Основи мови програмування C++*

Корх Олег

17 грудня 2007 р.

Зміст

1 Історія

Говорячи про C++ неможливо не згадати її предка — мову програмування C. То ж заглибимося трохи у історію. Почалося все ще у 1965–1969 роках коли фірма Bell Laboratories разом з General Electric вела розробку операційної системи Multics. Планувалося, що це буде величезна та дуже складна система для великих машин (напр. серії PDP). Але у 1969 році Bell Labs вийшла з проекту. Частина співробітників почала роботу над дещо менш амбіційною але простішою системою — UNIX, оскільки Multics будучи занадто складною була мертвою вже на момент свого народження. Група під головуванням Кена Томсона ставила за мету створити зручне оточення для досліджень в області програмування. Першу версію було написано на асемблері. Така собі революція пройшла у 1972 році, коли запрошений до роботи тоді ще молодий хакер Деніс Річі переписав систему на створеній ним мові програмування C. Перша версія C не була мовою програмування високого рівня у класичному розумінні, по суті це був своєрідний макроасемблер з деякими елементами структурного програмування. До 1978 року Річі разом з другом Брайаном Керніганом було підготовлено перший стандарт мови C, який по першим літерам їх прізвищ було названо «K&R C». У 1989 році було затверджено стандарт ANSI C розроблений компанією AT&T. Серед нововведень був дещо вдосконалений синтаксис та більш жорстка типізація. У 1999 році з'явився стандарт ANSI C99 у якому з'явилися динамічні масиви, комплексні числа, підтримка Unicode та ін. Стандарт C99 підтриманий рядом компіляторів, що вільно розповсюджуються, у тому числі популярним GCC. Натомість комерційні компілятори, як наприклад Microsoft Visual C, це розширення не підтримують, а вводять власні розширення та C++. C і досі широко використовується переважно у системному програмуванні, а крім того він сильно вплинув на багато інших мов програмування. У 1999 році Деніс Річі та Кен Томсон отримали національну медаль за досягнення в області технологій від тодішнього президента США Біла Клінтона, за створення операційної системи UNIX та мови програмування C.

C++[1] — є найбільш відомим і популярним нащадком C. Його розробив Бьярн Страуструп, який до 2002 року був головою відділу досліджень в області крупномасштабного програмування у компанії AT&T (Computer Science Research Center of Bell Telephone Laboratories). Нині професор Техаського университету A&M. Офіційно роком народження C++ вважається 1985, але насправді мова почала з'являтися трохи раніше коли Страуструп почав розробляти розширення до C під власні потреби (десь починаючи з 1979 року). Серед нововведень з'явилася підтримка об'єктно-орієнтованого програмування, шаблони, перевантаження операторів та функцій, простори імен, нові бібліотеки, типи та ін. У 1998 році було ратифіковано міжнародний стандарт C++: ISO/IEC 14882:1998 «Standard for the C++ Programming Language»; після прийняття деяких технічних виправлень до стандарту у 2003 році нинішня версія цього стандарту — ISO/IEC 14882:2003. На сьогоднішній день C++ є однією з найбільш популярних мов програмування.

2 Базові конструкції C++

2.1 Структура програми

Найпростіша програма на C++, що просто виводить на екран рядок тексту `Hello, World!` виглядає наступним чином:

```
#include <iostream>
using namespace std;
int main()
{
        cout << "Hello, World!" << endl;
        return 0;
}
```

У першому рядку зустрічається директива компілятора `#include` яка служить для підключення заголовкових файлів, тобто дозволяє використовувати додаткові бібліотеки. Якщо ім'я файлу заключене між символами `<...>`, то компілятор шукає файл у переліку відомих йому каталогів з бібліотеками, а якщо

*Коментарі та побажання щодо можливого вдосконалення цього довідничка можна залишати у моєму персональному блозі: *http://my.opera.com/bum/blog/*

[1] Приставка ++ походить від оператора інкременту у C і таким чином вказує на те, що C++ є C з розширеними можливостями.

між символами "...", то тоді у поточному каталозі. У даному випадку підключається бібліотека для підтримки потоків введення/виведення. Конструкція `using namespace` використовується для задання простору імен, на даному етапі заглиблюватися у це немає сенсу.

Кожна програма на C++ повинна містити функцію `main`, яка задає точку входу у програму. Саме ця функція автоматично виконується першою при запуску програми. Фігурні дужки обмежують тіло функції. Для виведення даних використовується стандартний потік виведення `cout`, який являє собою звичайну текстову консоль. Оператор `<<` служить для перенаправлення даних у потік, а `endl` говорить, що потрібно перевести курсор на новий рядок.

У кінці функції викликається оператор `return`, він служить для повернення результату функції, у даному випадку `0` надсилається операційній системі і говорить, що програму завершено успішно. Потрібно сказати, що виклик `return 0` у системах родини Windows є не обов'язковим, проте гарним стилем програмування все ж таки є використання подібних конструкцій. У операційних системах сумісних зі стандартами POSIX (Linux, BSD та ін.) ці значення натомість активно використовуються, тому з метою забезпечення кросплатформеності бажано притримуватися загальних підходів.

Ще один момент на який потрібно звернути увагу — це стиль оформлення коду. Насправді C++ досить лояльний до цього, при бажанні весь текст програми можна написати хоч в один рядок, проте дуже важливо, щоб код насправді міг вільно читатися. Саме тому рекомендується використовувати відступи для демонстрації структури програми, це дозволяє не лише краще орієнтуватися у тексті, але й дозволяє швидко виловлювати часті помилки з незакритою фігурною дужкою.

2.2 Коментарі

Програмісти використовують коментарі для того щоб залишати у коді програми пояснення щодо особливостей його роботи. Гарним стилем програмування є коментування основних ділянок коду під час написання програми, це дозволить через деякий час набагато легше згадати принципи роботи програми.

```
/* багаторядковий
                коментар */
// однорядковий коментар
```

Коментарі також можна використовувати на етапі налагодження програми для швидкого відключення/включення ділянок коду.

