Wykład 6

6. Funkcje

Funkcja jest grupą powiązanych instrukcji (fragmentem kodu programu), które realizują określone zadanie.

Funkcje są podprogramami, które mogą być wielokrotnie wykonywane dla różnych danych wejściowych.

6.1. Struktura funkcji

Struktura funkcji w języku C++ jest podobna do struktury funkcji main().

```
typ_wartości   nazwa_funkcji (lista_parametrów)
{
      lokalne stałe, typy danych, zmienne;

      <instrukcje>;

      return wartość zwracana;    // jeśli typ_wartości != void
}
```

Np.

```
double  suma(double x, double y)
{
      double wynik;     \\ zmienna lokalna

      wynik = x + y;

      return(wynik);    \\ zwracana wartość typu double
}


double  suma1(double x, double y)
{
      return(x+y);
}
```

Własności funkcji:

- typ zwracanej wartości jest podawany przed nazwą funkcji;
- lista parametrów funkcji może być pusta; można ten fakt podkreślić bezpośrednio posługując się słowem kluczowym *void*;
- funkcja może zwracać wynik dowolnego typu z wyjątkiem tablic i funkcji; jeśli nie podano typu funkcji, to domyślnie przyjmowany jest typ int;
- lista parametrów funkcji ma następującą postać:

  typ_1 par1, typ_2 par2, ...

  parametry deklarowane są tak jak zmienne;
  każdy parametr musi mieć indywidualnie zadeklarowany typ;
  parametry w liście są rozdzielone przecinkami;

- początek i koniec funkcji określają nawiasy klamrowe;
- w C++ parametry funkcji mogą być przekazywane przez *wartość*, *wskaźnik* lub *referencję*;
- wewnątrz funkcji nie mogą pojawiać się definicje innych funkcji;
- za pomocą słowa kluczowego *return* zwracana jest wartość funkcji; *return wartość* kończy działanie funkcji;
- jeżeli funkcja nie musi zwracać wyniku, to jako typu wartości należy podać *void*.

```
void  pisz( int liczba)
{
      printf("%3d\n", liczba);
}

void main(void)
{
      pisz(2);  pisz(7);  pisz(15);
}
```

Przykład 6.1. Wykorzystanie funkcji w programach.

```
double srednia(double x, double y)
{
  return ((x+y)/2);
}

double sum(double x, double y)
{
   return (x+y);
}

void min_max(float x, float y, float z)  //    min(x,y,z)   max(x,y,z)
{
  float min = x;
  float max = x;

if (y > max) max = y;
if (y < min) min = y;
if (z > max) max = z;
if (z < min) min = z;

printf("\nMinimum wynosi %10.4f",min);
printf("\nMaximum wynosi %10.4f",max);
}

void main(void)
{
 double w;

 clrscr();

  w = sum(2,3);
  cout << "\nSuma = " << w << " \nSuma = " << sum(2,3);
  w = srednia(2,3);
  cout << "\nSrednia = " << w << " \nSrednia = " << srednia(2,3);

 min_max(1,2,3);              // min = 1;  max = 3;
 min_max(7,4,2);              // min = 2;  max = 7;

 getch();
}
```

6.2. Deklaracja i definicja funkcji

- Deklaracja funkcji jest pojęciem logicznym. Stanowi informację dla kompilatora, że funkcja o określonej nazwie, typie wartości oraz liczbie i typie parametrów może zostać użyta (ale nie musi) w danym module programu. Deklaracja funkcji musi kończyć się średnikiem.

- Deklaracja funkcji nazywana jest *prototypem* funkcji. Pliki nagłówkowe *.h, dołączane dyrektywą #include, zawierają prototypy funkcji, których kody wynikowe dołączane są na etapie łączenia z bibliotek standardowych.

- Prototyp funkcji mówi kompilatorowi jakiego typu wartość jest zwracana przez funkcję oraz jakie są jej parametry. Dzięki temu możliwe jest podczas kompilacji sprawdzenie czy w poprawny sposób wykorzystywana jest wartość zwracana przez funkcję oraz czy podawane są parametry odpowiednich typów. Ponadto, prototypy pozwalają szybko zorientować się jakie funkcje są dostępne i jak z nich korzystać.

- *Definicja* funkcja mówi o tym co funkcja robi. Zawiera instrukcje umożliwiające realizację odpowiedniego fragmentu programu. W odróżnieniu od deklaracji przydzielany jest obszar pamięci, w którym znajdzie się kod wynikowy funkcji.

Np.
```
 double sum(double x, double y);          // prototyp funkcji – średnik;

void main(void)
{
 double w;
 clrscr();     w = sum(2,3);
  cout << "\nSuma = " << w << " \nSuma = " << sum(2,3);
getch();
}

double sum(double x, double y)            // definicja funkcji
{
   return (x+y);
}
```

Przykładowe prototypy

```
float kwadrat (float x);        // funkcja o wartościach typu float
                                // i jednym parametrze typu float

fun(int);                       // funkcja o wartościach typu int (domyślnie)
                                // i jednym parametrze typu int

void sum(int, long);            // funkcja nie zwracająca wartości
                                // o dwóch parametrach int i long
```

Wykorzystanie prototypów

Jeżeli typy argumentów przekazywanych do funkcji nie są zgodne z typami określonymi w definicji czy prototypie, to przeprowadzane są konwersje do typów określonych w prototypie.

