Міністерство освіти і науки України
Харківський національний університет радіоелектроніки
Кафедра КІТАМ

# КОМПЛЕКТ ЛЕКЦІЙ

навчальної дисципліни «Програмування»
підготовки бакалавра

спеціальності 151 «Автоматизація та комп'ютерно-інтегровані технології»
освітня програма «Автоматизація та комп'ютерно-інтегровані технології»,
«Системна інженерія»

Розробник І.К. Сезонова, проф. каф. КІТАМ, к.т.н., доцент
Схвалено на засіданні кафедри КІТАМ
Протокол від 22 серпня 2022 р. № 1

Харків 2022 р.

Конспект лекцій з дисципліни «Програмування» для студентів спеціальності 151 «Автоматизація та комп'ютерно-інтегровані технології», освітня програма спеціалізації «Автоматизація та комп'ютерно-інтегровані технології», «Системна інженерія» / Упоряд.: Сезонова І.К. – Харків, ХНУРЕ, 2022. – 85 с.

Упорядник: І.К. Сезонова, к.т.н., проф. каф. КІТАМ

Рецензент: О.М. Цимбал, докт. техн. наук, проф. каф. КІТАМ.

ЗМІСТ


Вступ 4
1. Історія створення та загальна характеристика C++…………………….. 6
2. Алфавіт. Ідентифікатори…………………………………………… 5
3. Проста програма на мові C++. Типи даних. Змінні………………… 7
4. Локальні та глобальні змінні. Область видимості. Директиви препроцесора………………………………………………………… 10
5. Покажчики. Адреса. Константи……………………………………… 12
6. Арифметичні і логічні операції. Оператор присвоювання…………… 14
7. Поняття алгоритму. Класифікація алгоритмів………………………… 17
8. Умовні оператори та оператор switch-case…………………………… 18
9. Введення/виведення та їх організація. Консольний та файловий вивід…………………………………………………………………… 20
10. Поняття циклу. Оператори циклу. Цикл while, do while…………….. 26
11. Масиви……………………………………………………………….. 28
12. Покажчик. Адреса…………………………………………………... 31
13. Арифметичні і логічні операції……………………………………… 32
14. Оператор керування………………………………………………… 35
15. Оператори циклу while і do while…………………………………….. 35
16. Сортування даних. Алгоритми сортування………………………….. 37
17. Функції в сі++. Прототипи функцій. Перевантаження функцій……… 41
18. Функції зі змінною кількістю параметрів. Структури. Об'єднання….. 44
19. Перерахування. Генератор випадкових чисел………………………… 47
20. Класи в C++……………………………………………………… 50
21. Конструктор. Деструктор…………………………………………… 51
22. Успадкування………………………………………………………… 58
23. Функції роботи з кирилицею, датою і часом……………………….. 61
24. Препроцесор………………………………………………………… 63
25. Динамічний розподіл пам'яті………………………………………… 68
26. Успадкування………………………………………………………... 71
27. Робота з файлами…………………………………………………… 74
28. Функціонали array і vector в C++……………………………………... 76
29. Перевантаження операторів…………………………………………... 79
30. Строкові класи std::string и std::wstring………………………………. 80
Перелік використаних джерел………………………………………… 82