2.3 Складений оператор

Складений оператор (іноді його можна називати блоком) використовується у тих випадках коли згідно правил C++ допустимо використовувати лише один оператор (наприклад конструкції `if, for, while`), а насправді потрібно вставити декілька. Такий оператор сприймається компілятором як один і задається парою символів `{ }`. Ще один момент який важливо знати, то це те, що будь-які змінні оголошені у рамках такого блоку не будуть видимі за його межами (тобто будуть локальними), натомість всі змінні оголошені за його межами видимі і всередині нього.

```
{ оператори }
{ int i = 3; }
int x = i; // помилка, за межами видимості!
```

2.4 Заголовкові файли

Заголовкові файли використовуються для підключення зовнішніх бібліотек.

```
#include <iostream>
#include "myclass.h"
#include <cmath>
```

Потрібно звернути увагу, що у C++ назви стандартних заголовкових файлів C прийнято писати дещо у іншому стилі, а саме, розширення імені файлу `.h` прибирається, а перед старою назвою додається буква `c`. Таким чином `math.h` перетворюється у `cmath`, `stdlib.h` перетворюється у `cstdlib` і т.д.

2.5 Ідентифікатори

Ідентифікатор — це послідовність букв, цифр та символів підкреслення (_). Ідентифікатор не може починатися з цифри. Великі та малі літери розрізняються.

```
i
size
w123
_foo
```

Наступні приклади не можуть бути ідентифікаторами:

```
for   // ключове слово
3w    // не можна починати з цифри
-fgh  // не плутайте - та _
_Sysfoo // підкреслення і велика літера
        // для системного використання
```

2.6 Оголошення змінних

Змінні у C++ оголошуються наступним чином:

```
тип_даних ідентифікатор[ = значення];
int a,b,c;
char c = 'A';
```

На відміну від C, у C++ не важливо у якому місці програми оголошено змінну. Вимагається лише єдине правило — кожну змінну потрібно оголосити до її використання. Якщо потрібно оголосити декілька змінних одного типу їх розділяють комами. Одночасно з оголошенням, змінну можна й ініціалізувати задавши початкове значення.

Модифікатор `const` використовується для оголошення констант. На відміну від звичайних змінних константі можна присвоїти значення лише один раз, під час її ініціалізації.

```
const float pi = 3.1415926;
pi = 2; // помилка!
```

2.7 Оператор присвоєння

Оператор присвоєння дозволяє встановлювати значення змінної як результат виконання виразу.

```
змінна = вираз;
i = 2 + 3;
c = sqrt(a*a+b*b);
```

Допускається об'єднувати оператор присвоєння з арифметичним оператором у наступній формі:

```
i += 2; // еквівалентно i=i+2
k -= 5;  // еквівалентно k=k-5
```

і т.д., у парі з оператором присвоєння допускається використання будь-якого арифметичного оператору.

2.8 Літерали

Літерали — це постійні значення, як наприклад 1 чи 3.1415926. Для кожного типу у C++ існують літерали, включаючи числа, рядки символів та ін. Ось деякі приклади:

`5`	ціла константа
`5U`	беззнакова ціла константа
`5L`	довге ціле
`05`	вісімкове число
`0x5`	шітнадцяткове число
`true`	булева константа
`5.0`	дробове число
`5.0F`	дробове одинарної точності
`5.0L`	довге подвійної точності
`'5'`	символьна константа
`'\n'`	символ переводу на новий рядок
`L'XYZ'`	`wchar_t` символ
`"Hello!"`	символьний рядок

Існують наступні спеціальні символьні константи:

'\a'	звуковий сигнал
'\\'	бекслеш
'\b'	повернення на крок
'\r'	повернення каретки
'\"'	подвійні лапки
'\f'	прогон аркуша
'\t'	табуляція
'\n'	новий рядок
'\0'	нульовий символ
'\''	апостроф
'\v'	вертикальна табуляція
'\101'	вісімковий ASCII-код 'A'
'\x041'	шістнадцятковий ASCII-код 'A'

2.9 Базові типи

Можна виділити наступні базові типи даних:

bool	логічний (true або false)
char	одиничний символ
wchar_t	символ Unicode
int	ціле число ($-32768 \dots 32767$)
long	довге ціле ($-2147483648 \dots 2147483647$)
float	дробове ($1,17549 \cdot 10^{-38} \dots 3,40282 \cdot 10^{+38}$)
double	дробове ($2,22507 \cdot 10^{-308} \dots 1,79769 \cdot 10^{+308}$)
long double	дробове ($3,36210 \cdot 10^{-4932} \dots 1,18973 \cdot 10^{+4932}$)

Допустимо також використовувати модифікатори `signed` або `unsigned`, які вказують відповідно чи може змінна приймати від'ємні значення чи ні. Потрібно зауважити, що об'єм який займає змінна у пам'яті при цьому не змінюється, а границі максимальних значення при цьому зміщуються.

2.10 Перерахований тип

Даний тип використовують у тому разі коли наперед відомо, що змінна може приймати чітко визначений набір значень. Наприклад таким типом є вже згаданий раніше `bool`, змінні якого можуть приймати лише два чітко визначених значення: `true` та `false`.

```
enum suit {clubs, diamonds, hearts, spades};
suit card;  // оголошуємо змінну типу suit
enum {lazy, hazy, crazy} why;
```

Зверніть увагу, що перераховані константи можуть оголошуватися анонімно, без задавання імені типу, так як це зроблено при оголошенні змінної `why`. Відповідно також можна об'єднувати оголошення нових типів і змінних цього ж типу у один рядок.

2.11 Масиви

Масиви дозволяють об'єднувати дані одного типу таким чином щоб до них можна було звертатися по назві самого масиву та індексу. Такі конструкції широко застосовуються для зберігання різних списків, матриць та ін.

```
int a[10];   // створює масив a[0] , ..... , a[9]
// ініціалізація масиву: a[0]=9, a[1]=8, a[2]=7
int a[3] = {9,8,7};
int b[3][5]; // двовимірний масив (матриця)
```

Нижня границя масиву завжди приймається за нуль, а число котре вказується при оголошенні масиву задає його розмір. Таким чином верхня границя масиву буде завжди на одиницю меншою. Доступ до елементів масиву відбувається по аналогії, наприклад `a[i], b[i][j]` і т.д. Присвоєння одного масиву іншому напряму не допускається, у таких випадках їх потрібно копіювати поелементно. Динамічні масиви, тобто масиви розміри яких би задавалися на етапі виконання, а не компіляції, напряму не підтримуються, проте можуть бути змодельовані з допомогою вказівників (див. 2.14).