Podczas kompilacji w trybie języka C++ wywołanie funkcji bez prototypu lub definicji traktowane jest jako błąd. Ponadto, błędy zgłaszane są w następujących przypadkach:

- podano za mało parametrów,
- podano zbyt wiele parametrów,
- występuje niezgodność typu parametrów (mimo konwersji).

Zaleca się dołączanie prototypów funkcji poprzez włączanie odpowiedniego zbioru nagłówkowego dyrektywą *#include*.

Jeżeli podane są prototypy to, definicje o takich samych nagłówkach powinny znajdować się poza funkcją main.

W prototypach zazwyczaj nie podaje się nazw argumentów, a jedynie ich typy. Nazwy podane w prototypach nie muszą być powtórzone w definicji, gdyż i tak sprawdzane są typy danych.
Prototyp: double sum(double x, double y);
Definicja: double sum(double a, double b) { return (a+b); }
Zaleca się zgodność nazw argumentów w prototypach i definicjach.

W języku C++ mogą istnieć funkcje o tych samych nazwach i różnych parametrach zwane *funkcjami przeciążonymi*. Na podstawie typu funkcji i liczby parametrów kompilator musi umieć wybrać właściwą funkcję.

6.3. Przekazywanie parametrów do funkcji

Lista *parametrów formalnych* funkcji podana w nagłówku określa sposób przekazywania parametrów (*argumentów*) podczas wywoływania funkcji.

Zasada ogólna: liczba parametrów aktualnych musi być identyczna z liczbą parametrów formalnych, a ich typy muszą być wzajemnie zgodne.

Parametry mogą być przekazywane przez:

- wartości,
- wskaźniki,
- referencje.

Wskaźnik – zmienna przeznaczona do przechowywania adresu innej zmiennej.

Zmienne referencyjne – reprezentują inne zmienne (aliasy zmiennych); operacje na zmiennej referencyjnej dotyczą zmiennej (i odwrotnie).

```
int a = 5;          // zmienna całkowita
int  *wsk;          // wskaźnik na zmienną całkowitą

int& ra = a;        // ra jest referencją zmiennej a (zawiera to samo co a).
```

W przypadku definicji bez inicjacji wskaźnik globalny (zdefiniowany poza funkcjami) wskazuje na adres zerowy (wsk = NULL) inne wskaźniki zawierają adresy przypadkowe

Wskaźnikowi można przypisać adres obszaru, który fizycznie istnieje wykorzystując operator adresu (&).

```
wsk = &a;          // wskaźnik zawiera adres zmiennej a
```

Aby zmienić lub odczytać zawartość zmiennej, która znajduje się w obszarze wskazywanym przez dany wskaźnik należy wykorzystać operator pobrania (*). W szczególności,

```
 int c =*wsk;          // zmienna c zawiera wartość a
*wsk = 20;             // zmienna a = 20
ra = 10;               // zmienna a = 10;
```

Przekazywanie parametrów przez wartości

W wyniku przekazywania przez wartość funkcja otrzymuje kopię zmiennej na stosie procesora lub w jego rejestrze.
Przekazany argument jest traktowany jak zmienna lokalna funkcji, tzn. wszelkie operacje na argumencie tracą ważność po zakończeniu funkcji.

typ_wartości nazwa_funkcji (typ_1 par1, typ_2 par2, ...)

Przykład 6.2. Przekazywanie parametrów przez wartości

```
void sum(int x, int y);

void plus1(int z);

void main(void)
{  int a, b;

 clrscr();  a = 5;   b = 10;
 cout << "\nWartosc przed wywolaniem funkcji a = " << a;  // a=5
 sum(a,b);                          //funkcja nie zwraca wyniku a = 15
 cout << "\nWartosc po wywolaniu funkcji a = " << a;      // a=5

 int n = 1;
 cout << "\nPrzed wywolaniem n = "<< n;              // n = 1
 plus1(n);                                          // n = 2
 cout << "\nPo wywolaniu n = " << n << endl;        // n = 1

 getch();
}

void sum(int x, int y)
{
   x = x + y;
   cout << "\nLiczba wewnatrz funkcji x = " << x;
}

void plus1(int z)
{
   ++z;
   cout << "\nWartosc wewnatrz funkcji z = " << z;
}
```

Przekazywanie parametrów przez wskaźniki

Jest to odmiana przekazywania parametrów przez wartość.
Funkcja otrzymuje kopie wskaźnika, który może określać adres zmiennej należącej do innej funkcji. Zmiana zawartości obszaru wskazywanego przez parametry wskaźnikowe prowadzi do zmiany wartości zmiennych utworzonych w innych funkcjach programu.

Przekazywanie parametrów przez wskaźniki jest stosowane:

- do przekazywania struktur danych o dużych rozmiarach (np. tablic);
- w celu umożliwienia zwrócenia więcej niż jednej wartości.

typ_wartości nazwa_funkcji (typ_1 *par1, typ_2 *par2, ...)

