**Лекція № 7. Використання функцій. Практичні поради щодо написання програм. Принципи структурного програмування.**

**Визначення функції**

**Функція** — це іменована логічно завершена сукупність оголошень і операторів, призначених для виконання певної задачі

Програма мовою С++ містить одну або декілька функцій, кожна з яких повинна бути оголошена та визначена до її першого використання. **Оголошення функці**ї (прототип, заголовок) задає ім’я функції, тип значення, що повертає функція (якщо воно є), а також імена та типи аргументів, які можуть передаватися як у функцію, так і з неї. **Визначення функції** — це завдання способу виконання операцій.

Серед функцій програми повинна бути одна з ім’ям **main (головна функція)**, яка може знаходитися в будь-якому місці програми. Ця функція виконується завжди першою і закінчується останньою.

**Структура функції**

Усі функції мають однакову структуру визначення у вигляді:

**[тип результату] ім'я функції ([список формальних аргументів])**

**{ // *тіло функції***

**опис даних;**

**оператори;**

**[return] [вираз]; };**

**Тип результату** — будь-який базовий або раніше описаний тип значення (за винятком масиву і функції), що повертається функцією (необов’язковий параметр). За відсутності цього параметра тип результату за замовчуванням буде цілий (**int**). Він також може бути описаний ключовим словом (**void**), тоді функція не повертає ніякого значення. Якщо результат повертається функцією, то в тілі функції є необхідним оператор **return вираз**;, де **вираз** формує значення, що співпадає з типом результату;

**ім’я функції** — ідентифікатор функції, за яким завжди знаходиться пара круглих дужок «**( )**», де записуються **формальні** аргументи. Фактично **ім’я функції** — це особливий вид покажчика на функцію, його значенням є адреса початку входу у функцію;

**список формальних аргументів** — визначає кількість, тип і порядок проходження переданих у функцію вхідних аргументів, які розділяються комою (**«,»**). У випадку, коли параметри відсутні, дужки залишаються порожніми або містять ключове слово (**void**). Формальні параметри функції локалізовані в ній і недоступні для будь-яких інших функцій.

Список формальних аргументів має такий вигляд:

**([const] тип 1 [параметр 1], [const] тип 2 [параметр 2], . . .)**

У списку формальних аргументів для кожного параметра треба вказати його тип **(не можна групувати параметри одного типу, вказавши їх тип один раз).**

Тіло функції може складатися з описів **змінних** і **операторів**. **Змінні**, що використовуються при виконанні функції, можуть бути **глобальні** і **локальні**. Змінні, що описані (визначені) за межами функції, називають **глобальними**. За допомогою глобальних параметрів можна передавати дані у функцію, не включаючи ці змінні до складу формальних параметрів. У тілі функції їх можна змінювати і потім отримані значення пере­давати в інші функції.

**Змінні,** що описані у тілі функції, називаються **локальними** або **автоматичними**. Вони існують тільки під час роботи функції, а після реалізації функції система видаляє локальні змінні і звільняє пам’ять. Тобто між викликами функції вміст локальних змінних знищується, тому ініціювання локальних змінних треба робити щоразу під час виклику функції. За необхідності збереження цих значень, їх треба описати як **статичні** за допомогою службового слова **static**, наприклад:

**static int**

**х, у;**

**або static float р = 3.25;.**

***Статична змінна схожа на глобальну,* але діє тільки у тій функції, в якій вона оголошена.**

На початку програми можна не описувати всю функцію, а записати тільки **прототип.** Запис прототипу може містити тільки перелік типів формальних параметрів без імен, а наприкінці прототипу завжди ставиться символ **«;»**, тоді як у описі (ви­значенні) функції цей символ після заголовка не присутній.

**Узагальнимо.**

*Визначення (definition) функції* включає наступні елементи (згідно затвердженому ANSI-стандарту мови С) :

1. вказівку про можливість використання функції поза даним файлом (за допомогою одного з службових слів static або extern; останнє вживається за замовчуванням);
2. тип результату, який повертає функція; відсутність результату позначається службовим словом void;
3. ідентифікатор (ім’я) функції;
4. кількість та типи *формальних параметрів* в круглих дужках; в разі їх відсутності вживається службове слово void;
5. важливо: всі параметри функції передаються **за значенням**
6. тіло функції, яке взяте в фігурні дужки; складається з визначення внутрішніх (локальних) змінних функції та інструкцій, що мають виконуватись під час роботи функції. Якщо функція повертає результат, то серед цих інструкції обов’язково має бути інструкція виду: return <вираз>;

причому тип виразу повинен збігатись з типом результату функції.