2.12 Структури

Структури використовують тоді, коли необхідно об'єднати декілька даних різних типів в одній змінній. Це зручно наприклад при реалізації баз даних хоча і далеко не лише для цього. Елементи структури називаються полями і для доступу до них у програмі використовують крапку для відділення імені структури від імені поля.

```
struct point {
   int x, y;
}
point p;
// доступ до елементів структури:
p.x = 10;
p.y = 7;
```

2.13 Об'єднання

Об'єднання чимось схоже на структуру, але його члени використовують пам'ять сумісно і їх значення перекриваються. Тобто значення такій змінній може бути присвоєно лише одне, але тип може бути різним, в залежності від того які ви передбачите.

```
union int_dbl {
   int i;
   double x;
}
int_dbl n = 7;
n = 5.67;
```

2.14 Вказівники

Вказівник на відміну від звичайних змінних зберігає лише адресу розташування даних у пам'яті. Насправді це дуже потужний інструмент з допомогою якого можна будувати складні структури даних, наприклад стеки, черги, кільця, зв'язані списки, двійкові дерева та ін. Це окрема велика тема, проте найпростіше вказівники можна використовувати для реалізації динамічних масивів.

```
int* p;    // вказівник на тип int
p = new int[10];    // розміщуємо масив p[0]...p[9]
delete[] p;    // звільняємо пам'ять зайняту масивом
```

Наведемо приклад програми, де користувач вводить розмір масиву, потім вводить сам масив, після чого програма повторно виводить його на екран.

```
#include <iostream>
using namespace std;
int main()
{
        int i, size;
        int* a;
        cout << "Введіть розмір масиву:";
        cin >> size;
        if (size==0) return 1;
        a = new int[size];
        for(i=0;i<size;++i) {
                cout << "a[" << i << "]=";
                cin >> a[i];
        }
        cout << "_____" << endl;
        for(i=0;i<size;++i) {
                cout <<"a["<<i<<"]="<<a[i]<< endl;
        }
        delete[] a;
        return 0;
}
```

При роботі з вказівниками потрібно пам'ятати, що вся відповідальність за виділення та звільнення пам'яті повністю лягає на програміста, тому не потрібно забувати про звільнення пам'яті як тільки змінна перестає бути потрібною (оператор `delete`).

2.15 Арифметичні вирази

У арифметичних виразах використовуються наступні операції:

+ , -	додавання , віднімання
* , /	множення , ділення
%	залишок від ділення
++	збільшення на одиницю (інкремент)
--	зменшення на одиницю (декремент)
>>	побітовий зсув праворуч
<<	побітовий зсув ліворуч
&	побітова операція І
|	побітова операція АБО

Частина операцій виконує дії напряму над операндами. Потрібно звернути увагу, що при використанні операцій `++` та `--` має значення їх розміщення відносно операнду. Наприклад `i++` збільшує змінну `i` на одиницю, але у якості результату повертає її значення до збільшення, а `++i` натомість повертає значення вже після збільшення. При операціях побітового зсуву вивільнені біти заповнюються нулями.

Існує ще один важливий момент. Результат операції ділення залежить від типу операндів, якщо у якості операндів задаються цілі числа, то виконується цілочислове ділення (дробова частина результату відкидається). Це дуже суттєвий момент який може бути не очевидним особливо програмістам на Pascal. Для того щоб результатом ділення двох цілих чисел було дробове число потрібно використати явне приведення типів. У C++ це робиться з допомогою імені типу взятого у круглі дужки розміщеного перед операндом. Наприклад: `f = (double)i/k`.

2.16 Оператор розгалуження

Оператор розгалуження використовується у тому випадку коли потрібно змінити порядок виконання операторів у залежності від якоїсь умови.

```
if (вираз)
        якщо так;
else
        якщо ні;
```

Ну наприклад:

```
if (a==2)
        cout << "Hi";
else
        cout << "Bye";
```

У логічних виразах використовуються наступні операції:

==	дорівнює
!=	не дорівнює
< , >	менше , більше
<= , >=	менше дорівнює , більше дорівнює
!	операція НІ (NOT)
&&	операція І (AND)
||	операція АБО (OR)

2.17 Оператор `switch`

Іноді потрібно проаналізувати значення змінної з набору наперед відомих констант, ну наприклад якщо програма підтримує параметри командного рядка, то потрібно аналізувати введені ключі. Ланцюжок утворений операторами `if` у такому разі буде занадто великий, тому в таких випадках використовують оператор `switch`:

```
switch (вираз) {
        case константа: оператори
        case константа: оператори
        . . . . . .
        default: оператори
}
```

Більш докладно це видно на прикладі:

```
switch (a) {
        case 1:
                cout << "One";
                break;
        case 2:
                cout << "Two";
                break;
        default:
                cout << "??";
}
```

Потрібно звернути увагу, що після виконання операторів які відповідають потрібній константі виконання оператору `switch` не закінчується, а починають виконуватися оператори які знаходяться далі, біля інших міток. Це відрізняє даний оператор від його аналогів у деяких інших мовах (наприклад `case` у Pascal). Для того щоб завершити виконання конструкції потрібно використати оператор `break`. Блок `default` задає оператори які будуть виконуватися у разі якщо значення змінної не відповідає жодному з наведених варіантів (тобто аналог `else`).