Podczas wywołania funkcji należy przekazać adresy argumentów:

zmienna = nazwa_funkcji(&arg1, &arg2, ...); // np. scanf("%d", &x);

Przekazywanie parametrów przez referencje

Jest to odpowiednik znanego z Pascala przekazywania parametrów przez zmienne. Operacje są przeprowadzane bezpośrednio na zmiennej, której referencję przekazano do funkcji.

typ_wartości nazwa_funkcji (typ_1 &par1, typ_2 &par2, ...)

Podczas wywołania funkcji należy przekazać argumenty:

zmienna = nazwa_funkcji(arg1, arg2, ...);

Jeżeli typ przekazywanego argumentu nie jest zgodny z typem oczekiwanego parametru referencyjnego, to tworzona jest zmienna tymczasowa, na której wykonywane są operacje wewnątrz funkcji, natomiast przekazany argument pozostanie bez zmian.

```
void kwadrat(double &x)  { x = x*x;  }

void main(void)  {  int a = 5;  double b = 5.0;
kwadrat(a);                cout << a;   // a = 5;
kwadrat( (double) a);    cout << a;   // a = 5;
kwadrat(b);                cout << b;   // b = 25;  }
```

Przykład 6.3. Przekazywanie parametrów przez wskaźniki

```
void sum(int *x, int y);

void plus1(int *z);

void main(void)
{  int a, b;

   clrscr();     a = 5; b = 10;

 cout << "\nWartosc przed wywolaniem funkcji a = " << a;  // a=5
 sum(&a,b);                        //funkcja nie zwraca wyniku a = 15
 cout << "\nWartosc po wywolaniu funkcji a = " << a;      // a=15

 int n = 1;
 cout << "\nPrzed wywolaniem n = "<< n;            // n = 1
 plus1(&n);                                        // n = 2
 cout << "\nPo wywolaniu n = " << n << endl;       // n = 2

 getch();
}

void sum(int *x, int y)
{
    *x = *x + y;
    cout << "\nLiczba wewnatrz funkcji x = " << *x;
}

void plus1(int *z)
{
    ++*z;
    cout << "\nWartosc wewnatrz funkcji z = " << *z;
}
```

Przykład 6.4. Przekazywanie parametrów przez referencje

```
void sum(int &x, int y);

void plus1(int &z);

void main(void)
{  int a, b;

   clrscr();    a = 5; b = 10;

 cout << "\nWartosc przed wywolaniem funkcji a = " << a;  // a=5
 sum(a,b);  //funkcja nie zwraca wyniku; a = 15
 cout << "\nWartosc po wywolaniu funkcji a = " << a;      // a=15

 int n = 1;
 cout << "\nPrzed wywolaniem n = "<< n;            // n = 1
 plus1(n);                                        // n = 2
 cout << "\nPo wywolaniu n = " << n << endl;       // n = 2

 getch();
}

void sum(int &x, int y)
{
    x = x + y;
    cout << "\nLiczba wewnątrz funkcji x = " << x;
}

void plus1(int &z)
{
    ++z;
    cout << "\nWartosc wewnątrz funkcji z = " << z;
}
```

6.4. Zmienne lokalne, zmienne globalne i widoczność

Zmienne lokalne – zmienne zdefiniowane wewnątrz funkcji; są tworzone w momencie wywołania funkcji i znikają po jej zakończeniu; jeżeli są poprzedzone atrybutem *static*, to zachowują wartość pomiędzy kolejnymi wywołaniami funkcji; nazwa i wartość zmiennej lokalnej są znane tylko funkcji, w której została ona zadeklarowana; zmienne lokalne muszą mieć unikalne nazwy; zmienne lokalne *nie są inicjowane* wartościami domyślnymi.

Zmienne globalne – zmienne dostępne w całym programie; muszą być deklarowane na początku programu, na zewnątrz wszystkich funkcji; jest im przypisywana wartość domyślna 0.

Należy unikać korzystania ze zmiennych globalnych. Jeżeli jednak zmienne globalne są używane i dochodzi do konfliktu między nazwą zmiennej globalnej i lokalnej, to priorytet jest dla zmiennej lokalnej.

W przypadku, gdy trzeba skorzystać ze zmiennej globalnej, która ma taką samą nazwę jak zmienna lokalna, to należy odwoływać się do zmiennej globalnej za pomocą operatora *widoczności globalnej* (::).

Przykład 6.5. Operator widoczności globalnej.

```
int liczba = 100;  // zmienna globalna

void pisz(int liczba)
{  static int n = 0;          n++;
    cout << "\nIteracja = " << n;
    cout << "\nZmienna lokalna = " << liczba;
    cout << "\nZmienna globalna = " << ::liczba;
}

void main(void)
{  int a = 20;

 clrscr();
 pisz(a);      // zm. lokalna liczba = 20; | zm. globalna ::liczba = 100; | n=1
 pisz(10);     // iteracja n = 2
 getch();
}
```

6.5. Klasy zmiennych: auto, static, extern, register

Klasa zmiennej określa:

- ***zasięg zmiennej***, tzn. jej widoczność w różnych częściach programu;
- ***łączność zmiennej***, czyli liczbę różnych miejsc, w których tą samą zmienną można zadeklarować;
- ***czas trwania zmiennej*** (okres pozostawania w pamięci).