## ВСТУП

Мова програмування (англ. Programming language) − це штучна мова, створена для передачі команд машинам, зокрема комп’ютерам. Мови програмування використовуються для створення програм, котрі контролюють поведінку машин, та запису алгоритмів. Мова програмування − це система позначень для опису алгоритмів та структур даних, певна штучна формальна система, засобами якої можна виражати алгоритми. Мову програмування визначає набір лексичних, синтаксичних і семантичних правил, що задають зовнішній вигляд програми і дії, які виконує виконавець (комп'ютер) під її управлінням Однією з проблем, які виникають на початкових етапах розробки програмного продукту, є вибір мови програмування, що в свою чергу впливає на вибір середовища програмування. Існує маса факторів, що впливають на вибір мови. Однак перш ніж прийняти рішення на користь тієї чи іншої мови програмування, слід мати уявлення про їх класифікації та особливості. Всі існуючі мови програмування поділяються на такі групи: − універсальні мови високого рівня; − спеціалізовані мови розробника програмного забезпечення; − спеціалізовані мови користувача; − мови низького рівня. Мови програмування високого рівня є зрозумілішими людині, ніж комп’ютеру. Особливості конкретних комп’ютерних архітектур в них не враховуються, тому створені програми легко переносяться з комп’ютера на комп’ютер. Здебільшого достатньо просто перекомпілювати програму під певну комп’ютерну архітектурну та операційну систему. Розробляти програми на таких мовах значно простіше і помилок допускається менше. Значно скорочується час розробки програми, що особливо важливо при роботі над великими програмними проектами. Наразі у середовищі розробників вважається, що мови програмування, які мають прямий доступ до пам'яті та регістрів або мають асемблерні вставки, потрібно вважати мовами програмування з низьким рівнем абстракції. Тому більшість мов, які вважалися мовами високого рівня до 2000-го року зараз вже такими не вважаються: Адресна мова програмування, Фортран, Кобол, Алгол, Pascal, PascalABC, Java, C, C++, Objective C, Smalltalk, C#, Delphi Недоліком мов високого рівня є більший розмір програм порівняно з програмами на мові низького рівня. Сам текст програм на мові високого рівня менший, проте, якщо взяти у байтах, то код початково написаний на асемблері буде компактніший. Тому в основному мови високого рівня використовуються для розробок програмного забезпечення комп'ютерів, і пристроїв, які мають великий обсяг пам’яті. А різні підвиди асемблеру застосовуються для програмування інших пристроїв, де критичним є розмір програми.

В групі універсальних мов високого рівня безумовним лідером сьогодні є мова С (разом з С ++), що мають цілий ряд дуже істотних переваг: − багатоплатформність − для всіх використовуваних в даний час платформ існують компілятори з мови С і С ++; − наявність операторів, що реалізують основні структурні алгоритмічні конструкції (умовну обробку, всі види циклів); − можливість програмування на низькому (системному) рівні з використанням адрес оперативної пам’яті; − величезні бібліотеки підпрограм і класів. Спеціалізовані мови розробника використовують для створення конкретних типів програмного забезпечення. До них відносять: − мови баз даних; − мови створення мережевих додатків; − мови створення систем штучного інтелекту і т. д. Спеціалізовані мови користувача зазвичай є частиною професійних середовищ, характеризуються вузькою спрямованістю і розробниками програмного забезпечення не використовуються. Мови низького рівня дозволяють здійснювати програмування практично на рівні машинних команд. При цьому отримують найоптимальніші, як з точки зору часу виконання, так і з точки зору обсягу необхідної пам’яті програми. Але ці мови абсолютно не годяться для створення великих програм і, тим більше, програмних систем Прикладом мови низького рівня є асемблер. Мови низького рівня орієнтовані на конкретний тип процесора і враховують його особливості, тому для перенесення програми на асемблері на іншу апаратну платформу її потрібно майже цілком переписати. За допомогою мов низького рівня створюють ефективні і компактні програми, оскільки розробник отримує доступ до всіх можливостей процесора. Мови низького рівня, як правило, використовують для написання невеликих системних програм, драйверів пристроїв, модулів стиків з нестандартним обладнанням, програмування спеціалізованих

мікропроцесорів, коли найважливішими вимогами є компактність, швидкодія і можливість прямого доступу до апаратних ресурсів В даний час мови типу асемблера зазвичай використовують: − при написанні порівняно простих програм, що взаємодіють безпосередньо з технічними засобами (наприклад, драйверів); − у вигляді вставок в програми на мовах високого рівня. Інколи в літературі та в Інтернеті згадують про п'ять поколінь мов програмування. Високорівневі мови програмування вважаються третім, асемблерні мови − другим, а машинний код − першим поколінням.

Класифікація мов на четверте і п’яте покоління проводиться різними авторами по різному по різних ознаках і різниця між мовами третього, четвертого та п'ятого покоління часто нечітка. Висновок. При виборі мови програмування слід керуватися такими міркуваннями: мова повинна бути зручною для програміста, придатною для даного комп’ютера, придатною для вирішення конкретного завдання.

