

Мова програмування. Поняття програми

Як було сказано раніше, **мова програмування** – фіксована система позначень та правил для опису структур даних та алгоритмів, призначених для виконання обчислювальними машинами.

З формальної точки зору **мова програмування** – це набір вихідних символів (*алфавіт*) разом із системою правил утворення з цих символів формальних конструкцій (*синтаксис*) та системою правил їх тлумачення (інтерпретації) (*семантика*), за допомогою яких описуються алгоритми. **Програма** – це алгоритм записаний за допомогою мови програмування.

Алфавіт, синтаксис та семантика — це три основні складові частини будь-якої мови програмування. До **алфавіту** мови програмування, як правило, входять літери латинського алфавіту, арабські цифри, знаки арифметичних операцій, розділові знаки, спеціальні символи. Із символів алфавіту будують послідовності, які називають *словами* (*лексемами*). Кожне слово у мові програмування має своє змістове призначення. **Правила синтаксису** пояснюють, як потрібно будувати ті чи інші мовні повідомлення для опису всіх понять мови, здійснення описів та запису вказівок. **Правила семантики** пояснюють, яке призначення має кожен опис та які дії повинна виконати обчислювальна машина під час виконання кожної із вказівок. Вказівки на виконання конкретних дій називають ще **командами** або **операторами** мови.

Усі слова або ж лексеми, поділяють на

- *службові* (зарезервовані або ж ключові) слова;
- *стандартні ідентифікатори*;
- *ідентифікатори користувача*;
- *знаки операцій*;
- *літерали*;
- *розділові знаки*.

Службові слова мають наперед визначене призначення і використовуються для формування структури програми, здійснення описів, позначення операцій, формування керуючих конструкцій (вказівок). Наприклад, службовими словами для мови C++ є: `abstract, do, in, protected, public, try, else, interface, enum, break, return, new, sizeof, using, class, const, for, operator, continue, struct, while, switch, default, if, this, private`.

C++Імена (позначення) для програмних об'єктів (типів даних, констант, змінних, функцій і т. п.) формують у вигляді ідентифікаторів. *Ідентифікатор* — це послідовність латинських літер, цифр і знака підкреслення, яка розпочинається з латинської літери. У мові C++максимальна довжина ідентифікатора є необмеженою.

C++ `switch, default`,*Стандартні ідентифікатори* використовуються як імена для стандартних (передбачених авторами мови програмування) типів даних, констант, підпрограм (зокрема, стандартних математичних функцій). Приклад: `int, long, double, char, bool` та ін.

Ідентифікатори користувача є іменами тих програмних об'єктів (констант, змінних, функцій тощо), які створює сам користувач. Службові слова та ідентифікатори користувача не повинні збігатися.

Коментар – невиконувана частина тексту програми, що ігнорується компілятором і служить для вставки деяких поміток у програмі тільки для програміста. Коментарі бувають однорядковими та багаторядковими. Однорядковий коментар починається з

символів «//» і закінчується у кінці рядка. Тобто всі символи до кінця рядка вважаються коментарем.

Приклад.

// Це однорядковий коментар

Багаторядковий коментар починається з символів «/*» і закінчується символами «*/».

Приклад.

/* Це
багаторядковий
коментар */

Літерал – це явно вказане значення деякого типу. Розрізняють такі типи літералів:

| <i>тип літерала</i> | <i>опис</i> | <i>приклади</i> |
|---------------------|---|---|
| цілочисловий | у десятковій системі числення – звичне нам ціле число | 125, -89, 108; |
| | у вісімковій системі числення – починається з 0 | 023, 075, 0416; |
| | у шістнадцятковій системі числення – починається з 0x або 0X | 0x4A, 0x6FE2, 0xABC |
| дійсного типу | у форматі з фіксованою крапкою – у записі числа є крапка, що розділяє цілу і дробову частити | 25.69, 2.0, 145.058 |
| | у форматі з плаваючою крапкою – у записі використано символ «e» або «E», що розділяє мантису від порядку | 3.5e5, 3e12, 678e2 |
| символьний | заданий у явному вигляді – символ записується у одинарних лапках | 'Z', 'a', 'E' |
| | прості ескейп-послідовності – службові символи починаються з символу «\» | \n -перехід на новий рядок, \t -горизонтальна табуляція, \' - апостроф, і т. д. |
| | ескейп-послідовності Unicode – символи «\u», за якими вказують код символу з чотирьох цифр у шістнадцятковій системі числення | '\u0123', '\u3A58' |
| рядковий | регулярний – у подвійних лапках вказується рядок символів (ескейп-послідовності обробляються) | "Я люблю C++" |
| логічний | може приймати два значення | true, false |

Розділові знаки служать для відокремлення однієї лексеми від іншої. Ними є пробіл, табуляція, символ нового рядка, символ «;» та коментарі.

ВЕЛИЧИНА. ТИП ВЕЛИЧИНИ

У своїй роботі програміст має справу з таким поняттям, як величина. З точки зору програмування **величини** – це дані, що обробляються програмами. *Дані* — це інформація, введена у пам'ять комп'ютера або підготовлена до введення.

Носіями даних у програмах є *константи*, *змінні* (значення яких зберігається в оперативній пам'яті) та *файли* (на зовнішніх носіях інформації).

Константи — це величини, значення яких у процесі виконання програми не змінюється. *Змінні* — це величини, значення яких у процесі виконання програми можуть змінюватися. Імена констант і змінних, як і інших програмних об'єктів, записують у формі ідентифікаторів. Кожна змінна і константа належать до визначеного типу.

Тип даних – це сукупність властивостей певного набору даних, від яких залежать: діапазон значень, якого можуть набувати ці дані, а також сукупність операцій, які можна виконувати над цими даними.

З іншого боку **тип даних** – це описання того, яку структуру, розмір мають комірки оперативної пам'яті при зберіганні відповідного елемента даних.

Елемент даних певного типу – це комірка або комірки оперативної пам'яті, що мають фіксовану адресу, розряди яких розшифровуються згідно описання даного типу даних.

З кожним типом даних зв'язано своє унікальне ім'я (ідентифікатор), яке є синонімом певного описання елемента даних відповідного типу. Наприклад, ідентифікатор `int` є синонімом опису : 32 послідовних розрядів містить ціле значення в діапазоні від (-2^{32}) до $(2^{32}-1)$ (у двійковому вигляді займає 4 байт).

У зв'язку з цим можна дати інше означення константи та змінної.

Якщо елемент даних не може змінювати свого значення, тобто завжди містить одне і те ж саме значення, то відповідний ідентифікатор називається *константою даного типу*. Якщо елемент даних певного типу може змінювати своє значення під час виконання програми, то ідентифікатор, що зв'язаний з цим елементом даних називається *змінною відповідного типу*. *Значення змінної* – це елемент даних, з якими ця змінна пов'язана.

Отже, у програмах змінна характеризується такими ознаками: *іменем*, *типом* і *значенням*.

Типи даних C++

У мові програмування визначено такі елементарні типи даних

| Ім'я типу | | Кількість байт | Діапазон |
|----------------------|----------------------------|----------------|---|
| <code>bool</code> | логічний тип | 1 | true – false |
| <code>char</code> | символьний тип | 1 | символи з таблиці кодів ASCII ($0 - (2^8-1)$) |
| <code>wchar_t</code> | двобайтовий символьний тип | 2 | $0 - (2^{16}-1)$ |
| <code>int</code> | цілочисловий тип | 4 | $(-2^{31}) - (2^{31}-1)$ |
| <code>float</code> | тип дійсних чисел | 4 | 3.4E-38 – 3.4E+38 |
| <code>double</code> | тип дійсних чисел | 8 | 1.7E-308 – 1.7E+308 |

Деякі з типів можна модифікувати ключовими словами **signed** (знаковий), **unsigned** (беззнаковий), **short** (короткий) и **long** (довгий).

[<модифікатор>] [<тип>]

При цьому, якщо тип не вказано, то вважається, що типом даних є тип `int`.

| Цілочислові типи | Кількість байт | Діапазон |
|------------------|----------------|----------|
|------------------|----------------|----------|

| | | |
|---|----------------|------------------------------|
| signed char | 1 | $-2^7 - (2^7 - 1)$ |
| unsigned char | 1 | $0 - (2^8 - 1)$ |
| short short int signed short signed short int | 2 | $(-2^{15}) - (2^{15} - 1)$ |
| unsigned short unsigned short int | 2 | $0 - (2^{16} - 1)$ |
| int signed signed int | 4 | $(-2^{31}) - (2^{31} - 1)$ |
| unsigned unsigned int | 4 | $0 - (2^{32} - 1)$ |
| long long int signed long signed long int | 8 | $(-2^{63}) - (2^{63} - 1)$ |
| unsigned long unsigned long int | 8 | $0 - (2^{64} - 1)$ |
| long long long long int signed long lon signed long long int | 16 | $(-2^{127}) - (2^{127} - 1)$ |
| unsigned long long unsigned long long int | 16 | $0 - (2^{128} - 1)$ |
| Дійсні типи | Кількість байт | Діапазон |
| long double | 10 | $3.4E-4932 - 3.4E+4932$ |

Створення псевдонімів типів даних. У С++ є можливість описувати для типів даних псевдоніми. Такий опис здійснюється з використанням оператора `typedef`.

| Загальний вигляд | Приклад |
|--|---|
| <code>typedef <Тип даних> <Псевдонім типу даних>;</code> | <pre>typedef long int lint; typedef double Myd; //Зараз опис long int m; //і опис lint m; //є еквівалентними</pre> |

Консольні програми у С++

Консольна програма – це програма, яка дозволяє виводити символічну інформацію на екран та вводити символічну інформацію з клавіатури. Точкою входу у будь-яку консольну програму є функція `main`.

Структура консольної програми

| Загальний вигляд | Приклад |
|------------------|---------|
|------------------|---------|

| | |
|--|--|
| <Підключення заготовочних файлів> | #include "stdafx.h" #include <iostream>; using namespace std; |
| <Оголошення глобальних змінних> <Оголошення функцій> <Оголошення класів > | int d=35; int sum(int x, int y); |
| int main () { <оператори > return 0; } | int main() { cout<<"Hello!!"<<endl; cout<<"d="<<sum(d,5)<<endl; system("pause"); return 0; } |
| <Реалізація функцій і методів класів> | int sum(int x, int y) { return x+y; } |

Програма у C++ , як і у інших мовах програмування, може складатися з багатьох файлів. Серед них можна виділити так звані *файли заголовків*, які можуть містити описи різного роду програмних об'єктів: функцій, констант, типів даних та ін. Розширення файлів заголовків – «.h». Якщо при написанні програми необхідно використати деякі чи стандартні, чи розроблені програмістом програмні об'єкти, що розміщено в іншому файлі заголовка, то необхідно підключити цей файл заголовка з використанням директиви include

include <ім'я файлу заголовка >;

Зауважимо, що при підключенні стандартних заготовочних файлів розширення «.h» не вказують. Так у наведеному прикладі було підключено файл заголовка `iostream`, що містить опис стандартних об'єктів `cout` та `cin`, які дають можливість здійснювати введення та виведення даних

#include <iostream>;

Опис змінних

Змінна – це іменована ділянка оперативної пам'яті, що використовується у програмі для збереження даних. Перш ніж використати змінну, її необхідно попередньо описати

| Загальний вигляд | Приклад |
|---|---------------------|
| [<Специфікатор>] [<Модифікатор типу>] < Тип > <Ім'я змінної>; | int x; double d; |

При описі змінної можна одразу надавати їй початкове значення, тобто ініціалізувати цю змінну

| Загальний вигляд | Приклад |
|--|------------------------------|
| [<Специфікатор>] [<Модифікатор типу>] <Тип> <Ім'я змінної>=<Значення>; | int x = 12; double d=2.5; |

Описуючи змінну, можна також вказувати *специфікатори* зберігання, які впливають на місце розташування змінної.

| Специфікатор | Призначення |
|--------------|---|
| auto | тип змінної визначається на основі значення, яким ця змінна ініціалізується |
| register | значення змінної буде зберігатися у регістрах процесора (як правило, такі змінні дуже часто використовуються) |
| extern | пояснює компілятору, що повний опис змінної знаходиться в іншому місці (наприклад, у іншому файлі) |
| static | використовується при описі змінних у середині функцій, що дозволяє зберігати значення між викликами цих функцій |

Визначення розміру змінних та типів. Кількість байтів, які виділяються для змінної можна визначити з використанням оператора `sizeof`, який має два формати

| Загальний вигляд | Приклад |
|--|------------------------------|
| <code>sizeof(<Змінна>)</code> | int x; cout<<sizeof(x); |
| <code>sizeof(<Тип даних>)</code> | cout<<sizeof(int); |

Динамічне визначення типів змінних. Тип змінної у процесі роботи програми можна визначити з використанням оператора `typeid`, який описано у файлі заголовка `typeinfo`. Оператор `typeid` повертає об'єкт класу `type_info`, який дозволяє отримати ім'я типу, за допомогою функції `name`.

| Загальний вигляд | Приклад |
|---|--|
| <code>typeid (<Змінна>).name()</code> | #include < typeinfo > int x; typeid (x).name(); // int |

Константи

Константи – це величини, значення яких не можуть змінюватися у процесі роботи програми. Їх описують з використанням ключового слова `const`

| Загальний вигляд | Приклад |
|--|--|
| <code>const <тип> <ім'я> =<значення>;</code> | const int x=55; const double d=3.7; |

Області видимості

Перш ніж змінну використати, її обов'язково необхідно описати. Описати змінну можна як глобальну (описану поза межами функцій), так і локальну (описану у середині якоїсь функції чи блоку).

Глобальна змінна описується поза межами функцій і може бути використана у будь-якій частині програми після її опису.

Локальна змінна описується у середині якогось блоку (всередині фігурних дужок) чи функції. Вона може бути використана тільки в межах блоку чи функції, у яких вона описана.

Якщо всередині блоку ім'я локальної змінної співпадає з іменем глобальної змінної, то кажуть, що локальна змінна приховує глобальну. Для доступу до глобальної змінної у межах цього блоку необхідно використати оператор « :: »

| Загальний вигляд | Приклад |
|------------------|---|
| :: <Змінна> | <pre>int x=20; //Опис глобальної змінної x int func(){ int x=3.5; //Опис локальної змінної x cout<<"Локальна x=" << x <<endl; //Локальна x=3.5 cout<<"Глобальна x="<< ::x<<endl; //Глобальна x=20 }</pre> |

Простори імен

При розробці великих і складних програм декількома програмістами часто виникає ситуація, коли одними і тими ж іменами позначають величини різних типів. Але ж у програмі не можна описати дві різні змінні з однаковим іменем. Для розв'язання цієї проблеми використовують так звані простори імен. *Простір імен* – це частина програми, іменування змінних і типів у якій ніяк не залежить від іншої частини програми. Розглянемо формат опису простору імен.

| Загальний вигляд | Приклад |
|--|---|
| <pre>namespace <Назва простору імен>{ <опис змінних, типів та ін.> }</pre> | <pre>namespace nsp1{ int g=20; } namespace nsp2{ double g=83.9; }</pre> |

Якщо у межах одного простору імен необхідно здійснити доступ до змінної чи типу, описаних у іншому просторі імен, то необхідно використати оператор « :: ».

| Загальний вигляд | Приклад |
|--|--|
| <Назва простору імен>::<ідентифікатор> | <pre>cout<<"nsp1::g="<<nsp1::g<<endl; //20 cout<<"nsp2::g="<<nsp2::g<<endl; //83.9</pre> |

Для спрощення доступу до програмних об'єктів, описаних у іншому просторі імен можна здійснити імпортування всіх описів або окремих ідентифікаторів цього простору імен з використанням оператора **using**.

| Загальний вигляд | Приклад |
|---|---|
| //Імпортування всіх описів using namespace <Назва простору>; | <pre>using namespace nsp1; cout<<"nsp1::g="<< g <<endl;</pre> |
| //Імпортування окремих ідентифікаторів using <Назва простору>::<ідентифікатор>; | <pre>using nsp1::g; cout<<"nsp1::g="<< g <<endl;</pre> |

При описі розглянутих просторів імен вказувались назви цих просторів (*іменовані простори імен*), але у С++ є також можливість описувати так звані *неіменовані (анонімні) простори*, при описі яких не вказують імен

```
namespace {
```

```
<ОПИС ЗМІННИХ, ТИПІВ ТА ІН.>
```

```
}
```

Ідентифікатори, описані у неіменованому просторі імен, доступні тільки у межах файлу, де цей простір імен описано. Звертання до ідентифікаторів цього простору імен здійснюється так само як і для глобальних змінних.

