C++ i Pascal
kurs podstawowy

Wykład: program, algorytm, kompilator, interpreter, debugger, linker, zmienne, typy danych, komentarze, instrukcje wejścia, wyjścia, operatory, instrukcja warunkowa if, pętla for, while, do while, repeat, until, sleep, czyszczenie ekranu, liczby pseudolosowe, tablice, funkcje matematyczne, sqrt, switch, case, łańcuchy, string, własne funkcje, parametry formalne, aktualne, void, wskaźniki, rezerwowanie pamięci, schematy blokowe

Podstawowe pojęcia w programowaniu

- **Program komputerowy:** zespół kodowanych instrukcji, określający dokładnie przebieg operacji arytmetycznych i logicznych do wykonania przez komputer.

- **Algorytm:** sposób w postaci ściśle określonych reguł i metod na rozwiązanie określonego zadania w skończonej liczbie kroków.

- **Cechy charakterystyczne poprawnego algorytmu:**

Poprawność - dla każdego przypisanego zestawu danych, po wykonaniu skończonej liczby czynności, algorytm prowadzi do poprawnych wyników.

Jednoznaczność - w każdym przypadku zastosowania algorytmu dla tych samych danych otrzymamy ten sam wynik.

Szczegółowość - wykonawca algorytmu musi rozumieć opisane czynności i potrafić je wykonywać.

Uniwersalność - algorytm ma służyć rozwiązywaniu pewnej grupy zadań, a nie tylko jednego zadania. Przykładowo algorytm na rozwiązywanie równań w postaci ax+b=0 ma je rozwiązać dla dowolnych współczynników a i b, a nie tylko dla jednego konkretnego zadania, np. 2x + 6 = 0

Pod względem konstrukcyjnym algorytmy dzielimy na:

- LINIOWE – poszczególne kroki wykonywane są jeden po drugim (np. pole prostokąta)

- ROZGAŁĘZIONE, w wyniku sprawdzenia warunku mogą być wykonywane różne kroki algorytmu (obliczania pierwiastków równania kwadratowego)

- ITERACYJNE, w których pewien zespół kroków wykonywany jest wielokrotnie w pętli, aż do momentu spełnienia określonego warunku

- REKURENCYJNE, w których proces wykonywania jakiegoś zadania wywołuje same siebie,

- MIESZANE – łączą konstrukcje różnych algorytmów.

Kompilator (ang. compiler): program służący do tłumaczenia kodu napisanego w jednym języku (np. C++, Pascal) na równoważny kod w języku maszynowym (zero-jedynkowym).

Interpreter: program komputerowy, który analizuje kod źródłowy programu, a przeanalizowane fragmenty wykonuje. Dzieje się tak inaczej niż w procesie kompilacji, podczas którego nie wykonuje się wejściowego programu (kodu źródłowego), lecz tłumaczy go do wykonywalnego kodu maszynowego.

KOMPILACJA - tłumaczenie kodu źródłowego na kod wynikowy

INTERPRETACJA - tłumaczenie z natychmiastowym wykonaniem programu

Zarówno kompilator jak interpretator można określić mianem translatora.

- **Konsolidator** (ang. **linker**) lub program, który łączy zadane pliki obiektowe i biblioteki tworząc w ten sposób plik wykonywalny.

- **Debugger** (czyt. „debager") – program komputerowy służący do dynamicznej analizy innych programów, w celu odnalezienia i identyfikacji zawartych w nich błędów, zwanych z angielskiego bugami (robakami). Proces nadzorowania wykonania programu za pomocą debuggera określa się mianem debugowania.

- Zintegrowane środowisko programistyczne (ang. Integrated Development Environment, **IDE**) jest to aplikacja lub zespół aplikacji (środowisko) służących do tworzenia, modyfikowania, testowania i konserwacji oprogramowania.

- Rapid Application Development (również **RAD**) oznacza "szybkie tworzenie aplikacji". Jest to ideologia i technologia polegająca na udostępnieniu programiście dużych możliwości prototypowania oraz dużego zestawu gotowych komponentów.

Tworzenie nowego projektu w Code::Blocks

Aby utworzyć nowy projekt zawierający program konsolowy, po wybraniu z menu ***File*** → ***New*** → ***Project...***, wskazujemy w kreatorze:

Następnie należy wybrać język ***C++***, nadać projektowi nazwę i przydzielić mu folder roboczy. Użyjemy domyślnego kompilatora ***GNU GCC***. Kod naszego programu znajdować się będzie w pliku ***main.cpp***:

Struktura programu - „Hello World!”