## 1. Історія створення та загальна характеристика С/ С++

Мова С++ була створена на початку 80-х років XX ст. співробітником фірми Bell Laboratories Берном Страуструпом, як розширення мови С. До початку офіційної стандартизації мова розвивалася здебільшого силами Б. Страуструпа у відповідь на запити програмістського співтовариства.

У 1998 р. був ратифікований міжнародний стандарт мови С++: ISO/IEC 14882:1998 "Standard for the C++ Programming Language"; після прийняття технічних виправлень до стандарту у 2003 р. — сучасна версія цього стандарту — ISO/IEC 14882:2003. Ідея створення нової мови бере початок від мови моделювання Сімула (Simula). Вона мала низку можливостей, що були корисні для розробки великого програмного забезпечення, але працювала занадто повільно. У той самий час мова BCPL досить швидка, але занадто близька до мов низького рівня й не підходить для розробки великого програмного забезпечення. Б. Страуструп почав працювати у Bell Laboratories над задачами теорії черг (у додатку до моделювання телефонних викликів). Він вирішив доповнити мову С (спадкоємець BCPL) можливостями, наявними у мові Сімула. Мова С, яка є базовою мовою системи UNIX, швидка та багатофункціональна. Б. Страуструп додав їй можливість роботи із класами й об'єктами. У результаті практичні задачі моделювання виявилися доступними для розв'язання як з погляду часу розроблення (завдяки використанню Сімула-подібних класів), так і з погляду часу обчислень (завдяки швидкодії С). Нововведеннями С++ порівняно із С є:

- підтримка об'єктно-орієнтованого програмування;
- підтримка узагальненого програмування через шаблони;
- додаткові типи даних;
- виключення;
- простір імен;
- вбудовані функції;
- перевантаження операторів;
- перевантаження імен функцій;
- посилання й оператори керування вільно розподіленою пам'яттю;
- доповнення до стандартної бібліотеки.

Отже, можна сказати, що мова С++ багато у чому є надмножиною мови С.

Варто також відзначити, що С++ має ряд недоліків, деякі з них успадковані від С:

- синтаксис, що провокує помилки, наприклад, операція присвоювання позначається як = , а операція порівняння як ==, їх легко переплутати; if (x = 0) {оператори}. Тут в умовному операторі помилково написане присвоювання замість порівняння. У результаті, замість того, щоб зрівняти поточне значення x з нулем, програма присвоїть x нульове значення;
- макроси (#define) є потужним, але небезпечним засобом. Вони збережені у С++, незважаючи на те, що в них немає необхідності завдяки шаблонам і вбудованим функціям. В успадкованих стандартних С-бібліотеках багато потенційно небезпечних макросів;
- деякі перетворення типів неінтуїтивні. Зокрема, операція над беззнаковим і знаковим числами видає беззнаковий результат;
- необхідність записувати break у кожному розгалужені оператора switch і можливість послідовного виконання кількох розгалужень при його відсутності провокує помилки через пропуск break. Ця сама особливість дає змогу робити сумнівні "трюки", що базуються на вибірковому незастосуванні break, і ускладнює розуміння коду.

Однак, незважаючи на свої недоліки, С++ є найпоширенішою мовою програмування для операційних систем Windows і Unix. Також, незважаючи на декларовану досконалість мов C# і Java, витиснути С++ ним так і не вдалося.

## 2. Алфавит. Ідентифікатори

Алфавіт мови програмування – це фіксований набір основних символів.

Алфавіт мови С++ складається з:

- великих і малих літер латинського алфавіту: ”A, ..., Z”, "a, ..., z";
- цифр 0, 1, ..., 9;
- спеціальних символів: " ’ ()[]{}<>•» ;:?! — *+— = / \ I # % $ & ~ ~ @ _

**Ідентифікатор** - це назва (ім’я), яку користувач надає об’єктам, наприклад, змінним, сталим, функціям. Усі ідентифікатори можуть складатися з рядкових чи прописних літер англійського алфавіту, цифр, а також містити символ підкреслення. Ідентифікатор завжди починається з букви або із символу підкреслення.

