Лекція 1

# Парадигми програмування. Елементи програм C++.

***В цій лекції...***



## 1.1. Парадигми програмування

За означенням Бьорна Страуструпа, C++ — *це мова програмування загального призначення із ухильністю в бік системного програмування, що підтримує абстракцію даних, а також об'єктно-орієнтоване і узагальнене програмування.*

Ця мова охоплює чотири парадигми програмування — процедурну, об'єктну, об'єктно-орієнтовну і узагальнену.

**Процедурне програмування** базується на такому принципі.

| *Програміст повинен визначити потрібні процедури і використати найкращі алгоритми.* |
|---|

Головний акцент в процедурному програмуванні робиться на обробці, тобто на *алгоритмі*. Основним механізмом процедурного програмування є функція. Процес розв'язання задачі в рамках процедурної парадигми називається *функціональною абстракцією*. Від дозволяє розробляти функції відносно ізольовано одну від одної, налагоджуючи зв'язки між ними за допомогою механізму передачі параметрів і повертання результатів.

Припустимо, що ми розробляємо програму, яка сортує список. Ми подаємо список у вигляді масиву і пишемо функцію, яка сортує його методом швидкого сортування. Дотримуючись принципу процедурного програмування, визначимо потрібні процедури: ввід — швидке сортування — вивід. Все, що нам потрібно, — визначити спосіб вводу, параметри функції сортування і вигляд виводу. Якщо нам знадобиться в подальшому змінити програму, наприклад, застосувати кращий метод сортування, її доведеться перекомпілювати заново.

**Модульне, або об'єктне програмування** більше уваги приділяє не проектуванню процедур, а організації структур даних. Це пов'язано із збільшенням розмірів програм. Принцип модульного программування формулюється так.

| *Програміст повинен вирішити, які потрібні модулі, а потім розбити їх так, щоб приховати дані в цих модулях.* |
|---|

Цей принцип також називається *принципом приховання даних*, або *інкапсуляцією*. Модулем називається набір зв'язаних процедур разом з даними, що піддаються обробці. Основним механізмом модульного програмування є простори імен і абстракція даних.

Абстракція даних зосереджує увагу на призначенні операцій, а не на деталях їх виконання. Інші модулі програми "знають", *що* робить та чи інша операція, але не "знають", *як* саме вона це робить. Абстракція даних використовує два поняття: *абстрактний тип*, тобто сукупність даних і операцій над ними, і *структуру даних*, тобто конструкцію, що визначена в мові програмування для зберігання набору даних.

Повертаючись до попереднього прикладу, уявімо, що наша програма — маленька частина великого проекту. Природно зробити так, щоб її модифікація мінімально впливала на решту модулів. Для цього потрібна її максимальна ізоляція і захист. В такому випадку, ми можемо приховати алгоритм в модулі, а сам модуль розмістити в бібліотеці, де його легко замінити. Основним механізмом модульного програмування і абстракції даних є типи, визначені користувачем — *класи*.

Імперативне і об'єктне програмування не відрізняються підвищеною гнучкістю. Чим більшими стають програми, тим складнішими стають способи узгодження взаємозв'язку між їхніми компонентами. Для організації розробки великих програм була запропонована **об'єктно-орієнтована парадигма**, яка доповнює інкапсуляцію ще двома принципами — успадкуванням і поліморфізмом.

*Програміст повинен визначити, які потрібні класи, описати повний набір операцій для кожного класу і виразити спільність через успадкування.*

- Інкапсуляція полягає в тому, що об'єкти об'єднують дані і операції в одне ціле.
- Успадкування дозволяє класам набувати властивості інших класів.
- Поліморфізм дозволяє об'єктам обирати відповідні операції під час виконання програми.

Механізмом успадкування в мові C++ є *ієрархія класів*, а поліморфізм забезпечується *перевантаженням функцій і операторів* (статичний поліморфізм) і віртуальними класами (динамічний поліморфізм).

Якщо алгоритм можна подати незалежно від деталей його організації, стає можливою **парадигма узагальненого програмування**.

*Програміст повинен визначити, які потрібні алгоритми, і параметризувати їх так, щоб вони могли працювати із більшістю потрібних типів і структур даних*

Механізм, яки забезпечує узагальнене програмування, складається із *шаблонних функцій і класів* — особливих конструкцій мови C++, які параметризуються типами даних. Використовуючи цей підхід, ми можемо розробити абстрактний список, що може містити дані, тип яких заданий за допомогою шаблонних параметрів. Велика кількість подібних абстрактних типів (списків, черг тощо) і узагальнених алгоритмів (сортування, перестановок тощо) були розміщені в стандартній бібліотеці шаблонів STL, яка стала невід'ємною частиною стандарту мови C++.

## 1.2. Алфавіт мови

Вивчення будь-якої іноземної мови починається з алфавіту. Саме з елементів алфавіту складаються слова, конструкції і вирази, що у підсумку утворять осмислений текст. Застосуємо цей принцип і до мови C++.

Стандарт мови C++ розділяє алфавіт на чотири частини: *основний набір текстових символів* (basic source character set), *множину універсальних символів* (universal character names), *основний набір керуючих символів* (basic execution character set) і *основний набір розширених керуючих символів* (basic execution wide-character set). У свою чергу, на їхній основі створюється *набір керуючих символів* (execution character set) і *набір розширених керуючих символів* (basic execution wide-character set). Розглянемо кожний з них докладніше.

*Основний набір текстових символів* складається з 96 символів: пробілу, чотирьох керуючих символів (горизонтальна табуляція (`\t`), вертикальна табуляція (`\v`), переведення сторінки (`\f`) і перехід на новий рядок (`\n`)) і символів ASCII (91 символ), а саме:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
```

Як правило, цих символів цілком достатньо для створення практично будь-якої програми.

*Універсальні символи* кодуються послідовностями з чотирьох чи восьми шістнадцятирічних цифр, що називаються *універсальними іменами* (universal-name-symbol).

```
/Uhhhhhhhhh
/uhhhh
```

Тут символ `h` позначає шістнадцятирічну цифру.

Крім символів, що використовуються для кодування елементів програм (ідентифікаторів, коментарів, літералів тощо), у мові C++ є так звані *керуючі символи*, за допомогою яких можна виконувати невелику кількість операцій, пов'язаних, як правило, із введенням і виведенням даних.

В *основний набір керуючих символів* входить основний набір текстових символів, до якого додані символи сигналу (`\a`), повернення на одну позицію (`\b`), повернення в початок рядка (`\r`) і нульового символу (`\0`).

*Основний набір розширених керуючих символів* складається з основного набору текстових символів, закодованих за допомогою універсальних імен, до якого додані символи сигналу (`\a`), повернення на одну позицію (`\b`), повернення в початок рядка (`\r`) і розширеного нульового символу (`\u0000`).

На основі цих двох наборів формується *набір керуючих символів* і *набір розширених керуючих символів*, що містять букви національних алфавітів і керуючі символи. Їхня реалізація залежить від конкретного компілятора.

## 1.3. Ключові слова

Дамо декілька необхідних означень. *Ім'я* — це рядок символів, що використовується для позначення якого-небудь елемента програми (змінної, константи, функції, класу тощо). *Ключовим словом* називається рядок символів, що має особливе значення лише у визначених ситуаціях. В інших випадках ключові слова не відрізняються від імен. Наприклад, у ранніх версіях мови FORTRAN слово REAL могло використовуватися і як ключове, визначаючи тип даних, і як ім'я змінної. *Зарезервованим словом* називається рядок символів, що в жодному разі не можна використовувати як ім'я. Як бачимо, поняття ключового слова є більш широким. Однак у мові C++ під ключовим словом звичайно розуміють саме зарезервоване, хоча це і не відповідає їх точному визначенню.

Стандарт мови C++ містить 63 ключові слова.