**Огляд процесу виконання виклику функції**

1. Виділяється пам'ять для точки повернення й параметрів функції. Посилання на точку повернення з функції запам'ятовується.

2. Обчислюються значення аргументів для параметрів значень, посилання на пам'ять аргументів для параметрів посилань. Відбувається підстановка аргументів.

3. Виділяється пам'ять, відповідна до локальних імен змінних (окрім локальних статичних змінних.

4. Виконуються інструкції тіла функції до інструкції повернення.

5. Якщо підпрограма не є **void**-функцією, то значення, що повертається з її виклику, копіюється у пам'ять функції, яка містила виклик.

6. Функція, що містила виклик, продовжується з точки повернення.

Змінні в локальній пам'яті функції не відповідають іменам у функції, що містила виклик, тобто ця пам'ять **недоступна** після того, як виклик закінчено. Вона **вважається звільненою**; її можна використовувати для наступного виклику цієї або іншої функції. Відбувається логічне звільнення, тобто зміст локальної пам'яті не змінюється, але стає недоступним.

**Автоматична пам'ять або програмний стек**

Ділянки пам'яті викликів функцій утворюються та звільняються в спеціальній області пам'яті процесу виконання програми – **автоматичній пам'яті**. Називається вона так тому, що за виконання викликів функцій пам'ять виділяється та звільняється без явних вказівок у програмі, написаній мовою високого рівня, тобто автоматично.

Під час виконання викликів функцій ділянки автоматичної пам'яті виділяються та звільняються за принципом "останньою зайнято – першою звільнено" ("*Last In –First Out*" (LIFO), тобто "останнім прийшов – першим пішов"). Тому автоматичну пам'ять програми також називають **програмним стеком**. коли послідовно виконуються два виклики функцій у тілі функції, то для другого виклику виділяється пам'ять, звільнена після першого.

Приклад.

**#include <iostream>**

**using namespace std;**

**int f(int x)**

**{ return ++x; }**

**int g(int x)**

**{ return x/=2; }**

**int main(){**

**int a=12;**

**cout << f(g(a)) << ' ';**

**cout << a << endl;**

**return 0; }**

З виклику функції **g** повертається **6** і присвоюється параметру **x** функції **f**. Пам'ять функції **g** звільняється. Тільки тепер виконуються інструкції функції **f**. Після їх закінчення в головну функцію повертається значення **7**. Пам'ять виклику функції **f** звільняється.

**Механізм передачі параметрів**

Механізм передачі параметрів є основним засобом обміну інформацією між функцією, що викликається, та функцією, яка викликає. Параметри, котрі зазначаються у заголовку опису функції називаються формальними, а параметри, які записані у операторах виклику функції — фактичними. Приклад.:

**double sqr (double); // прототип функції sqr()**

**main( ) // головна функція**

**{ cout << "Квадрат числа=" << sqr (10) << endl; }//виклик функції sqr()**

**double sqr (double p) //функція sqr()**

**{ return p\*p; } // повернення за значенням**

Функція завжди має бути визначена або оголошена до її виклику. В прикладі спочатку надається прототип функції як оголошення, а опис самої функції надається після **main**-функції, яка її викликає.

***При оголошенні, визначенні та виклику тієї самої функції типи та послідовність параметрів повинні співпадати.*** На імена параметрів обмежень на відповідність не існує, оскільки функцію можна викликати з різними аргументами, а в прототипах імена ігноруються компілятором (вони необхідні тільки для покращення читання програми). Тип значення, що повертає функція, та типи параметрів спільно визначають тип функції.

У найпростішому випадку при виклику функції потрібно вказати її ім’я, за яким у круглих дужках через кому – перелічити імена аргументів, що передаються. Виклик функції може здійснюватися у будь-якому місці програми, де за синтаксисом до­зволяється вираз того типу, що формує функція. Якщо тип значення, що повертає функція не **void**, вона може входити до складу виразів або розташовуватись у правій частині оператора присвоювання.