**Зауваження 1.** Однакові за змістом малі та великі літери у мові С++ вважаються різними символами.

Наприклад, імена MyName та myname позначають різні об’єкти.

**Зарезервовані ідентифікатори**

Мова С++ має зарезервований набір із 84 слів (включаючи версію C++17) для використання. Ці слова називаються ключовими словами, кожне з яких має особливе значення.

Ось список ключових слів у мові C++ (включаючи C++17):

| | | | |
|---|---|---|---|
| alignas | decltype | namespace | struct |
| (C++11) | (C++11) | new | switch |
| alignof (C++11) | default | noexcept (C++11) | template |
| and | delete | not | this |
| and_eq | do | not_eq | thread_local |
| asm | double | nullptr (C++11) | (C++11) |
| auto | dynamic_cast | operator | throw |
| bitand | else | or | true |
| bitor | enum | or_eq | try |
| bool | explicit | private | typedef |
| break | export | protected | typeid |
| case | extern | public | typename |
| catch | false | register | union |
| char | float | reinterpret_cast | unsigned |
| char16_t (C++11) | for | return | using |
| char32_t (C++11) | friend | short | virtual |
| class | goto | signed | void |
| compl | if | sizeof | volatile |
| const | inline | static | wchar_t |
| constexpr | int | static_assert | while |
| (C++11) | long | (C++11) | xor |
| const_cast | mutable | static_cast | xor_eq |
| continue | | | |

## 3. Проста програма на мові С++. Типи даних. Змінні

Найкоротша програма на мові Сі ++ виглядає так:

```
// Найпростіша програма
int main () {return 1;  }
```

Перший рядок у програмі - коментар, який служить лише для пояснення. Ознакою коментаря є два знака поділу поспіль (//).

main - це ім'я головної функції програми. З функції main завжди починається виконання. У функції є ім'я (main), після імені в круглих дужках перераховуються аргументи або параметри функції (в даному випадку у функції main аргументів немає). У функції може бути результат або повертається значення. Якщо функція не повертає ніякого значення, то це

позначається ключовим словом void. У фігурних дужках записується тіло функції - дії, які вона виконує. Оператор return 1 означає, що функція повертає результат - ціле число 1.

Якщо ми говоримо про об'єктно-орієнтованої програми, то вона повинна створити об'єкт будь-якого класу і послати йому повідомлення.

Щоб не ускладнювати програму, ми скористаємося одним з готових, визначених класів - класом ostream (потік вводу-виводу).

Цей клас визначено в файлі заголовків "iostream.h". Тому перше, що треба зробити - включити файл заголовків в нашу програму:

```
#include <iostream.h>
int main () {return 1;  }
```

Крім класу, файл заголовків визначає глобальний об'єкт цього класу cout. Об'єкт називається глобальним, оскільки доступ до нього можливий з будь-якої частини програми. Цей об'єкт виконує висновок на консоль.

У функції main ми можемо до нього звернутися і послати йому повідомлення:

```
#include <iostream.h>
int main ()
 {
   cout << "Hello world!"  << endl;
   return 1;
 }
```

Операція зсуву << для класу ostream визначена як "вивести". Таким чином, програма посилає об'єкту cout повідомлення "вивести рядок Hello world!" і "вивести новий рядок" (endl позначає новий рядок). У відповідь на ці повідомлення об'єкт cout виведе рядок "Hello world!" на консоль і переведе курсор на наступний рядок.

**Компіляція і виконання програми**

Програма на мові Сі ++ - це текст. За допомогою довільного текстового редактора програміст записує інструкцію, відповідно до якої комп'ютер буде працювати, виконуючи дану програму.

Для того щоб комп'ютер міг виконати програму, написану на мові Сі ++, її потрібно перевести на мову машинних інструкцій. Це завдання вирішує компілятор. Компілятор читає файл з текстом програми, аналізує її, перевіряє на предмет можливих помилок і, якщо таких не виявлено, створює виконуваний файл, тобто файл з машинними інструкціями, який можна виконувати.

