

ТЕМА: Функції, визначені програмою.

Концепція програмованих функцій

Складена програма часто має потребу багатократно виконати однакову послідовність інструкцій в різні моменти своєї роботи. Наприклад, задано три різні матриці, в кожній треба порахувати кількість нульових елементів. Зрозуміло, що алгоритм обчислення для однієї матриці буде таким самим і для іншої. Отже, треба три рази повторити алгоритм, лише для різних матриць. Програмний код буде відрізнятись лише іменами матриць.

Тому виникає ідея: однаковий програмний код узагальнити, визначити однократно і окремо від тексту головної функції і використовувати за потреби під час виконання програми в різних місцях.

Реалізація такої ідеї є головним інструментом структуризації, який зменшує розмір програмного коду, а заодно полегшує налагодження цілої програми. Такий підхід називають процедурною декомпозицією. Він полягає у визначенні власних функцій програми.

Щоб використати функцію в мові C++, необхідно виконати два кроки:

- 1) визначити функцію;
- 2) викликати функцію.



Загальний вигляд функції

Загальне визначення функції є таке:

```
тип_результату ім'я_функції ( список_параметрів )  
{  
    оператори тіла функції ;  
    return результат ;  
}
```

Виклик функції виконують у виразах і операторах шляхом використання імені:

```
#include <iostream>  
using namespace std;  
/* простий приклад визначення і виклику  
int simple() // визначення функції  
{  
    cout << "Congratulation !\n"; // 2  
    return 1;  
}  
int main()  
{  
    cout << "Hello, simple !\n"; // 1  
    simple(); // виклик функції  
    cout << "Good-bye !\n"; // 3  
    system("pause"); return 0;  
}
```

1,2,3 –
порядок
друкування

Деякі правила і особливості:

- 1) функція має бути визначена за текстом перед місцем виклику;
- 2) списку параметрів може не бути, але круглі дужки обов'язкові завжди – це є обов'язковий атрибут будь-якої функції C++;
- 3) результатом функції вважається вираз, записаний в операторі return; його можна використати або не використати у викликаючій функції.

Список параметрів – це аргументи функції. Кожний аргумент має бути оголошений окремо своїм типом.

Приклад: обрати найменше з трьох чисел:

```
int min3(float a, float b, float c) // визначення функції
{
    if (a<b && a<c) return a;
    else if (b<c) return b;
    else return c;
}
```

Список параметрів не можна записати скорочено, подібно до визначення змінних:
`int min3(float a,b,c)` - неправильно

Передача параметрів за значенням

В різних алгоритмічних мовах є два основні способи передачі аргументів до функції: передача параметрів за значенням (call by value); передача параметрів за посиланням (call by reference).

Передача параметрів за значенням відбувається шляхом присвоєння формальному параметру копії значення аргумента. Всі зміни параметра функції не впливають на аргумент. *Приклад: надрукувати задану літеру задану кількість разів.*

```
int kchars(char c, int k)
{
    while ( k-- > 0 )    cout << c ;
    cout << endl ;
    c = '#';
    // контроль параметрів
    cout << c << '\t' << k << endl;
    return 0;
}

int main()
{
    char let='A';  int cnt=15;
    kchars(let,cnt);
    cout << let << '\t' << cnt << endl; // без змін
    system("pause");  return 0;
}
```

Результати:

AAAAAAAAAAAAAAAAA	
#	-1
A	15

Передача параметрів за посиланням

За замовчуванням в мові C++ параметри функцій передають за значенням. Проте, їх можна передавати за посиланням. Для цього замість самого аргумента передають вказівник на нього. Функція отримує адресу аргумента, тобто, фактичного параметра, і має можливість змінити його значення. [Див. раніше розглянуте питання «Вказівники. Посилання на існуючі об'єкти»]

Приклад. Задано дійсне число, додатне або від'ємне. За допомогою функції отримати корінь числа (додатній чи від'ємний).

Є два способи передачі за посиланням.

1) Пряме використання вказівників:

```
void mysqrt(float * x)  // параметр функції - вказівник
{
    *x = (*x >= 0) ? sqrt(*x) : -sqrt(abs(*x)) ;
    // доступ оператором *
}
int  main()
{
    float test = -6.82;
    mysqrt(&test);  // адреса аргумента
    cout << test << endl;  // результат   -2.61151
    system("pause");  return 0;
}
```

2) Неявне використання вказівників (автоматичне розіменування):

```
void mysqrt(float &x)  // параметр функції - неявний вказівник
{
    x = (x >= 0) ? sqrt(x) : -sqrt(abs(x)) ;
    // доступ за неявним вказівником x
}
int main()
{
    float test = -6.82;
    mysqrt(test);  // аргумент передають безпосередньо
    cout << test << endl;    // -2.61151
    system("pause");
    return 0;
}
```

Функції типу void

Функція, оголошена специфікатором `void`, як це було в останньому прикладі, не повертає ніяких значень і не має оператора `return`. Тому їх не можна використати у виразах. Такі функції зазвичай опрацьовують аргументи на місці через вказівники і нових значень не будують.