```
asm          do             if                 return       typedef
auto         double         inline             short        typeid
bool         dynamic_cast   int                signed       typename
break        else           long               sizeof       union
case         enum           mutable            static       unsigned
catch        explicit       namespace          static_cast  using
char         export         new                struct       virtual
class        extern         operator           switch       void
const        false          private            template     volatile
const_cast   float          protected          this         wchar_t
continue     for            public             throw        while
default      friend         register           true
delete       goto           reinterpret_cast   try
```

Ключові слова можна розподілити на наступні категорії: оператори, специфікатори, модифікатори і кваліфікатори.

У мові C++ оператор — украй багатозначне слово. Як правило, їм називають символи операцій (+, * і тощо). Крім того, це слово традиційне використовується для позначення окремих команд, виконуваних програмою (наприклад, оператор присвоювання, оператор циклу тощо). У літературі по C++ можна зустріти спроби усунути неоднозначність цього слова. Для цього слово “оператор” резервують для позначення операцій, а команди програми називають “інструкціями”. Суперечка на цю тему неминуче набуває схоластичного характеру — адже перевантаження просто переміщується зі слова “оператор” на слово “інструкція”, що уже зв'язана з поняттям “машинна операція”. Утім, по контексту завжди можна визначити, про що йде мова — про інструкцію програми чи операцію. У нашому курсі ми продовжуємо дотримуватись історично сформованої традиції і називаємо оператором як позначення операції, так і конкретну інструкцію програми. Читачі легко зрозуміють, про що йде мова в кожному конкретному висловленні. Існує і третє значення слова “оператор” — ключове слово, що є основним елементом керуючої конструкції (`operator+`, `operator[]` тощо).

Ключові слова `break`, `case`, `catch`, `continue`, `delete`, `do`, `else`, `export`, `for`, `goto`, `if`, `namespace`, `new`, `reinterpret_cast`, `return`, `static_cast`, `switch`, `throw`, `try`, `typedef`, `typeid`, `using` і `while` є *операторами*.

Деякі ключові слова — це імена *типів*, тобто діапазону значень, що можуть набувати змінні, і набору операцій, які можна виконувати над ними. Ключові слова, що визначають тип і спосіб збереження змінної чи функції, називаються *специфікаторами*. До них відносяться *специфікатори основних типів* `bool`, `char`, `double`, `float`, `int`, `long`, `short`, `signed`, `unsigned` і `void`, *специфікатори складних типів* `class`, `struct` і `union`, *специфікатори збереження* `auto`, `extern`, `register` і `static`, *специфікатори доступу* `public`, `private` і `protected`, *специфікатори функцій* `inline` і `virtual`, а також *специфікатори дружніх класів і функцій* `friend` і *специфікатор шаблонів* `template`.

У визначених ситуаціях типи уточнюються за допомогою *модифікаторів* `const` і `volatile`, іменованих також *cv*-кваліфікаторами. (Не шукайте в цій назві зашифрованого змісту: *cv* — це проста абревіатура `const` і `volatile`.)

Кожен компілятор розширює цей набір ключових слів власними словами.

## 1.4. Ідентифікатори

*Ідентифікатор* — це синонім слова “ім'я”. Він зв'язується з конкретним елементом програми (змінною, константою, функцією тощо) і надалі використовується для їхнього позначення.

- Ідентифікатори можуть складатися тільки з букв, цифр і символу підкреслення.
- Ідентифікатори повинні починатися з букви або символу підкреслення.
- Ключові слова не можуть бути ідентифікаторами.
- Як ідентифікатори не можна використовувати імена стандартних функцій (`sin`, `cos` тощо).
- У стандарті мови C++ не встановлена гранична довжина ідентифікатора.
- Компілятор розрізняє прописні і малі літери, що входять до складу ідентифікатора.

Прокоментуємо кожне з цих обмежень.

Те, що ідентифікатор повинний складатися з букв і цифр, цілком природно. Дуже часто початківці називають змінні найпростішими іменами, накшталт `a1`, `i2`, `f4`. У невеликих програмах з очевидним змістом це цілком разумно. Однак варто зауважити, що досить швидко вони самі забувають, що саме мали на увазі, коли писали ідентифікатор `k3`. Тому, коли автори книг радять своїм читачам давати елементам програми осмислені імена, маються на увазі цілі фрагменти фраз, наприклад, `sum_of_integer_numbers`. Однак тут легко впасти в іншу крайність, намагаючись занадто докладно описати зміст ідентифікатора.

Ще одне утруднення зв'язане з вибором зовнішнього вигляду ідентифікатора. Існують три можливості: використовувати для зв'язування слів символ підкреслення, “верблюжий горб” чи так називаний угорський стиль. Першу можливість ми вже продемонстрували. Це дуже зручно і наочно. Ідентифікатор “верблюжий горб” виглядає так: `sumOfIntegerNumbers`. Він починається з малої літери, а кожне наступне слово починається з прописної. Що вибрати — справа смаку. Іноді люди погано розрізняють символи підкреслення і користаються другою можливістю, а іноді в них рябить в очах від чергування прописних і малих літер, і тоді вони застосовують символи підкреслення.

Угорський стиль полягає в тому, що кожна змінна повинна починатися з букви, що вказує на її тип. Наприклад, цілочисельні змінні повинні починатися з букви `i` (integer), а дійсні змінні — з букви `f` (float) чи `d` (double). Ця угода дуже нагадує правила, прийняті в мові FORTRAN, у якому всі змінні, що починаються з букв I, J, K, L, M чи N, за замовчуванням вважаються цілочисельними. До угорського стилю можна віднести також звичку програмістів починати імена класів з букви `T` (type).

Як правило, з підкреслення починаються ключові слова, зарезервовані компіляторами. Це дозволяє відразу розпізнати їх серед інших ідентифікаторів. У принципі, гарний ідентифікатор повинний не тільки описувати призначення елемента програми, але і своїм видом указувати на його природу. Простіше кажучи, по зовнішньому вигляду ідентифікатора програміст повинен легко розпізнавати змінні, константи, функції і класи. Наприклад, Герб Саттер (Herb Sutter) запропонував прийняти наступні угоди.

- Класи, функції і перелічувані типи необхідно називати так, щоб усі слова, що утворюють їхні імена, починалися з прописної букви.: `Stack`, `Queue`, `QueueWithPriority` тощо
- Імена змінних і перелічуваних значень повинні починатися з малої літери, наприклад: `stack`, `queue` і т.д.
- Імена змінних-членів повинні закінчуватися символом підкреслення: `x_`, `sum_` тощо

Крім того, у програмах на мовах C и C++ вже давно негласно прийнято при найменуванні змінних застосовувати малі літери (`variable`), а при назві констант — прописні (`PI`). Це полегшує читання програм, оскільки при цьому не потрібно повертатися назад, уточнюючи зміст ідентифікатора.

Отже, якщо ідентифікатор починається символом підкреслення — перед вами ключове слово, зарезервоване компілятором, а якщо ім'я змінної чи функції завершується символом підкреслення — перед вами член класу.

Варто підкреслити, що все сказане про зовнішній вигляд ідентифікатора — це не правила, а всього лише загальноприйняті угоди. Якщо хочете, можна застосувати власний стиль, але, з огляду на те, що програмування в даний час прийняло конвеєрний і колективний характер, це може викликати непорозуміння.

## 1.5. Літерали

Поряд із змінними невід'ємною частиною програм є літерали, тобто постійні значення, наприклад 3.141592 (число $\pi$) чи 2.71828 (константа Ейлера — число *e)*. Літерал може бути цілочисельним, символьним, дійсним, рядковим і булевим.

*Десятковий цілочисельний літерал* виглядає приблизно так.