Відкомпільовану програму можна виконувати багаторазово, з різними вихідними даними.

Не маючи можливості описати всі варіанти, зупинимося тільки на двох найбільш часто зустрічаються.

**Імена, змінні і константи**

Для символічного позначення величин, імен функцій і т.п. використовуються імена або ідентифікатори.

Ідентифікатори в мові Сі ++ - це послідовність знаків, що починається з букви. У ідентифікатори можна використовувати великі і малі латинські букви, цифри і знак підкреслення. Довжина ідентифікаторів довільна.

Приклади правильних ідентифікаторів:

Abc    A12    NameOfPerson    BITES_PER_WORD

У С++ є набір вбудованих типів даних для подання цілих і дійсних чисел, символів, а також тип даних "символьний масив", що служить для зберігання символьних рядків.

Тип char використовується для зберігання окремих символів та невеликих цілих чисел і займає 1 байт.

Типи short, int і long призначені для представлення цілих чисел. Вони розрізняються тільки діапазоном значень, які можуть приймати числа, а конкретні розміри перелічених типів залежать від реалізації.

Тип short займає половину машинного слова, int — одне слово, long — одне або два слова. У 32-бітних системах int і long, як правило, одного розміру.

Типи float, double і long double призначені для чисел із плаваючою крапкою й розрізняються точністю представлення (кількістю значущих розрядів) і діапазоном.

Тип float (одинарна точність) займає одне машинне слово, double (подвійна точність) — два, а long double (розширена точність) — три слова.

Тип bool має два значення — true і false — і займає 1 біт.

Char, short, int і long разом складають цілі типи, які, в свою чергу, можуть бути знаковими (signed) і беззнаковими (unsigned).

У знакових типах самий лівий біт використовується для зберігання знака (0 — плюс, 1 — мінус), а біти, що залишилися, містять значення.

У беззнакових типах всі біти використовуються для значення. 8-бітовий тип signed char може представляти значення від -128 до 127, а unsigned char — від 0 до 255.

Коли у програмі зустрічається деяке число, наприклад 1, то це число називається літералом, або літеральною константою. Літерали цілих типів можна записати у десятковому, восьмеричному й шістнадцятиричному вигляді. Наприклад, число 20, представлене десятковим, восьмеричним і шістнадцятиричним літералами, має вигляд:

- 20 // десятковий;
- 024 // восьмеричний;
- 0x14 // шістнадцятиричний.

Якщо літерал починається з 0, він трактується як восьмеричний, якщо з 0x або 0X, то як шістнадцятиричний.

За замовчуванням всі цілі літерали мають тип signed int. Можна явно визначити цілий літерал таким, що має тип long, приписавши наприкінці числа літеру L (використовується як прописна L, так і рядкова l ). Буква U (або u) наприкінці визначає літерал як unsigned int, а дві літери — UL або LU — як тип unsigned long.

Наприклад: 128u 1024UL 1L 8Lu.

Літерали, що представляють дійсні числа, можуть бути записані як з десятковою крапкою, так і в науковій (експонентній) нотації. За замовчуванням вони мають тип double. Для явної вказівки типу float потрібно використовувати суфікс F або f, а для long double — L або l, але тільки у випадку запису з десятковою крапкою.

Наприклад: 3.14159F 0/1f 12.345L 0.0 3el 1.0 E-3E 2. 1.0L.

Спеціальні символи (табуляція, повернення каретки) записуються як escape-послідовності.

Визначено наступні послідовності (вони починаються із символу зворотної косої риски):

- новий рядок \n;
- горизонтальна табуляція \t;
- забій \b;
- вертикальна табуляція \v;
- повернення каретки \r;
- прогін аркуша \f;
- дзвінок \a;
- зворотна коса риска \\;
- питання \?;
- одиночні лапки \’;
- подвійні лапки \”;
- escape — послідовність загального вигляду має форму \ooo, де ooo — від одного до трьох восьмеричних цифр. Це число є кодом символу.

Наприклад:

- \7 (дзвінок);
- \14 (новий рядок);
- \0 (null);

– \062 (‘2’).