2.18 Цикли

Цикли використовуються у тих випадках якщо у програмі потрібно реалізувати повторення певної ділянки коду в залежності від якоїсь умови. У C++ існує три конструкції які задають цикли. Перш за все потрібно згадати цикл з передумовою. Такий цикл виконується до тих пір доки результатом виразу вказаного у заголовку є значення `true`.

```
while (вираз)
    оператор;
```

Існує також його модифікація, цикл `do`, який задає цикл з післяумовою. Тобто умова перевіряється не на початку циклу, а вкінці, вся інша поведінка циклу є аналогічною `while`.

```
do
  . . . . . .
while (вираз);
```

Потрібно лише звернути увагу, що у циклі `do` не потрібно використовувати складений оператор якщо потрібно вставити декілька операторів, ключові слова `do` та `while` тут вже виконують роль операторних дужок.

Часто цикли використовують наприклад для роботи з масивами, коли при кожному проході циклу потрібно змінювати індекс, тоді зручно використовувати наступну конструкцію:

```
for (вираз1;вираз2;вираз3)
    оператор;
```

Зверніть увагу, що вирази відокремлюються крапкою з комою. Компілятор приводить цикл `for` до наступного вигляду:

```
вираз1;
while (вираз2) {
    оператор;
    вираз3;
}
```

Таким чином, якщо нам наприклад потрібно вивести на екран вміст якогось масиву `int a[10]` то можна використати таку конструкцію:

```
for (int i=0;i<10;++i)
    cout << a[i] << endl;
```

Також у циклах можна використовувати оператори `continue` (перериває поточний крок циклу і починає новий) та `break` (примусовий вихід із циклу).

2.19 Простори імен

C++ отримав у спадщину від C єдиний глобальний простір імен. Проте при об'єднанні програм написаних різними колективами, а тим більше оскільки C++ заохочує використання сторонніх бібліотек можуть бути конфлікти. Тому було введено концепцію простору імен.

```
namespace LMPin {
        int n;
        string s;
}
```

Тоді повний ідентифікатор буде складатися з назви простору імен та власне ідентифікатору розділених символом `::`. Наприклад `LMPin::n` чи `LMPin::s`. Можна також отримати доступ до всіх імен заданого простору виконавши `using namespace LMPin`, тоді ідентифікатори можна писати просто: `n` та `s`. У відповідності з новими стандартами ANSI стандартні заголовкові файли тепер використовують простір імен `std`, тому для роботи з ними потрібно вставляти у свої програми рядок `using namespace std`.

3 Створення власних функцій

3.1 Визначення функції

Функції використовуються для об'єднання ділянок коду які часто використовуються у межах одного блоку. Функція має назву, набір вхідних параметрів, блок операторів, а також може повертати результат.

```
тип ім'я_функції (тип параметр [,...]) {
    тіло функції
    return значення;
}
```

Якщо функція не повертає даних, то вказують тип `void`. Параметри задаються у такому ж стилі як і звичайне оголошення змінних. Оператор `return` служить для повернення значення, таким чином функції можна використовувати у виразах.

```
// площа прямокутника
float rect_area (float w,h) {
        return w*h;
}
```

Допускається всередині тіла функції посилатися на неї ж саму, такі функції називаються рекурсивними. Рекурсія дозволяє суттєво скоротити розмір програмного коду але потребує деякого тренування щоб зрозуміти принципи.

```
// факторіал, рекурсивна ф-ція
int factorial (int x) {
        if (x==1)
                return 1;
        else
                return x*factorial(x-1);
}
```

Параметри у функцію можна передавати не лише по значенню, але й за посиланням, для цього служить префікс `&` (у Pascal з тією ж метою використовується модифікатор `var`).

```
// передача значень через посилання
void get_int (int &v) {
   scanf("%d",&v);
}
// викликається така функція наступним чином:
get_int(i);
```

У цьому коді змінну `i` буде змінено. Зверніть увагу, що при передачі параметрів за посиланнями при виклику функції обов'язково потрібно задавати змінні, виклик даної функції у вигляді `get_int(5)` є помилкою.

3.2 Розміщення функцій у зовнішніх файлах

Часто дуже зручно буває винести набір власних функцій у окремий файл. Таким чином можна створити власну бібліотеку і потім використовувати її у інших програмах. Бібліотеки задаються у C++ парою файлів з розширеннями `.h` (заголовковий файл) та `.cpp` (файл реалізації). Заголовковий файл містить лише оголошення заголовків функцій і саме він підключається до основної програми з допомогою директиви `#include`, а безпосередньо самі функції описуються у файлі реалізації.

Наприклад, створимо бібліотеку `factorial`. Для цього потрібно створити два файли:

```
// factorial.h
int factorial (int x);

// factorial.cpp
int factorial (int x) {
        if (x==1)
                return 1;
        else
                return x*factorial(x-1);
}
```

Тепер функцію `factorial` можна використовувати у інших програмах вказавши директиву `#include "factotial.h"`. Насправді, правильна компіляція програми, яка складається з декількох файлів вимагає знання опцій компілятора та укладача (linker), які можуть відрізнятися в залежності від того які компілятори і на якій платформі використовуються. Тому на цьому акцентувати увагу не будемо, у будь-якому разі у розповсюджених IDE для C++ додавання файлу у проект автоматично означає його компонування (linking), тому особливо нічого складного насправді тут нема.

4 Стандартні бібліотеки

4.1 Введення/виведення даних

Історично існує два підходи до реалізації введення/виведення даних. Це може бути або класичний підхід C (з функціями типу `printf` та `scanf`) чи підхід C++ базований на концепції потоків. Розглянемо обидва підходи.

Для роботи з функціями введення/виведення у стилі C потрібно на початку вказати `#include <cstdio>`

```
printf("Площа дорівнює %d\n",rect_area(5,3));
scanf("%f",a&);
```

Функція `printf` служить для форматованого виводу даних, перший її параметр — це рядок у якому з допомогою символу `%` задаються місця куди потрібно вставити значення передані іншим параметрам. Повний опис формату наводиться у спеціалізованій літературі, а тут просто зауважимо, що `%d` служить для вставки цілого числа, а `%f` — дробового. Щоб округлити дробове число наприклад до трьох знаків після коми використовується формат `%.3f`. Функція `scanf` дозволяє вводити дані з клавіатури, зверніть увагу на оператор взяття адреси (`&`).