```
125   Цілочисельний літерал
125u  Цілочисельний літерал без знака
125U  Цілочисельний літерал без знака
```

```
125l  Розширений цілочисельний літерал
125L  Розширений цілочисельний літерал
```

Крім того, букви, що уточнюють тип літерала, можна комбінувати. Наприклад, розширений цілочисельний літерал без знака можна задати в такий спосіб.

```
125ul
125UL
125u
125Ul
```

*Вісімковий цілочисельний літерал* задається так: перед послідовністю вісімкових цифр (`0-7`) ставиться цифра `0`.

```
0125      Число 85
01        Число 1
031455    Число 13101
```

*Шістнадцятковий цілочисельний літерал* складається з префікса `0x` і напівбайтів, заданих шістнадцатирічними цифрами (`0-9`, `A-F`). Кількість напівбайтів, що вказуються, може коливатися від одного до чотирьох.

```
 0x125    Число 293
 0x1      Число 1
 0x3455   Число 13397
 0x34567  Число 214375
 0xf123   Число 61731
 0xf123U  Число 61731
-0xffff   Число -65535
 0xffff   Число 65535
 0xabcd   Число 43981
```

*Дійсний літерал* складається з десяткового числа, записаного у фіксованому чи науковому форматі, за яким можуть слідувати букви `f` чи `F`, а також `l` чи `L` (комбінувати букви `f` і `l`, а також `F` і `L` заборонено). Суфікс f означає тип `float`, а `L` — `long double`. Вони розрізняються лише діапазонами змінювання дійсного числа і кількістю знаків, який можна вважати вірогідними.

```
5.0       Число 5.000000
6.123F    Число 6.123000
7.432f    Число 7.432000
5.0l      Число 5.000000
12.4569L  Число 12.45690
15.05lf   Число 15.05000
```

Зважте на те, що більше шести знаків після десяткової крапки для літерала типу `float` задавати не варто — результат все рівно буде містити лише шістьох значущих цифр, причому остання буде заокруглена, навіть якщо вказати суфікс `L`. Інші цифри являють собою просто “сміття”. Для літерала типу `double` після десяткової крапки вдається удержати дев'ятнадцять знаків.

```
float x = 0.123456789             x = 0.123456791
float x = 0.123456789L            x = 0.123456791
double x = 0.123456789123456789L  x = 0.12345679123456781
double x = 1.234e+02              x = 123.4
double x = 0.12345678912E-02L     x = 0.0012345679120000008
```

Як бачите, задаючи літерали, потрібно стежити за типом змінної і пам'ятати про кількість цифр після десяткової крапки, яким можна довіряти.

*Символьні літерали* дуже широко поширені. Як правило, символьним літералом є символ ASCII, взятий в одинарні лапки, наприклад `'a'`, а також деякі керуючі символи, такі як `'\n'`. Розширені літерали починаються з букви `L`.

```
'1'       Символ 1 (ASCII-код дорівнює 49)
'A'       Символ A (ASCII-код дорівнює 65)
'a'       Символ a (ASCII-код дорівнює 97)
'\0'      Нульовий символ — ознака кінця рядка
'\n'      Символ переходу на новий рядок
L'b'      Розширений літерал b
```

*Керуючі символи*, чи *escape-послідовності*, грають особливо важливу роль при виводі даних, тому перелічимо їх окремо, указавши відповідні восьмеричний і шістнадцятирічний коди.

```
'\a'  '\07'  '\x07'    Звуковий сигнал
'\b'  '\08'  '\x08     Повернення на одну позицію назад
'\f'  '\14'  '\x0c     Прогін сторінки
'\n'  '\12'  '\x0a'    Перехід на новий рядок
'\r'  '\15'  '\x0d'    Повернення каретки
'\t'  '\11'  '\x09'    Горизонтальна табуляція
'\v'  '\13'  '\x0b'    Вертикальна табуляція
'\\'  '\134' '\x5c'    Зворотна коса риса
'\''  '\47'  '\x27'    Апостроф
'\"'  '\42'  '\x22'    Подвійні лапки
'\?'  '\77'  '\x77'    Знак питання
```

Кожна з escape-послідовностей являє собою одиночний символ, хоча записується за допомогою зворотної косої риси, а також букв і вісімкових чи шістнадцяткових цифр. Їх можна використовувати і як елементи форматованого виводу, і як самостійні символи.

```
printf("Результат = %d \n",sum);
char a = '\x77'; // Знак питання
char b = '\42';  // Подвійні лапки
```

*Рядковий літерал* являє собою послідовність символів, взяту в подвійні лапки (наприклад, "рядковий літерал"). Їх можна комбінувати із символьними літералами, забезпечуючи правильне форматування тексту.

```
printf("Рядок \n Наступний рядок");
```

Варто звернути увагу на одну корисну особливість строкових літералів — якщо два строкових літерала записати разом, вони автоматично поєднуються в одне ціле. Цю властивість можна використовувати для організації рядків форматування при виклику функції `printf`. Наприклад, наступні оператори виводять однакові рядки.

```
printf("Перша частина рядка ... "
       "друга частина рядка ... ");

printf("Перша частина рядка ... друга частина рядка ...");
```

*Булевими літералами* є ключові слова `true` і `false`.

## 1.6. Оператори і знаки пунктуації

Оператори в мові C++ позначають різні операції: арифметичні, логічні, поразрядні і порівняння. Пізніше ми розглянемо їх детально, а поки що нас цікавить лише, як вони позначаються.

```
+           -           *           /           %           ^
!           =           <           >           +=          -=
^=          &=          |=          <<          >>          <<=
<=          >=          &&          ||          ++          --
( )         [ ]         new         delete      &           |
~           *=          /=          %=          >>=         ==
!=          ,           ->          ->*         .           .*
?:
```

*Знаками пунктуації* в мові C++ називаються символи, що мають синтаксичний і семантичний зміст, але не позначають самостійні оператори. До них належать круглі, квадратні і фігурні дужки. Варто підкреслити, що знаки пунктуації `()`, `[]` і `{}` повинні використовуватися винятково парами. Порушення балансу дужок є однієї з найбільш грубих помилок.

## 1.7. Головний модуль

Процедура створення програми залежить від текстового редактора, компілятора й операційної системи. Однак у цілому її основні етапи залишаються незмінними.

- Створення вихідного коду
- Компіляція
- Редагування зв'язків
- Виконання

Перший етап — створення тексту програми — практично не залежить від компілятора й операційної системи. Отже, для створення і редагування тексту програми можна застосовувати будь-який доступний текстовий редактор. Однак оскільки цей текст має особливе призначення, на файл, що містить вихідний код програми, накладаються деякі обмеження. По-перше, цей файл повинний бути текстовим, тобто повинний

містити ASCII-коди. Жодні інші формати файлів не допускаються. По-друге, файл, що містить вихідний код, повинний мати розширення, передбачене компілятором (`.cpp`, `.cp` чи `.cxx`). Розглянемо наступний приклад.

**Найпростіша програма**