**Змінні**

Змінна, або об’єкт — це іменована область пам’яті, до якої ми маємо доступ із програми; туди можна записувати значення а потім їх використовувати. Кожна змінна C++ має певний тип, що характеризує розмір і розташування в області пам’яті, діапазон значень, який вона може зберігати, і набір операцій, які застосовуються до цієї змінної.

Наприклад: int student_count; double salary; bool on_loan; string street_address; char delimiter.

Початкове значення може бути задане прямо в операторі визначення змінної. У C++ припустимі дві форми ініціалізації змінної — явна, з використанням оператора присвоювання: int ival = 1024; string project = “Fantasia 2000”; і неявна, із завданням початкового значення у дужках: int ival (1024); string project (“Fantasia 2000”). Обидва визначення еквівалентні. Вбудовані типи даних мають спеціальний синтаксис для завдання нульового значення: // ival одержує значення 0, a dval — 0.0 int ival = int(); double dval = double().

Зі змінною асоціюються дві величини: 1) власне значення, або r-значення (від read value — значення для читання), що зберігається у цій області пам’яті й притаманне як змінній, так і літералу; 2) значення адреси області пам’яті, що асоційована зі змінної, або l-значення (від location value — значення місця розташування) — місце, де зберігається r-значення, притаманне тільки об’єкту.

Наприклад, у виразі ch = ch –‘0’; змінна ch перебуває і ліворуч, і праворуч від символу операції присвоювання. Праворуч розташоване значення для читання (ch і символьний літерал ‘0’): асоційовані зі змінної дані зчитуються з відповідної області пам’яті. Ліворуч — значення місця розташування: в область пам’яті, асоційовану зі змінної ch, міститься результат віднімання. Ім’я змінної, або ідентифікатор, може складатися з латинських літер, цифр і символу підкреслення. Прописні й малі літери в іменах розрізняються.

Мова C++ не обмежує довжину ідентифікатора, однак користуватися занадто довгими іменами типу незручно. Ключові слова не можна використовувати як ідентифікатори.

## 4. Локальні та глобальні змінні. Область видимості. Директиви препроцесора

Кожна змінна має свою зону видимості. За межами цієї області про цюзмінну нічого відомо не буде, тому використовувати її не можна.

Змінна знаходиться в області видимості, якщо до неї можна отримати доступ.

Змінні, оголошені всередині функції, називаються локальними. Локальні змінні мають свої області видимості, цими областями є функції, в яких оголошено змінні (принцип найменших привілеїв – основний принцип програмування).

Глобальні змінні оголошуються поза тілом будь-якої функції, і тому область видимості таких змінних поширюється на всю програму.

Зазвичай глобальні змінні оголошуються перед головною функцією, але можна оголошувати і після функції main(), але тоді ця змінна не буде доступна поза функції main().

Приклад.

```
#include <iostream>
int i = 2; // глобальна змінна (можна використовувати в будь-якій ділянці коду програми)
int sum()
{
int k = 2; // локальная змінна (видимість тільки всередині функції sum())
return i + k;}
int main()
{ std::cout << i << std::endl << sum() << std::endl;}
```

Область видимості поширюється і на внутрішні блоки

Приклад.

```
#include <iostream>
int i = 2;
int sum()
{ int k = 2;
for (int i = 0; i < 10; i++) // ця i - локальна, оголошена в тілі циклу k += 1;
return i + k; // ця i - глобальна, оголошена поза функцій
}
int main()
{ std::cout << i << std::endl << sum() << std::endl;}
```

Тут виявлені наступні області видимості:

- глобальна - i = 2 належить їй;
- локальна щодо функції - змінна k;
- локальна щодо циклу for () - друга i.

Незважаючи на те, що у for є своя область видимості, сам for належить функції sum(). А значить він, і все що в ньому знаходиться,підпорядковується локальної області видимості цієї функції. Тобто всі змінні, визначені в функції так само дійсні і в тілі for, що і дозволяє працювати

оператору k + = 1 (він же k = k + 1 або k ++).

Особливості змінною i, яка описана всередині for!