***У мові С++ визначено декілька способів передачі параметрів і повернення результату обчислень функцій, серед них найбільш широке використання набули:***

1. виклик функції з передачею параметрів за допомогою формальних аргументів-значень;
2. виклик функції з передачею адрес за допомогою параметрів-покажчиків;
3. виклик функцій з використанням посилань, коли доступ до переданих параметрів забезпечується за допомогою альтернативного імені (синоніма);
4. виклик функцій з передачею даних за допомогою глобальних змінних;
5. виклик функцій з застосуванням параметрів, що задані за замовчуванням, при цьому можна використовувати або всі аргументи, або їх частину.

**Ми розглянемо тільки 1, 4, 5 випадки.**

**Виклик функції з передачею значень** полягає у тому, що у функцію передаються не самі аргументи, а їх копії. Ці копії можна змінювати всередині функції, і це ніяк не позначиться на значеннях аргументів, що за межами функції залишаться без зміни, наприклад:

void fun (int **р)** // функція fun()

**{++p;**

**cout << " p = " << p << endl; }**

**void main ( )** // головна функція

**{ int x = 10;**

**fun (x);** // виклик функції

**cout << "x = " << x << endl; }**

Результат роботи цього фрагмента програми: **р=11, х=10,** оскільки для виклику функції fun(x) до неї передається копія значення, що дорівнює **10**. Всередині функції значення копії змінної збільшується на **1**, тобто **(++р)**, і тому виводиться **р = = 11**, але за межами функції параметр **р** не змінюється. У цьому випадку функція не повертає ніякого значення.

При цьому способі для звертання до функції достатньо написати її ім’я, а в дужках — значення або перелік фактичних аргументів. Фактичні аргументи повинні бути записані в тій же послідовності, що і формальні, і мати відповідний тип (крім аргументів за замовчуванням і перевантажених функцій).

Якщо формальними аргументами функції є параметри-значення і в ній не використовуються глобальні змінні, функція може передати у викликаючу її програму лише одне значення, що записується в операторі **return**. Це значення передається в місце виклику функції. Достроковий вихід з функції можна також організувати з використанням оператора return.

**Виклик функцій з передачею даних за допомогою глобальних змінних**

Використовувати глобальні змінні для передачі даних між функціями дуже легко, оскільки вони видимі в усіх функціях, де описані локальні змінні з тими ж іменами. Але такий спосіб не є поширеним, тому що ускладнює налагодження програми та перешкоджає розташуванню функції у бібліотеці загального використання. Потрібно прагнути, щоб функції були максимально незалежними, а їхній інтерфейс повністю визначався прототипом функції. Наведемо приклад використання глобальних змінних:

#include **< iostream.h >**

**int a, b, с;**             // глобальні параметри

**sum ( );**                //прототип функції

**main ( )**                //головна функція

{ **cin >> а >>b;**

**sum();**                // виклик sum()

**cout << с << endl; }**

**sum( )**                  // функція sum()

**{ с** = **а** **+ b;** **}**

**Передача даних за замовчуванням.**

При написанні функції всім аргументам або декільком з них присвоюються початкові значення і задовольняються такі вимоги: коли якому-небудь аргументу присвоєно значення за замовчуванням, то всі аргументи, що розташовані за ним (тобто записані праворуч), повинні мати значення за замовчуванням. Таким чином, список параметрів поділяється на дві частини: параметри, що не мають значення за замовчуванням, і параметри, що мають такі значення.

У випадку виклику функції для параметрів, що не мають значень за замовчуванням, обов’язково повинен бути фактичний аргумент, а для параметрів, що мають значення за замовчуванням, фактичні аргументи можна опускати, коли ці зна­чення не треба змінювати.