```
int main()
{
    return 0;
}
```

Перший рядок містить заголовок функції `main()`, що є невід'ємною частиною будь-якої програми. Після імені функції `main` указується пара дужок, усередині яких можуть задаватися аргументи командного рядка, але як правило, ці дужки залишаються порожніми. Пізніше ми розберемося, коли і як варто задавати аргументи функції `main()`, а поки зосередимося на інших питаннях. Тіло функції `main()` починається і закінчується фігурними дужками. Усередині цих дужок перелічуються оператори, що, власне, і визначають зміст програми. У даному випадку ця програма після запуску відразу повертає керування операційній системі за допомогою оператора `return`. Функція `main()` повертає ціле число, що аналізується операційною системою. Число 0 означає, що виконання програми завершилося успішно, інші значення кодують різні помилки. Слід зазначити, що донедавна передбачалося, що функція `main()` повертає ціле число за замовчуванням, тобто вказувати перед її ім'ям тип `int` було не обов'язково. Однак у сучасних компіляторах, що найбільше повно реалізують стандарт мови C++, це правило скасоване.

Якщо програмісту не важливо, як завершена програма, і він не передбачає визначеної реакції операційної системи, перед ім'ям `main()` варто поставити ключове слово `void`. Для того щоб повернути керування операційній системі можна або виконати оператор `return`, або дочекатися, поки потік керування досягне останньої фігурної дужки.

У першому варіанті програма може виглядати так.

```
void main()
{
     return; // Повернення керування
 }
```

В другому випадку програма стає ще простіше.

```
void main()
{
 } // Повернення керування
```

Зверніть увагу, що правило "`int` за замовчуванням" сучасними компіляторами не підтримується, отже, компілятор у цьому випадку видає повідомлення про помилку.

```
main()   // Помилка - не зазначений тип значення, що повертається
{
}
```

## 1.8. Оголошення і визначення

Зрозуміло, наведена вище найпростіша програма не має практичного змісту. Адже, як правило, програма повинна реалізовувати деякий алгоритм, використовуючи синтаксичні і семантичні особливості мови. Помітимо, що будь-який алгоритм можна сформулювати у виді послідовності операцій, виконуваних над деякими сутностями чи об'єктами. Цими сутностями можуть бути різні числа і структури даних (масиви, списки, дерева тощо). Яка б ні була природа алгоритму, ця особливість залишається незмінною. Таким чином, необхідно "населити" програму сутностями й описати осмислені дії над ними.

Виникає питання, що вважається сутністю в мові C++, інакше кажучи, що може містити програма. У відповідності зі стандартом, сутністю вважається значення, об'єкт, підоб'єкт, підоб'єкт базового класу, елемент масиву, змінна, функція, екземпляр функції, перерахування, тип, член класу, шаблон і простір імен. Багато хто з перерахованих понять нам поки що не знайомі, але усі вони мають одну загальну властивість — будь-яка сутність має *ім'я*, що є її *ідентифікатором*.

Основне правило мови C++ говорить: *будь-яка сутність перед використанням повинна бути оголошена*. Сутностями вважаються змінні, функції і багато інших елементів програми, що ми розглянемо в наступних главах. Однак оголошення не створює фізичний об'єкт. Воно лише описує властивості цього об'єкта, тобто називає його тип і ім'я.

Щоб створити реальний об'єкт, необхідно його *визначити*, тобто виділити для нього пам'ять. Тонкості оголошення і визначення об'єктів залежать від конкретної ситуації, але у відношенні змінних оголошення й опис збігаються. Виникає закономірне питання: як розрізняти оголошення й опис. Відповідь проста — *оголошення* не

виділяє пам'яті, а лише описує абстрактні властивості об'єкта. На відміну від оголошення *визначення* виділяє пам'ять для реально існуючих об'єктів.

Друге правило — *кожна сутність може мати тільки одне визначення і скільки завгодно оголошень*.

Тепер, коли ми сформулювали основні теоретичні поняття, перейдемо до практичних прикладів. Розглянемо програму, що підсумовує два цілих числа.

**Програма, що підсумовує два цілих числа**

```
int main()
 {
    int a;
    int b;
    int c;
    a = 1;
    b = 2;
    c = a + b;
    return 0;
 }
```

Відразу упадає в око, що для такої простої операції програма вийшла довгуватою. По-перше, кожна з змінних оголошена в окремому рядку. Це зовсім не обов'язково — их можна було б оголосити в одному рядку.

```
int a, b, c;
```

По-друге, значення кожної з змінних `a` і `b` задається окремим оператором присвоювання. Цю операцію можна сполучити з оголошенням і визначенням цих змінних, виконавши их *ініціалізацію*.

```
int a=1, b=2;
```

Отже, програму можна зробити вдвічі коротше.

**Компактна програма, що підсумовує два цілих числа**

```
int main()
 {
    int a=1, b=2, c;
    c = a + b;
    return 0;
 }
```

Прибічники крайнощів можуть на цьому не зупинятися і стиснути всю програму в один рядок. Для цього потрібно, по-перше, позбутися від оператора `return` і, по-друге, виконати операцію додавання безпосередньо в момент визначення змінної `c`. Для того щоб позбутися від оператора `return`, треба просто оголосити, що функція `main()` нічого не повертає, указавши перед її ім'ям тип `void`. Обчислення суми чисел у момент оголошення змінної можливо завдяки тому, що змінні можна ініціалізувати не тільки константами, але і результатами виразів, що містять раніше визначені змінні. У підсумку одержуємо шедевр мінімалізму.

**Надкомпактна програма, що підсумовує два цілих числа**

```
void main()
 {
     int a=1, b=2, c=a+b;
 }
```

Шанувальники стислості, що вважають її сестрою свого таланта, можуть записати цю функцію в одному рядку.

**Приклад мінімалізму**

```
void main(){ int a=1, b=2, c=a+b;}
```

З функціональної точки зору ці програми абсолютно еквівалентні, а спосіб их запису залежить від особистих смаків програмістів. У багатьох книгах по C++ установилася "ощадлива" практика використання фігурних дужок. Інакше кажучи, відкриваюча фігурна дужка ставиться відразу після заголовків функцій, структур, об'єднань і класів, а також операторів `if`, `while` та інших елементів програми, що мають тіла, узяті у фігурні дужки.

```
void name(){
...
 }
```

Іноді такий запис дозволяє зробити текст програми більш компактним. Однак можна уявити, скільки часу витратять програмісти на пошук загубленої закриваючої дужки, особливо якщо досвіду в них не занадто багато. Тому, хоча б спочатку, не треба економити місце, а писати фігурні дужки “драбинкою”. Це дозволить легко відстежити їхній баланс.