Виведення даних

Виведення даних може бути здійснено з використанням об'єкта `cout` з простору імен `std`, що описано у файлі заголовку `iostream`

| Загальний вигляд | Приклад |
|--|--|
| <pre>#include <iostream> std : : cout << <Дані1> << <Дані2><<...</pre> | <pre>#include <iostream> int x=36; std : : cout << "x=" << x << endl; std : : cout << "Hello" << endl;</pre> |

Спростити доступ до `cout` можна, виконавши імпорт описів з простору імен `std`.

| Загальний вигляд | Приклад |
|---|--|
| <pre>#include <iostream> using namespace std; cout << <Дані> << <Дані><<...</pre> | <pre>#include <iostream> using namespace std; int x=36; cout << "x=" << x << endl; // x=36 cout << "Hello" << endl; // Hello</pre> |

Введення даних

Введення даних може бути здійснено з використанням об'єкта `cin` з простору імен `std`, що описано у файлі заголовку `iostream`.

| Загальний вигляд | |
|--|---|
| <pre>#include <iostream> std : : cin >> <Змінна> ;</pre> | <pre>#include <iostream> int x; std : : cout << "x="; std : : cin >> x;</pre> |

Спростити доступ до `cin` можна, виконавши імпорт описів з простору імен `std`

| Загальний вигляд | |
|---|--|
| <pre>#include <iostream> using namespace std; cin >> <Змінна></pre> | <pre>#include <iostream> using namespace std; int x; cout << "x="; cin >> x;</pre> |

Оператор присвоєння. Перетворення типів

Оператор – це логічно завершена конструкція (вказівка), призначена для виконання конкретних дій.

Усі оператори можна поділити на **прості та складні**. **Прості** оператори не містять всередині себе інших операторів. **Складні** або **структурні** оператори представляють собою конструкції, що містять інші оператори (як складні так і прості). До простих операторів як правило відносять оператор присвоєння та виклик процедури.

Зрозуміло, якщо у програмі буде обчислене значення виразу, то його необхідно десь запам'ятати для подальшого використання. Для цього існує оператор присвоювання.

| Загальний вигляд | Приклад |
|---------------------------|-------------------------------|
| <ім'я змінної> = <вираз>; | int n; n=12; int i=n+2; |

Під час виконання цього оператора спочатку обчислюється значення виразу в правій частині при поточних значеннях змінних, що входять до нього, а потім отриманим результатом замінюється попереднє значення змінної, що вказана зліва.

Оператор присвоєння можна застосовувати до числових, логічних та символьних даних.

Приклад.

| |
|--|
| char c='d'; int n=9; double d=2.7; string s="перетворення типів"; bool b=true; |
|--|

При використанні оператора присвоєння необхідно слідкувати, щоб тип виразу у правій частині відповідав типу даних змінної в лівій. Але часто трапляються випадки коли змінній необхідно присвоїти значення виразу, тип якого не співпадає із типом змінної. Наприклад, коли змінній дійсного типу необхідно присвоїти значення цілого типу. В цьому випадку необхідно виконати перетворення (приведення) виразу до необхідного типу (типу змінної). У мові розрізняють два види перетворень змінних – *явне* та *неявне*.

Неявне перетворення типів використовується в тому випадку, коли дане перетворення є «природним». Тобто, наприклад, перетворюється величина типу `float` в величину типу `double`.

Приклад.

```
float f = 1.23;  
double d = f;    //Неявне перетворення
```

Таке перетворення є природним, оскільки обидва типи використовуються для представлення дійсних типів, причому цільовий тип має більший діапазон представлення і більшу точність. При такому перетворення не відбувається втрата інформації. При проведенні неявного перетворення немає необхідності вказувати цільовий тип (тип до якого здійснюється перетворення).

Явне перетворення типів вимагає явного задання цільового типу, до якого здійснюється перетворення. Цільовий тип вказується в дужках перед значенням тип якого перетворюється.

| Загальний вигляд | Приклад |
|--------------------------------------|---|
| <змінна> = (<цільовий тип>) <вираз>; | double d = 2.9; float f = (float) d; |

Для цього типу повинно існувати неявне перетворення до типу змінної, в якій буде збережено результат.

Арифметичні та логічні вирази

Вирази

Виразом називають послідовність операцій, операндів і розділових знаків, що задають деякі обчислення. В залежності від значення, яке одержується в результаті цих обчислень, вираз поділяють на арифметичні та логічні вирази.

Арифметичні вирази

Арифметичним виразом називають вираз, в результаті обчислення якого одержуємо числове значення. При цьому можуть використовуватися бінарні та унарні операції арифметичні операції.

Конструктивно означення **арифметичного виразу (АВ)** можна дати за допомогою правил його побудови:

1. Довільна числова константа – АВ
2. Довільна числова змінна – АВ
3. Довільний виклик підпрограми (процедури або функції) – АВ
4. Якщо А – АВ, то (А) – АВ
5. Якщо А, В – АВ, то А+В, А-В, А*В, А/В – арифметичні вирази

При складанні арифметичного виразу необхідно використовувати **наступні правила:**

1. Усі складові частини виразу записують в один рядок

$$\frac{a+b}{2} \text{ як } (a+b)/2, \quad 2a + \frac{4b-c}{(a+b)} \text{ як } 2*a + (4*b - c)/(a+b)$$

2. У виразах використовується тільки круглі дужки. При цьому кількість відкриваючих дужок має дорівнювати кількості закриваючих.
3. Обчислення арифметичного виразу здійснюється зліва направо, згідно із пріоритетом операцій.

У програмуванні, так само як і в математиці, існує пріоритет виконання арифметичних дій, тобто визначається, яким діям надається перевага перед іншими під час обчислення значення арифметичного виразу. Наведемо арифметичні операції мови C++ саме в порядку зменшення їх пріоритетності:

- *, / – множення і ділення;
- % – остача від ділення націло двох цілих чисел;
- +, – – додавання і віднімання.

Якщо операції, які йдуть безпосередньо одна за одною мають однаковий пріоритет, то вони виконуються в такому порядку в якому записані.

Приклад: $A * 2 * T + R / T * N - S$

- 1) $A * 2$ 2) $A * 2 * T$ 3) R / T 4) $R / T * N$ 5) $A * 2 * T + R / T + N$ 6) ... – S

Якщо з яких-небудь причин необхідно змінити порядок виконання операцій, то для цього використовуються круглі дужки (**Наприклад:** $S / (Q + T)$). Якщо у виразі є декілька вкладених дужок і вони вкладені одна в одну, то спочатку обчислюється вираз в самих внутрішніх дужках, а потім у зовнішніх.

$$A * (B + C * (D + E)) \quad 1) D + E \quad 2) C * (D + E) \quad \dots$$

В арифметичних виразах можуть використовуватися і **стандартні функції**. Пріоритетність обчислення функцій найвища. Отже, якщо в арифметичному виразі

використовуються функції, то спочатку буде обчислене їх значення, а потім над цими результатами будуть виконані інші дії.

Якщо аргумент функції є арифметичним виразом, то спочатку обчислюється цей вираз, а потім значення функції.

У мовах програмування аргументи функції вказуються в дужках. Тобто в математиці ви пишете $\sin x$, а в C++ треба писати $\sin(x)$.

Наведемо список функцій, які описано в файлі заголовку **math.h**

| Математичний запис | Запис на C++ | Призначення |
|--------------------------|---------------------------|-------------------------------------|
| $\cos x$ | $\cos(x)$ | косинус x – радіани |
| $\sin x$ | $\sin(x)$ | синус x – радіани |
| $\operatorname{tg} x$ | $\tan(x)$ | тангенс x – радиан |
| $\operatorname{ch} x$ | $\cosh(x)$ | гіперболічний косинус x – радіани |
| $\operatorname{sh} x$ | $\sinh(x)$ | гіперболический синус x – радиани |
| $\operatorname{th} x$ | $\tanh(x)$ | гіперболічний тангенс x – радіани |
| $\arccos x$ | $\operatorname{acos}(x)$ | арккосинус числа x |
| $\arcsin x$ | $\operatorname{asin}(x)$ | арксинус числа x |
| $\operatorname{arctg} x$ | $\operatorname{atan}(x)$ | арктангенс числа x |
| e^x | $\exp(x)$ | значення e в степені x |
| x^y | $\operatorname{pow}(x,y)$ | число x в степені y |
| $ x $ | $\operatorname{fabs}(x)$ | модуль числа x |
| $\sqrt{}$ | $\operatorname{sqrt}(x)$ | квадратний корінь числа x |
| $\ln x$ | $\log(x)$ | натуральний логарифм x |
| $\log_{10} x$ | $\log_{10}(x)$ | десятковий логарифм x |

Бінарні операції. Бінарні операції – це операції, для виконання яких необхідно два операнди.

| Операція | Позначення | Приклад |
|----------|--------------------|--------------|
| + | додавання | $z = x + y$ |
| – | віднімання | $z = x - y$ |
| * | множення | $z = x * y$ |
| / | ділення | $z = x / y$ |
| % | остача від ділення | $z = x \% y$ |

Окрім бінарних операцій в арифметичному виразі можуть бути присутні також унарні операції «+», «–» та операції інкременту «++» і декременту «– –». Унарний мінус використовується для зміни знаку. Операції інкременту і декременту використовують для збільшення та зменшення значення змінної на одиницю. Ці операції можуть вживатися у префіксній та постфіксній формі.

| Операція | Аналог |
|-----------------|-------------|
| $i++$ або $++i$ | $i = i + 1$ |
| $i--$ або $--i$ | $i = i - 1$ |

Якщо унарну операцію вжито у префіксній формі у виразі, то вона виконується до використання значення змінної у виразі. Якщо ж унарну операцію вжито у постфіксній формі, то операція виконується після використання значення змінної у виразі.

| Операція | Аналог з бінарними операціями |
|--------------------------------------|---|
| <code>int i = 5;</code> | <code>int i = 5;</code> |
| <code>int j = ++i; // j=6 i=6</code> | <code>i=i+1;</code> <code>int j = i; // j=6 i=6</code> |
| <code>int i = 5;</code> | <code>int i = 5;</code> |
| <code>int j = i++; // j=5 i=6</code> | <code>int j = i;</code> <code>i=i+1; // j=5 i=6</code> |
| <code>int i = 5;</code> | <code>int i = 5;</code> |
| <code>int j = --i; // j=4 i=4</code> | <code>i=i-1;</code> <code>int j = i; // j=4 i=4</code> |
| <code>int i = 5;</code> | <code>int i = 5;</code> |
| <code>int j = i--; // j=5 i=4</code> | <code>int j = i;</code> <code>i=i-1; // j=5 i=4</code> |

Побітові операції

У мові C# є можливість здійснювати побітові операції над розрядами аргументів

| Операція | Позначення | Приклад |
|--------------------|---------------------------|---|
| <code>&</code> | побітове «і» | <code>int x=10; //x=1010₍₂₎</code> <code>int y=7; //y=0111₍₂₎</code> <code>z=x&y // z=2=0010₍₂₎</code> |
| <code> </code> | побітове «або» | <code>int x=10; //x=1010₍₂₎</code> <code>int y=7; //y=0111₍₂₎</code> <code>z=x y //z=15=1111₍₂₎</code> |
| <code>^</code> | Побітове «виключаюче або» | <code>int x=10; //x=1010₍₂₎</code> <code>int y=7; //y=0111₍₂₎</code> <code>z=x^y //z=13=1101₍₂₎</code> |

Бінарні операції зсуву

| Операція | Позначення | Приклад |
|-----------------------|--|---|
| <code>>></code> | зсув розрядів вправо (змінна)=(змінна)>>(кільк. розрядів) | <code>int i=4; //i=100₍₂₎</code> <code>i=i>>1; //i=2=10₍₂₎</code> |
| <code><<</code> | зсув розрядів вліво (змінна)=(змінна)<<(кільк. розрядів) | <code>int i=4; //i=100₍₂₎</code> <code>i=i<<2; // i=16=10000₍₂₎</code> |

Операції присвоєння

Якщо при виконанні бінарної операції результат зберігається у змінній, що є першим аргументом, то можна використати так звані операції присвоювання.

| Операція | Операція | Аналог з бінарними операціями |
|-----------------|---|--|
| <code>+=</code> | <code>int i = 5;</code> <code>i += 3; // i=8</code> | <code>int i = 5;</code> <code>i=i+3; // i=8</code> |
| <code>-=</code> | <code>int i = 5;</code> <code>i -= 3; // i=2</code> | <code>int i = 5;</code> <code>i=i-3; // i=2</code> |
| <code>*=</code> | <code>int i = 5;</code> <code>i *= 3; // i=15</code> | <code>int i = 5;</code> <code>i=i*3; // i=15</code> |
| <code>/=</code> | <code>int i = 6;</code> | <code>int i = 6;</code> |

| | | |
|------------------------|---|--|
| | <code>i /= 3; // i=2</code> | <code>i=i/3; // i=2</code> |
| <code>>>=</code> | <code>int i = 5; i >>= 1; // i=2</code> | <code>int i = 5; i=i>>1; // i=2</code> |
| <code><<=</code> | <code>int i = 5; i <<= 1; // i=10</code> | <code>int i = 5; i=i<<1; // i=10</code> |
| <code>&=</code> | <code>int i = 5; int j = 7; i &= j; // i=5</code> | <code>int i = 5; int j = 7; i=i&j; // i=5</code> |
| <code>^=</code> | <code>int i = 5; int j = 7; i ^= 1; // i=2</code> | <code>int i = 5; int j = 7; i=i^j; // i=2</code> |
| <code> =</code> | <code>int i = 5; int j = 7; i = 1; // i=7</code> | <code>int i = 5; int j = 7; i=i j; // i=7</code> |

Логічні вирази

Логічним виразом називається такий вираз, внаслідок обчислення якого одержується логічне значення типу `bool` (*true* або *false*).

Прикладом логічного виразу є вираз, що містить операції порівняння

| Операція | Позначення | Приклад |
|--------------------|------------------|----------------------|
| <code>==</code> | рівність | <code>x==y</code> |
| <code>></code> | Більше | <code>x>y</code> |
| <code><</code> | Менше | <code>X<y</code> |
| <code>>=</code> | більше або рівно | <code>x>=y</code> |
| <code><=</code> | менше або рівно | <code>X<=y</code> |
| <code>!=</code> | не рівно | <code>x!=y</code> |

У арифметичному виразі можуть також використовуватися логічні операції

| x | y | x&&y (логічне «і») | x y (логічне «або») | x ^ y (виключаюче «або») | !x (заперечення) |
|-------|-------|--------------------------|------------------------------|--------------------------------|---------------------|
| false | false | false | false | false | true |
| false | true | false | true | true | true |
| true | false | false | true | true | false |
| true | true | true | true | false | false |

Конструктивно логічний вираз можна означити так:

1. Логічна константа (*true* або *false*) – Логічний Вираз (ЛВ).
2. Логічна змінна – ЛВ.
3. Якщо L – ЛВ то (L) – ЛВ.
4. Якщо L – ЛВ то $!L$ – ЛВ.
5. Якщо A_1, A_2 – арифм. вирази то
 $A_1 < A_2, A_1 > A_2, A_1 \leq A_2, A_1 \geq A_2, A_1 == A_2, A_1 != A_2$, – ЛВ
6. Якщо L_1, L_2 – ЛВ то $L_1 \&\& L_2, L_1 || L_2, L_1 \wedge L_2$, – ЛВ

Пріоритет операцій

| Пріоритет | Операції |
|-----------|--|
| 1 | () [] . ++(префікс) --(префікс) new sizeof typeof unchecked |
| 2 | ! ~ (ім'я типу) +(унарний) -(унарний) (постфікс)++ (постфікс)-- |
| 3 | * / % |
| 4 | + - |
| 5 | << >> |
| 6 | < > <= >= is |
| 7 | == != |
| 8 | & |
| 9 | ^ |
| 10 | |
| 11 | && |
| 12 | |
| 13 | ?: |
| 14 | = += -= *= /= %= &= = ^= <<= >>= |

Складений оператор.