Zmienna może posiadać jeden z trzech możliwych zasięgów:

- **zasięg plikowy -** zmienna jest widoczna od miejsca jej definicji, aż do końca pliku, w którym została zdefiniowana; np. zasięg plikowy posiadają wszystkie zmienne globalne zdefiniowane na początku pliku źródłowego;
- **zasięg blokowy -** zmienna jest widoczna od miejsca jej definicji do końca bloku zawierającego definicję; zasięg blokowy posiadają wszystkie zmienne zdefiniowane w bloku złożonym z nawiasów { ... }, np. zmienne lokalne i argumenty formalne funkcji;
- **zasięg prototypowy -** dotyczy zmiennych występujących w prototypach funkcji; obejmuje zakres od definicji zmiennej do końca prototypu; w przypadku deklaracji funkcji double dod(double x, double y) nazwy zmiennych nie są istotne dla kompilatora, a jedynie ich typy; wystarczy, więc deklarować prototyp funkcji w postaci double dod(double, double).

Zmienne mogą posiadać jeden z trzech rodzajów łączności:

- **łączność zewnętrzną -** zmienne tej klasy mogą być wykorzystywane w każdym pliku źródłowym należącym do programu;
- **łączność wewnętrzną -** zmienne tej klasy mogą być wykorzystywane tylko w tym pliku, w którym zostały zadeklarowane;
- **brak łączności -** zmienne tej klasy mogą być wykorzystywane tylko wewnątrz bloku, w którym zostały zdefiniowane.

Zmienne o łączności zewnętrznej lub wewnętrznej mogą występować w więcej niż jednej deklaracji, natomiast zmienne pozbawione łączności mogą być deklarowane tylko raz. Na przykład, zmienne o zasięgu blokowym lub prototypowym nie posiadają łączności, natomiast zmienne o zasięgu plikowym mogą posiadać łączność zewnętrzną (extern) lub wewnętrzną (static).

Czas trwania zmiennej może być:

- **statyczny** – zmienna istnieje przez cały czas działania programu (np. zmienne globalne);
- **automatyczny** – zmienne istnieją tylko wówczas, gdy program wykonuje blok, w którym zmienne zostały zdefiniowane (np. zmienne lokalne funkcji).

Zmienne automatyczne (klasa auto)

Zmienne zdefiniowane lokalnie w swoim bloku (zasięg blokowy), np. zmienne lokalne funkcji. Powstają i są inicjowane wartościami początkowymi (o ile są podane) za każdym razem, gdy sterowanie wejdzie do ich bloku. W momencie, gdy następuje koniec realizacji bloku – zmienne znikają (automatyczny czas trwania).

Domyślnie wszystkie zmienne lokalne funkcji są klasy auto. Można w tym przypadku pominąć w deklaracjach zmiennych słowo kluczowe auto. Umieszczenie tego słowa podkreśla zazwyczaj, że zmienna lokalna dubluje deklarację zmiennej zewnętrznej.

Zmienne automatyczne nie posiadają łączności. Nie można zadeklarować dwóch zmiennych o tej samej nazwie w tym samym bloku. Zmienne automatyczne o tych samych nazwach, zdefiniowane w różnych blokach, są różnymi obiektami. Deklaracja zmiennej wewnątrz bloku przesłania jej deklarację zewnętrzną, także globalną.

Domyślnie zmienne automatyczne nie otrzymują żadnej wartości początkowej. Nie można zakładać, że wartość początkowa zmiennej automatycznej jest równa 0.

```
int y = 21; // definicja zmiennej globalnej;

void main()
{
  int x;                  // deklaracja zmiennej klasy auto
  auto int y = 5;         // definicja  zmiennej klasy auto
                          // przykrywa def. zmiennej globalnej
  cout << x << endl;      // wartość x przypadkowa
  cout << y << endl;      // y = 5
  cout << ::y << endl;    // ::y = 21 (operator :: tylko dla C++)
     {
          int y = 11;             // definicja lokalna
          cout << y << endl;      // y = 11
     }
          cout << ++y << endl;    // y = 6
}
```

Zmienne zewnętrzne (klasa extern)

Zmienne zdefiniowane poza funkcjami (tylko to decyduje, że zmienne są zewnętrzne, a nie specyfikator extern). Istnieją i zachowują swoje wartości podczas wykonywania całego programu (statyczny czas trwania). Powstają i są inicjowane wartościami początkowymi raz przed rozpoczęciem wykonywania programu. W przypadku braku jawnej inicjalizacji otrzymują wartość początkową 0. Inicjacja zmiennych zewnętrznych może nastąpić tylko raz w miejscu ich definicji. Zmienne te mogą być wykorzystywane do komunikacji między funkcjami należącymi do różnych plików programu (łączność zewnętrzna).

Deklaracja zmiennej ze słowem kluczowym extern, np. extern int w, nie powoduje zarezerwowania w pamięci miejsca dla zmiennej. Informuje ona kompilator, że definicja zmiennej zewnętrznej znajduje się w innym miejscu programu, być może w innym pliku. Jeżeli zmienna jest rzeczywiście zdefiniowana w innym pliku lub po definicji funkcji, w której jest wykorzystywana, to użycie specyfikatora extern jest obowiązkowe. W pozostałych przypadkach może być pominięte.