Domyślny program – tzw. *„Hello World!”* (*Witaj Świecie!*) ma następującą strukturę:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      cout << "Hello world!" << endl;
8
9      return 0;
10 }
```

`#include <iostream>` oznacza dodanie do programu biblioteki iostream, która jest biblioteką strumienia wejścia/wyjścia (input/output stream).

`using namespace std;` oznacza używanie w programie przestrzeni nazw biblioteki standardowej, co w praktyce pozwala nam korzystać z uproszczonych zapisów – zamiast pisać:

```
std::cout << "Hello world!" << endl;
```

możemy używać zapisu:

```
cout << "Hello world!" << endl;
```

`int main()` główna funkcja programu – rozpoczyna się klamrą otwierającą: `{`, a kończy klamrą zamykającą: `}`. Funkcja main() ma zwrócić wartość typu int, czyli liczbę całkowitą – dokonuje tego zapis: `return 0;`

`cout << "Hello world!" << endl;` linia ta odpowiada za wyświetlenie na ekranie tekstu zawartego w cudzysłowach (oznaczonego niebieskim kolorem). Wyrażenie `endl;` tłumaczymy jako *end line*, czyli wstawienie znaku końca linii i jednoczesne umieszczenie kursora w nowej. Operator `<<` pełni rolę separatora oddzielającego poszczególne instrukcje.

Struktura programu – porównanie C++/Pascal

Porównanie struktury programu „Hello World!” w C++ i Pascalu:

C++	Pascal

`#include <iostream>`
 	
`using namespace std;`
 	
`int main()`
`{`
 	
`cout<<"Hello world!"<<endl;`
 	
`return 0;`
`}`
	`Program hello;`
	
	`uses crt;`
	
	`begin`
	
	`writeln('Hello world');`
	
	`end.`

Zmienne, typy zmiennych, komentarze

Zmienna to niejako „szufladka” w pamięci, w której przechowujemy dane określonego typu – np. liczbę całkowitą, rzeczywistą, napis (łańcuch), lub wartość logiczną. Zmienną w C++ definiujemy używając następującego zapisu:

```
typ_zmiennej nazwa;
```

Na przykład:

```
int liczba;
```

Taki zapis to informacja dla kompilatora: *zarezerwuj w pamięci miejsce o nazwie* „liczba”*, gdzie będzie przechowywana liczba całkowita*. Od tego momentu wszędzie gdzie kompilator napotka słowo „liczba”, będzie w domyśle pracował na tym obszarze pamięci.

Definiowanie zmiennej w Pascalu:

```
var nazwa:typ_zmiennej;
```

Na przykład:

```
var liczba:integer;
```

Słowo kluczowe **var** oznacza tutaj *variable* – zmienną.

Typy zmiennych występujące w C++/Pascalu:

C++	Pascal	Co przechowuje?	Zakres
	byte	Liczby całkowite	0 .. 255
int	integer	Liczby całkowite	-32768 .. 32767
short int	shortint	Liczby całkowite	-128 .. 127
long int	longint	Liczby całkowite	-2147483648 .. 2147483647
float	real	Liczby zmiennoprzecinkowe	3.4e +/- 38 (7 znaków)
double	double	Liczby zmiennoprzecinkowe	1.8e +/- 308 (15 znaków)
long double		Liczby zmiennoprzecinkowe	1.1e +/- 4932 (15 znaków)

bool	**boolean**	Wartości logiczne	true/false
char	**char**	Pojedynczy znak	od 0 do 127 kody ASCII
string	**string**	Łańcuchy	

W naszych pierwszych aplikacjach, tworzyć będziemy każdą zmienną jako zmienną globalną, to znaczy widzianą w całym programie. Zatem deklaracje zmiennych umieszczać będziemy pomiędzy using namespace std; a rozpoczęciem funkcji głównej: int main()

```
1  #include <iostream>
2
3  using namespace std;
4
5  int liczba;  //tutaj wstawiamy zmienne
6  string imie; //uzywane w programie
7
8  int main()
9  {
10     cout << "Hello world!" << endl;
11
12     return 0;
13 }
```

W liniach 5 i 6 użyto komentarza – jest to tekst poprzedzony znakami: `//`. Komentarze są ignorowane przez kompilator – wstawiamy je tylko dla naszej informacji – aby wiedzieć co robi dana zmienna czy funkcja; łatwiej wtedy wrócić do własnego kodu po upływie dłuższego czasu. Kompilator oznacza komentarze na szaro. Przykład użycia komentarza:

```
int x; // x - liczba cukierków
```

Zmiennej możemy nadawać wartość już na etapie tworzenia:

```
string imie="Jan";
int x=67; // x - liczba cukierków
int y=31; // y - liczba uczniow
```

lub wewnątrz funkcji main(), lecz po wcześniejszym zadeklarowaniu:

```
x=67;
y=31;
```

Instrukcje wejścia/wyjścia

W zmiennych przechowujemy dane wprowadzane przez użytkownika. Aby wczytać do zmiennej wartość podaną z klawiatury, używamy instrukcji wejścia/wyjścia. Na przykład aby pobrać imię użytkownika:

C++	Pascal
`string imie;`	`imie:string;`
`cout<<endl<<"Podaj imie: ";`	`writeln('Podaj imie: ');`
`cin >> imie;`	`readln(imie);`

Analogicznie wczytujemy liczbę, zmieniając jedynie typ danych.

Operatory w C++/Pascalu

Oto zestawienie podstawowych operatorów arytmetycznych i logicznych używanych podczas programowania w C++ (dla przypomnienia analogiczne operatory z Pascala):

C++	OPIS DZIAŁANIA	PASCAL
+ - * /	działania matematyczne	**+ - * /**
< <= > >=	porównania	**< <= > >=**
== !=	równość, nierówność	**= <>**
%	reszta z dzielenia	**mod**
&&	iloczyn logiczny	**AND**
||	suma logiczna	**OR**
!	negacja	**NOT**
=	przypisanie wartości	**:=**

Działanie operatorów logicznych OR i AND (dla dwóch warunków):

WARUNEK 1	WARUNEK 2	OR	AND
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	

Instrukcja warunkowa if

Instrukcja warunkowa to rozgałęzienie w działaniu programu (*if* znaczy *jeżeli*). W zależności od tego, czy warunek zawarty w instrukcji jest prawdziwy lub fałszywy, wykonane zostają inne instrukcje. Klauzula **else** jest opcjonalna, to znaczy nie musi koniecznie wystąpić – zależy to od nas i rozpatrywanego problemu. Składnia w pseudokodzie:

```
if(warunek_logiczny)
{
instrukcje jeśli warunek_logiczny prawdziwy;
}
else
{
instrukcje jeśli warunek_logiczny fałszywy;
}
```

Rozważmy przykład bankomatu. Aby móc dokonać transakcji, użytkownik musi podać poprawny numer PIN. Sprawdzenia poprawności możemy dokonać instrukcją warunkową – niech poprawny numer pin to 1945:

```
if(PIN==1945)
{
 cout << "Poprawny PIN" << endl;
}
else
{
 cout << "Niepoprawny nr PIN!" << endl;
}
```

Zwróć uwagę na operator porównania ==, będący podwójnym znakiem równości (w odróżnieniu od operatora przypisania). Pamiętaj, żeby nie umieszczać średnika w linii if, gdyż wówczas kompilator potraktuje średnik jako pustą instrukcję, zaś linie wewnątrz klamer wykonają się zawsze, niezależnie od warunku:

```
if(PIN==1945); //średnik zmienia działanie instrukcji!
{
 cout << "Poprawny PIN" << endl;
}
```

Warunki mogą być złożone, zaś łącznikami są operatory: `&&`, `||`, `!`. Rozważmy przykład logowania do systemu operacyjnego:

```
if((login=="admin")&&(haslo=="admin"))
{
cout << "Poprawny PIN" << endl;
}
```

Albo wprowadzania odpowiedzi na pytanie testowe małą lub duża literą:

```
if((odp=="a")||(odp=="A"))
{
cout << "Poprawny PIN" << endl;
}
```

Zauważ, że oba warunki zamknięte są w dodatkowym nawiasie. Działanie operatorów logicznych:

WARUNEK 1	WARUNEK 2	OR (`||`)	AND (`&&`)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

WARUNEK	NOT (`!`)
0	0
1	1

Instrukcje warunkowe można zagnieżdżać, zatem po klauzuli **`else`** może pojawić się kolejna instrukcja warunkowa **`if`**

Instrukcja **`if`** w Pascalu:

```
if(PIN=1945)then
begin
writeln('Poprawny PIN');
end
```

```
else

begin
writeln('Niepoprawny nr PIN!');
end;
```

PĘTLE W C++ I PASCALU

Pętla for

Pętle służą do zdefiniowania szeregu instrukcji, które będą powtarzane wielokrotnie. W przypadku pętli for z góry wiemy ile razy pętla ma się wykonać. Aby zdefiniować w C++ pętlę wykonującą się 10 razy:

```
for(int i=1; i<=10; i++)
{
instrukcje do wykonania;
}
```

Zmienna `i` to liczba typu całkowitego nazywana *zmienną sterująca pętlą*. W powyższym przykładzie w pierwszym wywołaniu pętli przypisujemy jej wartość `i=1;` W każdym przebiegu pętli inkrementujemy jej wartość (zapis `i++`), czyli zwiększamy o jeden (`i=i+1`). Warunek `i<=10;` określa koniec iterowania (powtarzania) instrukcji – tzn. pętla wykonuje się dopóki warunek zwraca wartość `true`.

Pętla może zliczać *w dół*:

```
for(int i=10; i>=1; i--)
{
instrukcje do wykonania;
}
```

wówczas dokonujemy w każdym kroku zmniejszenia wartości zmiennej `i` o jeden – dekrementujemy jest wartość. Zmienił się też sprawdzany warunek.

Możemy także dodawać/odejmować w każdym wywołaniu pętli dowolną wartość całkowitą, zamiast inkrementować/dekrementować:

```
for(int i=1; i<=10; i=i+2)
{
cout<<endl<<i;
}
```

Podobnie jak przy instrukcji warunkowej, na końcu linii definiującej pętle nie może znajdować się średnik. Wobec tego poniższy zapis jest niepoprawny (zmieni działanie instrukcji – 10 razy wykonana zostanie pusta instrukcja (średnik), zaś linie kodu między klamrami wykonają się zawsze jeden raz):

```
for(int i=1; i<=10; i=i+2);
{
cout<<endl<<i;
}
```

Składnia pętli for w Pascalu:

```
for i:=1 to 10 do
begin
writeln(i);
end;
```

```
for i:=10 downto 1 do
begin
writeln(i);
end;
```

Pętla while, do..while

Pętle te sterowane są warunkiem, zatem instrukcje powtarzane są wielokrotnie, dopóki warunek w nawiasie jest spełniony. W przeciwieństwie do pętli for nie musimy od razu definiować po ilu iteracjach pętla zakończy działanie. Różnica między pętlą while i do..while polega na tym, iż w przypadku pętli while warunek sprawdzany jest na początku, zaś w do..while na końcu bloku instrukcji. Stąd linie kodu zawarte w pętli do..while wykonają się zawsze przynajmniej jeden raz.

Składnia pętli while na przykładzie odgadywania liczby „pomyślanej” przez komputer:

```
while(strzal!=liczba)
{
instrukcje realizujące odgadywanie;
}
```

Dopóki (while) liczba którą wpisaliśmy z klawiatury (strzal) jest różna od wylosowanej przez komputer liczby (liczba) powtarzaj instrukcje odpowiedzialne za odgadywanie. Taka sama pętla do..while ma

następującą składnię:

```
do
{
instrukcje realizujące odgadywanie;
}while(strzal!=liczba);
```

Zwróć uwagę, że tym razem wymagany jest na końcu średnik.

Pętla while w Pascalu:

```
while(strzal<>liczba)do
begin
instrukcje realizujące odgadywanie;
end;
```

Pętli do..while w Pascalu odpowiada pętla **repeat..until** (powtarzaj..dopóki):

```
repeat
instrukcje realizujące odgadywanie;
until(strzal<>liczba);
```

Dźwięk w programie

Głośnik systemowy odezwie się po wpisaniu instrukcji:

```
cout << "\a";
```

Litera `a` wzięła się od słowa *alarm*.

Dźwięk w Pascalu:

```
Sound;
```

Kolory czcionki w konsoli

Do zmiany koloru czcionki w konsoli użyjemy funkcji Windows API. Do bibliotek używanych w programie dodajemy moduł:

```
#include <windows.h>
```

Zmianę koloru realizujemy następująco:

```
SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),x);
```

gdzie wstawiając za `x` poniższe wartości otrzymamy kolory – na przykład:

10 = green 15 = white 6 = brown 12 = red
13 = magenta 9 = blue 5 = purple 4 = dark red

Kolor czcionki w Pascalu:

```
TextColor(x);
```

Funkcja Sleep()

Aby wprowadzić opóźnienie w wykonaniu instrukcji i wstrzymać program na dany czas, wystarczy użyć funkcji Sleep(). Do bibliotek używanych w programie dodajemy moduł:

```
#include <windows.h>
```

Argumentem funkcji Sleep() jest czas wyrażony w milisekundach – stąd aby wstrzymać wykonanie programu na sekundę, należy użyć:

```
Sleep(1000);
```

Wstrzymanie programu w Pascalu:

```
Delay(1000);
```

Czyszczenie ekranu

Do bibliotek używanych w programie, podobnie jak w poprzednich przypadkach dodajemy moduł:

```
#include <windows.h>
```

Czyszczenie ekranu realizujemy przy użyciu:

```
system("CLS");
```

Czyszczenie ekranu w Pascalu:

```
clrscr;
```

Liczby pseudolosowe

Do bibliotek używanych w programie, podobnie jak w poprzednich przypadkach dodajemy moduły:

```
#include <windows.h>
#include <time.h>
```

Generator liczb pseudolosowych należy zainicjować (ale tylko raz!) w programie:

```
srand(time(NULL));
```

Liczbę pseudolosową z zakresu 1..100 uzyskujemy w następujący sposób:

```
liczba=rand()%100+1;
```

Funkcja rand() zwraca liczbę z zakresu od zera do liczby po operatorze `%`. W naszym przypadku aby uzyskać zakres 1..100 najpierw definiujemy sto liczb (od 0 do 99), po czym dodajemy do wylosowanej liczby 1.

Generator liczb pseudolosowych należy inicjowanie w Pascalu:

```
Randomize;
```

Liczbę pseudolosową z zakresu 1..100 uzyskujemy w Pascalu w następujący sposób:

```
liczba=random(99)+1;
```

Tablice w C++/Pascalu

Tablice to uporządkowane zbiory danych i występują praktycznie w każdym języku programowania. W momencie kiedy tworzymy w programie zmienną – np.:

```
int liczba;
```

kompilator rezerwuje w pamięci miejsce na przechowywanie liczby całkowitej. Jest to jakby „szufladka” w pamięci. Jeżeli natomiast potrzebujemy przechować np. 5 liczb w pamięci, możemy zastosować tablicę. Wtedy zamiast tworzyć 5 zmiennych i pisać:

```
int liczba1, liczba2, liczba3, liczba4, liczba5;
```

możemy zadeklarować tablicę przechowującą 5 liczb:

```
int liczby[5];
```

Powstaje wówczas w pamięci 5 „szufladek” na liczby – ponumerowanych od zera do czterech:

Liczba przechowywana w pamięci	45	15	14	144	17
Indeks (nr szufladki)	0	1	2	3	4

Ponumerowanych od zera, ponieważ stosowana jest notacja amerykańska. Zatem jeżeli chcę wyświetlić liczbę przechowaną w „szufladce” o numerze dwa, posłużę się instrukcją:

```
cout<<liczby[2];
```

Liczba przechowywana w pamięci	45	15	**14**	144	17
Indeks (nr szufladki)	0	1	**2**	3	4

Tablice w Pascalu:

```
var liczby: array[0..4] of integer;
```

Słowo **var** oznacza *variable* - zmienne, słowo **array** oznacza tablicę.

Funkcje matematyczne w C++

Funkcje matematyczne zawarte są w bibliotece math.h. Zatem, aby móc z nich skorzystać w tworzonym programie, wymagane jest dołączenie tego modułu do listy używanych bibliotek:

```
#include <math.h>
```

Oto lista funkcji matematycznych używanych w pierwszych programach, wraz z przykładami ich zastosowania:

Funkcja	Zastosowanie	Przykład użycia
`sqrt();`	Zwraca pierwiastek kwadratowy	`wynik=sqrt(liczba);`

`pow();`	Zwraca wynik podniesienia liczby do potęgi określonej wewnątrz funkcji po przecinku	`wynik=pow(liczba,2);`
`fabs();`	Zwraca wartość bezwzględną liczby zmiennoprzecinkowej	`wynik=fabs(liczba);`

Przełącznik w C++

Instrukcja **switch**..**case** pozwala zastąpić szereg wywołań instrukcji warunkowych. Znakomicie nadaje się do budowy menu w programach konsolowych. Składnia pokazana została w prostym kalkulatorze:

```
cout << "KALKULATOR- MENU GLOWNE:" << endl;
cout << "1. Dodawanie" << endl;
cout << "2. Odejmowanie" << endl;
cin >> wybor;

    switch(wybor)
    {
        case 1:
          //instrukcje realizujace dodawanie liczb
        break;

        case 2:
         //instrukcje realizujace odejmowanie liczb
        break;

        default:
        cout << "Zly wybor!" << endl;
        break;
    };
```

Instrukcję czytamy następująco: **przełącz** (w zależności od wartości zmiennej `wybor`) - **w przypadku** wartości `wybor==1` wykonaj dodawanie i **przerwij** działanie switcha, **w przypadku** wartości `wybor==2` wykonaj odejmowanie i **przerwij** działanie switcha, jeśli nie napotkałeś żadnej z wymienionych wartości, **domyślnie** wypisz na ekranie napis „Zly wybor!”.

Jeżeli dla kilku wartości zmiennej sterującej należy wykonać te same instrukcje, nie trzeba zapisywać ich kilkakrotnie. Za przykład niech

posłuży fragment programu wypisujący liczbę dni danego miesiąca, w zależności od jego numeru (przypomnij sobie zajęcia) – tutaj zapis dla miesięcy mających 30 dni:

```
    switch(nr_miesiaca)
    {
        case 4:
        case 6:
        case 9:
        case 11:

           cout << "Ten miesiac ma 30 dni!" << endl;

        break;
    };
```

Przełącznik w Pascalu

Instrukcji **switch**..**case** w Pascalu odpowiada instrukcja **case**. Składnia pokazana została poniżej, zaś działanie przełącznika jest analogiczne jak w C++:

```
case wybor of

1: begin //instrukcje realizujace dodawanie end;
2: begin //instrukcje realizujace odejmowanie end;

else writeln('Zly wybor');

end;
```

Warto zauważyć że defaultowi odpowiada tutaj else (jedyny taki else w Pascalu, przed którym może stać średnik ☺)

Łańcuch jako tablica znaków

Słowa (łańcuchy) zapisane są w pamięci komputera najczęściej jako tablice znaków (stąd nazwa łańcuch – ogniwami słowa (łańcucha) są poszczególne jego litery). Jedyna różnica polega tutaj na tym, iż na końcu tablicy znakowej znajduje się NULL (aby poinformować komputer w którym miejscu słowo się kończy).

Zatem, aby dostać się do konkretnej litery w słowie posługujemy się charakterystycznymi dla tablic operatorami nawiasów kwadratowych. Załóżmy, że istnieje następująca zmienna:

```
string slowo="Bajka";
```

Wówczas, aby uzyskać w programie dostęp do trzeciej litery tego słowa („j") należy odwołać się do niego tak jak do tablicy:

```
slowo[2];
```

Pamiętaj o notacji amerykańskiej, czyli numerowaniu od zera, a nie europejskiej (od jedynki). Zatem nasze słowo istnieje w pamięci jak następująca tablica:

Znak	B	a	j	k	a	NULL
Znak	0	1	2	3	4	5

Długość łańcucha

W C++ istnieje szereg funkcji operujących na łańcuchach. Na początku poznamy tą najczęściej używaną – do pobierania długości łańcucha. Zastosowanie pokazano na przykładzie:

```
string slowo="Bajka";

int dlugosc=slowo.length();
```

Funkcje jako podprogramy

W bardzo rozbudowanych programach bardzo często zachodzi potrzeba wielokrotnego powtarzania pewnego bloku instrukcji. C++ umożliwia nam niejako „oderwanie" od programu głównego pewnego bloku instrukcji, który możemy określić mianem podprogramu lub funkcji.

Funkcja to podprogram mający za zadanie wykonać pewne (znane tylko sobie) instrukcje na danych wejściowych i zwrócić głównemu programowi wynik swoich działań. Z punktu widzenia głównego programu możemy powiedzieć, że mamy tu do czynienia z podejściem „czarnej skrzynki",

ponieważ interesują nas tylko dane wejściowe i rezultaty na wyjściu funkcji.

Definiowanie własnej funkcji w C++:

```
typ_zwracanej_wartosci nazwa_funkcji(typ_danych zmienna_we)
{
    return zmienna_wyjsciowa;
}
```

Jak to zwykle bywa, najłatwiej zrozumieć działanie funkcji na przykładzie. Rozważmy kalkulator wykonujący dodawanie lub odejmowanie dwóch liczb:

```
#include <iostream>

using namespace std;

float liczba1, liczba2;

float dodaj(float a, float b)
{
    return a+b;
}

float odejmij(float a, float b)
{
    return a-b;
}

int main()
{
    cout<<"Kalkulator"<<endl;

    cout<<"Podaj pierwsza liczbe: ";
    cin>>liczba1;

    cout<<"Podaj pierwsza liczbe: ";
    cin>>liczba2;

    cout<<"Suma = "<<dodaj(liczba1,liczba2);
    cout<<endl;
    cout<<"Roznica = "<<odejmij(liczba1,liczba2);
```

```
    return 0;
}
```

W powyższym programie zastosowano dwie własne funkcje o nazwach: `dodaj` oraz `odejmij`. Obie funkcje mają zwrócić do głównego programu liczbę typu **`float`**, obie obliczają ją „po swojemu”. Zwrócenie wartości określane jest słowem kluczowym **`return`**.

Wywołanie działania funkcji następuje w liniach:

```
cout<<"Suma = "<<dodaj(liczba1,liczba2);

cout<<"Roznica = "<<odejmij(liczba1,liczba2);
```

Wywołania dokonujemy poprzez podanie nazwy funkcji oraz parametrów wejściowych – tutaj dwóch liczb typu **`float`**. Po wykonaniu funkcji w miejsce wywołania zwracana jest wartość wyjściowa, czyli odpowiednia liczba (suma lub różnica) zostaje wyświetlona na ekranie.

Parametry formalne i aktualne funkcji

Zauważ, że liczby zdefiniowane w naszym programie są nazwane jako zmienne `liczba1, liczba2`. Natomiast wewnątrz funkcji nazwano parametry wejściowe jako `a` oraz `b`. Otóż funkcja po prostu oczekuje na dwie liczby typu **`float`** – niekoniecznie na konkretnie nasze dwie zmienne. Tej samej funkcji moglibyśmy np. użyć do sumowania czy odejmowania czyichś zarobków.

Parametry formalne to argumenty wejściowe, które nazywa po swojemu funkcja aby je rozpoznawać, natomiast parametry aktualne, to argumenty dla których funkcja została wywołana. Czyli w naszym przypadku:

- parametry formalne: `a, b`
- parametry aktualne: `liczba1, liczba2`

Typ zwracanej wartości void

Oczywiście nie zawsze chcemy, aby funkcja coś zwracała. Chcielibyśmy uzyskać odpowiednik procedury w Pascalu, czyli funkcję która nic nie zwraca, a tylko wykonuje jakieś instrukcje. Wówczas posługujemy się typem danych **`void`** (z języka angielskiego: „próżny”). Przykład:

```
void dodaj(float a, float b)
```

```
{
     cout<<"Suma = "<<a+b;
}
```

Nieco zmieniliśmy działanie funkcji. Teraz wywołanie w funkcji main() wyglądałoby tak:

```
dodaj(liczba1,liczba2);
```

Nie ma tutaj couta, bo sama funkcja zawiera go w sobie, więc to ona wypisze na ekran wynik dodawania.

Przesyłanie argumentów przez wartość i referencję

Dla naszego bezpieczeństwa po wywołaniu funkcja otrzymuje jedynie kopię zmiennej, tak aby przypadkowo nie uszkodziła (zmieniła) jej wartości. Takie domyśle przesyłanie kopii nazywamy przesyłaniem przez wartość.

Oczywiście istnieje możliwość posłania do funkcji oryginałów zmiennych, zwłaszcza w przypadku, gdy chcemy aby funkcja zmieniła wartość posyłanego do niej argumentu. Przesyłanie oryginałów argumentów nazywamy przesyłaniem przez referencję. Przykład:

```
float dodaj(float &a, float &b)
{
     return a+b;
}
```

Jak widać, używamy tutaj operatora ampersand (adresu): &, który sprawia, iż zamiast kopii wartości zmiennej, funkcja otrzymuje adres komórki pamięci, gdzie przechowywana jest wartość oryginału przesyłanego argumentu.

Trzy sposoby definiowania funkcji

Funkcje można definiować na różne sposoby. Pierwszy sposób już znamy: zarówno nagłówek jak i ciało (instrukcje) funkcji umieszczamy przed funkcją główną main().

Drugi sposób polega na podaniu przed funkcją main() jedynie nagłówków funkcji (zakończonych średnikami), zaś ich nagłówki (ponownie, ale bez średników) i ciała umieszczamy pod funkcją główną:

```
#include <iostream>
```

```
using namespace std;

float liczba1, liczba2;

float dodaj(float a, float b);
float odejmij(float a, float b);

int main()
{
    cout<<"Kalkulator"<<endl;

    cout<<"Podaj pierwsza liczbe: ";
    cin>>liczba1;

    cout<<"Podaj pierwsza liczbe: ";
    cin>>liczba2;

    cout<<"Suma = "<<dodaj(liczba1,liczba2);
    cout<<endl;
    cout<<"Roznica = "<<odejmij(liczba1,liczba2);


    return 0;
}

float dodaj(float a, float b)
{
    return a+b;
}

float odejmij(float a, float b)
{
    return a-b;
}
```

Przydatne, jeśli nasze funkcję mają dużo linii kodu w sobie – nie trzeba wówczas przewijać w dół kodu źródłowego, aby zobaczyć gdzie zaczyna się main().

Trzeci sposób polega na umieszczeniu funkcji we własnej bibliotece i użyciu dyrektywy `#include`, aby ją dołączyć do programu – tego jednak nauczymy się później.

Zastosowanie wskaźników

Wskaźniki, charakterystyczne dla języków C/C++, znajdują następujące zastosowanie:

- rezerwowanie / zwalnianie obszarów pamięci
- zwiększenie szybkości zapisu/odczytu elementów tablicy, dzięki posługiwaniu się adresami komórek pamięci
- w funkcjach mogących zmieniać wartości przesyłanych do nich argumentów (funkcje otrzymują adres „oryginału" zmiennej)
- dostęp do wybranych komórek pamięci (współpraca z urządzeniem zewnętrznym, np. miernikiem temperatury)

Rezerwowanie/zwalnianie obszarów pamięci

Poniżej przedstawiono rezerwowanie w pamieci tablicy przechowującej kolejne wyrazy ciągu Fibonacciego, o rozmiarze (ilości liczb) podanym przez użytkownika (zmienna `ile`)

```
    int ile;

    cout<<"Ile wyrazow ciagu wyswietlic: ";
    cin>>ile;

long double *ciag;
ciag=new long double[ile];
```

Najpierw tworzony jest wskaźnik o nazwie `ciag` (tutaj typu **`long double`**, aby pomieścić jak największe liczby), a następnie używany jest operator **`new`** rezerwujący w pamięci miejsce na tablicę typu **`long double`**, która ma dokładnie `ile` elementów.
Ponieważ zarezerwowaliśmy pamięć, to w momencie gdy tablica `ciag` przestaje nam być w programie potrzebna, możemy usunąć ją, tak by nie zajmowała już niepotrzebnie miejsca w RAMie. Używamy do tego operatora **`delete`**:

```
delete [] ciag;
```

Należy to czytać jako: **usuń** (co?) tablicę (bo `[]`) o nazwie (jakiej?) `ciąg`.

Odczyt z tablicy za pomocą wskaźników

Załóżmy, że nasza tablica ciag zawiera już obliczone wyrazy ciagu Fibonacciego. Ny wyświetlić wyniki dotychczas posługiwaliśmy się zapisem tablicowym (który wymaga korzystania ze „spisu treści” - przypomnij sobie o czym mówiliśmy na zajęciach):

```
//Wyswietlanie wynikow
    for(int i=0; i<ile; i++)
    {
        cout<<endl<<ciag[i];
    }
```

Ten sam odczyt zrealizowany na wskaźnikach (szybszy):

```
//definiowanie wskaznika
    long double *wskaznik;
    wskaznik=&ciag[0]; //albo: wskaznik=ciag;

//Wyswietlanie wynikow
    for(int i=0; i<ile; i++)
    {
        cout<<*wskaznik++<<endl;

    }
```

Utworzony został wskaźnik o nazwie `wskaznik`, mogący pokazywać na typ **`long double`**. Następnie ustawiono go tak, aby wskazywał na pierwszy element tablicy ciag. Można użyć operatora ampersand i adresu zerowego elementu (`&ciag[0]`), albo tylko nazwy tablicy (`ciag`), bo jak pamiętamy nazwa tablicy jest jednocześnie adresem jej zerowego elementu. Zapis `*wskaznik++` oznacza, iż po pierwsze chcemy odczytać zawartość komórki na którą wskazuje wskaźnik (`*`), a po drugie w każdym następnym przebiegu pętli wskaźnik będzie pokazywał na następny element tablicy (`++`).
Odczyt (lub zapis) z użyciem wskaźników odbywa się szybciej, ponieważ znając typ danych na który pokazuje wskaźnik oraz aktualnie wskazywany element tablicy program jest w stanie domyślić się pod jakim adresem kryje się następny element.

Wypiszmy na ekranie adresy kolejnych 5 komórek pamięci zawierających liczby Fibonacciego:

```
//Wyswietlanie adresow kolejnych elementow tablicy
    for(int i=0; i<ile; i++)
    {
       cout<<"adres: "<<(unsigned long)wskaznik<<endl;
    }
```

Rezultat wywołania:

```
adres: 4077828
adres: 4077840
adres: 4077852
adres: 4077864
adres: 4077876
```

Ponieważ wartość typu **long double** zajmuje 12 bajtów, stąd wiadomo iż kolejne elementy tablicy będą zapisane pod adresami zwiększającymi się o 12.

Funkcje działające na oryginałach przesyłanych do nich argumentów

Dla naszego bezpieczeństwa po wywołaniu funkcja otrzymuje jedynie kopię zmiennej, tak aby przypadkowo nie uszkodziła (zmieniła) jej wartości. Takie domyśle przesyłanie kopii nazywamy przesyłaniem przez wartość.

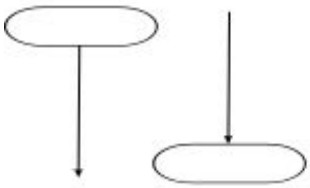
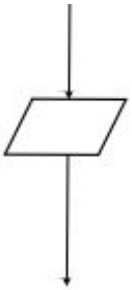
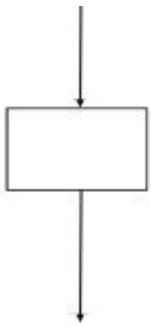
Oczywiście istnieje możliwość posłania do funkcji oryginałów zmiennych, zwłaszcza w przypadku, gdy chcemy aby funkcja zmieniła wartość posyłanego do niej argumentu. Przesyłanie oryginałów argumentów nazywamy przesyłaniem przez referencję. Przykład:

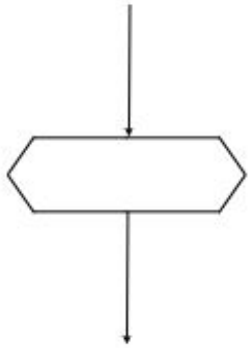
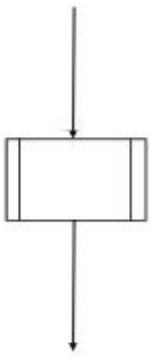
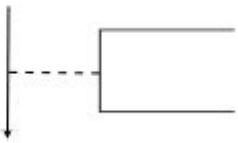
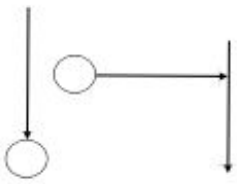
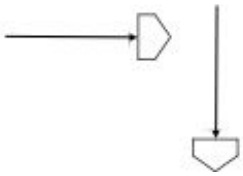
```
float dodaj(float &a, float &b)
{
    return a+b;
}
```

Jak widać, używamy tutaj operatora ampersand (adresu): &, który sprawia, iż zamiast kopii wartości zmiennej, funkcja otrzymuje adres komórki pamięci, gdzie przechowywana jest wartość oryginału przesyłanego argumentu.

Schematy blokowe

Schemat blokowy to graficzna reprezentacja algorytmu, ma na celu ukazanie sposobu rozwiązania problemu w postaci bloków (prostych figur geometrycznych). Poznaj podstawowe bloki:

	Bloki startu / stopu (początku i zakończenia algorytmu, oba inaczej nazywane blokami granicznymi)
	Blok wejścia / wyjścia (wprowadzania / wyprowadzania informacji) – w programie zawsze następuje wtedy kontakt z użytkownikiem, bo albo prosimy o wpisanie danych z klawiatury lub wskazanie pliku z danymi albo pokazujemy wyniki obliczeń na ekranie lub zgrywamy wyniki do wskazanego pliku
	Blok operacyjny (wykonania działania, wykonawczy) – następują tutaj najczęściej jakieś obliczenia lub dodatkowe operacje na danych, np. operacja przypisania jakiejś wartości do zmiennej

	Blok wywołania podprogramu (własnej funkcji lub procedury) – używamy w miejscu wywołania podprogramu – np. wewnątrz funkcji głównej o nazwie "main()" używamy podprogramu o nazwie "sqrt(16)" do policzenia pierwiastka z liczby 4
	To jest blok wywołania ZEWNĘTRZNEGO programu – chodzi o sytuację, w której nasz program uruchamia w systemie inny plik exe – np. przeglądarkę z wpisanym adresem strony www.
	Jest to komentarz, czyli wyjaśnienie "jak to działa" dla oglądającego schemat człowieka, umieszczone wewnątrz prostokąta. Łatwo rozpoznać blok komentarza po przerywanej linii – tylko ten blok takową posiada.
	Są to tzw. łączniki wewnętrzne – czasami chcemy przerwać rysowanie algorytmu bo np. chcielibyśmy umieścić na stronie akapit tekstu. Wtedy przerywamy rysowanie umieszczając znacznik i kontynuujemy pod akapitem tekstu rozpoczynając dalszą część algorytmu od łącznika. Powiązane ze sobą łączniki oznaczone są dodatkowo tym samym oznaczeniem
	Łączniki zewnętrzne – łączą dwie części algorytmu, który jest opublikowany na dwóch różnych stronach dokumentu. Również powiązane ze sobą łączniki oznaczone są tym samym oznaczeniem.

	Blok kolekcyjny (kolektor) – można luźno kojarzyć z kolektorem ściekowym – zbiera wszystkie strumienie w jeden strumień, tylko że w informatyce będą to strumienie danych, a nie strumienie ścieków. Zatem taki blok umieścimy tam, gdzie np. zbieramy wyniki z dwóch różnych funkcji w programie żeby potem użyć obu wyników w dalszej części programu.