Складений оператор – це складний оператор, який об'єднує декілька операторів в одну групу.

Форма його запису наступна:

```
{
    оператор 1;
    оператор 2;
    .....
    оператор n;
}
```

В даній конструкції дужки “{” та “}” мають назву операторних дужок. “{” – відкриваюча операторна дужка; “}” – закриваюча операторна дужка. Складений оператор визначається як єдиний оператор. Його можна вставляти в довільне місце програми, де дозволено використання одного простого оператора.

Керуючі конструкції мови C++

Конструкція мови C++ називається керуючою, якщо вона може змінювати послідовність виконання дій. До керуючих конструкцій відносяться умовний та циклічний оператори.

Умовний оператор

Дозволяє вибрати оператор, який буде виконуватися в залежності виконання чи невиконання деякої умови. Існують дві форми для даного оператора повна та скорочена.

Повна форма

| Програмна структура | Аналог на мові блок-схем | Приклад |
|--|--------------------------|---|
| <pre>if (<умова>) <оператор1>; else <оператор2>;</pre> | | <pre>if (x>y) max=x; else max=y;</pre> |

Скорочена форма

| Програмна структура | Аналог на мові блок-схем | Приклад |
|--|--------------------------|---------------------------------|
| <pre>if (<умова>) <оператор1>;</pre> | | <pre>if (x!=0) z=1/x;</pre> |

Якщо в умовному операторі при виконанні чи невиконанні умови необхідно виконати декілька операторів, то необхідно ці оператори помістити в складений оператор.

Повна форма

| Програмна структура | Аналог на мові блок-схем | Приклад |
|--|--------------------------|---|
| <pre> if (<умова>) { <оператор1.1>; <оператор1.N>; } else { <оператор2.1>; <оператор2.M>; } </pre> | | <pre> if (x>y) { max=x; min=y; } else { max=y; min=x; } </pre> |

Скорочена форма

| Програмна структура | Аналог на мові блок-схем | Приклад |
|---|--------------------------|--|
| <pre> if (<умова>) { <оператор 1>; <оператор N>; } </pre> | | <pre> if (x>0) { z=1/x; l=y/x; } </pre> |

Приклад. Знайти максимальне та мінімальне із двох дійсних чисел.

```

#include <vcl.h>
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
    double a,b;
    cout<<"a=";
    cin>>a;
    cout<<"b=";
    cin>>b;
    double max,min;
    if (a>b)
    {
        max=a;
        min=b;
    }
    else
    {
        max=b;
        min=a;
    }
}

```



```
cout<<"max="<<max<<endl;
cout<<"min="<<min<<endl;
system("pause");
return 0;
}
```

Результати роботи програми

```
a=3
b=8
max=8
min=3
```

Умовний оператор (?:)

Якщо значення виразу може дорівнювати одному із двох значень в залежності від виконання чи невиконання деякої умови, то можна скористатися умовним оператором (?:) .

<умова>?<значення1>:<значення 2>

Якщо умова виконується (логічний вираз має значення true), то результатом виразу є **значення1**, інакше – **значення2**.

Часто цей оператор використовують у правій частині оператора присвоєння.

| | |
|---|--|
| Загальний вигляд умовного оператора (?:) | Аналог з використанням умовного оператора if |
| <змінна>=<умова>?<значення1>:<значення 2>; | <pre>if (<умова>) <змінна>=<значення 1>; else <змінна>=<значення 2>;</pre> |

Приклад. Знайти максимальне із двох дійсних чисел.

| | |
|---|---|
| З використанням умовного оператора (?:) | З використанням умовного оператора if |
| max=(x>y)? x : y ; | <pre>if (x>y) max=x; else max=y;</pre> |

Оператор вибору switch

Оператор switch дозволяє передавати керування одному з декількох операторів в залежності від значення виразу, який називають селектором вибору. У якості селектора вибору може бути вираз цілого типу, типу char, типу або типу enum.

| | |
|--|---|
| <u>Загальна форма</u> | <u>Приклад.</u> Вводиться оцінка – цифра, вивести оцінку прописом (селектор вибору цілого типу). |
| <pre>switch (<селектор вибору>) { case <константа 1> : <оператор 1>;</pre> | <pre>#include <vcl.h> #include <iostream> using namespace std; int main(int argc, char* argv[]) { int mark; cout<<"mark="; cin>>mark; switch (mark) { case 2: cout<<"Незадовільно";</pre> |

| | |
|---|---|
| <pre> break; case <константа 2> : <оператор 2>; break; case <константа N> : <оператор N>; break; default : <оператор N+1>; break; } </pre> | <pre> break; case 3: cout<<"Задовільно."; break; case 4: cout<<"Добре"; break; case 5: cout<<"Відмінно"; break; default: cout<<"Неправильна оцінка."; break; } system("pause"); return 0; } </pre> |
|---|---|

Якщо для декількох варіантів необхідно виконати одні і ті ж оператори, то ці оператори вказують тільки для одного з варіантів, а для всіх інших не вказуємо ні необхідних операторів, ні операторів `break`.

Приклад. З клавіатури вводиться оцінка у національній шкалі, необхідно вивести повідомлення про те, чи зараховано студенту залік.

```

#include <vcl.h>
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
    int mark;
    cout<<"mark=";
    cin>>mark;
    switch (mark)
    {
        case 1:
        case 2: cout<<"незараховано ";
                break;

        case 3:
        case 4:
        case 5: cout<<"зараховано";
                break;
    }

    system("pause");
    return 0;
}

```

Приклад. З клавіатури вводиться буква у нижньому регістрі, з'ясувати, чи є буква голосною. При розв'язанні цього завдання використаємо оператор **switch**, у якому селектор вибору та константи вибору типу **char**.

```
#include <vcl.h>
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
    char c;
    cout<<"c=";
    cin >> c;
    switch (c)
    {
        case 'a':
        case 'o':
        case 'y':
        case 'и':
        case 'i':
        case 'e': cout<<"голосна";
                  break;
        default: cout<<"приголосна";
                  break;
    }
    system("pause");
    return 0;
}
```

Оператори циклу

Оператор циклу **while**

Оператор **while** циклічно виконує свою інструкцію до тих пір, поки умова виконується (логічний вираз приймає значення **true**).

| Програмна структура | Аналог на мові блок-схем | Приклад. Знайти суму перших n чисел. |
|---------------------------------------|--------------------------|---|
| while (<умова>) <оператор>; | | <pre>int sum=0; int i=1; while (i<=n) sum=sum+i++;</pre> |

Якщо тіло циклу складається з більше ніж одного оператора, то необхідно використати складений оператор.

| Програмна структура | Аналог на мові блок-схем | Приклад. Знайти суму і добуток перших n чисел. |
|--|--------------------------|---|
| <pre>while (<умова>) { <оператор 1>; <оператор N>; }</pre> | | <pre>int sum=0; int mult=1; int i=1; while (i <= n) { sum=sum+(i++); mult=mult*(i++); i++; }</pre> |

Оператор циклу do-while

Оператор циклу do-while відрізняється від оператора while тим, що перевірка умови виконується не до, а після виконання інструкції (оператора).

| Програмна структура | Аналог на мові блок-схем | Приклад. Знайти суму перших n чисел. |
|---|--------------------------|--|
| <pre>do { <оператор>; } while (<умова>)</pre> | | <pre>int sum=0; int i=1; do { sum=sum+i++; } while (i <= n)</pre> |

Оператор циклу for

Загальна форма оператора наступна

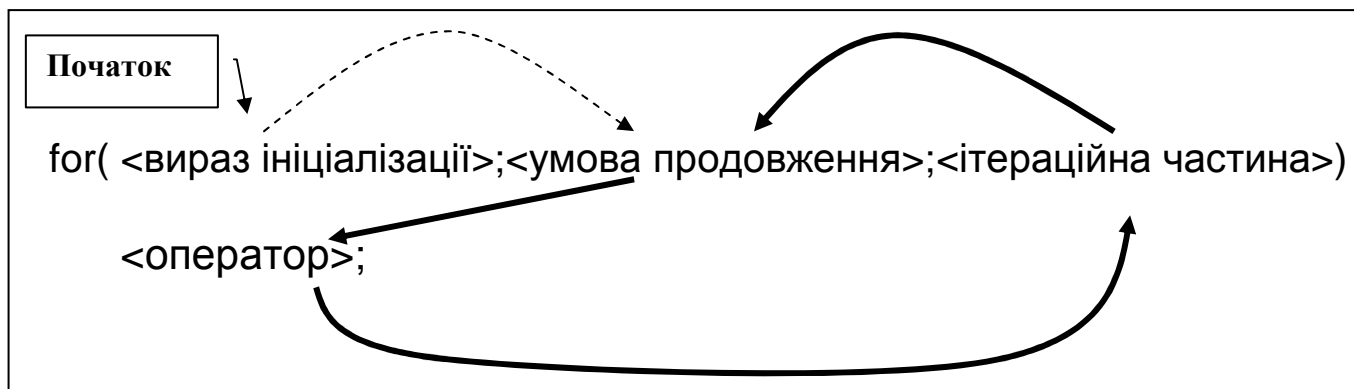
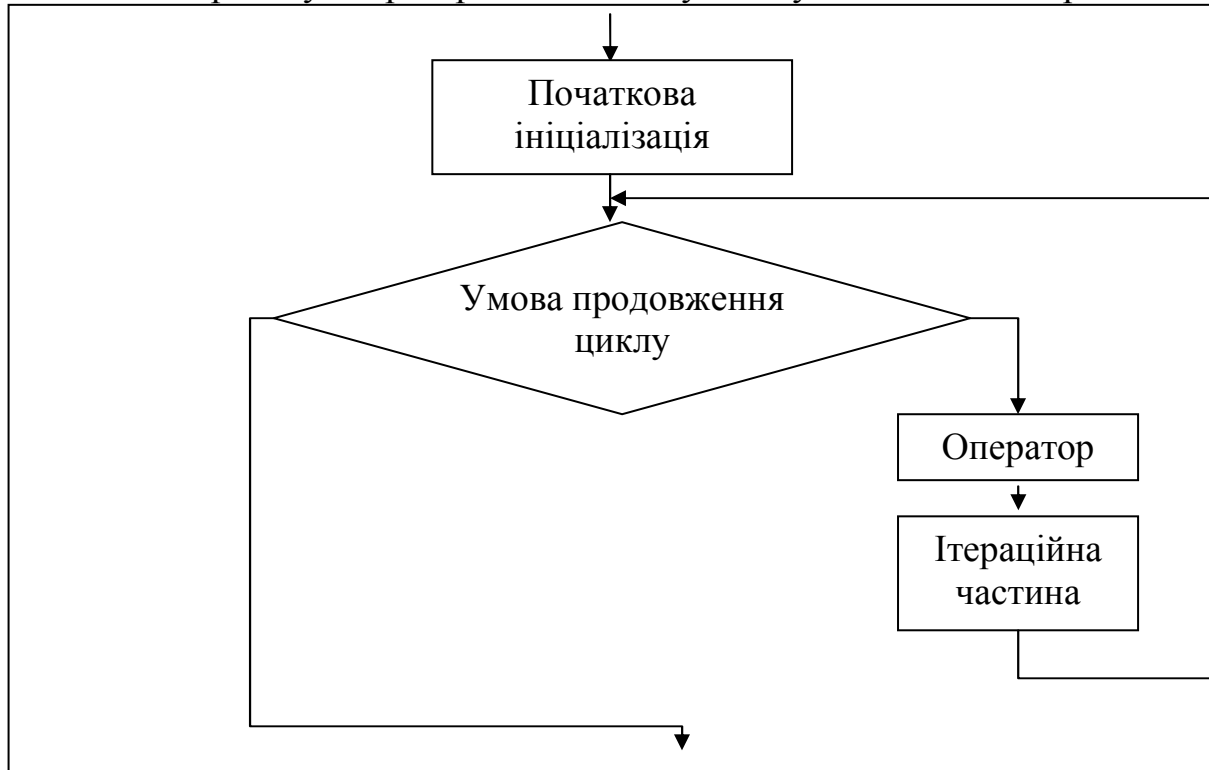
for (<вираз ініціалізації>; <логічна умова>; <ітераційна частина>)
 <оператор>;

Оператор for виконує наступні дії:

1. Обчислює вираз ініціалізації. У цій частині допустима ініціалізація декількох лічильників циклу.
2. Перевіряється логічна умова. Якщо умова невірна то робота циклу завершується і передається управління наступному оператору.

3. Якщо умова істинна, виконується інструкція даного оператора.
4. Виконується приріст одного або декількох лічильників цикл (або виконується довільна інша операція).
5. Здійснюється перехід до кроку 2.

Схематично роботу оператора поки виконується умова можна зобразити так



Приклад. Знайти суму перших n натуральних чисел.

```

#include <vcl.h>
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
    int n;
    cout<<"n=";
    cin >> n;
    int s=0;

    for(int i=1;i<=n;i++)
  
```

```

    s=s+i;

    cout<<"s="<<s<<endl;

    system("pause");
    return 0;
}

```

Проілюструємо випадок, коли частина ініціалізації та ітераційна частина мість декілька виразів, розділених комою.

Приклад. Обчислити значення виразу

$$\frac{1}{0,3} + \frac{2}{0,4} + \dots + \frac{n}{0,3 + 0.1 * (n - 1)}$$

```

#include <vcl.h>
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{

    int n;
    cout<<"n=";
    cin>>n;
    double d;
    int c;
    double sum=0;

    for (c = 1, d=0.3 ; c <= n ; c++, d+=0.1)
        sum +=c/d;

    cout<<"sum="<<sum<<endl;

    system("pause");
    return 0;
}

```

Кожна з частин оператора циклу може бути відсутньою, але розділові знаки “;” є обов’язковими. Так нескінчений цикл може бути задано так

```

for (    ;    ;    )
    <оператор>;

```

Оператори **break** та **continue**

Оператор **continue** може бути використаний у будь-якому із циклів у випадку, коли немає потреби виконувати всі оператори у тілі циклу у поточній ітерації, а необхідно одразу перейти до наступної ітерації.

Приклад. Знайти добуток непарних натуральних чисел, що менші за К.

```

#include <vcl.h>
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
    int K;
    cout<<"K=";
    cin>>K;

    int mult=1;
    for (int i = 1;i<=K ; i++)
    {
        if ((i % 2) == 0) continue;
        mult *= i;
    }

    cout<<"mult="<<mult<<endl;

    system("pause");
    return 0;
}

```

Оператор **break** у циклах використовують для негайного завершення самого внутрішнього циклу, у тілі якого він знаходиться.

Приклад. Знайти найменше значення факторіалу натурального числа, що перевищує число **K**.

```

#include <vcl.h>
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
    int K;
    cout<<"K=";
    cin>>K;

    int fakt=1;
    for (int i = 1; ; i++)
    {
        fakt *= i;
        if (fakt > K) break;
    }
    cout<<"fakt="<<fakt<<endl;
    system("pause");
    return 0;
}

```

```
}
```

Приклад. Вивести на екран усі натуральні двоцифрові числа, у яких друга цифра не перевищує першу.