```
void name()
{
   ...
    {
     ...
    }
   ...
}
```

Ніколи не слід забувати, що програми, як правило, читає не тільки компілятор, але й інші програмісти. Зневага наочністю програми може привести до великих утрат часу при її аналізі.

## 1.9. Функції

У такій простій програмі, як наведена вище, немає необхідності виділяти окремі операції і процедури алгоритму. Усе настільки очевидно, що уміщається в один рядок. Звісно, такі алгоритми зустрічаються лише в підручниках. І все ж навіть у “іграшкових” прикладах є деякий зміст — з ними можна експериментувати, випробуючи можливості компілятора. Експериментувати з реальними, до того ж уже налагодженими програмами зважиться не кожний програміст. Як говориться, “краще — ворог гарного”. Отже, повернемося до наших іграшок і задамо собі питання: “А що якщо ...?”. А що якщо нам треба не тільки складати, але і віднімати два цілих числа? А що якщо нам потрібно їх перемножити? Невже доведеться переписувати функцію `main()` щоразу заново?

На щастя, програмісти давно уже використовують принципи структурного програмування, розбиваючи програму на модулі, що реалізують окремі операції алгоритму. Кожен модуль вирішує свою вузькоспеціалізовану задачу й обмінюється даними з іншими модулями, зберігаючи свою незалежність. Поняття модуля в мові C++ конкретизується у вигляді функції.

*Функція* — це набір операторів, що виконують обчислення. Вхідна інформація функції подається у вигляді списку її параметрів, а результат повертається нею в модуль, що її викликав. Зауважимо, що функція може не мати параметрів, а також може нічого не повертати як результат.

Як і будь-яка сутність, функція повинна бути оголошена. Для цього в модулі, що її викликає, вказується *прототип* функції, у якому перелічуються тип значення, що повертається нею, а також ім'я і типи параметрів (імена вказувати не обов’язково, оскільки компілятор їх проігнорує).

Доповнимо нашу програму операціями складення, віднімання і множення цілих чисел, оформивши їх у виді функцій.

**Програма для арифметичних обчислень**

```
int plus(int, int);
int minus(int, int);
int multiply(int, int);
int main()
{
   int a=1, b=2,c,d,e;
   c = plus(a,b);
   d = minus(a,b);
   e = multiply(a,b);

   return 0;
}

int plus(int a, int b)
{
   int c;
   c = a + b;
   return c;
}

int minus(int a, int b)
{
   int c;
   c = a - b;
```

```
    return c;
}

int multiply(int a, int b)
{
    int c;
    c = a * b;
    return c;
}
```

Перевага модульного підходу яскраво виявляється у зовнішньому вигляді функції `main()`. Тепер вона виглядає як проста послідовність викликів функцій, що розкриває зміст програми й одночасно приховує технічні деталі реалізації алгоритму.

На прикладі цієї програми видно, що кожна функція має *заголовок* і *тіло*. Заголовок складається з типу значення, що повертається, імені функції і списку параметрів. Фактично заголовок функції збігається з її прототипом, за винятком двох особливостей: 1) у прототипі не обов'язково вказувати імена параметрів; і 2) прототип обов'язково завершується крапкою з комою, а заголовок, навпаки, жодних крапок з комою не допускає. Тіло функції взято у фігурні дужки і складається з набору операцій, що розв'язує поставлену перед нею задачу. Зверніть увагу також на те, що *функцію не можна визначати усередині інших функцій*. Крім того, *функція може повертати тільки одне значення*.

Відзначимо ще дві особливості цієї програми. По-перше, кожна з функцій використовує аргументи `a` і `b`, а також власну змінну `c`, у яку записується її результат. Їхні імена збігаються з іменами змінних, використовуваних у функції `main()`, при цьому вони належать до зовсім різних змінних. По-друге, тимчасова змінна `c` у кожній з цих функцій зайва. Більш короткий і ощадливий варіант програми виглядає так.

**Ощадлива програма для арифметичних обчислень**

```
int plus(int, int);
int minus(int, int);
int multiply(int, int);
int main()
{
    int a=1, b=2, c, d, e;
    c = plus(a,b);
    d = minus(a,b);
    e = multiply(a,b);

    return 0;
}

int plus(int a, int b)
{
    return a + b;
}

int minus(int a, int b)
{
    return a - b;
}

int multiply(int a, int b)
{
    return a * b;
}
```

Більше заощадити не на чому — програма містить лише самі необхідні елементи: прототипи і функції.

Застосування однакових імен у різних функціях ілюструє дуже важливе поняття — *область видимості* функцій і змінних.

Уважні читачі уже помітили, що в попередній програмі прототипи функцій зазначені на самому початку програми. Тим самим вони зроблені глобальними — тепер их можна викликати з будь-якої точки програми.

Перепишемо функцію `main()` інакше.

**Функція main(), що містить прототипи локальних функцій**

```
int main()
{
    int plus(int, int);
```

```
    int minus(int, int);
    int multiply(int, int);

    int a=1, b=2, c, d, e;
    c = plus(a,b);
    d = minus(a,b);
    e = multiply(a,b);

    return 0;
}
```

У цьому випадку функції `plus()`, `minus()` і `multiply()` можна викликати лише у функції `main()` (власне, у даному прикладі інші виклики просто відсутні). Однак у більш складних алгоритмах може виникнути необхідність викликати функції з інших функцій, а не тільки з головного модуля. Тоді варто зробити усі функції глобальними, помістивши их прототипи поза тілом функції `main()`. Звідси випливає правило: *глобальними* є сутності, оголошені поза функцією `main()`, а *локальними* — сутності, оголошені усередині інших функцій чи блоків. Оскільки блоком вважається будь-яка послідовність операторів, укладена у фігурні дужки, остання фраза означає, що локальна сутність завжди з'являється усередині якої-небудь пари фігурних дужок, що обмежують її *область видимості*. У той же час областю видимості глобальної сутності є вся програма, починаючи з точки оголошення цієї сутності.

Проілюструємо сказане наступним прикладом.

**Невірна програма з двома глобальними функціями**

```
#include <iostream.h>

int f2(); // Прототип глобальної функції

int main()
{
   cout << f1() << endl; // Помилка — у цій точці функція f1() недоступна
   cout << f2() << endl;
   return 0;
}

int f1() // Глобальна функція
{
   return 1;
}

int f2() // Глобальна функція
{
   return 2*f1();
}
```

До речі, ця програма демонструє ще одну тонкість, зв'язану з прототипами функцій. У старих компіляторах оголошення і визначення функції можна було сполучати, тобто замість прототипу можна було просто визначити тіло функції. Тепер компілятори відносяться до цього суворіше і, хоча не припиняють компіляцію, але усе-таки видають попередження про те, що функція `f1()`, наприклад, не має прототипу. Для того щоб усунути і помилку, і попередження, варто помістити прототип функції `f1()` перед функцією `main()`.

**Правильна програма з двома глобальними функціями**

```
#include <iostream.h>

int f1(); // Прототип глобальної функції
int f2(); // Прототип глобальної функції

int main()
{
   cout << f1() << endl;
   cout << f2() << endl;
   return 0;
}

int f1() // Глобальна функція
{
```

```
    return 1;
 }

int f2() // Глобальна функція
 {
    return 2*f1();
 }
```

А що якщо в програмі для арифметичних обчислень ми помістимо прототипи функцій як поза, так і усередині функції `main()`?

**Програма з глобальними і локальними функціями**

```
 int plus(int, int);
 int minus(int, int);
 int multiply(int, int);

 int main()
 {
    int plus(int, int);
    int minus(int, int);
    int multiply(int, int);

    int a=1, b=2, c, d, e;
    c = plus(a,b);
    d = minus(a,b);
    e = multiply(a,b);

    return 0;
 }
```

Нічого страшного. У цьому випадку існують два види сутностей: глобальні функції `plus()`, `minus()` і `multiply()` і локальні функції з тими ж іменами, область видимості яких обмежена тілом функції `main()`. Це типовий приклад *маскування сутностей*, коли їхні області видимості перекриваються. У даному випадку подвійне оголошення функцій не має змісту — вони відносяться до тих самих сутностей.

На закінчення, відзначимо ще декілька цікавих особливостей мови C++, пов'язаних з функціями. По-перше, функція може викликати саму себе. Така функція називається *рекурсивної*. По-друге, будь-яка функція може викликати будь-яку доступну функцію, у тому числі і функцію `main()`. По-третє, дві функції можуть викликати одна одну. Такі функції називаються *взаємно рекурсивними*. Правда, необхідно відзначити, що рекурсивний виклик функцій витрачають дуже багато пам'яті. У свій час ми докладно розберемо причини цього явища.

## 1.10. Заголовкові файли

Тривіальна програма, розглянута вище, звісно, є виключенням. Кожна реальна програма виконує, принаймні, операції вводу і виводу, а для цього їй потрібні стандартні функції, що входять у процедурну систему вводу–виводу мови C, чи оператори, що є частиною об'єктно-орієнтованої системи вводу–виводу мови C++. Щоб мати можливість використовувати ці функції й оператори, необхідно включити в програму заголовковий файл `<stdio.h>` чи заголовок `<iostream>`. Для цього в директиві `#include` вказується ім'я відповідного заголовного файлу, укладене в кутові дужки.