У C++ для організації введення/виведення рекомендується використовувати концепцію потоків. Для цього потрібно підключити бібліотеку `iostream`.

```
cout << "Hello, World!" << endl;
// виведення даних
cin >> a;    // введення даних
```

Потоки `cout` (виведення) та `cin` (введення) є звичайними класами, а операція побітового зсуву (`<<`) перевантажена для роботи з потоками. Модифікатор `endl` служить для переведення рядка, хоча ніщо не заважає використовувати старий стиль з `'\n'`.

4.2 Символьні дані

У C не існувало спеціалізованого типу для представлення рядків символів, замість цього доводилося використовувати символьні масиви. Причому краще використовувати вказівники, тоді немає проблем з розміром рядка, така змінна зберігає адресу першого символа рядка у пам'яті, а закінчувався такий рядок нульовим символом (`NULL` або `'\0'`). Такі ж текстові рядки можна використовувати і у C++.

```
char* s = "Hello!";
cout << s << endl;
```

Для того щоб проводити з такими рядками певні дії потрібно підключити бібліотеку `cstring`.

`char* strcat(s1,s2)`	конкатенація двох рядків
`int strcmp(s1,s2)`	порівняти рядки
`char* strcpy(s1,s2)`	копіювати рядок `s2` у `s1`
`int strlen(s)`	довжина рядка
`int strchr(s,c)`	знайти перше входження символу `c`
`int atoi(s)`	конвертує рядок у ціле число
`double atof(s)`	конвертує рядок у дробове число
	і т.д.

Для конвертації числа у символьний рядок можна використовувати функцію `sprintf`, яка аналогічна `printf` проте не виводить результат на екран, а записує його у змінну вказану у першому параметрі (з `&` звичайно).

У C++ з'явився новий клас `string`, який інкапсулює у собі всю функціональність потрібну для роботи з символьними рядками. Змінним типу `string` можна присвоювати значення змінних типу `char*`, а от зворотнє присвоєння не допустимо, у таких випадках використовують метод `c_str` класу `string`. Для того щоб даний клас був доступним потрібно підключити бібліотеку `string`.

```
string s;    // оголошуємо пустий рядок s
string s1("Привіт!"); // ініціалізуємо рядок
s = "Ги-ги"    // присвоєння значення рядку
```

```
s = s + " :-) " + s1;    // конкатенація
int len = s.size; // довжина рядка
cout << s.substr(0,4); // виділити підрядок
int loc = s.find(":-)",0); // шукаємо підрядок
s1.insert(0,s);  // вставити підрядок
s.erase(0,4);  // видалити підрядок
```

Щоб зконвертувати рядок символів у число можна скористатися функціями `atoi` чи `atof` (ціле та дробове числа відповідно).

```
integer=atoi(buffer.c_str());
flt=atof(buffer.c_str());
```

А от конвертація числа у рядок типу `string` вже далеко не така очевидна. Для цього можна створити власний символьний потік і записавши у нього число потім отримати вже символьний результат. Для цього потрібно підключити бібліотеку `sstream`. Наведемо приклад функції яка конвертує ціле число у символьний рядок:

```
string int2string(const int& number)
{
        ostringstream oss;
        oss << number;
        return oss.str();
}
```

4.3 Робота з файлами

Для роботи з файлами потрібно підключити бібліотеку `cstdio`. Кожен файл у програмі представляється вказівником на тип `FILE`.

```
FILE *f;    // задаємо файлову змінну
// відкриваємо файл на запис
f = fopen("filename.txt","w");
fprintf(f,"Hello!\n");   // виводимо дані у файл
fclose(f);    // закриваємо файл
// відкриваємо файл на читання
f = fopen("filename.txt","r");
// читаємо рядок з файлу у змінну s
fscanf(f,"%s",&s);
// відкриваємо файл на додавання
f = fopen("filename.txt","a");
```

Існують і інші режими відкриття файлів, зокрема файл можна відкривати одночасно і для запису і для читання, можна працювати з бінарними файлами та ін. Повну інформацію про використання даних функцій можна знайти у відповідній літературі.

Крім того, у C++ з'явився ще один спосіб роботи з файлами, у стилі тих же потоків. Якщо підключити бібліотеку `fstream`, то стає можливим використовувати класи `ifstream` та `ofstream` для створення та керування відповідно вхідними та вихідними потоками. Наприклад щоб вивести дані у файл можна скористатися таким способом:

```
ofstream f_out;
f_out.open("filename.txt");
f_out << "Перший рядок" << endl;
f_out << "Другий рядок" << endl;
f_out.close();
```

Читати дані з вхідного файлу можна абсолютно аналогічно. Потрібно лише використовувати клас `ifstream`, а сам процес зчитування даних є повністю аналогічним роботі зі стандартним потоком вводу `cin`.

4.4 Математичні функції

Функція взяття абсолютного значення (модуля) числа `abs` знаходиться у бібліотеці `cstdlib`, інші математичні функції вимагають бібліотеку `cmath`.

`sin, cos, tan`	тригонометричні функції
`asin, acos, atan`	обернені тригонометричні функції
`sinh, cosh, tanh`	гіперболічні функції
`exp(x)`	e^x
`log`	натуральний логарифм
`log10`	десятковий логарифм
`pow(a,b)`	a^b

4.5 Випадкові числа

Функції для генерації випадкових чисел зосереджені у бібліотеці `cstdlib`. Перш за все, щоб виключити можливість повторення чисел потрібно з допомогою функції `srand` ініціалізувати лічильник довільним значенням, найкраще для цього підійде поточний час:

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

Для отримання власне випадкового числа використовують функцію `rand`, яка генерує випадкове число у діапазоні від `0` до `RAND_MAX` (на сучасних комп'ютерах це 2147483647). Якщо потрібно знайти число у діапазоні від 0 до 1, то результат потрібно відповідним чином перерахувати.

```
x = rand();
// знаходимо випадкове число у діапазоні 0..1
x = (float)rand()/RAND_MAX;
```

5 Об'єктно-орієнтоване програмування

5.1 Поняття класу та об'єкту

Об'єктно-орієнтоване програмування — це парадигма програмування, у якій основною концепцією є використання поняття об'єкт який ототожнюється з об'єктом у предметній області. Першою об'єктно-орієнтованою мовою програмування була Simula-67 яка на момент своєї появи, у кінці 60-х, серйозно випередила свій час і не набула широкого вжитку через не готовність тодішніх програмістів сприйняти нову концепцію програмування. Саме Simula надихнула Бьярна Страуструпа на реалізацію класів у C++. Чому виникла дана парадигма? Справа у тому, що рано чи пізно під час ускладнення програмного продукту наступає такий момент, що у рамках традиційного процедурного підходу розв'язок проблеми стає дуже складним. Об'єктно-орієнтований підхід дозволяє суттєво спростити задачі моделювання складних систем.