При розв'язанні цього завдання використаємо оператор **break**, який здійснює вихід тільки із вкладеного циклу (робота зовнішнього циклу продовжується).

```
#include <vcl.h>
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
    for (int i = 1; i <= 9; i++)
    {
        for (int j = 1; j <= 9; j++)
        {
            if (j > i) break;
            cout<<i<<j<<endl;
        }
    }

    system("pause");
    return 0;
}
```

Результат роботи програми

```
11
21
22
31
32
33
..
99
```


Масиви

Можна дати декілька означень масиву.

Масив – це структура даних, що являє собою однорідну (за типом), фіксовану (за розміром і конфігурацією) сукупність елементів, упорядкованих за номерами.

Масив – це нумерована послідовність елементів одного типу.

Масиви відносяться до структур з прямим або довільним доступом. Щоб визначити окремий елемент масиву потрібно вказати його *індекси*. У якості індексів використовуються значення цілого типу. В одновимірному масиві індекс – це порядковий номер елементу масиву. Нумерація елементів завжди починається з нуля. Кількість індексів називають *розмірністю*, кількість дозволених значень кожного індексу – *діапазоном*, а сукупність розмірності та діапазону – *формою масиву*.

Класифікація масивів може здійснюватися за багатьма ознаками:

- 1) можливістю зміни значень елементів масиву (масиви-константи та масиви-змінні);
- 2) кількістю індексів, які необхідно вказати для того, щоб ідентифікувати елемент у масиві (масиви одновимірні, двовимірні, тривимірні і т.д.);
- 3) можливістю зміни кількості елементів масиву в процесі виконання програми (статичні та динамічні масиви).

Статичні масиви

Одновимірні масиви

Описуючи статичний одновимірний масив, наперед необхідно вказати максимально допустиму кількість елементів.

| Загальний вигляд | Приклад |
|---|----------------------------|
| <Тип> <Ім'я масиву> [<Кількість елементів>] | int a [3]; double b[7]; |

Описуючи масив його можна одразу ініціалізувати, тобто надати елементам масиву початкові значення. Для цього використовують *ініціалізатор масиву* – у фігурних дужках через кому наводиться список початкових значень елементів.

| Загальний вигляд | Приклад |
|---|--------------------|
| <Тип><Ім'я масиву>[<Кількість елементів>]= { <список значень> }; | int a [3]={2,8,5}; |

Якщо кількість значень є меншою ніж кількість елементів масиву, то всі інші елементи до кінця масиву заповнюються нульовими значеннями.

Зауважимо, що у випадку, якщо ініціалізація елементів масиву здійснюється одразу після його опису і кількість значень співпадає з кількістю елементів масиву, то кількість елементів масиву під час його опису вказувати не обов'язково.

| Загальний вигляд | Приклад |
|--|--------------------|
| <Тип><Ім'я масиву>[]= { <список значень> }; | int a []={2,8,5}; |

Для ідентифікації елемента масиву необхідно вказати ім'я масиву та індекс елемента, який записується у квадратних дужках після імені масиву

<ім'я масиву> [<індекс>]

Приклад.

```
a [ 0 ] = 2;
a [ 1 ] = 9;
a [ 2 ] = a[ 0 ]+3;
```

Наведемо декілька важливих зауважень:

- 1) елементи масиву можуть бути довільного, але тільки одного типу;
- 2) діапазон не може змінюватися під час виконання програми;
- 3) значення індексу не повинно виходити за межі описаного діапазону;
- 4) оскільки процедури введення/виведення розраховані на введення та виведення значень простих типів, то масив потрібно вводити/виводити поелементно, тобто кожен елемент окремо.

Як було зауважено, межі зміни діапазону індексів повинні бути константами, і діапазон не може змінюватися під час виконання програми. Але ж при розв'язанні задач, як правило, кількість елементів масиву вводить користувач під час виконання програми. Тому в залежності від задачі при описі масиву необхідно вказати гранично можливий діапазон зміни індексу. Для збереження ж справжнього діапазону зміни індексу описують додаткові змінні, значення яких не можуть виходити за вказані гранично можливі.

Приклад. Якщо необхідно зберігати роки народження учнів у класі (учнів не більше 40), то масив описують так

int Y[40];

Для збереження дійсної кількості учнів у класі описуємо додатково змінну цілого типу n , значення якої буде задавати користувач під час виконання програми. Зрозуміло, що значення змінної n у даному випадку не може бути більшим за 40.

| | |
|------------|--|
| int Y[40]; | // Y – масив років народження |
| int n; | // n – кількість учнів у класі ($n \leq 40$) |

Тут

Y[0] – рік народження 1-го учня;
 Y[1] – рік народження 2-го учня;

 Y[n – 1] – рік народження n -го учня.

Наведемо текст програми, що знаходить рік народження найстаршого учня.

```
#include <vcl.h>
#pragma hdrstop
#pragma argsused
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
```

```

int Y[40];           //Опис масиву
int n;               //Кількість елементів масиву

cout<<"n=";          //Введення кількості елементів масиву
cin>>n;

                        //Введення елементів масиву
for(int i=0;i<n;i++)
{
    cout<<"Y["<<i<<"]=";
    cin>>Y[i];
}
int min=Y[0];         //Знаходження найменшого елемента масиву
for(int i=1;i<n;i++)
{
    if (Y[i]<min) min=Y[i];
}

cout<<"Рік нар. найстаршого учня ="<<min<<endl;
system("pause");
return 0;
}

```

Приклад. Необхідно знайти загальну масу автомобілів у парку.

Будемо вважати, що кількість автомобілів не перевищує 100.

```

double C[100];        // C – масив для збереження маси автомобілів
int n;                 // n – кількість автомобілів ( $n \leq 100$ )

```

Тут

C[0] – маса 1-го автомобіля;
 C[1] – маса 2-го автомобіля;

 C[n – 1] – маса n-го автомобіля.

Приклад. Дано $a, b \in R^n$, знайти $c = a + b$.

Розв'язання

Для збереження векторів використаємо однойменні одновимірні масиви a, b і c . Будемо вважати, що $n \leq 50$.

```

#include <vcl.h>
#pragma hdrstop
#pragma argsused
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
    int a[50];           //Опис масивів
    int b[50];
    int c[50];
    int n;

```

```

cout<<"n=" ;
cin>>n;

                                //Введення масиву a
cout<<"----- Input a -----"<<endl;
for(int i=0;i<n;i++)
{
    cout<<"a["<<i<<"]=";
    cin>>a[i];
}

                                //Введення масиву b
cout<<"----- Input b -----"<<endl;
for(int i=0;i<n;i++)
{
    cout<<"b["<<i<<"]=";
    cin>>b[i];
}

                                //Знаходження масиву c
for(int i=0;i<n;i++)
{
    c[i]=a[i]+b[i];
}

                                //Виведення масиву c
cout<<"----- Output c -----"<<endl;
for(int i=0;i<n;i++)
{
    cout<<"c["<<i<<"]="<<c[i]<<endl;
}
system("pause");
return 0;
}

```

Представлення одновимірного масиву в пам'яті ЕОМ

Для збереження масивів в пам'яті ЕОМ використовується послідовне представлення. Тобто для збереження масиву наперед виділяється ділянка оперативної пам'яті величиною

<кількість елементів> * <кількість байтів для збереження одного елемента>

і елементи розміщуються послідовно один за одним.

Приклад.

```
int A[10];
```

Масив А в пам'яті ЕОМ буде розміщено наступним чином

| | | | | | | | |
|------------------|------|------|------|-----|------|-----|--------------------|
| ... | A[0] | A[1] | A[2] | ... | A[9] | ... | Елементи масиву |
| Область масиву А | | | | | | | |

Одновимірні масиви-константи

У мові C++ є можливість описання масиву-константи. При цьому необхідно вказати всі елементи описуваного масиву.

```
const <тип елементів> <ім'я масиву> [<кільк. елем.>] = {<елемент 0>,<елемент 1>,<елемент N-1>};
```

Приклад.

```
const int v[3]={10,21,37};
```

Спроба змінити значення масиву-константи призведе до помилки компіляції.

Багатовимірні масиви

У C++ також є можливість описувати багатовимірні масиви

```
<Тип><Ім'я масиву>[<Кількість елементів 1>][<Кількість елементів 2>] ... [<Кількість елементів N>];
```

Найчастіше використовують двовимірні масиви, які асоціюють з матрицями (що складаються з рядків і стовпців).

| Загальний вигляд | Приклад |
|--|---------------|
| <Тип><Ім'я масиву>[<Кількість рядків>][<Кількість стовпців>] | int a [3][2]; |

Як і одновимірні масиви, двовимірні масиви можна ініціалізувати одразу після їх опису.

| Загальний вигляд | Приклад |
|---|--|
| <Тип><Ім'я масиву>[<Кільк. рядків>][<Кільк.стовпців>]={<Список значень>}; | int a [3][2]={2,8, 7,4, 7,5}; |
| <Тип><Ім'я масиву>[<Кільк. рядків>][<Кільк.стовпців>]={ {<Список значень рядка1>},{<Список значень рядка2>}, ... }; | int a [3][2]={ {2,8}, {7,4}, {7,5} }; |
| <Тип><Ім'я масиву> [] [<Кількість стовпців>]={ {<список значень рядка1>},{<список значень рядка2>}, ... }; | int a [] [2]={ {2,8}, {7,4}, {7,5} }; |
| // Можна не вказувати кількість рядків | |

Для ідентифікації елементу масиву необхідно вказати ім'я та індекси

```
<ім'я масиву> [<індекс 1>][<індекс2>] ... [<індекс N>]
```

Приклад.

a[2][1] – елемент двивимірного масиву a, що знаходиться у рядку під номером 2, і стовпці під номером 1.

Представлення статичного двовимірного масиву в пам'яті ЕОМ

Для збереження статичних двовимірних масивів, як і для одновимірних, в пам'яті ЕОМ використовується послідовне представлення. При цьому наперед виділяється ділянка оперативної пам'яті величиною

<кількість рядків> * <кількість стовпців> * <розмір одного елемента масиву>
і елементи розміщуються послідовно, один за одним. Спочатку елементи першого рядка, потім другого, третього і т.д.

Приклад.

```
int A[3][2];
```

Змінну А

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \\ a_{2,0} & a_{2,1} \end{pmatrix}.$$

у пам'яті ЕОМ буде розміщено наступним чином:

| | | | | | | | | |
|-----|------------------|--------|---------|--------|---------|--------|-----|--------------------|
| | | | | | | | | Комірки |
| ... | A[0,0] | A[0,1] | A[1,0] | A[1,1] | A[2,0] | A[2,1] | ... | Елементи масиву |
| | Рядок 1 | | Рядок 2 | | Рядок 3 | | | |
| | Область масиву А | | | | | | | |

Показчики

Показчик – це змінна, яка може містити адресу оперативної пам'яті. Як правило, показчики використовують для роботи з динамічною пам'яттю та описанні формальних параметрів підпрограм, що можуть змінюватися у цій підпрограмі. Опис показчика містить символ «*».

| Загальний вигляд | Приклад |
|---------------------|---------|
| <Тип> * <Показчик>; | int *p; |

Знак «*» можна вказувати як біля типу даних, так і біля показчика. Як правило, його вказують біля показчика.

Показчик може містити адресу іншої змінної (для взяття адреси використовується оператор «&»). При цьому тип змінної повинен співпадати з типом, який використано при описі показчика показчика.

| Загальний вигляд | Приклад |
|----------------------------|--|
| < Показчик > = &<Змінна >; | int *p; //Опис показчика на тип int int d; //Опис змінної типу int p=&d; //Показчик p одержує адресу змінної d |

Доступ до ділянки пам'яті, адресу якої містить показчик здійснюється через операцію розіменування (символ «*» ставиться перед показчиком).

| Загальний вигляд | Приклад |
|-------------------------------|--|
| * < Показчик > = <значення >; | int k; int *m=&k; *m=27; double r; double *d=&r; |

| | |
|--|----------------------|
| | <code>*d=2.5;</code> |
|--|----------------------|

Адресна арифметика. При збільшенні (зменшенні) покажчика на деяке ціле число `count` значення покажчика (адреса) збільшується (зменшується) на величину, що дорівнює добутку

`count*sizeof (<Тип покажчика>)`

| Оператор | Результат |
|--|--|
| <code>int *p; p=p+1; // Або ж p++</code> | Значення покажчика (адресу) <code>p</code> збільшили на <code>1*sizeof(int)=4</code> байти |
| <code>int *p; p=p+3; // Або ж p++</code> | Значення покажчика (адресу) <code>p</code> збільшили на <code>3*sizeof(int)=12</code> байтів |
| <code>double *d; d=d+3;</code> | Значення покажчика (адресу) <code>d</code> збільшили на <code>3*sizeof(double)=24</code> |

Виділення динамічної пам'яті

Динамічна пам'ять може бути виділена за допомогою оператора `new` і звільнена за допомогою `delete`, або ж виділена за допомогою `malloc` і звільнена за допомогою оператора `free`.

| Загальний вигляд (<code>new – delete</code>) | Приклад |
|---|--|
| <code><покажчик> = new <тип покажчика>; //Виділення пам'яті //Використання пам'яті delete <покажчик>; //Звільнення пам'яті</code> | <code>int *p; p= new int; delete p;</code> |

Функції `malloc` та `free` описано у файлі заголовку `cstdlib`

```
void *malloc(size_t Size);
```

```
void free (void *Memory);
```

тому для їх використання необхідно цей файл заголовка підключити. Оскільки функція `malloc` типу `void*`, то при її використанні необхідно виконувати приведення типу до необхідного типу даних. Зауважимо, що у різних операційних системах кількість байтів, що виділяється для типів даних може бути різною, тому, щоб програма була незалежною від операційної системи, при виділенні пам'яті необхідно використовувати функцію `sizeof`.

| Загальний вигляд (<code>malloc – free</code>) | Приклад |
|---|--|
| <code>..... //Виділення пам'яті <Покажчик>=(Тип *) malloc(<Кількість байтів>; //Використання пам'яті //Звільнення пам'яті free ((void *) <Покажчик>);</code> | <code>#include <cstdlib> int *p; p=(int*)malloc(sizeof(int)); free((void *) p);</code> |

Посилання

Посилання – це змінні, які можуть бути псевдонімами для інших змінних. При цьому змінні та посилання на них (їх псевдоніми) звертаються до однієї і тієї ж комірки пам'яті. Вони, як правило, використовуються при

описі формальних параметрів функцій, щоб уникнути виділення додаткової пам'яті для збереження копії фактичних параметрів (значення цих змінних у функції можуть змінюватися). Змінній-посиланню можна присвоїти деяке значення тільки при її описі. Тип змінної, для якої посилання є псевдонімом, повинен співпадати з типом змінної-посилання. При описі посилань використовується символ «&».

| Загальний вигляд | Приклад |
|--------------------------------|--|
| <Тип> &<Посилання> = <Змінна>; | int d; int &n=d; //n є посиланням на змінну d n=2; // Еквівалентно d=2 |

Динамічні масиви

Динамічні масиви – це масиви, пам'ять для яких виділяється у процесі роботи програми. Динамічне виділення пам'яті здійснюється з використанням покажчиків.

| Загальний вигляд (new-delete) | Приклад |
|--|--|
| <Тип елемента масиву> * <Покажчик>; <Покажчик>=new <Тип елем. масиву>[<К-сть елементів>] delete [] <Покажчик>; | int *p; int Size; //Кільк. елем. cout<<"Size="; cin>>Size; p=new int [Size]; delete [] p; |

| Загальний вигляд (malloc-free) | Приклад |
|--|--|
| <Тип елементів масиву> * <Покажчик>; //Виділення пам'яті для масиву <Покажчик>=(Тип елем.*) malloc (<Кільк. елем.>*sizeof(<Тип елем.>)); //Використання масиву free((void *) <Покажчик>; //Звільнення пам'яті | #include <cstdlib> int *p; int Size; //Кільк. елем. cout<<"Size="; cin>>Size; p=(int*)malloc (count * sizeof(int)); free((void *) p); |

Доступ до елементів динамічного масиву можна здійснювати по різному.

| Загальний вигляд | Приклад |
|-------------------------------|------------|
| <Покажчик> [<Номер елемента>] | p[i] =8; |

| | |
|----------------------------------|------------|
| <Номер елемента> [<Показчик>] | i [p] = 8; |
| *(<Показчик> + <Номер елемента>) | *(p+i) =8; |
| *(<Номер елемента> + <Показчик>) | *(i+p) =8; |

Наведемо приклад використання динамічних масивів.