Якщо деякий параметр має значення за замовчуванням та для нього відсутній фактичний аргумент, то і для всіх наступних (тобто записаних пізніше) параметрів фактичні аргументи повинні бути відсутні, тобто їхні значення передаються до функції за замовчуванням, наприклад:

**void functl (float х, int у, int z = 80)**

**{ cout << "x = " << x << " у = " << у << "z = " << z** **<< endl;}**

**void funct2 (float x, int у = 25, int z = 100)**

**{ cout << "x = " << x << "y = " << у << "z = " << z << endl; }**

**void funct3 (float x = 3.5, int у = 40, int z = 200)**

**{ cout << "x = " << x << "y = " << у << "z = " << z << endl; }**

**main ( ) {**

**functl (5.1, 10);**//пo замовчуванню передається один аргумент — z

**funct2 (10.2);** // пo замовчуванню передаються два аргумента — у, z

**funct3 ( );**     // пo замовчуванню передаються всі аргументи

}

На екрані буде виведено:

**х = 5.1       у = 10      z = 80**

**х = 10.2    у = 25      z = 100**

**х = 3.5      у = 40      z = 200**

**Аргумент за замовчуванням — це той аргумент, значення якого задане при описі заголовка функції, а при її виклику його можна не вказувати.**

Якщо замість параметра, заданого за замовчуванням при звертанні до функції, записується інше значення фактичного параметра, то значення за замовчуванням перекривається заданим фактичним значенням. Так, наприклад, в останньому програм­ному фрагменті при виклику функції **functl (13.5, 75);** на екрані буде виведено:

**х = 13.5**        **у = 75       z = 80,** тобто **z** — прийнято за замовчуванням.

Якщо виникає необхідність збереження значень локальних змінних між викликами функції, то вони повинні бути оголошені як статичні, тобто з описом **static**, наприклад: **static int num = 10;**

**Шаблони функцій**

У процесі розв’язання багатьох задач необхідно використовувати функції, в яких алгоритм обчислення однаковий, а типи даних відрізняються. Прикладом є задачі пошуку і сортування. Особливістю програмування таких задач мовою С++ є використання шаблонів функцій. Шаблонні функції і шаблони типів є основними елементами *узагальненого програмування* у C++.

*Шаблонні функції* (template functions) призначені для запису узагальнених функцій, що можуть працювати з даними різних типів.

Шаблони функцій — потужний засіб параметризації. За допомогою шаблона функції можна визначити алгоритм, який буде застосовуватися до даних різних типів, а конкретний тип даних передається функції у вигляді параметра на етапі компіляції.

Шаблон функції — це деяка узагальнена функція (родова функція) для сімейства функцій, призначених для розв’язання даної задачі. Визначається така шаблонна функція у заголовному файлі і має такий вигляд:

**template <class Т>**

**type\_func my\_func (type paraml, type param2, …)**

**{**

**// оператори тіла функції**

**}**

Де **template <class T>** — зарезервований вираз (заголовок шаблону), який вказує компілятору оголошений користувачем ідентифікатор типу Т;

**type\_func** — тип шаблонної функції;

**my\_func** — довільний ідентифікатор шаблонної функції;

**type param1, type param2** — формальні параметри, з яких хоча б один повинен мати або наведений у заголовку **(template cclass type>)** тип **Т**, або покажчик**\*param** на змінну типу **Т (Т** **\*param)**, або посилання **&param** на змінну типу **Т (Т&param)**;

**оператори тіла функції** — схема реальних операторів, що генеруються компілятором у відповідну функцію, враховуючи тип даних, вказаних при виклику.

Список формальних параметрів шаблона не може бути порожнім.

У шаблоні функції може бути оголошено декілька формальних типів даних, а також використано параметри означених раніше типів. Наприклад:

**template <class ТІ, class T2>**

**typefunc my\_func(Ti a,double x,T2 b,int c.char s)**

**{**

**//оператори тіла функції**

**}**

Таким чином, оголошення шаблонів функцій завжди починається з ключового слова **template** **(шаблон)**, за ним у кутових дужках визначається список формальних типів, перед кожним з яких вказується ключове слово **class** (або **typename**, що більше відповідає сучасному стандарту) за яким йде ідентифікатор. Ім'я формального параметра в списку повинне бути унікальним. Далі йде звичайний опис функції. При цьому формальні типи, представлені у заголовку шаблону, можна використовувати в опису функції для задання типів аргументів функції, типу значення, що повертається, а також для оголошення змінних усередині тіла функції.

Формальні параметри шаблонів можуть використовуватися для визначення типу результату і формальних параметрів шаблонної функції. У тілі шаблонної функції також можуть використовуватися формальні параметри шаблона.