Незважаючи на ім'я, ідентичне з глобальної змінної описаної вище, це вже інша змінна.

Приклад.

```
#include <iostream>
int main()
{ int a = 0;
{ int a = 2;
std::cout << a << std::endl; }
std::cout << a << std::endl;}
```

**Простір імен** (англ. *namespace)* - деяка множина, під якою мається на увазі модель, абстрактне сховище або оточення, створене для логічного угрупування унікальних ідентифікаторів (тобто імен ).

Ідентифікатор, зосереджений у просторі імен, асоціюється з цим простором . Один і той самий ідентифікатор може бути незалежно визначений у декількох просторах. Значення, пов'язане з ідентифікатором, який знаходиться в одному просторі імен, може мати (чи не мати) таке ж саме значення, як і той самий ідентифікатор, визначений у іншому просторі.

**Про́стір іме́н** — концепція у програмуванні, призначена для розмежування різних множин ідентифікаторів і уникнення конфліктів між їхніми іменами.

**Простір імен STD**

Уся стандартна бібліотека мови C ++ визначена всередині namespace std .

Кожен заголовок зі стандартної бібліотеки мови Сі включений в стандартну бібліотеку мови C ++ під різними іменами, створеними шляхом відсікання розширення .h і додаванням 'c' на початку, наприклад, 'time.h' став 'ctime'.

Також, функції повинні бути поміщені в простір імен std :: (хоча деякі компілятори самі роблять це).

**Константи**

Константа – це обмежена послідовність символів алфавіту мови, що представляє собою зображення фіксованого (незмінного) об'єкта.

Всі константи незалежно від типу даних можна поділити на дві

категорії: іменовані константи і константи, які не мають власного імені.

Приклад.

- 25 - константа цілого типу;
- 3.14 - дійсна константа;
- 'A' - символьна константа.

Приклад.

int k = 25; // змінна k ініціалізована константою - цілим числом 25.

У мові C ++ константи – це змінні, які не можна змінювати після ініціалізації.

Приклад.

```
const double PI = 3.14; // тут PI – константна змінна
double t;
t = PI * 2;
```

Приклад.

```
extern const int a;
```

**Директиви препроцесора**

Директива #include

Директива #include вставляє код із зазначеного файлу в поточний файл,тобто, просто підключивши інший файл, ми можемо користуватися його функціями, класами, змінними.

Заголовки зазвичай знаходяться або в поточній директорії, або в стандартному системному каталозі

Підключення заголовних файлів виконується під час компіляції, або як файл, який є частиною вашого проекту.

Параметри директиви #include записуються в «» або <>.

Приклад.

```
#include “path”
 #include <path>
```

#include "path" шукає в директорії в якій знаходиться поточний файл, і потім шукає там же де і #include <path>.

#include <path> шукає в шляхах переданих компілятору за допомого командного рядка; в шляхах, заданих змінними оточення; в шляхах, вшитих в компілятор.

Директива #define

Директива #define приймає дві форми:

- визначення констант;
- визначення макросів.

Взначення констант:

```
#define nameToken value
```

При використанні імені константи nameToken, воно буде замінено на value.

Приклад.

```
#include <iostream>
#define TEXT "C++" // визначення константи
int main()
{
std::cout << TEXT;
return 0;}
```

## 5. Покажчики. Адреса. Константи

Покажчик — це об’єкт, що містить адресу іншого об’єкта і дає змогу побічно маніпулювати цим об’єктом. Найчастіше покажчики використовуються для роботи з динамічно створеними об’єктами для побудови зв’язаних структур даних, таких як зв’язані списки й ієрархічні дерева, і для передачі у функції великих об’єктів — масивів і об’єктів класів.

Кожний покажчик асоціюється з деяким типом даних, причому їх внутрішнє подання не залежить від внутрішнього типу: і розмір пам’яті, що займає об’єкт типу покажчик, і діапазон значень у них однаковий. Різниця полягає в тому, як компілятор сприймає об’єкт, що адресується. Покажчики на різні типи можуть мати те саме значення, але область пам’яті, де розміщуються відповідні типи, може бути різною: покажчик на int, що містить значення 9 адреси 1000, вказує на область пам’яті 1000–1003 (у 32-бітній системі); покажчик на double, що містить значення адреси 1000, вказує на область пам’яті 1000–1007 (у 32-бітній системі).