Приклад. Задано послідовність цілих чисел a_1, a_2, \dots, a_n . Знайти найбільший елемент цієї послідовності.

| new-delete | malloc-free |
|---|---|
| <pre> #include "stdafx.h" #include <iostream> using namespace std; int main(int argc, char* argv[]) { int n; //Введення кількості елементів cout<<"n= "; cin>> n; //Виділення пам'яті int *p=new int[n]; //Введення елементів for(int i=0;i<n;i++) { cout<<"a["<<i<<"]="; cin>>p[i]; } //Знаходження максимального int max=p[0]; for(i=0;i<n;i++) { if(max<p[i]) max=p[i]; } cout<<"max="<<max<<endl; //Звільнення пам'яті delete[] p; system("pause"); return 0; } </pre> | <pre> #include "stdafx.h" #include <iostream> #include <cstdlib> using namespace std; int main(int argc, char* argv[]) { int n; //Введення кількості елементів cout<<"n= "; cin>> n; / //Виділення пам'яті int *p=(int*) malloc(n*sizeof(int)); //Введення елементів for(int i=0;i< n;i++) { cout<<"a["<<i<<"]="; cin>>p[i]; } //Знаходження максимального int max=p[0]; for(i=0;i<n;i++) { if(max<p[i]) max= p[i]; } cout<<"max="<<max<<endl; //Звільнення пам'яті free((void *) p); system("pause"); return 0; } </pre> |

Структури

Будь-яка діяльність людини так чи інакше пов'язана з дослідженням та маніпуляцією різного роду об'єктами. Широко вживаним методом дослідження є метод *моделювання*, який ґрунтується на розробці та дослідженні моделей об'єктів реальної дійсності. *Модель* – це матеріальний або уявний об'єкт, що відображає важливі для дослідження властивості об'єкта-оригінала і використовується для дослідження об'єкта-оригінала. В обчислювальних системах широко використовується *інформаційна модель* – сукупність даних, що відображають основні властивості об'єкта-оригінала, його структуру та зв'язок з оточуючим середовищем. Для збереження даних в обчислювальних системах використовують ту чи іншу *структуру даних* – спосіб організації даних, що дозволяє відобразити зв'язки та відношення між даними.

У C++ сукупність даних, що описують властивості деякого об'єкта реальної дійсності, може бути представлена, наприклад, за допомогою структури даних, яку називають структурою (*struct*). *Структура* – це складна структура даних, яка може містити дані (*поля*) різного типу. *Поля* – це змінні, які використовують для збереження даних про об'єкт.

Розглянемо найпростіший вигляд описання структур, коли кожна із властивостей реального об'єкта, що описується, представляється окремим полем.

```
struct [ <Ім'я типу> ] {
    <тип поля 1> <ім'я поля 1> ;
    <тип поля 2> <ім'я поля 2> ;
    .....
} [<Список змінних типу структури>] ;
```

Наведемо декілька прикладів опису структур.

| Об'єкт, що моделюється | Відповідна програмна структура |
|---|---|
| <u>Прямокутник</u> Сторона <i>a</i> } Сторона <i>b</i> } Дані (поля) | <pre>struct TRectangle{ double a; double b; };</pre> |
| <u>Студент</u> Прізвище та ініціали } Рік народження } Дані (поля) Розмір стипендії | <pre>struct TStudent{ char Name[40]; int Year; double Grant; };</pre> |

Зауважимо, що у наведених прикладах вказано тільки імена типу структур, а змінні не описано, їх ми можемо описати пізніше. У C++ при описі структури

можна не вказувати ім'я типу структури, а тільки змінні типу структури. У будь-якому випадку, опис структур повинен завершуватись знаком « ; ».

| Вказано ім'я типу і список змінних | Вказано тільки ім'я типу | Вказано тільки список змінних |
|--|--|--|
| struct TRectangle{ double a; double b; } t1,t2; | struct TRectangle{ double a; double b; }; | struct { double a; double b; } t1,t2; |

Якщо описано тип даних структура і вказано ім'я типу, то у подальшому, як і для будь-якого іншого типу, можна описати змінні та покажчики цього типу.

| Загальний вигляд опису | Приклад |
|------------------------------|--------------------------|
| <Тип структура> <Змінна>; | TRectangle Rectangle1; |
| <Тип структура> *<Покажчик>; | TRectangle *pRectangle1; |

Перевагою такої структури даних є те, що всі розглядувані властивості деякого окремого об'єкта містяться в одній структурі даних. У цьому випадку об'єкт реальної дійсності представляється за допомогою однієї змінної.

Приклад. Два прямокутники можуть бути представлені за допомогою двох змінних типу структури



Після опису змінної типу структури компілятор автоматично виділяє необхідний об'єм оперативної пам'яті. Як і для інших типів, кількість байт, що необхідні для збереження структури, можна визначити за допомогою оператора `sizeof`. При описі покажчика на структуру, як і будь-якого іншого покажчика, пам'ять для структури автоматично не виділяється, і її необхідно виділити, використовуючи оператор `new` або `malloc`, та, відповідно, після використання звільнити пам'ять за допомогою операторів `delete` або `free`.

Доступ до полів змінної типу структури. Доступ до полів змінної типу структури здійснюється з використанням оператора крапка « . ».

| Загальний вигляд | Приклад |
|------------------|---------|
|------------------|---------|

| | |
|--|--|
| <code><Змінна типу структури>.<Ім'я поля>=<Значення>;</code> <code><Змінна> = <Змінна типу структури>.<Ім'я поля>;</code> | <code>TRectangle Rectangle1;</code> <code>Rectangle1.a = 10;</code> <code>Rectangle1.b = 6;</code> <code>double d= Rectangle1.a;</code> |
|--|--|

Доступ до полів структури через покажчик. Доступ до полів структури, адресу якої містить покажчик, можна здійснити, виконавши операцію розіменування (знак «*» вказується перед покажчиком) та використанням оператора крапка, або ж, використовуючи оператор «->».

| Загальний вигляд | Приклад |
|--|--|
| <code>(*<Покажчик>).<Ім'я поля>=<Значення>;</code> <code><Змінна> = (*<Покажчик>).<Ім'я поля>;</code> <code><Покажчик> -> <Ім'я поля>=<Значення>;</code> <code><Змінна> = <Покажчик> -> <Ім'я поля>;</code> | <code>TRectangle *p;</code> <code>p= new TRectangle;</code> <code>double d;</code> <code>//Звертання до полів</code> <code>(*p).a = 10;</code> <code>d= (*p).a;</code> <code>// Еквівалентна форма</code> <code>p ->a = 10;</code> <code>d= p ->a;</code> |

Бітові поля. У C++ можна описувати структури, які окрім полів інших типів можуть містити так звані бітові поля (поля, які використовують вказану кількість біт оперативної пам'яті). Зауважимо, що мінімальний розмір структури, що містить бітові поля дорівнює розміру типу int. Наведемо загальний вигляд опису структур, що містять бітові поля

| |
|---|
| <pre> struct [<Ім'я типу>] { <тип поля 1> [<ім'я поля 1>] : <Кількість біт> ; <тип поля 2> [<ім'я поля 2>] : <Кількість біт> ; } [<Список змінних типу структури>] ; </pre> |
|---|

Якщо поле займає один біт, то при описі цього поля використовується ключове слово `unsigned`.

Приклад.

| |
|---------------------------------|
| <pre> struct TMyStruct { </pre> |
|---------------------------------|

```

        unsigned : 2;
        unsigned a: 1;
        unsigned b: 1;

    } s1,s2;

```

Доступ до бітових полів здійснюється як і для інших полів структури.

Приклад.

```

s1.a=1;
s1.b=0;

```

Вкладені записи. При описі структур можна описувати поля типу структури. Такі поля називають вкладеними структурами. Тип структури, як типу поля, можна описувати як попередньо, так і безпосередньо при описі поля. Наведемо можливі варіанти опису типу структури, що містять вкладені структури.

| Без вкладених структур | З вкладеними структурами (з попереднім описом типу) | З вкладеними структурами (опис вкладеної структури безпосередньо при описі поля) |
|---|--|--|
| <pre> struct TStudent{ char Name[40]; int day; int month; int year; double Grant; }; </pre> | <pre> struct TDate{ int day; int month; int year; }; struct TStudent{ char Name[40]; TDate Date; double Grant; } s; </pre> | <pre> struct TStudent{ char Name[40]; struct { int day; int month; int year; } Date; double Grant; } s; </pre> |

Доступ до полів вкладених структур здійснюється через ім'я поля.

| Загальний вигляд | Приклад |
|---|--|
| <структура>.<поле типу структури>.<поле вклад. структури>=<значення>; | <pre> s.Date.day = 5; s.Date.month=1; </pre> |

Об'єднання

Об'єднання – це структура даних, яка дає можливість використовувати одну й ту ж ділянку пам'яті для одного з полів різних типів. Розмір ділянки пам'яті, що виділяється для об'єднання, дорівнює розміру найбільшого поля.

Наведемо загальний вигляд об'єднання

```
union [ <Ім'я типу> ] {
    <тип поля 1> <ім'я поля 1> ;
    <тип поля 2> <ім'я поля 2> ;
    .....
} [<Список змінних типу об'єднання>] ;
```

Приклад.

```
union TMyU {
    bool b;
    double d;
    int n;
} u1;
```

Розмір комірки пам'яті для змінної u1 буде дорівнюватиме розміру типу double. Ця комірка спільно буде використовуватись для всіх полів, тому звертання до декількох полів у одній змінній може призвести до втрати значень цих полів.

Приклад.

```
u1.n=12;
u1.d=0;
cout<<u1.n; // 0
```

Тип об'єднання може використовуватись для опису полів структур та інших об'єднань. Часто об'єднання використовують при описі групи споріднених об'єктів, які мають багато спільних властивостей (описуємо як звичайні поля структури), але є і властивості, якими вони відрізняються (описуємо у вигляді об'єднання). Очевидно, для того щоб знати яке з альтернативних полів необхідно використати серед спільних полів описують поле, яке дозволяє визначати до якої з споріднених груп належатиме об'єкт.

Приклад. Опишемо структуру, яка міститиме дані про студентів: прізвище та ініціали, рік народження, стать, для хлопців – наявність військового квитка, а для дівчат – розмір стипендії. Поле стать використовується для того, щоб відрізнити хлопців від дівчат.

| | |
|-------------------------------------|----------------|
| Студент | |
| Прізвище та ініціали | char Name[40]; |
| Рік народження | int Year; |
| Стать | char Statj; |
| Наявн. військ. квитка (для хлопців) | bool Army; |
| Розмір стипендії (для дівчат) | double Grant; |
| | }; |
| | }; |

Анонімні (без опису імені типу) об'єднання також можна використати у програмі для опису множини альтернативних змінних з метою економії оперативної пам'яті. При описі альтернативних глобальних змінних використовується службове слово **static**.

Приклад.

```
static union {
    int p1;
    double p2;
};

int main(){
    union {
        int pp1;
        bool pp2;
    };
    p1=3; p2=0;
    cout<<"p1="<<p1<<endl;    // 0
    pp2=true; pp1=0;
    cout<<"pp2="<<pp2<<endl;    // 0
}
```

Приклад. З метою економії пам'яті можна також описати масиви цілих і дійсних чисел. Спочатку використати масив цілих чисел, а потім масив дійсних чисел. При цьому, зрозуміло, одночасне їх використання буде некоректним.

```
int main(){  
    union {  
        int a[10];  
        double b[17] ; } Альтернативні локальні змінні  
    };  
    .....  
}
```


Класи

Більшість реальних об'єктів характеризується не тільки певними властивостями, а й набором функціональних можливостей. Іншими словами, об'єкти можуть характеризуватися властивостями та діями, що можуть виконувати самі об'єкти або які можна виконувати над ними.

Приклад. Для прямокутника можна визначити наступні дії:

- 1) знаходження периметру;
- 2) знаходження площі;
- 3) визначення рівності з іншим прямокутником, який задається за допомогою своїх сторін;
- 4) масштабування (збільшення або зменшення всіх сторін прямокутника у певну кількість разів).

Отже, необхідно описати наступну логічну структуру

Прямокутник



Для повноцінного опису об'єктів реальної дійсності використовується тип даних, який має назву *клас* і який дозволяє описувати не тільки властивості об'єкту (за допомогою полів), а і його поведінку або ж функціональні можливості (за допомогою функцій, які називають *методами*). Поля і методи, описані у класі, прийнято називати *методами*. Слід зазначити, що при визначенні класу описуються властивості й методи, які характерні для всіх об'єктів відповідного класу (групи) реальних об'єктів. Як довільний тип даних, клас визначає певну множину елементів або екземплярів цього типу. Окремо взятий екземпляр деякого класу називають *об'єктом*. Прийнято вважати, що *клас* – це шаблон, на основі

якого може бути створено конкретний програмний *об'єкт*, що моделює реальний об'єкт певної предметної області. З точки зору мови програмування, *клас* – це тип даних, а *об'єкт* – це змінна типу клас. *Клас* і *об'єкт* є фундаментальними поняттями технології об'єктно-орієнтованого програмування. Суть цієї технології ґрунтується на моделюванні досліджуваних об'єктів предметної області з використанням класів.

Наведемо загальну схему опису класу

```
class [ <Ім'я класу> ] {
    <Опис закритих членів класу>
public:
    <Опис відкритих членів класу>
private:
    <Опис закритих членів класу>
protected:
    <Опис захищених членів класу>
} [<Список змінних типу клас (об'єктів)>] ;
```

Як і для структур, можна не вказувати імені а тільки змінні типу клас. У будь-якому випадку опис повинен завершуватись символом «;».

При описі членів класу можна вказувати специфікатори доступу, які визначають де вказані члени є доступними (де до них можна звернутись):

- **public** – відкриті члени класу. Доступ до них може бути здійснений поза межами класу;
- **private** – закриті члени класу. Доступні тільки у методах цього ж класу та дружніх функціях і класах;
- **protected** – захищені члени класу. Доступні у методах цього ж класу та методах його нащадків.

За замовчуванням, якщо не вказано специфікатор доступу, вважається до член класу є закритим.

Для створення та знищення об'єктів у описі класу можуть бути спеціальні методи, які називаються, відповідно, конструктором та

деструктором (імена цих методів співпадають з іменем класу, у якому вони описані).

Загальний вигляд опису конструктора

```
<Ім'я класу> ( [ <Опис форм. параметрів> ] );
```

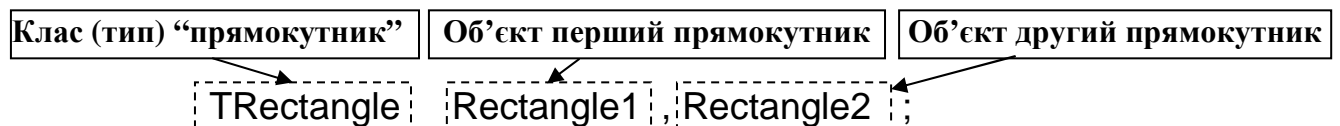
Загальний вигляд опису деструктора (відрізняється від опису конструктора використанням перед іменем деструктора символу «~»)

```
~ <Ім'я класу> ( );
```

Приклад. Опишемо розглядувану логічну структуру “Прямокутник” за допомогою класу у C++.

| Об'єкт предметної області | Структура даних class |
|--|---|
| <u>Прямокутник</u> Сторона <i>a</i> Сторона <i>b</i> Дія створення трикутника (конструктор) Дія знаходження периметру Дія знаходження площі Дія порівняння (з іншим прямокутником) Дія масштабування Дія знищення трикутника (деструктор) | class TRectangle{ public: double a; double b; TRectangle (double a, double b); double Perimeter (); double Square(); bool IsEqual(double a, double b); void Scale(double k); ~TRectangle (); }; |

Зараз розглядувані раніше два прямокутники можуть бути представлені за допомогою двох змінних типу клас TRectangle:



Змінні Rectangle1 і Rectangle2 називають об'єктами класу TRectangle.