**Порядок звернення до шаблонних функцій**

Бувають випадки, коли для якихось конкретних типів потрібно дати особливе визначення шаблонної функції. У цьому випадку програміст повинен задати свій спеціальний варіант функції. Наприклад, шаблон функції min() працює для типів, для яких визначена операція "<":

**template** <**class** Type>

Type min(Type a, Type b)

{

**return** a < b ? a : b;

}

Порядок виклику функцій буде таким.

1. Досліджуються всі нешаблонні варіанти функції.
2. Досліджуються всі шаблонні варіанти функції.
3. Повторно досліджуються всі нешаблонні варіанти функції, з застосуванням перетворення типів.

Для того, щоб можна було конкретизувати шаблон, компілятор повинен бачити не тільки оголошення, але і визначення функції. Тому визначення шаблонних функцій можна і треба поміщати в заголовні файли.

Під час виклику функції фактичний параметр шаблона можна вказати явно, наприклад:

**int** i = min<**int**>(2, 3);

**Практичні поради щодо написання програм.**

Програми повинні бути красивими і зрозумілими. Хороша програма красива, як у своїй концепції – використовуваний алгоритм. Хороший код не рясніє тернарними операторами і арифметикою покажчиків, і, принаймні, не повинен бути оптимізованим, щоб заощадити кілька секунд операції, яка виконується раз на кілька місяців. Але читання коду, навіть найідеальнішого, часто потребує деякої допомоги від російського, англійської (або іншого) мов. Іноді алгоритм занадто складний для швидкого або повного розуміння без будь-яких пояснень, або в коді використовується яка-небудь езотерична функція із загадковим і вводить в оману ім'ям. Коментування коду також сприяє глибокому розумінню програми і може допомогти виявити помилки до початку тестування. З естетичних і практичних міркувань хороше коментування є важливим навиком програмування. Але потрібно не піддаватися спокусі і не коментувати занадто багато.

По-перше, коментарі не повинні бути занадто довгими. Коментарі не повинні містити деталі заради деталей. Тільки якщо який-небудь факт необхідний або цікавий, він повинен бути донесений до відома читача програми. Потрібно включати деякі заголовки у верхню частину вашої програми. Ця інформація може включати ваше ім'я та контактну інформацію, дату останньої зміни коду, мета програми і, за потреби, короткий виклад алгоритму або проектні рішення. Ви також можете включити список відомих помилок, недоліків або пропозицій щодо поліпшення. Це зручно для демаркації цієї частини файлу програми в великий блок коментарів.

|  |  |
| --- | --- |
| 1  2  3  4 | /\*\*\*\*\*\*\*\*  \*       \*  \*       \*  \*\*\*\*\*\*\*\*/ |

Цю форму коментаря найкраще робити в кінці програми, за винятком, мабуть, огляду алгоритму, який може допомогти висловити ваші думки, коли ви маєте справу з новими, заплутаними або складними поняттями.

По-друге, при створенні нового класу або нового визначення функції, ви повинні додати коментар, пояснює клас або функцію. Що стосується класу, вам слід пояснити його мета, доступні функції і змінні, будь-які обмеження класу, а також інформацію, яка може знадобитися програмісту, якщо він або вона захоче успадкувати від класу. При визначенні функції ви повинні описати, що робить ця функція і чи є у неї побічні ефекти, такі як зміна глобальних змінних, взаємодіючих з користувачем, і так далі. Це також зручно для опису аргументів, які приймає функція, і значень, якщо такі є, які вона повертає: наприклад, якщо у вас є функція findTime(int distance, int speed), ви хотіли б повідомити користувачеві, що відстань вимірюється в сажнях, швидкість – в аршинах за три дні, і, що функція повертає час подорожі в епохах.

Ви могли б зробити імена змінних більш описовими, але необхідно збільшити інформативність всередині функції, тому що відносини між відстанню і швидкістю – суттєва особливість, а не відносини між distanceinrods і speedinfurlongsperfortnight. Як ви бачите, довгі імена занадто складні і ускладнюють пошук друкарських помилок. Загалом, імена змінних повинні бути описовими настільки, щоб висловити відносини між змінними. Всі деталі реалізації слід відобразити в коментарях у відповідних місцях. Наприклад, у наведеній вище функції, викликає функція повинна знати одиниці, але ці одиниці не мають значення усередині функції (крім випадків, коли робляться перетворення, але програміст повинен зробити запис про це в момент перетворення). Щоб уникнути плутанини, слід уникати скорочень; слова можуть бути скорочені декількома способами, і одна абревіатура може позначати різні слова. Ні скорочень – немає проблем.