Deklaracja extern x jest równoważna deklarcji extern int x.

```
// **********   modul_1.cpp  ************

      int y = 5;

      void main()
      {
            extern int a;       // deklaracja zmiennej zewnętrznej
            extern double z;    // deklaracja zmiennej zewnętrznej

            cout << y << endl;      // y = 5
            cout << a << endl;      // a = 0
            cout << z << endl;      // z =145.2
      }
            int a;          // definicja zmiennej int a = 0

// ***********  modul_2.cpp  *************

      double z = 145.2;       // definicja zmiennej
```

Zmienne statyczne (klasa static)

Zmienne o zasięgu blokowym (tak jak zmienne automatyczne), ale w odróżnieniu od zmiennych automatycznych istniejące i zachowujące swoje wartości przez cały czas wykonywania programu (statyczny czas trwania). Zmienne statyczne są inicjowane wartościami początkowymi tylko raz w miejscu ich definicji. W przypadku, gdy sterowanie opuści blok, w którym zmienna jest zdefiniowana aktualna wartość zmiennej zostanie zachowana. W przypadku, gdy zmienne statyczne nie są inicjowane żadnymi wartościami, podobnie jak zmienne zewnętrzne, otrzymują domyślnie wartość 0.

Zmienne statyczne można definiować lokalnie wewnątrz funkcji. Są one wówczas widoczne tylko wewnątrz tych funkcji. Można również zdefiniować zmienną statyczną zewnętrzną. Zmienna tego typu, w odróżnieniu od zwykłej zmiennej zewnętrznej, jest widoczna tylko przez funkcje znajdujące się w tym samym pliku, co jej definicja.

```
static int plikowa;        // zmienna statyczna zewnętrzna

void pisz(double  x )
{
      static int licznik;  //  zmienna statyczna licznik = 0

      cout << x << endl;

      licznik++;           // zwiększ licznik wywołań funkcji
}
```

Zmienne statyczne i zewnętrzne mają zarezerwowane miejsce w kodzie wykonywalnym programu (exe) i są ładowane do pamięci razem z programem.

Zmienne rejestrowe (klasa register)

Zmienne automatyczne mogą być umieszczane w szybkich rejestrach procesora, co przyspiesza obliczenia. W środowisku BC++ możliwość taka istnieje tylko w przypadku zmiennych typu char, int lub zmiennych wskaźnikowych.

Kompilator umieszcza zmienne w rejestrach w miarę ich dostępności.

W odniesieniu do zmiennych rejestrowych nie można stosować operatora adresu &.

```
int sum(register int a, register int b) {  return (a + b); }

void main()
{
      register int  fast;        // deklaracja zmiennej rejestrowej
               int  slow;        // deklaracja  zmiennej automatycznej
}
```

Podsumowanie

Klasa	Słowo kluczowe	Czas	Zasięg i łączność
Zmienne zadeklarowane wewnątrz funkcji			
automatyczna	auto	chwilowy	lokalne
rejestrowa	register	chwilowy	lokalne
statyczna	static	ciągły	lokalne
Zmienne zdefiniowane poza funkcjami			
zewnętrzna	extern (przy ponownych deklaracjach)	ciągły	globalne (wszystkie pliki)
zewnętrzna statyczna	static	ciągły	globalne (jeden plik)

6.6. Kwalifikatory zmiennych: const, volatile

Zmienne definiowane przy pomocy słowa kluczowego *const* nie mogą ulegać zmianie podczas działania programu. Zmienne te muszą być inicjowane wartościami w momencie ich definicji. Najczęściej są one wykorzystywane do przechowywania stałych parametrów programu o ustalonym typie. Często ich identyfikatory są pisane dużymi literami.

```
const int TEMP = 100;
const double PI = 3.14;  // istnieje stała stand. M_PI
const N = 11;
```

Kwalifikator *volatile* jest wykorzystywany w deklaracjach zmiennych, które mogą zmieniać swoją zawartość pod wpływem czynników innych niż sam program, np. urządzeń zewnętrznych.

```
volatile unsigned char status_urz;

//  stan zmiennej status_urz może być zmieniany przez czynniki
//  zewnętrzne pomiędzy jej kolejnymi wystąpieniami w programie
```

6.7. Zasięg funkcji

Pojęcie klas dotyczy również funkcji. Domyślnie dla każdej funkcji przyjmowana jest klasa zewnętrzna. Funkcja zewnętrzna jest dostępna dla wszystkich funkcji znajdujących się we wszystkich plikach programu.