Опис полів у класі здійснюється як і опис звичайних змінних, але не вказується специфікатор зберігання.

Реалізацію методів можна вказувати як всередині класу, так і після опису класу. Якщо реалізацію методу вказано всередині класу, то його вміст вставляється у місця його виклику. Такий метод називають *вбудованим*.

Приклад. Наведемо реалізацію методів **Square** та **Perimeter** як вбудованих методів

```
class TRectangle{
public:
    double a;
    double b;
    .....
    double Perimeter { return a*b; }
    double Square() { return 2*(a+b); }
    .....
};
```

Якщо у класі вказано тільки прототип методу, то при описі реалізації методу перед його назвою додатково вказують ім'я класу, в якому він описаний.

```
<Тип> <Ім'я класу>::<Ім'я метода>(<Форм. параметри>){
    .....
}
```

Приклад. Наведемо реалізацію методів **Square** та **Perimeter**, для яких при описі класу вказано прототипи.

```
class TRectangle{
public:
    double a;
    double b;
    .....
    double Perimeter ();                //Прототипи методів
    double Square();
    .....
};                                     //Реалізація методів

TRectangle:: Square ()
{
    return a*b;
}
double TRectangle::Perimeter ()
{
    return 2*(a+b);
}
```

Всередині методів класу можна звертатись до інших членів цього ж класу за їх іменами. Якщо ж імена формальних параметрів методу співпадають з іменами полів, описаних у класі, то доступ до цих полів можна здійснити з використанням покажчика на об'єкт класу **this**.

Приклад. Наведемо реалізацію конструктора

| | |
|---|--|
| <pre> TRectangle:: TRectangle (double a, double b) { this->a = a; this->b = b; } </pre> | <p>//Реалізація конструктора</p> <p>Формальні параметри</p> <p>Поля, описані у класі</p> |
|---|--|

Доступ (звертання) до полів та методів об'єкта. Доступ до полів і методів об'єкта, як і до полів структур (struct), здійснюється за допомогою оператора крапка.

| Загальний вигляд | Приклад |
|---|--|
| <p><Ім'я об'єкта> . <Ім'я поля></p> <p><Ім'я об'єкта> . <Ім'я методу></p> | <pre> TRectangle r1; r1.a=3; r1.b=5; cout<<"Square = "<<r1.Square()<<endl; cout<<"Perimeter= "<<r1.Perimeter()<<endl; </pre> |

Якщо ж доступ здійснюється через покажчик, що містить адресу об'єкта, то використовується оператор «->»

| Загальний вигляд | Приклад |
|--|--|
| <p><Показчик > -> <Ім'я поля></p> <p>< Показчик > -> <Ім'я методу></p> | <pre> TRectangle r1; TRectangle *p=&r1; p->a=3; p->b=5; cout<<"Square = "<< p->Square()<<endl; cout<<"Perimeter="<< p->Perimeter()<<endl; </pre> |

або ж розіменування покажчика

| Загальний вигляд | Приклад |
|--|---|
| <p>(*<Показчик >) . <Ім'я поля></p> <p>(*<Показчик >). <Ім'я методу></p> | <pre> TRectangle r1; TRectangle *p=&r1; (*p).a=3; (*p).b=5; cout<<"Square = "<< (*p).Square()<<endl; </pre> |

| | |
|--|---|
| | cout<<"Perimeter="<<(*p).Perimeter()<<endl; |
|--|---|

Конструктори.

Як було зазначено раніше, для створення об'єктів використовуються спеціальні методи, які називають *конструкторами*. Ім'я конструктора співпадає з іменем класу. Тип для методу-конструктора не вказують. Конструктори можуть мати параметри, які дозволяють здійснити *ініціалізацію об'єкта*, тобто надати його полям початкових значень. У залежності від кількості та типу параметрів, чи їх відсутності, серед конструкторів можна виділити декілька типів.

Конструктор без параметрів. Як і для змінних типу структури, якщо описано об'єкти (змінні) деякого класу, компілятор автоматично виділить для них необхідний об'єм оперативної пам'яті. При цьому автоматично (неявно) викликається конструктор без параметрів. Якщо такого конструктора не описано, то автоматично створюється стандартний конструктор за замовчуванням. Такий конструктор тільки виділить пам'ять для полів об'єкта, але поля не буде ініціалізовано. Тому бажано розробляти власний конструктор без параметрів.

Приклад. Для класу **TRectangle** наведемо можливі варіанти опису та реалізації конструктора без параметрів, який ініціалізує поля нульовими значеннями.

| Одразу вказуємо реалізацію | Реалізацію вказуємо після опису класу |
|---|--|
| <pre>class TRectangle{ public: double a; double b; TRectangle () { a=0; b=0; } };</pre> | <pre>class TRectangle{ public: double a; double b; TRectangle () ; }; TRectangle::TRectangle () { a=0; b=0; }</pre> |

Виклик конструктора без параметрів може бути неявним (описавши змінну типу клас) або явним. Розглянемо можливі варіанти виклику конструктора без параметрів

| Варіанти виклику | Приклад |
|--|--|
| <Ім'я класу> <Ім'я об'єкта>; <Ім'я класу> <Ім'я об'єкта>= <Ім'я класу>(); | TRectangle r1; //Неявний виклик TRectangle r2=TRectangle(); //Явний |

Як і для структур, при описі покажчика на об'єкт деякого класу пам'ять для об'єкта автоматично не виділяється, тому її необхідно виділити, а після використання – звільнити. При створенні об'єкта може викликатись конструктор без параметрів.

| Загальний вигляд | Приклад |
|---|---|
| <pre> //Виділення пам'яті <Покажчик> =new <Ім'я класу> ; <Покажчик> =new <Ім'я класу> () ; //Звільнення пам'яті delete <Покажчик>; </pre> | <pre> TRectangle *r4=new TRectangle; TRectangle *r5=new TRectangle(); delete r4; delete r5; </pre> |

Конструктори з параметрами. Як було зазначено раніше, опис конструктора може містити параметри, які, як правило, використовують для ініціалізації полів об'єктів.

Приклад. Наведемо можливу реалізацію конструктора з параметрами для класу TRectangle.

| Одразу вказуємо реалізацію | Реалізацію вказуємо після опису класу |
|---|--|
| <pre> class TRectangle{ public: double a; double b; TRectangle (double a, double b) { this->a=a; this->b=b; } }; </pre> | <pre> class TRectangle{ public: double a; double b; TRectangle (double a, double b); }; TRectangle::TRectangle(double a, double b) { this->a=a; this->b=b; } </pre> |

Розглянемо можливі варіанти виклику конструктора з параметрами.

| Варіанти виклику | Приклад |
|---|---|
| <Ім'я класу> <Ім'я об'єкта>(<Список параметрів>; <Ім'я класу> <Ім'я об'єкта>= <Ім'я класу>(<Список параметрів>); | TRectangle r1(2,3); TRectangle r2=TRectangle(2,3); |

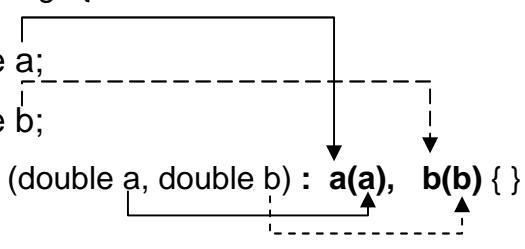
Конструктори з параметрами можуть містити так звані *списки ініціалізації*, які призначені для ініціалізації полів створюваного об'єкта. Список ініціалізації може міститися у описі конструктора після опису

формальних параметрів (ставиться знак «:») і перед тілом конструктора. В середині списку поступово через кому зазначають назву поля і в дужках його початкове значення.

`<Ім'я констр.>(<Форм.пар.>) :<Поле_1>(Значення 1),...,<Поле N>(Знач. N) { ... }`

Як правило, в якості початкових значень полів використовують значення формальних параметрів цього конструктора.

Приклад. Наведемо можливу реалізацію конструктора з списком ініціалізації.

| Без списку ініціалізації (ініціалізацію виконуємо у тілі конструктора) | З списком ініціалізації (ініціалізацію виконуємо за допомогою списку ініціалізації) |
|--|---|
| <pre>class TRectangle{ public: double a; double b; TRectangle (double a, double b) { this->a=a; this->b=b; } };</pre> | <pre>class TRectangle{ public: double a; double b; TRectangle (double a, double b) : a(a), b(b) { } };</pre>  |

Конструктор з одним параметром. Серед конструкторів з параметром виділяють конструктор з одним параметром. Особливістю такого конструктора є те, що він, крім розглянутих варіантів виклику конструктора з параметрами, може бути використаний для створення об'єкта на основі значення відповідного типу

`<Ім'я класу> <Ім'я об'єкта>= <Значення >;`

Приклад. Опишемо конструктор класу **TRectangle** з одним параметром, який буде використовуватись для ініціалізації обох полів.

| Без списку ініціалізації | З списком ініціалізації |
|---|---|
| <pre>class TRectangle{ public: double a; double b; TRectangle (double d) { a=d; b=d; } };</pre> | <pre>class TRectangle{ public: double a; double b; TRectangle (double d) : a(d), b(d) { } };</pre> |

Розглянемо можливі варіанти виклику конструктора з одним параметром.

| Варіанти виклику | Приклад |
|--|--|
| <code><Ім'я класу> <Ім'я об'єкта>(<Список параметрів>);</code> <code><Ім'я класу> <Ім'я об'єкта>= <Ім'я класу>(<Список параметрів>);</code> <code><Ім'я класу> <Ім'я об'єкта>= <Значення >;</code> | <code>TRectangle r1(5);</code> <code>TRectangle r2=TRectangle(5);</code> <code>TRectangle r2 = 5 ;</code> |

Створюючи об'єкт на основі значення і виклику конструктора з параметрами, відбувається неявне перетворення типу цього значення в об'єкт відповідного класу. Щоб заборонити такий варіант створення об'єктів перед описом класу необхідно вказати ключове слово **explicit**.

```
class TRectangle{
public:
    double a;
    double b;
    explicit TRectangle (double d) { a=d; b=d; }
    .....
};
```

Зараз можливим є виклик конструкторів

```
TRectangle r1(5);
TRectangle r2=TRectangle(5);
```

але заборонено виклик

```
TRectangle r2 = 5 ; // Помилка
```

Конструктор копіювання. Якщо при створенні об'єкта деякого класу його ініціалізують іншим об'єктом, передачі об'єкта як параметра значення у функцію або у випадку, коли об'єкт є результатом роботи функції неявно викликається конструктор копіювання. Якщо конструктор копіювання не описано, то використовується конструктор копіювання за замовчуванням. У цьому випадку створюється побітова копія об'єкта (копіюються значення усіх полів). У деяких випадках, наприклад, коли у класі є поля, для яких пам'ять виділяється динамічно, необхідно виконати не побітову копіювання покажчика, а копіювання динамічно виділеної копії пам'яті. Тоді опис конструктора копіювання є обов'язковим. Загальний вигляд опису конструктора копіювання

```
<Ім'я класу> (const <Ім'я класу> & <Ім'я об'єкта>);
```

Приклад. Опишемо можливий конструктор копіювання для класу **TRectangle**.

```
class TRectangle{
public:
    double a;
    double b;
    TRectangle (const TRectangle & r ) { a= r.a ; b=r.b; }
    .....
};
```

Розглянемо випадки, коли викликається конструктор копіювання:

1) створюється об'єкт і одразу ініціалізується іншим об'єктом;

```
class TRectangle{
public:
    double a;
    double b;
    TRectangle (const TRectangle & r ) { a= r.a ; b=r.b; }
    .....
};
int main( )
{
    TRectangle r1;    //Виклик конструктора без параметрів
    r1.a=2; r1.b=7;
    TRectangle r2=r1; //Виклик конструктора копіювання
    .....
}
```

2) викликається функція, у якій одним із формальних параметрів є формальний параметр-значення типу клас. Тоді під час виклику функції такому формальному параметру ставлять у відповідність фактичний параметр – об'єкт відповідного класу;

```
class TRectangle{
public:
    double a;
    double b;
    TRectangle (const TRectangle & r ) { a= r.a ; b=r.b; }
    .....
};

//rect- Формальний параметр-значення типу TRectangle
void Show(TRectangle rect)
{
    cout<<"a="<<rect.a<<" b="<<rect.b<<endl; }
int main( )
{
    TRectangle r1;    //Виклик конструктора без параметрів
    r1.a=2; r1.b=7;

    //У функцію передається фактичний параметр об'єкт r1
    Show(r1); //Виклик конструктора копіювання
    .....
}
```

3) об'єкт створюється всередині функції і повертається як результат функції;

```

class TRectangle{
public:
    double a;
    double b;
    TRectangle (const TRectangle & r ) { a= r.a ; b=r.b; }    //Конструктор копіювання
    void Show(){cout<<" a= "<<a<<" b= "<<b<<endl;}
};

//Результатом функції є об'єкт
TRectangle Create()
{
    TRectangle NewRectangle;    //Створення об'єкта NewRectangle
    cout<<"a=" ;
    cin>>NewRectangle.a;
    cout<<"b=" ;
    cin>>NewRectangle.b;
    return NewRectangle;    //Об'єкт повертається як результат функції
}

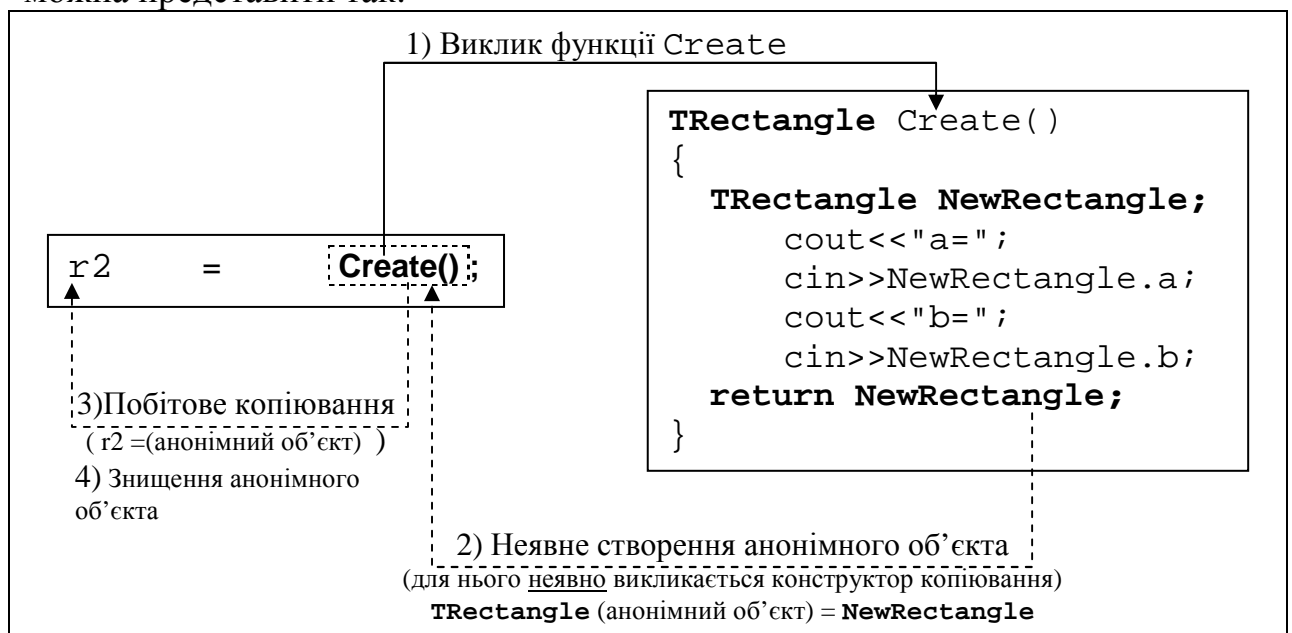
int main()
{
    TRectangle r2;    //Виклик конструктора без параметрів
    r2=Create();    //Виклик конструктора копіювання
    .....
}

```

у даному прикладі конструктор копіювання викликається при поверненні об'єкта із функції **Create**. При цьому створюється анонімний (без імені) об'єкт і саме для нього викликається конструктор копіювання (анонімний об'єкт ініціалізують об'єктом, який вказано у функції **Create** після **return**). В той же час при виконанні операції присвоєння для об'єкта **r2** відбувається побітове копіювання анонімного об'єкта. Схематично виконання оператора

r2=Create();

можна представити так:



Оскільки виклик функції, що повертає об'єкт, призводить до створення анонімного об'єкта, то є можливість звернутися до його членів.