Взагалі тема об'єктно-орієнтованого програмування заслуговує окремої великої розмови, тут ми поки просто обмежимося особливостями реалізації. Отже, клас (`class`) — це певна структура даних, яка сильно нагадує `struct` проте дозволяє включати у собе не лише змінні (властивості) але й функції (методи). Такий підхід називається інкапсуляцією. Покажемо реалізацію простого класу який інкапсулює механізм роботи з символьними рядками.

```
class my_string {
public:
        void assign(const char* str);
        int length() const { return len; }
        void print() const
        { cout << s << endl; }
private:
        char s[225];
        int len;
}; // зверніть увагу, тут має бути крапка з комою

void my_string::assign(const char* str) {
        delete[] s;
        len = strlen(s);
        s = new char[len+1];
        strcpy(s,str);
}
```

Розглянемо наведений код докладніше. Як бачимо, клас дійсно багато чим нагадує структуру. За єдиним винятком — клас може включати функції, які прийнято називати методами. Ключові слова `public` та `private` задають області видимості членів класа. У даному випадку змінні `s` та `len` оголошені у приватній частині класу, а це означає, що доступ до них мають лише методи даного класу, ззовні звернутися до них неможливо. По замовчуванню всі елементи класу є приватними якщо не вказано іншого, це дозволяє виключити непередбачені зміни структури даних. Це особливість об'єктно-орієнтованого програмування, всі маніпуляції з даними прийнято робити з допомогою методів. Методи `length` та `print` служать для повернення довжини рядка та виведення рядка на екран відповідно. Директива `const` вказує на те, що тіла методів задаються тут же всередині класу. Реалізація ж методу `assign` який служить для передачі символьного рядку у приховану змінну `s` винесена назовні. Методи описуються так же само як і звичайні

функції але перед ім'ям обов'язково потрібно вказувати ім'я класу відокремлене символами ::.

Тепер поглянемо як можна використати даний клас:

```
int main() {
        my_string st;
        st.assign("Привіт!");
        st.print();
        cout << "Довжина st =" << st.length();
        return 0;
}
```

Як бачимо для виклику методів використовується оператор «крапка» (так я і у випадку зі структурою). Змінна `st` згідно термінології ООП називається об'єктом класу `my_string`.

5.2 Конструктори та деструктори

Метод, єдина робота якого полягає в ініціалізації об'єкту класу називається конструктором (constructor). Часто ініціалізація передбачає виділення пам'яті. Конструктор викликається кожного разу як створюється новий об'єкт даного класу. Конструктор з одним вхідним параметром може виконувати приведення типів якщо тільки не задано ключове слово `explicit`. Деструктор (destructor) — це метод, задача якого полягає у тому, щоб завершити існування змінної класу. Зазвичай це передбачає звільнення зайнятої пам'яті. Конструктор оголошується як метод з ім'ям ідентичним імені класу, а деструктор аналогічно але першою літерою ставиться символ «тильда» (~).

```
class my_string {
public:
        explicit my_string(int n) // конструктор
        { s = new char[n+1]; lqn = n; }
        ~my_string() { delete[] s; } // деструктор
        void assign(const char* str);
        int length() const { return len; }
        void print() const
        { cout << s << endl; }
private:
        char s[225];
        int len;
};
}
```

Тепер у основній програмі можна використовувати клас наступним чином:

```
my_string a(7);
a.assign("Привіт!");
a.print();
```

Для одного й того ж класу можна задати декілька конструкторів, тоді компілятор буде обирати з них той який підходить по вхідним параметрам. Наприклад ми можемо додати ще один конструктор наступного вигляду:

```
my_string (const char* p) {
        len = strlen(p);
        s = new char[len+1];
        strcpy(s,p);
}
```

Тепер об'єкт можна оголошувати і так:

```
my_string a("Привіт!");
```

Взагалі рекомендується також передбачити ще й конструктор без аргументів, але не будемо зосереджувати на цьому увагу.

5.3 Перевантажування

Перевантажуванням (overloading) називають практику призначення декількох значень оператору чи функції. Наприклад можна створити декілька методів з однаковим ім'ям але з різним набором параметрів і компілятор автоматично буде обирати необхідний метод в залежності від того з якими параметрами даний метод викликається. Наприклад до метода `print` без параметрів можна додати ще й такий:

```
void print(int n) const {
        for(int i=0;i<n;++i)
                cout << s << endl;
}
```

Тепер якщо ми напишемо `st.print()`, то рядок буде надруковано як і раніше один раз, а якщо напишемо `st.print(5)`, то рядок надрукується п'ять разів.

Перевантажувати можна не лише методи але й оператори. Наприклад ми можемо навчити стандартний оператор додавання `+` виконувати операцію конкатенації двох рядків типу `my_string`.

```
my_string& operator+(const my_string& a,
                     const my_string& b) {
        my_string* temp=new my_string(a.len+b.len);
        strcpy(temp->s,a.s);
        strcat(temp->s,b.s);
        return *temp;
}
```

Тепер можна використовувати наступні конструкції:

```
my_string one("Hello "), two("C++!");
my_string both;
both = one + two;
// тепер both містить рядок "Hello C++!"
```

Використання `&` у описі оператору означає, що параметри передаються за посиланням, а ключове слово `const` вказує, що їх не можна змінювати. Це стандартна форма запису для бінарних операторів. Оператор `->` використовується замість крапки у тому випадку коли об'єкт створюється на етапі виконання (оператор `new`), а не компіляції.