Istnieje również możliwość zadeklarowania funkcji jako statycznej. Może ona być wówczas wykorzystywana tylko w obrębie pliku, który zawiera jej definicję. Inny plik może zawierać osobną funkcję o tej samej nazwie. Klasa statyczna umożliwia tworzenie funkcji prywatnych dla danego modułu.

```
void sun(int x, int y)          // funkcja zewnętrzna – domyślnie
{  return x + y;  }

extern void pisz(int n)         // funkcja zewnętrzna
{ cout << n << endl; }

static double oblicz(double x)      // funkcja prywatna modułu
{ return  pow(x,3); }
```

6.8. Funkcje z atrybutem *inline*

Wywołanie funkcji wiąże się z umieszczeniem argumentów na stosie oraz wykonaniem skoku pod odpowiedni adres. W przypadku funkcji o niewielkich rozmiarach czas związany z jej wywołaniem może być porównywalny z czasem wykonania funkcji.

Jeżeli chcemy, aby kod funkcji był umieszczany w miejscu jej wywołania, to należy poprzedzić definicję funkcji atrybutem *inline*. Pozwala to zwiększyć szybkość odwołań do funkcji.

Funkcje z atrybutem inline nie powinny zawierać słów kluczowych: **do, while, for, goto, switch, case, break, continue**. Wystąpienie jednego z wymienionych słów spowoduje, że kompilator potraktuje funkcję jako zwykłą funkcję statyczną.

Przykład 6.6. Atrybut inline.

```
inline double pot(double x)
{  return x*x*x; }

inline int max(int a, int b)
{ return (a > b ? a : b); }

void main(void)
{   int a = 2;  int b = 7;

 clrscr();
 cout << "\na^3 = " << pot(a);        //  a^3 = 8
 cout << "\nMax = " << max(9, max(a,b));       // max = 9
 getch();
}
```

W przypadku wywołania max(9, max(a,b)) funkcja max zostanie dwukrotnie włączona do kodu wynikowego.

6.9. Funkcje rekurencyjne

Rekurencyjne wywołanie funkcji polega na ponownym jej wywołaniu jeszcze przed jej zakończeniem. Przy każdym wywołaniu rekurencyjnym tworzona jest kopia środowiska funkcji łącznie ze wszystkimi parametrami i zmiennymi lokalnymi.

Wywołanie każdej funkcji powoduje umieszczenie na stosie:

- informacji umożliwiającej powrót z funkcji do miejsca wywołania,
- parametrów funkcji,
- zmiennych lokalnych (automatycznych) istniejących w momencie wywołania funkcji.

Ze względu na ograniczoną pojemność stosu funkcje rekurencyjne nie powinny posiadać zbyt dużej liczby parametrów i zmiennych lokalnych ani też zmiennych o dużych rozmiarach.

Sposób działania rekurencji ilustrują następujące przykłady.

Przykład 6.7. Funkcje rekurencyjne.

```
void rek(void)
{
  static int m = 0;
  m++;
  if (m < 3) rek();
  cout << "\nKopia nr : " << m;
  m--;
}
```

W przypadku funkcji rek() zmienna m jest zwiększana o jeden, a następnie jeśli m < 3 funkcja wywołuje sama siebie. Zmienna m jest statyczna dlatego zachowuje wartość pomiędzy kolejnymi wywołaniami funkcji. Jeśli m = 3 rozpocznie się proces powrotów z kolejnych kopii funkcji. Dla każdej kopii wykonywane są operacje: wyprowadzenie m na ekran, zmniejszenie m, przekazanie sterowania do kolejnej kopii.

```
void pisz(int n)
{ // pisz znakowo liczbe dziesietna
     int c;
     c = n / 10;   if (c) pisz(c);     // jeśli c != 0 ponowne wywołanie
     printf("%c-", n % 10 + '0');
 }

double silnia(unsigned n)
{    // n! = n*(n-1)!
      if (n>1) return (double) n * silnia(n-1);
      else return 1.0;
}

double fib(unsigned n)
{  // f(1) = f(2) = 1;  f(n) = f(n-1) + f(n-2)
   double f1, f2;

   if (n<3) return 1.0;
   else
   {   f1 = fib(n-1); f2 = fib(n-2);
       return f1+f2;
   }
}
```

```
void main(void)
{ double w;

  clrscr();
  rek();    // pojawia sie kolejno
            // Kopia nr : 3
            // Kopia nr : 2
            // Kopia nr : 1
  cout << endl;

  pisz(134);  cout <<" "<< 134;  // 1-3-4- 134

  w = silnia(5); cout << "\nWartosc silni = "<< w;              // 120

  w = fib(6);    cout << "\nWartosc fib = " << w;               // 8

  getch();
}
```

6.10. Funkcje przeciążone

Przeciążanie funkcji pozwala zdefiniować w programie rodzinę funkcji o tej samej nazwie, lecz różniących się parametrami (liczbą i /lub typem parametrów).

Funkcje przeciążone nie muszą zwracać wartości tego samego typu, ale dwie funkcje nie mogą się różnić jedynie typem zwracanego wyniku.

Wybór funkcji z rodziny funkcji przeciążonych jest dokonywany na podstawie analizy parametrów wywołania. Poszukiwana jest funkcja przeciążona o typach i liczbie parametrów zgodnych z wywołaniem. Jeżeli funkcja taka nie zostanie znaleziona, to dokonywane są standardowe konwersje typów i poszukiwanie jest kontynuowane.

Przeciążanie funkcji upraszcza programowanie, gdyż pozwala programiście używać jednej nazwy funkcji do konkretnego zadania.