Приклад.

```
( Create() ).Show();
```

Зауважимо, що виконуючи оператор

```
TRectangle r3=Create();
```

анонімний об'єкт не створюється, а одразу викликається конструктор копіювання для r3.

| Конструктор | Особливості виклику | Загальний вигляд | Приклад |
|--------------------------------|--|---|--|
| конструктор без параметрів | може викликається неявно; якщо не описано, то створюється автоматично | <u>Опис</u> <Ім'я класу> (); | TRectangle () { a=b=0; }; |
| | | <u>Виклик</u> <Ім'я класу> <Ім'я об'єкта>; <Ім'я класу> <Ім'я об'єкта>= <Ім'я класу>(); | TRectangle r1; TRectangle r2=TRectangle(); |
| | | <u>З використанням покажчика</u> <Покажчик> =new <Ім'я класу> ; <Покажчик> =new <Ім'я класу> () ; | TRectangle *r4=new TRectangle; TRectangle *r5=new TRectangle(); |
| конструктор копіювання | викликається при створенні з ініціалізацією, передачі у функцію як параметра-значення та поверненні з функції | <u>Опис</u> <Ім'я класу> (const <Ім'я класу> & <Ім'я об'єкта>); | TRectangle(const TRectangle &c){ a=c.a; b=c.b; } |
| | | <u>Виклик</u> <Ім'я класу> <Ім'я об'єкта>=<Інший об'єкт>; <Ім'я класу> <Ім'я об'єкта> (<Інший об'єкт>); <Ім'я класу> <Ім'я об'єкта>=<Ім'я класу>(Ін. об.); | TRectangle r2=r1; TRectangle r2 (r1); TRectangle r2= TRectangle (r1); |
| | | <u>З використанням покажчика</u> <Покажчик> =new <Ім'я класу> (<Інший об'єкт.>); | TRectangle *r5=new TRectangle(r1); |
| | | <u>При передачі у функцію об'єкта, якому відповідає об'єкт як формальний параметр-значення</u> | <u>Опис функції</u> TRectangle MyFunc(TRectangle r){...} |
| | | <u>Коли об'єкт є результатом роботи функції</u> | <u>Виклик функції</u> MyFunc(r1); <u>Опис функції</u> TRectangle Create() {...} ; <u>Виклик функції</u> r1= Create(); |
| конструктор з одним параметром | може використовуватись при створенні об'єктів з використанням ініціалізаторів (якщо при описі не вжито explicit) | <u>Опис</u> <Ім'я класу> (<Тип парм.> <Ім'я парам.>); | TRectangle (double d); |
| | | <u>Виклик</u> <Ім'я класу> <Ім'я об'єкта> (<Факт. парам.>); <Ім'я класу> <Ім'я об'єкта>=<Ім'я класу>(<Ф. пар.>); <Ім'я класу> <Ім'я об'єкта>=<Значення>; | TRectangle p1(7); TRectangle p2=TRectangle(7); TRectangle p3=4; //Викор. ініціалізатора |
| | | <u>З використанням покажчика</u> <Покажчик> =new <Ім'я класу> (<Факт. парам.>); | TRectangle *p4=new TRectangle(7); |
| інші конструктори | параметри вказуються при створенні об'єктів | <u>Опис</u> <Ім'я класу> (<Опис форм. параметрів>); | TRectangle (double a, double b): a(a), b(b){}; |
| | | <u>Виклик</u> <Ім'я класу> <Ім'я об'єкта> (<Факт. парам.>); <Ім'я класу> <Ім'я об'єкта>=<Ім'я класу>(<Ф. пар.>); | TRectangle d1(3,7); TRectangle d2=TRectangle(3,7); |
| | | <u>З використанням покажчика</u> <Покажчик> =new <Ім'я класу> (<Факт. парам.>); | TRectangle*d4=new Rectangle(3,7); |

У класі може бути описано декілька конструкторів, які, як переважані методи, повинні відрізнятися кількістю або типом параметрів.

Приклад. Наведемо приклад опису конструкторів для класу TRectangle.

```
class TRectangle {
public:
    double a,b;
    TRectangle ();                //Конструктор за замовчуванням
    TRectangle (const TRectangle &c); //Конструктор копіювання
    TRectangle (double d);        //Конструктор з одним парам.
    TRectangle (double a, double b); //Конструктор з параметрами
    .....
};

TRectangle :: TRectangle (){
    a=b=0;
}
TRectangle :: TRectangle (const TRectangle &c){
    a = c.a;
    b = c.b;
}
TRectangle :: TRectangle (double d){
    a = b = d;
}
TRectangle :: TRectangle (double a, double b){
    this->a = a;
    this->b = b;
}
}
int main( )
{
    //Виклик конструктора за замовчуванням
    TRectangle r1;
    TRectangle r2=TRectangle();
    TRectangle *r4=new TRectangle;
    TRectangle *r5=new TRectangle();
    //Виклик конструктора копіювання
    TRectangle pk1=r1;
    //Виклик конструктора з одним параметром
    TRectangle p1(7);
    TRectangle p2=TRectangle(7);
    TRectangle p3=4;
    TRectangle *p4=new TRectangle(7);
    //Виклик конструктора з двома параметрами
    TRectangle d1(3,7);
    TRectangle d2=TRectangle(3,7);
    .....
}
```

} Реалізація конструкторів

Деструктор.

Як було зазначено раніше, при створенні об'єктів деякого класу явно чи неявно (автоматично) викликається конструктор. У конструкторі, як правило, здійснюють ініціалізацію полів об'єкта, динамічно виділяють необхідний об'єм пам'яті, підключають деякі додаткові ресурси та інші дії. Аналогічно, при знищенні об'єктів автоматично викликається метод, який називають деструктором. Деструктор викликається безпосередньо перед знищенням об'єкта, тому усередині деструктора виконують звільнення усіх виділених для цього об'єкта ресурсів: звільнення динамічно виділеної пам'яті, закритті файлів та ін. Якщо додаткових ресурсів при створенні об'єкта не виділяється, то власний деструктор описувати не обов'язково. Зазначимо, що деструктори викликають у порядку, зворотному до порядку виклику конструкторів.

Загальний вигляд деструктора

```
~<Ім'я класу> ( );
```

Приклад. Розглянемо клас, що моделює динамічний масив цілих чисел. У конструкторі цього динамічно виділяємо пам'ять для масиву, а у деструкторі відбувається її звільнення.

```
class TArray{
public:
    int *p;
    int Count;
    TArray(){p=0; Count=0;}
    TArray(int Count){                                //Конструктор
        this->Count=Count;
        p=new int[Count];                            //Динамічне виділення пам'яті
    }
    ~TArray(){                                         //Деструктор
        delete[] p;                                  //Звільнення пам'яті
    }
};
. . . . .
```

Статичні поля та методи.

звертатись тільки до статичних членів класу. Всередині методів класу, де описано статичний метод, до нього можна звертатися як до звичайного (нестатичного) метода. За межами класу до статичних методів можна звертатися як і до нестатичних, з використанням об'єкта,

`<Ім'я об'єкта> :: <Ім'я статичного метода>`

так і з використанням класу

`<Ім'я класу> :: <Ім'я статичного метода>`

Приклад. Зробимо метод знаходження площі прямокутника статичним.

| Загальний вигляд | Приклад |
|---|--|
| <pre>//Опис статичного методу</pre> | <pre>class TRectangle { public: static double S(double a,double b){return a*b;} }; int main(int argc, char* argv[]) { TRectangle r1; cout<<"S(2,9) ="<<TRectangle::S(2,9)<<endl; cout<<"s1="<< r1.S(2,9)<<endl; }</pre> |
| <pre>//Виклик як метода класу //Виклик як метод об'єкта</pre> | |

Інкапсуляція. Інкапсуляція – основний принцип об'єктно-орієнтованого програмування, згідно з яким поля є внутрішніми даними об'єкта, а тому безпосереднє звертання до полів поза межами класу є недопустимим. Таке безпосереднє звертання несе у собі небезпеку їх неправильного використання. Наприклад, у класі TRectangle поля **a** і **b** використовують для збереження довжини сторін прямокутника. Але якщо ці поля описати з специфікатором доступу **public**, то у програмі можна створити об'єкт цього класу і надати полям неприпустимі від'ємні значення.

```
TRectangle r1;
r1.a=-2;
r1.b=-9;
```

Для того, щоб уникнути таких помилок поля класу описують як закриті (зі специфікатором доступу **private**), тим самим забороняючи доступ до полів

поза межами класу. Для коректного звертання до закритого поля об'єкта у класі додатково розробляють функції для зчитування та встановлення його значення, у яких можна здійснити перевірку коректності значень, що необхідно надати закритому полю. Іноді, при описі закритого поля до його імені додають знак нижнього підкреслювання. Функція, яка повертає значення закритого поля, як правило, має префікс «**Get_**», а функція, яка використовується для встановлення значення закритого поля – «**Set_**».

Приклад. Опишемо клас `TRectangle`, у якому поля для збереження довжин сторін прямокутника будуть закритими.

```
class TRectangle {
private:
    double a_;           //Опис закритих членів класу
    double b_;
public:
    double Get_a(){return a_;}           //Зчитування a_
    void Set_a(double a){                 //Встановлення a_
        if (a>=0)a_=a;                   //Перевірка коректності значення a
    }
    double Get_b(){return b_;}           //Зчитування b_
    void Set_b(double b){                 //Встановлення b_
        if (b>=0)b_=b;                   //Перевірка коректності значення b
    }
    TRectangle (double a, double b) {
        Set_a(a); //Виклик функцій для встановлення значень закритих полів
        Set_b(b);
    }
    . . . . .
};
```

Зараз звертання до закритих полів об'єкта у програмі можливе тільки з використанням описаних функцій зчитування та встановлення значень.

```
int main()
{
    TRectangle r(3,8);
    r.Set_a(2);   r.Set_b(9);           //Встановлення значень
    double s= r.Get_a()*r.Get_b();      //Зчитування значень
    . . . . .
}
```

Дружні функції. Як зазначалося, згідно з принципом інкапсуляції, поля є закритими даними об'єкта, і поза межами класу доступними є тільки ті члени, які описані після специфікатора доступу **public**. Але у C++ є можливість описати функції, які можуть здійснювати доступ (звертатися) не тільки до відкритих, а й до закритих та захищених членів об'єкта, який передається у функцію як фактичний параметр. Такі функції називається дружніми. Прототип дружньої функції описують усередині класу з використанням ключового слова **friend**. Одним з параметрів дружньої функції повинен бути об'єкт класу, в якому описано дружню функцію, або **посилання** на нього.

Приклад. Опишемо дружню функцію **Replace** для класу **TRectangle**, яка буде здійснювати обмін значень закритих полів **a_** і **b_** об'єкта, що буде передано у функцію.

```
class TRectangle {
private:
    double a_;
    double b_;
public:
    . . . . .
friend void Replace(TRectangle &c); //Прототип функції
    . . . . .
};

void Replace(TRectangle &c) //Реалізація функції
{
    double temp=c.a_; //Звертання до закритих полів об'єкта c
    c.a_=c.b_;
    c.b_=temp;
}
int main()
{
    TRectangle r(2,6);
    Replace(r); //Виклик дружньої функції
    . . . . .
}
```

Дружні класи. Якщо всередині методів деякого класу необхідно здійснювати звертання до закритих членів об'єкта іншого класу, то можна ці методи (по аналогії з дружніми функціями) описати дружніми, або ж описати

дружнім увесь клас. В цьому випадку усі його методи будуть вважатися дружніми. Для опису дружнього класу необхідно розмістити неповне оголошення класу, перед яким вказано **ключове** слово **friend**, **усередині** оголошення іншого класу

```
friend class <Ім'я дружнього класу>;
```

Приклад. В якості приклада, створимо клас `TRectangleOperation`, який містить функції для знаходження площі та периметра прямокутника, і оголосимо його як дружній для класу `TRectangle`.

```
class TRectangle {
private:
    double a_;
    double b_;
public:
    TRectangle (double a,double b):a_(a),b_(b){};
                                     //Оголошення дружнього класу
    friend class TRectangleOperation;
};
class TRectangleOperation{
public:
    //Методи цього класу є дружніми для класу TRectangle
    double S(TRectangle &r){return r.a_*r.b_;}
    double P(TRectangle &r){return 2*(r.a_+r.b_);}
};
int main(int argc, char* argv[])
{
    TRectangle r(2,6);
    TRectangleOperation rop;
    cout<<"S="<<rop.S(r)<<endl;
    cout<<"P="<<rop.P(r)<<endl;
    system("pause");
    return 0;}

```

Успадкування

Однією із проблем у програмуванні є повторне використання створеного коду та його модифікація. У об'єктно-орієнтованому програмуванні цю проблему можна розв'язати з використанням успадкування. *Успадкування* – це основний принцип об'єктно-орієнтованого програмування, який ґрунтується на описі нових класів (класів-нащадків) на основі базових класів (класів-предків). При цьому клас-нащадок автоматично успадковує функціональні можливості класів-предків (поля і методи). У класі-нащадка успадковані методи можна перевизначити (описати метод з таким же заголовком, але іншим призначенням) та описати власні нові поля та методи. Розглянемо формат опису класу-предка

```
class <Клас-нащадок>: [<Специфікатор доступу>] <Клас-предок>{...};
```

Приклад. Опишемо клас, який моделює паралелепіпед TParalelepiped, як нащадок від класу TRectangle.

```
class TRectangle {                                //Опис класу TRectangle
    . . . . .
};

class TParalelepiped: public TRectangle {
    . . . . .
};
```

При успадкуванні можна вказувати один із специфікаторів доступу:

- **public** – відкрите успадкування. Всі відкриті і захищені члени базового класу (класу-предка) стають відповідно відкритими і захищеними членами класу-нащадка.
- **private** – закрите успадкування. Всі відкриті і захищені члени класу-предка стають закритими членами класу-нащадка.
- **protected** – захищене успадкування. Всі відкриті і захищені члени класу-предка стають захищеними членами класу-нащадка.

Якщо специфікатор доступу не вказаний, то використовується **закрите успадкування**.

При використанні **закритого успадкування** можна **змінити** рівень доступу деяких **відкритих** і захищених членів базового класу. Для цього необхідно при описі класу-нащадка оголошення потрібних членів класу-предка додатково включити у відповідний розділ класу-нащадка. Оголошення членів класу-предка можна зробити двома способами:

```
<Клас-предок> : : <Член класу>;
using <Клас-предок> : : <Член класу>;
```

Як правило, використовують варіант з використанням ключового слова **using**.