5.4 Шаблони

Шаблони використовуються у C++ для забезпечення узагальненого програмування. Ну наприклад ви пишете клас який би інкапсулював у собі функціональність стеку даних, але при цьому ви можете не обмежуватися реалізацією стеку для якогось одного типу даних, а замість цього можете оперувати з якимось абстрактним типом, назвемо його наприклад `TYPE`. Тоді все, що потрібно то це модифікувати оголошення класу наступним чином:

```
template <class TYPE>
class stack {
        . . . . .
private:
        . . . . .
        TYPE* s;
};
```

А об'єкт такого класу створюється наступним чином:

```
stack<char> stk_ch; // стек символів
stack<double> stk_ch; // стек дробових чисел
```

Таким чином узагальнене програмування дозволяє суттєво економити об'єм коду, бо написаний одного разу код можна використовувати багато разів.

5.5 Наслідування

Особливістю ООП є надання переваги повторному використанню коду за допомогою механізму наслідування. Наприклад у нас є клас `fruit` який описує якийсь узагальнений фрукт, такий клас називається базовим. Тепер ми можемо створити клас `apple` породжений від класу `fruit` і який окрім вже описаних у базовому класі властивостей фрукта інкапсулює ще й оригінальні властивості яблука. Далі ми можемо від того ж класу `fruit` породити клас `pear` який буде описувати груші, а від класу `apple` породити класи `macintosh` та `crispin` які додатково ще будуть описувати властивості різних сортів яблук. Ось таким чином будується ієрархія класів. Щоб вказати компілятору, що даний клас породжений від іншого класу потрібно після імені класу дописати : `public` та ім'я базового класу.

```
class student {
        . . . . .
};
```

```
class grad_student : public student {
        . . . . .
};
```

При цьому всі методи та змінні базового класу автоматично будуть представлені і у новому класі, дублювати при описі їх не потрібно. Натомість можна додавати додаткові змінні та методи. Допускається також множинне наслідування, тоді імена базових класів відокремлюються комами, але множинного наслідування краще по можливості не допускати.

5.6 Поліморфізм

Під поліморфізмом розуміють ситуацію коли одна й та ж функція чи оператор мають декілька варіантів. Наприклад стандартний оператор ділення у C++ є поліморфним, бо коли він застосовується до дробових чисел, то результатом його виконання також буде дробове число, а якщо діляться цілі числа, то і результат буде цілим числом навіть якщо для цього доведеться відкинути дробову частину. Якщо говорити про класи, то тут під поліморфізмом розуміють ситуацію коли у ієрархії класів зустрічаються однакові методи, тоді при виконанні програми буде обиратися необхідний. Ну наприклад ми маємо клас `shape` та породжені від нього класи `rect` та `circle`, і у кожному з них є метод `draw` який служить для виведення на екран даної фігури.

```
class shape {
        . . . . .
public:
        virtual void draw()
        { draw_nothing; }
};

class rect : public shape {
        . . . . .
public:
        virtual void draw()
        { draw_rect; }
};

class circle : public shape {
        . . . . .
public:
        virtual void draw()
        { draw_circle; }
};
```

Директива `virtual` задає віртуальний метод (як і у Pascal), це означає, що даний метод тепер буде поліморфним. Що це означає можна показати на прикладі:

```
shape* s1, s2;
s1 = new rect();
s2 = new circle();
s1->draw(); // Малюємо прямокутник
s2->draw(); // Малюємо коло
```

І ось тут ми бачимо у дії один з найважливіших принципів ООП. Поліморфізм дозволяє звертаючись до одного й того ж методу базового класу отримувати різний результат в залежності від того якого ж в дійсності класу є об'єкт. Насправді, ми можемо навіть про це не знати, але грамотно побудувавши ієрархію класів ми можемо вільно оперувати об'єктами різних класів навіть про це не думаючи. Потрібно тільки звернути увагу, що якби ми не використали директиву `virtual` у базовому класі, то скористатися поліморфізмом ми б не змогли, в обох би випадках викликався б метод `draw` базового класу `shape`. Активне використання поліморфізму часто призводить до появи у ієрархії так званих абстрактних класів з абстрактними методами. Такі класи не інкапсулюють у себе жодну функціональність, а просто є шаблонами для створення на їх базі нових класів і для того щоб можна було до них звертатися.

5.7 Особливі ситуації

Дуже важливим моментом при написанні якісних програм є передбачення помилок які можуть виникати внаслідок дій користувача і коректне відновлення роботи після обробки такої ситуації. Для обробки помилок у C++ передбачено термін «особлива ситуація» (exception). Для слідкування за такими ситуаціями служить блок `try`, обробники задаються блоками `catch`, а безпосередньо особлива ситуація генерується оператором `throw`. Ну наприклад ми маємо клас `vect`, його конструктор намагається при створенні об'єкту виділити пам'ять і у випадку якщо це не вдається генерує особливу ситуацію з допомогою `throw`. Оскільки об'єкти `a` та `b` класу `vect` створюються у межах оператору `try`, то у разі виникнення такі ситуації перехоплюють і далі управління передається відповідному по набору вхідних параметрів блоку `catch`.

```
vect::vect (int n) : size (n) {
        if(n < 1)
                throw(n);
        p = new int[n];
        if(p == 0)
                throw("Немає пам'яті");
}

void g() {
        try {
                vect a(n), b(n);
                . . .
        }
        // невірний розмір
        catch(int n) { . . . }
        // немає пам'яті
        catch(char* error) { . . . }
}
```

Таким чином у разі виникнення помилки у програми є шанс прийняти необхідні міри і спробувати повторити заплановану дію. Потрібно розуміти, що слідкування за особливими ситуаціями не ліквідовує помилки, а лише дозволяє їх локалізувати і захищає від аварійного завершення програми.

6 Бібліотека шаблонів STL

6.1 Контейнери

Стандартна бібліотека шаблонів (STL, Standard Template Library) є стандартною бібліотекою C++ яка надає можливості узагальненого програмування. Трьома її компонентами є: контейнери, ітератори та алгоритми. Бібліотеку побудовано з використанням шаблонів тому її компоненти можуть працювати з будь-якими типами даних.

