablica);
```

Pasuje to dokładnie do takiej wersji funkcji `fun`

```
void fun(int ttt[]);
```

Jeśli rzeczywiście taką funkcję mamy, to dopasowanie jest dosłowne.

---

### 9.9.2      Etap 2. Dopasowanie dosłowne, ale z tzw. trywialną konwersją

Przykładowo: do wywołania

```
int tablica[10] ;
fun(tablica);
```

nie znaleziono w etapie 1 funkcji

```
void fun(int ttt[]);
```

natomiast jest funkcja

```
void fun(int *wshtab) ;
```

Wiemy przecież, że tablicę wystaną do funkcji można tam odebrać albo jako tablicę, albo jako wskaźnik do niej. To właśnie jest ta sytuacja. Odebranie tablicy jako wskaźnika jest tzw. trywialną konwersją.

Oto zestawienie innych konwersji uznawanych za trywialne. T jest symbolem jakiegoś typu.

| Konwersja   |                | Przykład          |                          |
|-------------|----------------|-------------------|--------------------------|
| z           | do             | wywołanie funkcji | jej deklaracja           |
| T           | T&             | fun(7);           | fun(int& a);             |
| T&          | T              | fun(przezvisko);  | fun(int b);              |
| T[]         | T*             | fun(tablica);     | fun(int * wskaznik);     |
| T(argument) | (*T)(argument) | fun(funkcyjka);   | fun((*wskfun)(int));     |
| T           | const T        | fun(5);           | fun(const int n);        |
| T           | volatile T     | fun(21);          | fun(volatile int m);     |
| T*          | const T*       | fun(wsk);         | fun(const int * www);    |
| T*          | volatile T*    | fun(wsk);         | fun(volatile int * www); |

Ważny jest tu fakt, że obie sytuacje opisane jako etap 1) i etap 2) są dopasowaniem dosłownym. Dosłownym dlatego, że argument wywołania funkcji może dosłownie (czyli bez żadnych przeróbek) zainicjalizować określony argument formalny.

Oczywiście dopasowanie bez konwersji trywialnej jest dokładniejsze niż takie, w którym ta konwersja musi nastąpić. Czyli, że dopasowanie

`T[]` —————> `T[ ]`

jest lepsze niż

`T[]` —————> `T*`



Następne etapy to już nie dopasowanie dosłowne. Jeśli więc, mimo dotychczasowych prób dopasowania, do wywołania nie udało się dopasować żadnej z funkcji - trzeba zacząć lekko zmieniać typ argumentów wywołania. Lepiej tak, niż nie dopasować wcale.

### 9.9.3 Etap 3. Dopasowanie z awansem

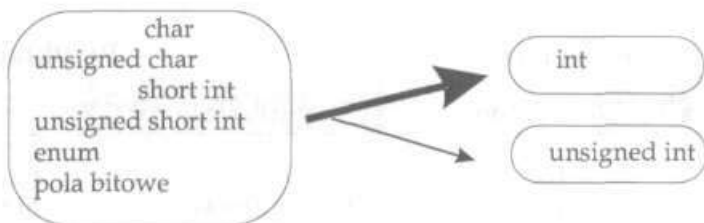
Tak! Argumenty wywołania awansują. Droga awansu dla argumentów zmienoprzecinkowych jest taka: float awansuje na typ double.

`float` —————> `double`

Jeśli po takim awansie uda się dopasować jednoznacznie jakąś funkcję - to dopasowanie się udało. Jak widać jest to awans do "większej dokładności".

Dla argumentów całkowitych droga awansu jest taka:

Wersje `signed` i `unsigned` następujących typów: `char`, `short`, `int`, typy wyliczeniowe (`enum`), pola bitowe (patrz. str 323) awansują do typu `int` - jeśli taki awans nie powoduje utraty części informacji. W przeciwnym razie jest awans do typu `unsigned int`.



### Dygresja

Pamiętasz na pewno, że liczby całkowite w różnych typach komputerów mogą być reprezentowane w inny sposób. Nic więc dziwnego, że również awans może odbyć się różnie na różnych komputerach. (Czasem na typ `int`, a czasem na typ `unsigned int`).

Jeżeli jednak nie zamierzasz uruchamiać swojego programu na różnych typach komputerów to na razie nie musisz się tym martwić.

Przykład dopasowania z awansem.

Mamy takie dwie funkcje `fff`:

```
void fff(int);
void fff(float *) ;
```

Wówczas wywołanie

```
char znak = 'A' ;
fff(znak)
```

rozważane jest etapami tak:

Etap 1 - dopasowanie dosłowne: nierealne.

Etap 2 - dosłowne z trywialną konwersją: też nie wychodzi

Etap 3 - awans. Argument `znak` awansuje do typu `int`. Po tym awansie wywołanie pasuje do funkcji

```
void fff(int) ;
```

Problem więc rozwiązany.

## 9.9.4 Etap 4. Próba dopasowania za pomocą konwersji standardowych

Konwersjom standardowym poddawany jest oczywiście argument wywołania funkcji.

Do konwersji standardowych oprócz wspomnianych wyżej awansów należą

### ❖ Konwersja typu całkowitego