```
#include <stdio.h>
```

У попередній програмі значення змінних `a` і `b` задавалися в момент оголошення, що обмежує застосування нашої програми. Крім того, нам ніколи б не удалося довідатися, чому ж дорівнює сума, різниця і добуток цих чисел, якби заздалегідь ми не знали результат — адже програма нічого не виводить у зовнішній світ.

Отже, нам необхідні засоби введення чисел `a` і `b` і виводу результату. Для цього програму доведеться переписати.

**Удосконалена програма для арифметичних обчислень**

```
 #include <iostream.h>
 int plus(int, int);
 int minus(int, int);
 int multiply(int, int);
 int main()
 {
    int a, b, c, d, e;
    cin >> a;
```

```
    cin >> b;
    c = plus(a,b);
    d = minus(a,b);
    e = multiply(a,b);
    cout << "Сума = " << c << endl;
    cout << "Різниця = " << d << endl;
    cout << "Добуток = " << e << endl;

    return 0;
}

int plus(int a, int b)
{
   return a + b;
}

int minus(int a, int b)
{
   return a - b;
}

int multiply(int a, int b)
{
   return a * b;
}
```

Таким чином, ми одержали повноцінну програму, яку можна застосовувати для додавання, віднімання і множення довільних цілих чисел, а не раз і назавжди заданої пари.

Як бачимо, для цього нам довелося включити в програму заголовковий файл `iostream.h`, що містить визначення операторів вводу і виводу. Щораз, коли нам потрібно викликати яку-небудь стандартну функцію чи звернутися до убудованої константи, необхідно включати в програму відповідний заголовковий файл. Для цього, зокрема, призначений препроцесор мови C++.

## 1.11. Директиви препроцесора

Отже, у нашій програмі з'явився новий елемент — *директива препроцесора*.

```
#include <iostream.h>
```

*Препроцесор* — це програма, що обробляє текст програми на першому етапі компіляції. Вона виконує макропідстановку, умовну компіляцію і включення іменованих файлів. Кожна з цих операцій кодується у виді особливого оператора (*директиви*), що починається символом #. Як правило, препроцесор використовується для включення в програму файлів, у яких визначені стандартні функції й інші сутності.

Препроцесор не виконує синтаксичний аналіз тексту. Він просто розпізнає макропідстановки і виконує їх. Директиви препроцесора не залежать від синтаксису мови, за одним виключенням — їх імена чуттєві до регістра букв (вони перераховані нижче).

```
#define          #elif          #else          #endif
#error           #if            #ifdef         #ifndef
#include         #line          #pragma        #undef
```

Кожна директива повинна розташовуватися в окремому рядку програми. При цьому необхідно уважно стежити за пробілами усередині директив. Деякі з них не мають значення, а інші приводять до помилок. Розглянемо як приклад фрагмент програми, що містить дуже розповсюджену директиву `include`.

```
#include <iostream.h>  #include <math.h>
# include <string.h>
#include < conio.h>
#include <stdlib.h >
#INCLUDE <alloc.h>
#include<INCLUDE\IOSTREAM.H>
```

Перший рядок невірний, оскільки, як зазначено вище, кожна директива повинна знаходитися в окремому рядку. Третій рядок помилковий, тому що усередині кутових дужок повинне знаходитися ім'я заголовка. У даному випадку ім'я заголовного файлу, зазначеного в третьому рядку, починається з пробілу, що, безсумнівно, є помилкою. Пробіл після імені заголовкового файлу, як показано в четвертому рядку, сучасні компілятори також вважають помилкою, хоча старі компілятори розглядали его як незначний символ і правильно розпізнавали ім'я файлу.

Другий рядок вірний, оскільки символ # є окремою лексемою. Між першою і другою лексемами директиви препроцесора можуть розташовуватися лише пробіли і символи горизонтальної табуляції. Будь-які інші роздільники заборонені. Незважаючи на те що символ # — окрема лексема, він є невід'ємною частиною імені директиви.

П'ятий рядок невірний, тому що ім'я директиви `#include` повинне складатися лише з малих літер. Шостий рядок також є абсолютно правильним, хоча і виглядає досить незвично.

Докладний аналіз директив препроцесора чекає нас попереду, а поки розглянемо лише их загальне призначення і місце в тексті програми.

Зверніть увагу на одну обставину: оскільки директиву препроцесора можна розміщати в довільному місці програми, а директива `#include` замінюється зазначеним файлом, можна "згортати" вихідний текст програми, розподіляючи його фрагменти по різних файлах.

```
#include <stdio.h>

main()

{
 ...
    #include "file1.txt"
 ...
    #include "c:\text\file2.txt"
 ...
    return 0;
}
```

Цей фрагмент ілюструє одну цікаву властивість директиви `#include`. З її допомогою можна вставляти в текст програми не тільки стандартні, але і власні текстові файли. Для цього слід застосовувати лапки, а не кутові дужки. У цьому випадку пошук файлу виконується в поточному каталозі, а не в каталозі `INCLUDE`. Якщо такого файлу в поточному каталозі немає, виконується пошук у каталозі `INCLUDE`. Регістр букв при наборі імені файлу не має значення. Крім того, у директиві `#include` можна явно вказувати шлях до файлу. Імена каталогів у системі Windows розділяються зворотною косою рисою, а в системі UNIX — косою рисою.

## 1.12. Коментарі

Як правило, програму читає не тільки ее автор, але і його колеги. Однак навіть якщо ви пишете програму для себе, будьте упевнені, що незабаром ви самі забудете, для чого призначена та чи інша функція, а також той чи інший оператор, і будете витрачати багато часу на болісні спогади. Щоб цього не трапилося, програму документують, вставляючи в неї *коментарі*. У мові C++ є два види коментарів — *багаторядкові* й *однорядкові*. Багаторядкові коментарі успадковані від мови C и являють собою пояснення, укладені в символи `/*` і `*/`. Ці коментарі називаються багатостроковими, тому що компілятор ігнорує усе, що міститься між символами `/*` і `*/`, включаючи самі обмежувачі, і отже, між ними можна вставляти скільки завгодно рядків тексту. Однорядковий коментар ліворуч обмежений символами `//`, а праворуч — кінцем рядка програми. Отже, коментарі цього виду можуть знаходитися лише праворуч від оператора. Звичайно вкладені коментарі не заохочуються, однак, компілятори, як правило, допускають вкладення багаторядкових коментарів. Вкладення однорядкових операторів не має змісту, оскільки лівий обмежник вкладеного коментарю при цьому нічим не відрізняється від інших символів.

Як правило, багаторядкові коментарі містять докладний опис функцій і класів (так називаний *контракт*, що складається із передумов і постумов) і тому розміщаються на початку їх опису. Передумови функції описують її призначення та вхідну інформацію, а постумови описують, що функція повертає та як змінюється стан програми після повернення керування у точку виклику. Однорядкові коментарі пояснюють зміст окремих рядків програми. Крім того, коментарі дозволяють локалізувати помилки при налагодженні програми. "Закоментувавши" підозрілий фрагмент програми, можна досить швидко знайти місце можливої помилки.

Отже, оформимо нашу програму за всіма правилами.

**Приклад оформлення програми за допомогою коментарів**