По-третє, при додаванні коментарів безпосередньо в коді, будьте стриманими. Якщо ви спробуєте бути точним, ви зробите коментарі кориснішими. Код повинен відповідати алгоритму, а не ходити навколо.

По-четверте, хороші коментарі можуть поліпшити ваше програмування. Ви можете використовувати коментарі для організації коду, вставляючи їх перед блоком коду, а не всередині його – наприклад, коли у вас є довгий блок умовних операторів, ви можете прокоментувати те, що має статися при виконанні різних умов перед самим кодом. Роблячи таким чином, ви позбавите себе від тягаря запам'ятовування подробиць всієї програми, що дозволить зосередитися на реалізації одного з аспектів.

Нарешті, майте на увазі, що, то, що здається очевидним зараз, може здатися зовсім навіть не очевидним пізніше. Хоча вам не слід коментувати занадто багато, обов'язково прокоментуйте нестандартні алгоритми, яким би простим він вам не здавався.

**Принципи структурного програмування.**

При створенні середніх за розміром застосувань (декілька тисяч рядків вихідної коди) використовується структурне програмування, ідея якого полягає в тому, що структура програми повинна відображати структуру вирішуваного завдання, аби алгоритм рішення був ясно видний з вихідного тексту. Для цього треба мати засоби для створення програми не лише за допомогою трьох простих операторів, але і за допомогою засобів, що точніше відображають конкретну структуру алгоритму. З цією метою в програмування введено поняття підпрограми – набору операторів, що виконують потрібну дію і не залежних від інших частин вихідної коди. Програма розбивається на безліч дрібних підпрограм, кожна з яких виконує одну з дій, передбачених вихідним завданням. Комбінуючи ці підпрограми, удається формувати підсумковий алгоритм вже не з простих операторів, а із закінчених блоків коди, що мають певне смислове навантаження, причому звертатися до таких блоків можна по назвах.

Структурне програмування засноване на модульній структурі програмного продукту і типових структурах алгоритмів обробки даних різних програмних модулів, що управляють.

**Типи структур, що управляють:**

– послідовність;

– альтернатива (умова вибору);

– цикл.

**Поширено дві методики (стратегії) розробки програм, що відносяться до структурного програмування:**

– програмування зверху «вниз»;

– програмування «від низу до верху».

**Програмування зверху «вниз», або низхідне програмування – це методика розробки програм, при якій розробка починається з визначення цілей вирішення проблеми, після чого йде послідовна деталізація, що закінчується детальною програмою.**

Спочатку виділяється декілька підпрограм, вирішальних найглобальніші завдання (наприклад, ініціалізація даних, головна частина і завершення), потім кожен з цих модулів деталізує на нижчому рівні, розбиваючись у свою чергу на невелике число інших підпрограм, і так відбувається до тих пір, поки все завдання не виявиться реалізованим.

В даному випадку програма конструюється ієрархічно - зверху вниз: від головної програми до підпрограм самого нижнього рівня, причому на кожному рівні використовуються лише прості послідовності інструкцій, цикли і умовні розгалуження.

Такий підхід зручний тим, що дозволяє людині постійно мислити на наочному рівні, не опускаючись до конкретних операторів і змінних. Крім того, з'являється можливість деякі підпрограми не реалізовувати відразу, а тимчасово відкладати, поки не будуть закінчені інші частини. Наприклад, якщо є необхідність обчислення складної математичної функції, то виділяється окрема підпрограма такого обчислення, але реалізується вона тимчасово одним оператором, який просто привласнює заздалегідь вибране значення (наприклад, 5). Коли все застосування буде написано і відлагоджене, тоді можна приступити до реалізації цієї функції.

Програмування «від низу до верху», або висхідне програмування – це методика розробки програм, що починається з розробки підпрограм (процедур, функцій), у той час коли опрацювання загальної схеми не закінчилося.

Така методика є менш переважною в порівнянні з низхідним програмуванням оскільки часто наводить до небажаних результатів, переробок і збільшення часу розробки.