Przykład 6.8. Funkcje przeciążone.

```
void pisz(int a, int b)
{   printf("\n%5d %5d", a, b);  }

void pisz(float a, float b)
{  printf("\n%5.2f %5.2f", a, b); }

void pisz(char a)
{  printf("\n%c", a); }

double potega(double podst, double wyk)
{  return (podst > 0) ? exp(wyk*log(podst)) : -1E+20;  }

long potega (int podst, int wyk)
{   long w=1;
    if (podst > 0 && wyk > 0)  for (int i = 1; i <= wyk; i++) w = w * podst;
    else w = -1;
    return w;
}

void main(void)
{
   int a = 5;   int b = 10;
  float x = 1.2f;  float y = 20.0f;
  char z = 'A';

  clrscr();
  pisz(a,b);    pisz(x,y);     pisz(z);
  cout << endl;
  cout << potega(2.0,0.5) << endl;        // 1.414214
  cout << potega(2,3) << endl;              // 8
  getch();
}
```

6.11. Domyślne wartości parametrów funkcji

W języku C liczba argumentów w wywołaniu funkcji musi zgadzać się z liczbą jej parametrów formalnych. W języku C++ istnieje możliwość nadawania parametrom wartości domyślnych. Czynimy to przypisując parametrom wartości w prototypie funkcji lub w nagłówku definicji funkcji (jeśli brak prototypu). Wówczas, funkcję taką można wywoływać z mniejszą liczbą argumentów niż podano na liście parametrów formalnych.

Np. prototyp:

```
int sum(int a,  int b = 1, int c = 2);
```

Przykładowe wywołania: x = sum(2); x = sum(1,3,5);

Własności:

- parametry bez domniemań muszą poprzedzać wszystkie parametry z domniemaniami;
- jeżeli pewne parametry zostaną pominięte przy wywołaniu funkcji, to przyjęte zostaną dla nich wartości domyślne;
- wartości parametrów podane w wywołaniu funkcji „przebijają" wartości domniemane;
- jeżeli w wywołaniu funkcji pominięty zostanie pewien parametr, to muszą zostać pominięte wszystkie parametry znajdujące się za nim;
- wartości domniemane można podać tylko raz (w jednej deklaracji).

W przypadku funkcji przeciążonych może istnieć dwuznaczność spowodowana określeniem domyślnych wartości elementów.

Np.

```
int  fun(int a, float x =10.0);

int fun(int a);
```

Takie prototypy zostaną skompilowane, ale jeśli kompilator spotka wywołanie funkcji fun z parametrem całkowitym, np. fun(5), to nie będzie w stanie stwierdzić, która z funkcji przeciążonych ma zostać wywołana. Wówczas pojawi się błąd kompilatora.

Przykład 6.9. Argumenty domniemane.

```
long sum(int a, int b, int c=1, int d=2);

double sum(double a=3.0, double b=4.0);

void main(void)
{
  clrscr();
```

```
  cout << sum(1,2,3,4) << endl;         // 10
  cout << sum(1,2) << endl;      // 6

  cout << sum(1.0,2.0) << endl;    // 3
  cout << sum(1.1) << endl;      // 5.1

  getch();
}

long sum(int a, int b, int c, int d)
{ return(a+b+c+d); }

double sum(double a, double b)
{  return a + b; }
```

6.12. Funkcje o zmiennej liczbie parametrów

Język C++ umożliwia deklarowanie i definiowanie funkcji o zmiennej liczbie parametrów. Deklaracja takiej funkcji ma postać:

<typ_wartości> *nazwa_funkcji* (parametry_ustalone, ...);

Trzy kropki oznaczają zmienną liczbę parametrów.

Dostęp do parametrów nieustalonych jest realizowany następująco:

- deklaracja zmiennej, np. *pocz* typu *va_list* (*va_list pocz*); zmienna wspomnianego typu zawierać będzie adres początku listy zmiennych parametrów;
- wywołanie makrorozkazu *va_start(pocz, c)*, gdzie c oznacza nazwę ostatniego ustalonego parametru;
- pobranie argumentu za pomocą makrorozkazu *va_arg*, np. zmienna = *va_arg(pocz, typ_arg)*; przyjmuje się, że argument jest typu typ_arg; kolejne wywołania makrorozkazu będą powodowały pobieranie kolejnych parametrów funkcji;
- wywołanie makrorozkazu *va_end(pocz)*; rozkaz powinien być wywoływany tylko po przeczytaniu wszystkich argumentów; jego wywołanie pozwoli na normalne zakończenie funkcji.

Liczbę zmiennych parametrów przekazuje się do funkcji za pomocą oddzielnego parametru ustalonego. Po wyczerpaniu listy parametrów nieustalonych można uzyskać ponownie dostęp do początku listy poprzez wywołanie *va_start*.

Należy pamiętać, aby typ pobieranego argumentu zgadzał się z typem podanym w makrorozkazie *va_arg*.

Z powodu automatycznych konwersji nie można używać w wywołaniach *va_arg* typów **char**, **unsigned char** i **float**.

Korzystanie z makrorozkazów *va_start*, *va_arg* i innych wymaga dołączenia pliku nagłówkowego *stdarg.h*.