```
/*
  Програма призначена для додавання, віднімання і множення двох цілих чисел.
*/
 #include <iostream.h>          // Директива препроцесора
 int plus(int, int);            // Прототип функції plus()
 int minus(int, int);           // Прототип функції minus()
 int multiply(int, int);        // Прототип функції multiply()
 int main()
```

```
{
   int a, b, c, d, e;          // Оголошення локальних змінних
   cin >> a;                   // Уведення першого аргументу
   cin >> b;                   // Уведення другого аргументу
   c = plus(a,b);              // Додавання двох аргументів
   d = minus(a,b);             // Вирахування двох аргументів
   e = multiply(a,b);          // Множення двох аргументів
   // Вивод результатів
   cout << "Сума = " << c << endl;
   cout << "Різниця = " << d << endl;
   cout << "Добуток = " << e << endl;

   return 0;
}

int plus(int a, int b)
/*
Передумова: функція одержує два цілих аргументи a і b.
Постумова:  функція обчислює суму аргументів, не змінюючи їхнього значення.
*/
{
   return a + b;
}

int minus(int a, int b)
/*
Передумова: функція одержує два цілих аргументи a і b.
Постумова:  функція обчислює різницю аргументів, не змінюючи їхнього значення.
*/
{
   return a - b;
}