**Дуже важлива характеристика підпрограм – це можливість їх повторного використання. З інтегрованими системами програмування поставляються великі бібліотеки стандартних підпрограм, які дозволяють значно підвищити продуктивність праці за рахунок використання чужої роботи із створення часто вживаних підпрограм.**

**Підпрограми бувають двох видів – процедури і функції. Відрізняються вони тим, що процедура просто виконує групу операторів, а функція додатково обчислює деяке значення і передає його назад в головну програму (повертає значення). Це значення має певного типа.**

Аби робота підпрограми мала сенс, їй треба отримати дані із зовнішньої програми, яка цю підпрограму викликає. Дані передаються підпрограмі у вигляді параметрів або аргументів, які зазвичай описуються в її заголовку так само, як змінні.

**Підпрограми викликаються, як правило, шляхом простого запису їх назви з потрібними параметрами.**

**Підпрограми активізуються лише у момент їх виклику. Оператори. Які знаходяться усередині підпрограми, виконуються, лише якщо ця підпрограма явно викликана.**

**Підпрограми можуть бути вкладеними – допускається виклик підпрограми не лише з головної програм, але і з будь-яких інших програм.**

У деяких мовах програмування допускається виклик підпрограми з себе самій. Такий прийом називається рекурсією і небезпечний тим, що може привести до зациклення – нескінченному самовиклику.

Достоїнства структурного програмування:

– підвищується надійність програм (завдяки хорошій структуризації при проектуванні, програма легко піддається тестуванню і не створює проблем при відладці);

– підвищується ефективність програм (структуризація програми дозволяє легко знаходити і коректувати помилки, а окремі підпрограми можна переробляти (модифікувати) незалежно від інших);

– зменшується час і вартість програмної розробки;

– покращується читабельність програм.

Таким чином, технологія структурного програмування при розробці серйозних програмних комплексів, заснована на наступних принципах:

– програмування повинне здійснюватися зверху вниз;

– весь проект має бути розбитий на модулі (підпрограми) з одним входом і одним виходом;

– підпрограма повинна допускати лише три основні структури – послідовне виконання, розгалуження (if, switch) і повторення (for, while).

– недопустимий оператор передачі управління в будь-яку крапку програми (goto);

– документація повинна створюватися одночасно з програмуванням у вигляді коментарів до програми.

Структурне програмування ефективно використовується для вирішення різних математичних завдань, що мають алгоритмічний характер.

***Контрольні запитання для самоперевірки***.

1. Чи обов’язково вказувати типи параметрів функції в прототипі? Чи обов’язково вказувати імена параметрів?
2. Опишіть призначення функцій та особливості їх оголошення.
3. Охарактеризуйте особливості передачі параметрів функції значенням, масивів; замовчувані значення параметрів.
4. Дайте пояснення, що таке «Області видимості функцій».
5. Охарактеризуйте особливості повернення та використання значення функцій.
6. Визначення і оголошення глобальних об’єктів і функцій.
7. Тривалість життя об’єктів: автоматичне, статичне і динамічне.
8. Визначте типи формальних параметрів.

**Для самостійного вивчення** *(2 години)*: Вивчення лекційного матеріалу та додаткових джерел. Розгляд запитань і виконання завдань для самостійної роботи, запропонованих на лекції.

**Рекомендована література**

1. Ковалюк Т. В. Алгоритмізація та програмування: Підручник. — Львів: «Магнолія 2006», 2013. — 400 с., ил.
2. Павловская Т.А. С/С++. Программирование на языке высокого уровня. СПб.: Питер, 2003. – 461 с. URL: <http://www.ph4s.ru/bookprogramir_1.html>
3. Вступ до програмування мовою С++. Організація обчислень: навч. посіб. / Ю. А. Бєлов, Т. О. Карнаух, Ю. В. Коваль, А. Б. Ставовський. – К. : Видавничо-поліграфічний центр "Київський університет", 2012. – 175 с. с.: іл. ISBN (укр.) . URL: <http://csc.knu.ua/uk/library/books/belov-24.pdf>
4. Джейс Либерти Освой самостоятельно С++ за 21 день: 3-е изд. пер. с англ.: Уч. пос. – М.: Издательский дом „Вильямс”, 2001. – 816 с.: ил..