Przykład 6.10. Funkcje o zmiennej liczbie parametrów.

```
double obl_wiel(double x, int n, ...)
// obliczanie wartosci wielomianu stopnia n
{  double sum = 0.0;    double wsp;

   va_list pocz;              // początek obszaru zmiennych argumentów
   va_start(pocz,n);       // adres ostatniego ustalonego argumentu

   while (n)
   {   wsp = va_arg(pocz, double);
       sum += wsp * pow(x,n);         // a * x^n
       n--;
   }
        sum += va_arg(pocz, double);    //  sum = sum + a0

   va_end(pocz);

   return sum;
}
```

```
float srednia(int n, ...)
{
   int k = n;    float sum = 0.0f;

   va_list pocz;          // początek obszaru zmiennych argumentów
   va_start(pocz,n);       // adres ostatniego ustalonego argumentu

   while (k--) sum += va_arg(pocz, int);
   va_end(pocz);
   return sum /n;
}

void main(void)
{ double wyn;

  clrscr();     wyn = obl_wiel(-2.0, 3,  1.0, 1.0, -1.0, 3.0);
  // x = -2;   f(x) = x^3 + x^2 - x + 3; f(-2) = 1
  cout << "\nWartosc wielomianu = " << wyn;

  wyn = srednia(4, 1,2,3,4);     cout << "\nSrednia = " << wyn;
  getch();
}
```

6.13. Parametry funkcji main

W języku C istnieje możliwość przekazywania argumentów do uruchamianego programu za pomocą wiersza poleceń.
Przekazywane argumenty są kojarzone z parametrami wywołania funkcji main, która może występować w następujących postaciach.

- int main(int argc, char *argv[]);

- funkcja dwuparametrowa, zwracająca wynik typu int;
- argc – liczba argumentów wywołania funkcji zwiększona o 1 (argumenty + argument uwzględniający nazwę programu);
- argv[ ] – tablica wskaźników do tekstów opisujących argumenty; przyjmuje się, że argv[0] jest wskaźnikiem (adresem) do tekstu opisującego pełną nazwę programu (uwzględniającą ścieżkę dostępu); natomiast argv[1], ..., argv[argc-1] są wskaźnikami do teksów odpowiadających argumentom wywołania funkcji;

W szczególności wywołanie programu lista.exe 1 2 3, znajdującego się na dysku C: , spowoduje następujące skojarzenia argumentów z parametrami funkcji main:
argc = 4,
argv[0] – wskaźnik na ”C:\\lista.exe”,
argv[1] – wskaźnik na ”1”,
argv[2] – wskaźnik na ”2”,
argv[3] – wskaźnik na ”3”.

- void main(int argc, char *argv[]);

  - funkcja dwuparametrowa, nie zwracająca wyniku;

- int main(void);

  - funkcja bezparametrowa, zwracająca wynik typu int;

- void main(void);

  - funkcja bezparametrowa, nie zwracająca wyniku;

Przykład 6.11. Przekazywanie parametrów za pomocą funkcji main.

```
/*
 Przekazywanie parametrów za pomocą funkcji (obliczanie sumy arg.)

main(int argc, char *argv[])

 argc - liczba argumentów programu + 1;

 char *argv[] - tablica wskaźników na łańcuchy opisujące kolejne
                argumenty.

Konwersja liczby zapisanej w postaci tekstu do typu double - funkcja atof

Ostatnie wystąpienie znaku c w łańcuchu s - funkcja strrchr(s, c)
*/
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
```

```
int main(int argc, char *argv[])
{
  double pom = 0, suma = 0;  int i = 0;

  clrscr();
  if (argc < 2) {
  printf("Wywołanie programu: %s argumenty oddzielone
          spacja\n",argv[0]);   exit(0);   // argv[0] – nazwa programu
    }          // w nazwie ścieżki znak \  jest zapisywany jako '\\'
  if (argc > 1) {
    printf("Ścieżka programu: %s\n", argv[0]);
    printf("Nazwa właściwa programu: %s\n", strrchr(argv[0],'\\')+1 );
    printf("Liczba argumentów programu: %d\n", argc-1);
    for (i=1; i < argc; i++) {
    printf("Arg. %d = %9s Liczba zn. %d\n", i, argv[i], strlen(argv[i]));
    pom = atof(argv[i]);  // konwersja łańcucha do liczby double
    if (pom != HUGE_VAL) suma+=pom; //  spraw. czy konwersja jest
                                    //  prawidłowa?
    else exit(1);  }
       }
  printf("Suma argumentów = %lg\n", suma);
  getch();     return 0;
}
```

Wyniki działania programu uruchomionego z kartoteki D:\TEMP jako

P_MAIN.EXE 1 2 4 5

Ścieżka programu: D:\TEMP\P_MAIN.EXE

Nazwa właściwa programu: P_MAIN.EXE

Liczba argumentów programu: 4

Arg. 1 = 1 Liczba zn. 1
Arg. 2 = 2 Liczba zn. 1
Arg. 3 = 4 Liczba zn. 1
Arg. 4 = 5 Liczba zn. 1

Suma argumentów = 12