int multiply(int a, int b)
/*
Передумова: функція одержує два цілих аргументи a і b.
Постумова: функція обчислює добуток аргументів, не змінюючи їхнього значення.
*/
{
   return a * b;
}
```

Зрозуміло, програміст може сам вирішувати, наскільки докладними повинні бути коментарі. Тут необхідно дотримувати як здорового глузду, так і правил оформлення документації.

## 1.13. Фази компіляції

Після створення вихідний текст компілюється. Етап компіляції і редагування зв'язків розпадається на кілька фаз.

1. Вихідний файл кодується основним набором текстових символів, причому наприкінці кожного рядка програми розставляються символи переходу на новий рядок. Кожен символ, що не входить в основний набір текстових символів, замінюється відповідним універсальним символом.
2. З проміжного тексту видаляються символи переходу на новий рядок, і фізичні рядки вихідного коду стають логічними. Якщо в процесі трансформації тексту випадково виникнуть універсальні символи, поводження компілятора стандартом не регламентується. Це стосується і ситуації, коли непорожній текстовий файл не завершується символом переходу на новий рядок.
3. Вихідний файл розкладається на лексеми і послідовності роздільників, включаючи коментарі. При цьому текст програми не повинний завершуватися незакінченою лексемою чи незавершеним коментарем (наприклад, `<include` чи `/* неповний коментар`). Кожен коментар замінюється пробілом. Символи переходу на новий рядок зберігаються. Обробка інших роздільників залежить від конкретного компілятора. Інтерпретація символів, що утворюють текст програми, залежить від контексту (наприклад, символ `<` у директиві `#include` <*ім'я файлу*> і символ `<` в умовному вираженні обробляються по-різному).
4. Виконуються директиви препроцесора і макровизначення. Якщо в процесі конкатенації лексем випадково утвориться універсальний символ, поводження компілятора не регламентується. У результаті

виконання директиви `#include` у вихідний текст програми вставляється текст відповідного заголовного файлу, до якого рекурсивно застосовуються фази 1–4.

5. Кожен символ з основного набору текстових символів, escape-послідовність чи універсальний символ, що є частиною символьного чи рядкового літерала, конвертується у відповідний елемент із набору керуючих символів.
6. Суміжні рядкові і розширені рядкові літерали конкатенуються.
7. Роздільники між лексемами ігноруються. Кожна лексема препроцесора перетворюється в звичайну лексему. Отримані лексеми піддаються синтаксичному і семантичному аналізу. Поняття "вихідний файл" і "одиниці трансляції" є абстрактними — їм не обов'язково відповідають фізичні файли. Інакше кажучи, щоб скомпілювати програму, її не обов'язково зберігати у файлі. (Це зауваження стосується, скоріше, до інтегрованих систем програмування, що поєднують текстовий редактор, компілятор і редактор зв'язків єдиним інтерфейсом.)
8. Розпізнаються всі зв'язки програми з зовнішніми об'єктами і функціями. Для зв'язування функцій і об'єктів, не визначених у процесі трансляції, підключаються бібліотечні компоненти. У результаті виникає образ програми, що містить всю інформацію, необхідну для виконання програми (exe-модуль).

Отже, ми розглянули всі питання, зв'язані з конструюванням програм. Тепер ми можемо написати і налагодити найпростішу програму, що представляє собою деякий каркас. Залишилося лише наповнити його реальним змістом. Для цього нам необхідно зовсім небагато — вивчити іншу частину мови C++.

## 1.14. Резюме

- Алфавіт мови C++ складається з чотирьох частин: *основного набору текстових символів*, *множини універсальних символів*, *основного набору керуючих символів* і *основного набору розширених керуючих символів*.
- *Ім'я* — це рядок символів, що використовується для позначення якого-небудь елемента програми (змінної, константи, функції, класу тощо).
- *Ключове слово* — це рядок символів, що має особливе значення лише у визначених ситуаціях.
- *Зарезервованим словом* називається рядок символів, що в жодному разі не можна використовувати як ім'я.
- Ключові слова розподіляються на оператори, специфікатори, модифікатори і кваліфікатори.
- *Тип* — це діапазон значень, що можуть набувати змінні, і набір операцій, які можна виконувати над ними.
- Ключові слова `break`, `case`, `catch`, `continue`, `delete`, `do`, `else`, `export`, `for`, `goto`, `if`, `namespace`, `new`, `reinterpret_cast`, `return`, `static_cast`, `switch`, `throw`, `try`, `typedef`, `typeid`, `using` і `while` є *операторами*.
- Ключові слова, що визначають тип і спосіб збереження змінної чи функції, називаються *специфікаторами*. До них відносяться *специфікатори основних типів* `bool`, `char`, `double`, `float`, `int`, `long`, `short`, `signed`, `unsigned` і `void`, *специфікатори складних типів* `class`, `struct` і `union`, *специфікатори збереження* `auto`, `extern`, `register` і `static`, *специфікатори доступу* `public`, `private` і `protected`, *специфікатори функцій* `inline` і `virtual`, а також *специфікатори дружніх класів і функцій* `friend` і *специфікатор шаблонів* `template`.
- У визначених ситуаціях типи уточнюються за допомогою *модифікаторів* `const` і `volatile`, іменованих також *cv*-кваліфікаторами.
- *Ідентифікатор* — це синонім слова "ім'я". Він зв'язується з конкретним елементом програми (змінною, константою, функцією тощо) і надалі використовується для їхнього позначення.
- Ідентифікатори можуть складатися тільки з букв, цифр і символу підкреслення.
- Ідентифікатори повинні починатися з букви або символу підкреслення.
- Ключові слова не можуть бути ідентифікаторами.
- Як ідентифікатори не можна використовувати імена стандартних функцій (`sin`, `cos` тощо).
- У стандарті мови C++ не встановлена гранична довжина ідентифікатора.
- Компілятор розрізняє прописні і малі літери, що входять до складу ідентифікатора.

- Літералом називається постійне значення, яке визначається на етапі компіляції. Літерал може бути цілочисельним, символьним, дійсним, рядковим і булевим.
- Цілочисельний літерал — це ціле число, записане у десятковій, вісімковій або шістнадцятковій системах числення і може супроводжувтися певними префіксами і суфіксами.
- Символьним літералом є ASCII–код, взятий в одинарні лапки, або керуючий символ.
- Оператори в мові C++ позначають різні операції: арифметичні, логічні, поразрядні і порівняння.
- *Знаками пунктуації* в мові C++ називаються символи, що мають синтаксичний і семантичний зміст, але не позначають самостійні оператори. До них належать круглі, квадратні і фігурні дужки.
- Будь-яка програма на мові C++ складається із функцій, серед яких є одна головна — функція `main`().
- У відповідності зі стандартом, сутністю вважається значення, *об'єкт*, *підоб'єкт*, *підоб'єкт базового класу*, *елемент масиву*, *змінна*, *функція*, *екземпляр функції*, *перерахування*, *тип*, *член класу*, *шаблон* і *простір імен*. Усі вони мають одну загальну властивість — будь-яка сутність має *ім'я*, що є її *ідентифікатором*.
- Основне правило мови C++: *будь-яка сутність перед використанням повинна бути оголошена.*
- Оголошення не створює фізичний об'єкт. Воно лише описує властивості цього об'єкта, тобто називає його тип і ім'я.
- Щоб створити реальний об'єкт, необхідно його *визначити*, тобто виділити для нього пам'ять. Тонкості оголошення і визначення об'єктів залежать від конкретної ситуації, але у відношенні змінних оголошення й опис збігаються. *Оголошення* не виділяє пам'яті, а лише описує абстрактні властивості об'єкта. На відміну від оголошення *визначення* виділяє пам'ять для реально існуючих об'єктів.
- Друге правило мови C++ — *кожна сутність може мати тільки одне визначення і скільки завгодно оголошень*.
- *Функція* — це набір операторів, що виконують обчислення. Вхідна інформація функції подається у вигляді списку її параметрів, а результат повертається нею в модуль, що її викликав. Функція може не мати параметрів, а також може нічого не повертати як результат.
- Як і будь-яка сутність, функція повинна бути оголошена. Для цього в модулі, що її викликає, вказується *прототип* функції, у якому перелічуються тип значення, що повертається нею, а також ім'я і типи параметрів (імена вказувати не обов'язково, оскільки компілятор їх проігнорує).
- Кожна функція має *заголовок* і *тіло*. Заголовок складається з типу значення, що повертається, імені функції і списку параметрів. Фактично заголовок функції збігається з її прототипом, за винятком двох особливостей: 1) у прототипі не обов'язково вказувати імена параметрів; і 2) прототип обов'язково завершується крапкою з комою, а заголовок, навпаки, жодних крапок з коми не допускає. Тіло функції взято у фігурні дужки і складається з набору операцій, що розв'язує поставлену перед нею задачу.
- Бажано, щоб в кожній функції був явно указаний *контракт*, що складається із *перед-* і *постумови*. *Передумова* описує ситуацію, в якій викликається функція: тип і кількість параметрів, що вона отримує, та її призначення. *Постумова* описує ситуацію, що утворюється після виконання функції: тип значення, що повертається, та можливі побічні ефекти (наприклад, зміна значень глобальних змінних).
- *Функцію не можна визначати усередині інших функцій*.
- *Функція може повертати тільки одне значення*.
- Функція може викликати саму себе. Така функція називається *рекурсивною*.
- Будь-яка функція може викликати будь-яку доступну функцію, у тому числі і функцію `main()`.
- Дві функції можуть викликати одна одну. Такі функції називаються *взаємно рекурсивними*.
- *Область видимості* будь-якої сутності в програмі — це область, в межах якої сутність вважається існуючою, тобто над нею можна виконувати операції. Область видимості починається з точки оголошення сутності і закінчується найближчою фігурною дужкою, що визначає межу блоку або функції.
- Сутність, яка є видимою в межах лише певного блоку або функції, називається *локальною*.

- Сутність, яка є видимою в межах всієї програми або файлу (починаючи з точки оголошення), називається *глобальною*. Глобальні сутності оголошуються за межами функції `main()` — в глобальному просторі.
- *Маскування сутностей* виникає, коли їхні області видимості перекриваються.
- *Препроцесор* — це програма, що обробляє текст програми на першому етапі компіляції. Вона виконує макропідстановку, умовну компіляцію і включення іменованих файлів. Кожна з цих операцій кодується у виді особливого оператора (*директиви*), що починається символом #. Як правило, препроцесор використовується для включення в програму файлів, у яких визначені стандартні функції й інші сутності.
- Препроцесор не виконує синтаксичний аналіз тексту. Він просто розпізнає макропідстановки і виконує їх. Директиви препроцесора не залежать від синтаксису мови, за одним виключенням — їх імена чуттєві до регістра букв.
- У мові C++ є два види коментарів — *багаторядкові* й *однорядкові*. Багаторядкові коментарі успадковані від мови C и являють собою пояснення, розташовані між символами `/*` і `*/`. Ці коментарі називаються багаторядковими, тому що компілятор ігнорує усе, що міститься між символами `/*` і `*/`, включаючи самі обмежувачі, і отже, між ними можна вставляти скільки завгодно рядків тексту. Однорядковий коментар ліворуч обмежений символами `//`, а праворуч — кінцем рядка програми.

## Контрольні питання

1.1. З яких частин складається алфавіт мови C++?

1.2. Що таке *ім'я* сутності?

1.3. Що таке *ключове слово*?

1.4. Що таке *зарезервоване слово*?

1.5. Чи *оголошення* відрізняється від *означення*?

1.6. На які категорії розподіляються ключові слова?

1.7. Що таке *тип* змінної?

1.8. Назвіть всі ключові слова мови C++.

1.9. Перерахуйте всі оператори мови C++.

1.10. Перерахуйте ключові слова, що є *операторами*.

1.11. Перерахуйте ключові слова, що визначають тип і спосіб збереження змінної чи функції.

1.12. Перерахуйте *cv*-кваліфікатори.

1.13. Що таке *ідентифікатор*?

1.14. За якими правилами утворюються ідентифікатори?

1.15. Що таке *літерал*?

1.16. Що таке *цілочисельний літерал*?

1.17. Що таке *символьний літерал*?

1.18. На які категорії розподіляються оператори мови C++?

1.19. Що називається *знаком пунктуації*.

1.20. З яких елементів складаються програми C++?

1.21. Що таке сутність?

1.22. Сформулюйте основне правило мови C++.

1.23. Сформулюйте друге правило мови C++.

1.24. Що таке *функція*?

1.25. Що таке *прототип* функції?

1.26. Що називається *заголовком* і *тілом* функції.

1.27. Що таке *контракт функції*? З яких частин від складається?

1.28. За якими правилами визначаються функції?

1.29. Що таке рекурсивна функція?

1.30. Які функції називаються *взаємно рекурсивними*?

1.31. Що таке *область видимості* сутності?

1.32. Які сутності називаються *локальними*?

1.33. Які сутності називаються *глобальними*?

1.34. В чому проявляються *маскування сутностей*?

1.35. Що називається *препроцесором*? Як він працює?

1.36. Що таке *коментар*? Які види коментарів існують в мові C++?