Приклад. Змінимо рівень доступу функції Diagonal класу TRectangle у класі TParalelepiped.

```

class TRectangle {                                     //Опис класу TRectangle
    . . . . .
protected:
    int r;
public:
    double Diagonal();
};

class TParalelepiped: private: TRectangle {
    . . . . .
public:
    using TRectangle::r;                               //Зміна рівня доступу
    using TRectangle::Diagonal;
    . . . . .
};

```

Закрите успадкування

Створюючи клас-нащадок можна використовувати класи-предки, які, в свою чергу, також є класами-нащадками від інших класів і т.д. Таким чином вибудовується цілі ієрархічні ланцюги успадкування. При **створенні** об'єкта класу-нащадка **у разі** ієрархічного успадкування спочатку **викликається** конструктор найпершого класу-предка в ланцюзі успадкування, а далі **викликаються** послідовно конструктори решти всіх класів-предків і лише потім **викликається** конструктор класу-нащадка. Деструктори **викликаються** у зворотному **порядку**. Для передачі параметрів конструкторам класів-предків використовується список ініціалізації, який повинен розташовуватися **після** двокрапки між списком параметрів і тілом конструктора класу-нащадка. **Всередині** списку ініціалізації **вказується** ім'я класу-предка, **після** якого **всередині** круглих дужок передаються значення.

```
<Конструктор нащадка>(<Форм.парам.>): <Конструктор предка> (<Значення>) {...}
```

В якості значень можуть використовуватися формальні параметри, що описано у конструкторі класу-нащадка.

Приклад. Опишемо конструктор класу TParalelepiped, при описі якого використовується виклик конструктора класу TRectangle.

```

class TRectangle {
    . . . . .
    TRectangle (double a,double b){. . .}
    . . . . .
};

class TParalelepiped: public: TRectangle{
    . . . . .
    TParalelepiped (double a,double b,double h): TRectangle(a,b) {...}
    . . . . .
};

```

Виклик
конструктора
предка

Приклад. Опишемо клас, який моделює паралелепіпед TParalelepiped, як нащадок від класу TRectangle.

```
class TRectangle {
private:
    double a_;
    double b_;
public:
    void Set_a(double a){if(a>=0)a_=a;}
    double Get_a(){return a_;}
    void Set_b(double b){if(b>=0)b_=b;}
    double Get_b(){return b_;}
    double S(){return a_*b_;}
    double P(){return 2*(a_+b_);}
    TRectangle (double a,double b){
        Set_a(a);Set_b(b);
    };
};

class TParalelepiped: public TRectangle{
private:
    double h;
public:
    void Set_h(double a){if(a>=0)a_=a;}
    double Get_h(){return a_;}
    double S(){return
2*(TRectangle::S()+2*Get_a()*h+2*Get_h()*h);}
    double V(){return h_*(TRectangle::S()); }
    TParalelepiped (double a,double b,double h):
    TRectangle(a,b) {
        Set_h(h);
    };
};
```

Множинне успадкування. Мова C++ підтримує також **множинне** успадкування, коли у якості базових використовують одразу декілька класів-предків. У цьому випадку використовується **наступний** формат опису класу-нащадка;

```
class <Клас-нащадок> : [<Специфікатор доступу>] <Клас-предок 1>,
                        [<Специфікатор доступу>] <Клас-предок 2>,
                        .....
                        [<Специфікатор доступу>] <Клас-предок N>
{
    <Опис членів класу-нащадка>;
} [<Список об'єктів класу-нащадка>;
```

Приклад. Опишемо клас TPrism, який моделює правильну трикутну призму. В основі призми правильний трикутник, а бічними гранями

прямокутник, тому клас TPrism опишемо як клас-нащадок від двох класів TRTriangle (правильний трикутник) та TRectangle (прямокутник) .

```
class TRectangle{
    . . . . .
};
class TRTriangle{
    . . . . .
};
class TPrism: public TRTriangle, public TRectangle {
    . . . . .
};
```

При **множинному** успадкуванні спочатку **викликається** конструктор базового класу, ім'я якого **розташоване** у списку першим. Після цього **викликається** конструктор базового класу, ім'я якого **розташоване** у списку успадкування правіше і так далі. Лише потім **викликається** конструктор похідного класу. Деструктори **викликаються** у зворотному **порядку**. Для передачі параметрів конструкторам базових класів-предків використовується **наступний синтаксис**:

```
<Конструктор нащадка>(<Форм.парам.>): <Конструктор предка 1> (<Значення>),
                                         <Конструктор предка 2> (<Значення>),
                                         . . . . .
                                         <Конструктор предка N> (<Значення>)
{...}
```

Значення конструкторам базових класів передаються через список ініціалізації, **усередині** якого **вказуються** імена базових класів-предків **після** яких **усередині** круглих дужок передаються значення.

Приклад. Розглянемо конструктори класу TPrism , який є класом-нащадком від двох класів предків TRTriangle та TRectangle.

```
class TRectangle{
private:
    double a_;
    double b_;
public:
    TRectangle(double a,double b){. . .}
    . . . . .
};
-----
class TRTriangle{
private:
    double a_;
public:
    TRTriangle(double a){. . .}
    . . . . .
};
-----
class TPrism: public TRTriangle, public TRectangle {
public :
    TPrism(double a, double h): TRTriangle(a), TRectangle(a,h) {...}
};
```

Зауваження. Як відомо, якщо не описано конструктор без параметрів, то при описі об'єкта такий конструктор може створюється автоматично і неявно

викликатися. Якщо ж клас є нащадком від декількох класів, то конструктор без парметорів описувати обов'язково.

Приклад. Опишемо можливі конструктори без параметрів для класів TRTriangle, TRectangle, TPrism.

```
class TRectangle{
private:
    double a_;
    double b_;
public:
    TRectangle(){a_=0;b_=0;};
    . . . . .
};

class TRTriangle{
private:
    double a_;
public:
    TRTriangle(){a_=0;};
    . . . . .
};

class TPrism: public TRTriangle, public TRectangle {
public :
    TPrism(): TRTriangle(),TRectangle(){}
    . . . . .
};
```

Якщо клас-нащадок **успадковує** декілька класів, то виникає неоднозначність, **оскільки** члени з однаковими іменами може бути описано у декількох класах-предках.

Приклад. Наведемо можливу реалізацію класу TPrism , який є класом-нащадком від двох класів предків TRTriangle та TRectangle

```
class TRectangle{
private:
    double a_;                //Довжини сторін прямокутника
    double b_;
public:
    void Set_a(double a){if(a>=0) a_=a;}
    double Get_a(){return a_;}
    void Set_b(double b){if(b>=0) b_=b;}
    double Get_b(){return b_;}
    TRectangle(){a_=0;b_=0;};
    TRectangle(double a,double b){Set_a(a); Set_b(b);}
    double S(){return a_*b_;}           //Площа прямокутника
    double P(){return 2*(a_+b_);}      //Периметр прямокутника
};

class TRTriangle{
private:
```

```

    double a_;           //Довжина сторони рівностороннього трикутника
public:
    void Set_a(double a){if(a>=0) a_=a;}
    double Get_a(){return a_;}
    TRTriangle(){a_=0;};
    TRTriangle(double a){Set_a(a);}
    double S(){return a_*a_*sqrt(3)/4;} //Площа рівн.трикутн.
    double P(){return 3*a_;}           //Периметр рівн.трикутн.
};
-----
class TPrism: public TRTriangle, public TRectangle {
public :
    TPrism(): TRTriangle(),TRectangle(){}
    TPrism(double a, double h): TRTriangle(a), TRectangle(a,h) {}
    double S(){return 2*TRTriangle::S()+3*TRRectangle::S();}
};
-----
int main(int argc, char* argv[])
{
    TPrism p(2,3);
    p.TRectangle::Set_a(31);
    p.TRTriangle::Set_a(9);
    cout<<"p.TRectangle::a="<<p.TRectangle::Get_a()<<endl;
    cout<<"p.TRTriangle::a="<<p.TRTriangle::Get_a()<<endl;
    system("pause");
    return 0;
}

```

У класі **TPrism** є два успадкованих закритих поля **a_** (одне у **TRTriangle** – сторона правильного трикутника, а друге у **TRectangle** – довжина однієї із сторін прямокутника) та по дві функції обчислення площі (трикутника і прямокутника) **S** та периметру **P** відповідно. В цьому випадку для усунення неоднозначності при звертанні до полів та методів зазначають ім'я класу.

```

<Ім'я класу>::<Ім'я поля>
< Ім'я класу >::< Ім'я метода>

```

Приклад. Звертання до членів класів-предків у методі **S ()** класу-нащадка для знаходження площі поверхні правильної трикутної призми (площа двох основ (трикутників) і трьох бічних сторін (прямокутників))

```

class TPrism: public TRTriangle, public TRectangle {
    .
    .
    .
    double S(){return 2*TRTriangle::S()+3*TRRectangle::S();}
};

```

Площа трикутника

Площа прямокутника

Звертання до членів об'єкта

```

int main(int argc, char* argv[])
{
    TPrism p(2,3);
    //< Ім'я об'єкта>.< Ім'я класу> ::<Ім'я члена>
    p.TRectangle::Set_a(31);
}

```

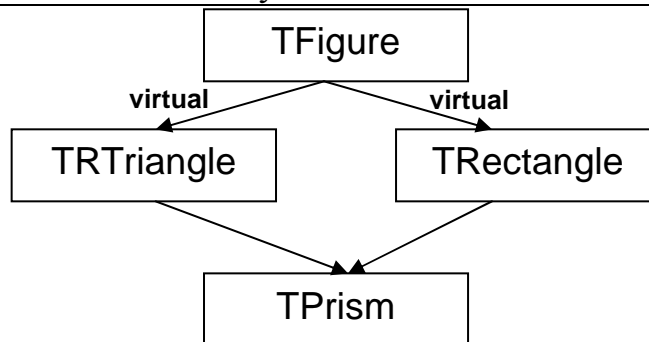
```

        p.TRTriangle::Set_a(9);
        p.Set_b(23); //Клас зазначати не потрібно, бо немає неоднозначності
        cout<<"p.TRectangle::a_="<<p.TRectangle::Get_a()<<endl;
        cout<<"p.TRTriangle::a_="<<p.TRTriangle::Get_a()<<endl;
        cout<<"p.b_="<<p.Get_b()<<endl;
        . . . . .
    }

```

У випадку, якщо клас-нащадок створюється на основі декількох базових класів-предків, які у свою чергу є нащадками деякого спільного класу, то у класі-нащадка буде декілька екземплярів однакових методів такого спільного класу, успадкованих від різних класів-нащадків. Для уникнення неоднозначності можна, як і у попередньому випадку, зазначати ім'я одно з класів-нащадків. Для того, щоб у класі-нащадка методи спільного класу існувати тільки в одному екземплярі, необхідно при описі класів-предків, які успадковують спільний клас, у їх списку успадкування зазначати спільний клас як віртуальний. Очевидно, в цьому випадку неоднозначності не буде.

Приклад. Нехай клас **TFigure**, у якому описано функцію **Show**, є спільним предком для класів **TRTriangle** та **TRectangle**, а ці класи, в свою чергу, є класами-предками для класу-нащадка **TPrism**.



```

class TFigure{
public:
    void Show(){. . .}
};
-----
class TRectangle: virtual public TFigure{
    . . . . .
};
-----
class TRTriangle: virtual public TFigure{
    . . . . .
};
-----
class TPrism: public TRTriangle, public TRectangle {
    . . . . .
};
-----
int main(int argc, char* argv[])
{
    TPrism p(2,3);
}

```

```

    p.Show();           //Неоднозначності немає
    . . . . .
}

```

Показчики на об'єкти. Як і для структур, адресу об'єкта можна зберегти в показчику відповідного класу. Оголошення показчика на об'єкт проводиться так, як і на будь-якого іншого **типу** даних

<Ім'я класу> * <Показчик>

Для **отримання** адреси об'єкту використовується оператор **&**, а для звертання до членів застосовується оператор **«->»**

```

<Показчик>   -> <Поле об'єкта>
<Показчик>   -> <Метод об'єкта>

```

або виконують операцію розмінування показчика

```

(*<Показчик>) . <Поле об'єкта>
(*<Показчик>) . <Метод об'єкта>

```

Приклад. Розглянемо клас **TSquare**, який моделює квадрат. Доступ до членів об'єкта здійснимо з використанням показчика **S1**.

```

class TSquare{
public:
    double a;
    TSquare(double a):a(a){}
    double S(){return a*a;}
};

int main(int argc, char* argv[])
{
    TSquare s(5);           //Створення об'єкта s
    TSquare *s1;           //Опис показчика s1
    s1=&s;                   //Показчика s1 одержує адресу об'єкта s
    s1->a=8;                 //Звертання до поля об'єкта a
    cout<<"S="<<s1->S()<<endl;           //Виклик метода S
    return 0;
}

```

Особливістю показчиків на об'єкти класів є те, що показчик класу-предка, може містити адресу об'єкта класу-нащадка. Але при цьому допустимими є звертання тільки до тих відкритих членів об'єкта класу-нащадка, які клас-нащадок успадкував від класу-предка. Якщо ж необхідно звернутися до членів класу-нащадка, які оголошено безпосередньо у класі-нащадка то необхідно додатково виконати приведення типу показчика до типу показчика класу-нащадка

((<Клас-нащадок> *) <показчик>) -> <Член об'єкта>

Приклад. Розглянемо клас **TCube**, який моделює куб і є класом-нащадком розглянутого раніше класу **TSquare**.

```

class TSquare{

```

```

public:
    double a;
    TSquare(double a):a(a){}
    double S(){return a*a;}           //Площа квадрата
};
-----
class TCube: public TSquare{
public:
    TCube(double a): TSquare(a){}
    double S(){return 6*TSquare::S();} //Площа поверхні куба
    double V(){return a* TSquare::S();} //Об'єм куба
};
-----
int main(int argc, char* argv[])
{
    TCube c1(5);                      //Створюємо об'єкт c1 типу TCube
    TSquare *p=&c1;                    //Створюємо покажчик p типу TSquare
    //Покажчик класу TSquare p одержує адресу об'єкта класу-нащадка TCube c1
    p=&c1;
    p->a=8;
                                // p->S() площа квадрата
    cout<<"TSquare::S ="<<p->S()<<endl;
                                //((TCube*)p )->S() площа поверхні куба
    cout<<"TCube::S ="<<((TCube*)p )->S()<<endl;
                                //((TCube*)p )->V() об'єм куба
    cout<<"TCube::V ="<<((TCube*)p )->V()<<endl;
    system("pause");
    return 0;
}

```

У даному прикладі хоча покажчик містить адресу об'єкта класу-нащадка TCube, все одно при звертанні до методу S

p->S()

знаходиться площа квадрата, а не площа поверхні куба.

Для того, щоб знайти площу поверхні об'єкта-куба, яка міститься у покажчику p додатково необхідно виконати приведення типу покажчика до типу покажчика на клас-нащадок TCube

((TCube*)p)->S()

Нехай у класі-предка і класі-нащадка описано методи, заголовки яких співпадають (не перевантажуються). Якщо необхідно, щоб метод який необхідно викликати визначався у залежності від того, адреса якого об'єкта міститься у покажчику необхідно цей метод у класі-предка описати як віртуальний, з використанням ключового слова **virtual**. Таке явище називають *динамічним поліморфізмом*.

virtual <Тип результату> <Ім'я метода>(<Форм. параметри>){. . .}

Приклад. Модифікуємо попередній приклад, описавши метод S у класі TSquare як віртуальний.

```

class TSquare{

```

```

public:
    double a;
    TSquare(double a):a(a){}
    virtual double S(){return a*a;} //Метод S - віртувальний
                                     //Площа квадрата
};
-----
class TCube: public TSquare{
public:
    TCube(double a): TSquare(a){}
    double S(){return 6*TSquare::S();} //Площа поверхні куба
    double V(){return a* TSquare::S();} //Об'єм куба
};
-----
int main(int argc, char* argv[])
{
    TCube c1(5);
    TSquare s1(5);
    TSquare *p;

    TCube c1(5); //Створюємо об'єкт c1 типу TCube
    TSquare s1(5); //Створюємо об'єкт s1 типу TSquare

    //Показчик класу TSquare p одержує адресу об'єкта класу TSquare s1
    p=&s1;
    cout<<"TSquare::S ="<<p->S()<<endl; //Площа квадрата

    //Показчик класу TSquare p одержує адресу об'єкта класу TCube c1
    p=&c1;
    cout<<"TCube::S ="<< p->S()<<endl; //Площа поверхні куба
    system("pause");
    return 0;
}

```