Функції і структури [факультативно]

Структури можна передавати функціям за значенням і за посиланням.

```
#include <iostream>
#define _USE_MATH_DEFINES
    // ця директива #define потрібна перед #include <math.h>
#include <math.h>    // математична бібліотека
using namespace std;

struct polar // полярні координати точки
{
    double distance ; // відстань
    double angle ;    // кут напрямку в радіанах
} ;

// надрукувати полярні координати, подавши кут в градусах
void show_polar ( polar t )
{
    const double Rad_to_deg = 57.29577951 ; // ~ 180/pi
    cout << "distance = " << t.distance ;
    cout << ", angle = " << t.angle * Rad_to_deg ;
    cout << " degrees\n" ;
}
```

```
int main()
{
    polar an1 = { 16.27, M_PI / 6 }; // M_PI - константа пі
    // передача структури an1 за значенням:
    show_polar(an1); // distance = 16.27, angle = 30 degrees
    system("pause");
    return 0;
}
```

Щоб передати структуру за посиланням, достатньо визначити заголовок інакше:

```
void show_polar ( polar &t ) // лише знак адресування &
```

Все решту залишається без змін, проте, функція може змінити аргумент.

Масиви як параметри функцій

Масиви в ролі параметрів функцій є винятком загального правила, коли всі параметри передають за значенням. Якщо аргументом функції є масив, тоді функції передають адресу масиву. Отже, функція має доступ через вказівник до всіх його елементів і може їх змінити.

Приклад 1. У текстовому рядку замінити задану букву пропуском.

```
void lettersub(char * string, char c)
// у рядку string замінити всі літери 'c' пропуском
{
    int i=0;
    while ( string[i] )
    { if(string[i] == c) string[i]=' ';
      i++;
    }
}
```

Виклик функції може виглядати так:

```
char sentence[60] = "abcdefgh12345eetyureee45tgeoeoi";
lettersub(sentence, 'e');
```

Аналогічні можливості є щодо всіх типів елементів масивів, зокрема, для числових.

Приклад 2. Вибіркове сумування елементів масиву.

```
int sum ( int * arr, int n ) // вказівник на масив і розмір
{
    int total = 0 ;
    for ( int i=0; i<n; i++ ) total += arr[i] ;
    return total ;
}
int main()
{
    int test[10] = { 3, 6, 9, 2, 5, 8, 1, 4, 7, 20 } ;
    int s1 = sum ( test, 4 ); // сума перших чотирьох: 20
    int s2 = sum ( test+6, 4 ) ; // сума останніх чотирьох: 32
    // . . . . .
}
```

test – адреса найпершого елемента, тобто, цілого масиву

test+6 – адреса 7-го елемента (рахуючи від 1)

Повернення результатів функцій

Всі функції повертають деякий результат, за винятком тих, які оголошені як `void`.

Функція, оголошена за типом результату, може бути використана як елемент виразу. Наприклад, для визначеної функції `sum()`, можна будувати подібний вираз:

```
int select = sum(test,2) + sum(test+4,2) + sum(test+8,2);
```

Тип результату функції має співпадати з типом виразу `return`. Наприклад, у функції `sum()` не можна оголосити тип величини `total` як `int*`. Аналогічно – для типу результату.

Невідповідність типу результату до типу виразу `return` можлива лише в межах автоматичного перетворення типів, які допускає компілятор.



Повернення вказівників

Результатом функції може бути вказівник. Це може бути вказівник на деяку існуючу величину, або вказівник на нову побудовану величину за виконанням функції.

Приклад. Функція будує текстовий рядок, який складається з однакових літер.

```
char * nletters ( char c, int n )  // утворити рядок з n літер c
{
    char * p = new char[n+1] ;  // надати пам'ять рядку
    p[n] = '\0' ;  // кінець рядка
    while ( n-- > 0 )  p[n] = c ;
    return p ;  // повернути вказівник типу char*
}

//. . . . .
char *s1 = nletters('*',12); // зберегти вказівник на новий рядок
char *s2 = nletters('h',8);
//. . . . .
delete [] s1;  delete [] s2; // не забути звільнити пам'ять
                        // (зовні від nletters)
```


Прототипи функцій

В мові C++ всі функції мають бути оголошені до першого виклику за текстом програми. Для полегшення запису комплексу взаємодіючих функцій і можливості будови різних технологій і стилів проектування використовують оголошення прототипів функцій. Прототип – це копія лише заголовку функції зі знаком «крапка з комою» в кінці, з деякими можливими спрощеннями. Стандарт мови C++ рекомендує використовувати прототипи.

```
#include <iostream>
using namespace std;
// 1) спочатку треба записати прототипи
int sum ( int * arr, int n ) ; // вказівник на масив і розмір
char * nletters ( char c, int n ); // утворити рядок з n літер c
void mysqrt(float &x) ;

// 2) головна функція
int main()
{
    // ..... виклики функцій в довільному порядку
    cout << "test prototype function" << endl;
    system("pause");
    return 0;
}
```

```
// 3) повне визначення функцій - окремо
int sum ( int * arr, int n )
{ // .....
  return 1;
}
char * nletters ( char c, int n )
{ // .....
  return new char[32];
}
void mysqrt(float &x)
{ // .....
}
```

Оголошення прототипів вирішує проблему довільної перехресної взаємодії функцій. 

В прототипах вказувати імена параметрів не обов'язково:

```
int sum ( int *, int ) ;
char * nletters ( char, int ) ;
void mysqrt(float &) ;
```

Якщо функцію визначити повністю до першого виклику, як це було в попередніх прикладах, тоді окремих прототипів не потрібно.



Принцип локалізації

Код функції є закритим і недоступним для будь-яких операторів, записаних в інших функціях, - крім імені самої функції.

Отже:

- 1) всі змінні, визначені всередині функції, є локальними і відомими лише всередині функції, звернутись до них зовні не можна;
- 2) локальні змінні існують лише протягом часу виконання функції;
- 3) імена формальних параметрів так само є локальними, безпосередньо звернутись до них не можна, а лише підставити фактичні параметри;
- 4) перейти до функції можна лише через заголовок, до певного оператора функції — не можна.