```
int —————> unsigned int
unsigned int —————> int
```

### ❖ Konwersja typów zmiennoprzecinkowych

float                      →                      double                      (było w etapie 3)  
double                      →                      float

❖ Konwersja między typami całkowitym a zmiennoprzecinkowym

typ zmiennoprzecinkowy → typ całkowity

typ całkowity → typ zmiennoprzecinkowy

❖ Konwersje arytmetyczne

- czyli takie konwersje jak te, które zwykle robione są automatycznie przy wykonywaniu wyrażeń arytmetycznych.  
Przykładowo: jeśli trzeba pomnożyć dwie liczby: jedną long a drugą short, to liczba short najpierw zamieniana jest na liczbę long, po czym dopiero na dwóch liczbach long wykonywane jest działanie.  
Takie konwersje nigdy nie powodują utraty jakiejkolwiek części informacji.

❖ Konwersje wskaźników

- 0 (zero) może być zamienione na wskaźnik do NULL
- wskaźnik dowolnego nie-const i nie-volatile typu może być zamieniony na wskaźnik typu void\*

(dalsze uwagi dotyczą klas – zakładam, że czytasz książkę po raz drugi)

- wskaźnik do klasy pochodnej może być zamieniony na wskaźnik do klasy podstawowej, od której ta klasa pochodna się wywodzi.

❖ Konwersja referencji

- referencja do klasy pochodnej może być zamieniona na referencję do klasy podstawowej.

W czwartym etapie dopasowania argument wywołania funkcji poddawany będzie powyższym konwersjom standardowym. Możesz więc wysłać do przeładowanej funkcji argument, któremu (dzięki tym konwersjom) kompilator znajdzie jakąś pasującą do niego funkcję.

Jednak rada jest taka:



Nie licz zbyttnio na standardowe konwersje, bo może się zdarzyć, że przy konwersji część informacji zostanie stracona. Przykładowo:

```
void fff(int) ;
void fff(char *) ;

float pi = 3.14 ;

fff(pi) ;
```

Po dopasowaniu do funkcji void fff(int) wartość 3.14 zostanie zamieniona na 3 – Tego chciałeś?

## 9.9.5 Etap 5. Próba dopasowania z użyciem konwersji zdefiniowanych przez użytkownika.

Jak pewnie już wiesz – z pierwszego czytania tej książki - w zdefiniowanej przez siebie klasie możesz umieścić funkcję zamieniającą obiekt jakiegoś typu na obiekt typu tej klasy. (Patrz str. 399).

Jeśli argument wywołania da się według takiej konwersji zamienić na typ odpowiadający argumentowi formalnemu, to dopasowanie jest pomyślne.



Ale uważaj: dla argumentu robiona jest tylko jedna taka konwersja. Nawet jeśli zdefiniowałeś jak z typu A zrobić typ B, oraz jak z typu B zrobić typ C, a także jak z typu C zrobić typ D. Nic z tego. Kompilator zastosuje najwyżej tylko jedną z nich. Nie robi całej „kaskady”.

(Bardziej szczegółowo o tym na stronie 416).

## 9.9.6 Etap 6. Próba dopasowania do funkcji z wielokropkiem

Jest to już rozpaczliwa próba. Jeśli do tej pory nie powiodło się żadne dopasowanie, to szuka się funkcji mającej w liście argumentów wielokropek. Wielokropek oznacza: dowolna liczba argumentów dowolnego typu.

Np.

```
void fun(int) ;
void fun (float) ;
void fun(...) ;
// -----a wywołanie wygląda w ten sposób
fun("napis") ;
```

Żadna z zadeklarowanych nie daje się dopasować, bo argument wywołania to przecież wskaźnik. Jest jednak funkcja, która ma na liście argumentów formalnych wielokropek. Dopasowanie następuje do tejże funkcji.

Jeśli na liście jest kilka argumentów formalnych, a dopiero potem wielokropek, to pamiętajmy, że:

Wielokropek na liście argumentów formalnych funkcji obejmuje argumenty od tej pozycji, na której stoi, oraz ewentualne dalsze.

## 9.10 Dopasowywanie wywołań z kilkoma argumentami

Jeśli w wywołaniu funkcji jest kilka argumentów, to wówczas procedura dopasowania argumentów odbywa się na każdym z nich.

Ze wszystkich wersji funkcji wybrana zostaje ta wersja, do której parametry pasują tak samo lub lepiej niż do innych wersji. Co to oznacza ? Wyobraźmy sobie, że mamy wywołanie dwuargumentowe

```
fun(2, 5) ;
```



a funkcje, spośród których będzie kompilator wybierał to:

```
fun(float, unsigned int);
fun(float, float);
fun(int *, float) ;
```

Ostatnia wersja – ta ze wskaźnikiem od razu odpada. Nie da się przecież konwersjami standardowymi zamienić liczby 2 na wskaźnik do `int`.

Zostają zatem tylko dwie funkcje. Pierwszy argument wywołania czyli 2 pasuje tak samo źle do obu wersji, no ale w końcu po standardowej konwersji można wytrzymać.

Natomiast drugi argument wyraźnie lepiej pasuje do wersji

```
fun(float, unsigned int);
```

Wyraźnie lepiej – dlatego, że wystarczy jego awans do typu `unsigned int`. Awans, jak wiemy, jest lepszy niż ewentualna konwersja standardowa podobna do tej, jaką zrobiliśmy z argumentem pierwszym. Zatem – skoro pierwszy argument pasował tak samo (tak samo źle) od obu wersji funkcji, natomiast drugi argument pasował lepiej do wersji

```
fun(float, unsigned int);
```

to ta wersja zostanie przez kompilator wybrana.