Контейнери діляться на дві основні родини: послідовні контейнери та асоціативні контейнери. Послідовні контейнери включають вектори (vectors), списки (lists) та двосторонні черги (deques). Такі контейнери працюють з послідовностями елементів. Асоціативні контейнери включають множини (sets), мультимножини (multisets), відображення (maps) та мультивідображення (multimaps), і містять ключі для пошуку елементів. Усі варіанти контейнерів мають схожий інтерфейс.

Вектор задає звичайний динамічний масив. Для створення нового вектору використовується конструкція `vector<тип> змінна`. Метод `push_back` служить для додавання елемента у кінець вектору. Метод `size` повертає розмір векторі, а використовуючи квадратні дужки до елементів вектору можна звертатися так же само як і до елементів масиву. Потрібно ще звернути увагу на поняття ітератора. Його можна уявити як удосконалений вказівник, якщо проаналізувати текст прикладу, то можна легко зрозуміти принципи роботи з ітераторами і без додаткових пояснень.

```
#include <iostream>
#include <vector> // потрібно підключити
#include <string>

using namespace std;

main()
{
    vector<string> SS;

    SS.push_back("Число 10");
    SS.push_back("Число 20");
    SS.push_back("Число 30");

    cout << "Виводимо вектор:" << endl;
```

```
    int ii;
    for(ii=0; ii < SS.size(); ii++)
    {
        cout << SS[ii] << endl;
    }

    cout << endl << "Ітератор:" << endl;

    vector<string>::const_iterator cii;
    for(cii=SS.begin(); cii!=SS.end(); cii++)
    {
        cout << *cii << endl;
    }

    cout << endl << "Зворотній ітератор:" << endl;

    vector<string>::reverse_iterator rii;
    for(rii=SS.rbegin(); rii!=SS.rend(); ++rii)
    {
        cout << *rii << endl;
    }

    cout << endl << "Приклади виводу:" << endl;

    cout << SS.size() << endl;
    cout << SS[2] << endl;

    swap(SS[0], SS[2]);
    cout << SS[2] << endl;
}
```

З допомогою векторів можна задавати і багатовимірні масиви, наприклад матриці.

```
vector<vector<int>> vI2Matrix(3,vector<int>(2,0));

vI2Matrix[0][0] = 0;
vI2Matrix[0][1] = 1;
vI2Matrix[1][0] = 10;
vI2Matrix[1][1] = 11;
vI2Matrix[2][0] = 20;
vI2Matrix[2][1] = 21;
```

Список є схожим на вектор але дозволяє вставляти та видаляти об'єкти у будь-якому порядку.

```
#include <iostream>
#include <list>
using namespace std;

main()
{
    list<int> L;
    L.push_back(0); // Додаємо елемент в кінець
    // Вставляємо елемент на початку
    L.push_front(0);
    // Вставляємо "2" перед першим елементом
    L.insert(++L.begin(),2);

    L.push_back(5);
    L.push_back(6);

    list<int>::iterator i;

    for(i=L.begin(); i != L.end(); ++i)
        cout << *i << " ";
    cout << endl;
    return 0;
}
```

6.2 Асоціативні контейнери

Асоціативні контейнери містять доступні по ключу елементи та відношення `Compare`, яке є порівняльним об'єктом асоціативного контейнеру.

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main() {
        map<string,int,less<string>> name_age;

        name_age["Гаррі"]=12;
        name_age["Фред"]=14;
        name_age["Персі"]=17;
        cout << "Гаррі " << name_age["Гаррі"]
             << "років." << endl;
}
```

Тут відображення `name_age` є асоціативним масивом у якому ключом є тип `string`. Об'єкт `Compare` — це `less<string>`.

Більше про бібліотеку шаблонів STL можна взнати з відповідної літератури. Це досить широка тема яку навряд чи можна освітити стисло.

7 Інтегровані середовища для C++

Microsoft Visual C++ — потужна комерційна система розробки на C++ для операційних систем родини Windows розроблена фірмою Microsoft. Після версії 6.0 Microsoft провела своєрідну революцію з допомогою власної технології .NET. З цього моменту намітився відхід від вже морально застарілої бібліотеки MFC до використання WinForms. Працює лише на платформі Windows.

Borland C++ Builder — комерційне інтегроване середовище фірми CodeGear (бувша Borland) для розробки програм на C++. Сильно схожий на Delphi за винятком власне мови програмування. Працює на платформі Windows.

Eclipse — система розробки з відкритим кодом спонсорована фірмою IBM. Створювалася як середовище розробника на Java але завдяки гнучкій архітектурі з підтримкою плагінів дозволяє розробляти проекти на різних мовах, у тому числі на C++. Для компіляції використовує зовнішній компілятор, наприклад GCC. Підтримує Linux, Windows, MacOS X, Solaris, AIX, QNX.

KDevelop — базове інтегроване середовище для KDE на платформі UNIX-like з відкритим кодом. Система орієнтується на використання бібліотек Qt фірми Trolltech для програмування графічного інтерфейсу. У якості компілятора використовується зовнішній, зазвичай GCC. Підтримує практично всі UNIX-like системи, хоча більше орієнтується на Linux.

Anjuta DevStudio — інтегроване середовище з відкритим кодом, що розробляється у рамках проекту Gnome. Для побудови графічного інтерфейсу система схиляється до бібліотек GTK+. Використовує зовнішній компілятор GCC. Підтримує практично всі UNIX-like системи.

Набагато більший перелік різноманітних інтегрованих середовищ розробки для C++ можна знайти на сайті Wikipedia: *http://en.wikipedia.org/wiki/Category:Integrated_development_environments*

Web-ресурси

1. *http://www.cppreference.com/*
2. *http://www.cprogramming.com/*
3. *http://www.cplusplus.com/doc/tutorial/*
4. *http://www.yolinux.com/TUTORIALS/*

Література

1. Айра Пол. Объектно-ориентированное программирование на C++. 1999.
2. Robert L Wood. C Programming for Scientists & Engineers. 2002.
3. Charles Wright. Visual C++ 6 for Dummies. Quick Reference.