Наприклад:
int *ip1, *ip2;
complex *cp;
string *pstring;
vector *pvec;
double *dp.
Розглянемо приклади використання покажчиків:
int ival = 1024;
//pi ініціалізовано нульовою адресою
int *pi = 0;
// pi2 ініціалізовано адресою
ival int *pi2 = &ival;
// правильно: pi і pi2 містять адресу
ival pi = pi2;
// pi2 містить нульову адресу
pi2 = 0;
// помилка: pi не може приймати значення
int pi = ival;
double dval;
double *ps = &dval;
// помилки компіляції, неприпустиме присвоювання типів даних: int* //<== double* pi = pd pi = &dval;
//void* може містити адреси будь-якого типу
void *pv = pi;
pv = pd;
// непряме присвоювання змінній ival значення

**Специфікатор const**

Специфікатор const використовується для оголошення константи: Наприклад:
const int bufSize = 512;
// помилка: неініціалізована константа const double pi.
Можна також використовувати покажчики на константи
const double *pc = 0;
const double minWage = 9.60;
// правильно: не можемо змінювати minWage за допомогою pc;
pc = &minWage;
double dval = 3.14; // правильно: не можемо змінювати minWage за допомогою pc, хоча dval і не константа;
pc = &dval; // правильно;
dval = 3.14159; //правильно;
*pc = 3.14159. // помилка.
Існують і константні покажчики
int errNumb = 0;
int *const currErr = &errNumb.

У цьому випадку curErr — константний покажчик на неконстантний об’єкт. Це значить, що ми не можемо присвоїти йому адресу іншого об’єкта, хоча сам об’єкт допускає модифікацію. Спроба присвоїти значення константному покажчику викличе помилку компіляції:
curErr = &myErNumb; // помилка.

## 6. Арифметичні і логічні операції. Оператор присвоювання

Програма оперує з даними. Числа можна додавати, віднімати, множити, ділити. Знаки можна порівнювати і т.д. Тобто з різних величин можна складати вирази, результат обчислення яких - нова величина.

Наведемо приклади виразів:

```
X * 12 + Y // значення X помножити на 12 і до
           // результату додати значення Y
val <3 // порівняти значення val з 3
 -9 // константний вираз -9
```

Вираз, після якого стоїть крапка з комою - це оператор-вираз. Його зміст полягає в тому, що комп'ютер повинен виконати всі дії, записані в даному виразі, інакше кажучи, обчислити вираз.

```
x + y - 12;  // скласти значення x і y і потім
           // відняти 12
a = b + 1;  // додати одиницю до значення b і
           // запам'ятати результат у змінній
```

**Оператор присвоювання**

Присвоєння - це теж операція, вона є частиною виразу. Значення правого операнда присвоюється лівому операнду.

```
x = 2;  // змінної x привласнити значен
cond = x <2;  // ня 2, змінної cond
            // привласнити значення true,
            // якщо x менше 2, в іншому
            // разі привласнити значення
3 = 5;  // false помилка, число 3
            // не здатне змінювати своє
            // значення
```

**Оператори**

Запис дій, які повинен виконати комп'ютер, складається з операторів. При виконанні програми оператори виконуються один за іншим, якщо тільки оператор не є оператором управління, який може змінити послідовне виконання програми.

Розрізняють оператори оголошення імен, оператори управління і оператори-вирази.

Вираз, після якого стоїть крапка з комою, - це оператор-вираз. Його зміст полягає в тому, що комп'ютер повинен виконати всі дії, записані в даному виразі, інакше кажучи, обчислити вираз. Найчастіше в операторі-вираженні коштує операція присвоювання або виклик функції. Оператори виконуються послідовно, і всі зміни значень змінних, зроблені в попередньому операторі, використовуються в наступних:

```
a = 1;
b = 3;
m = max (a, b);
```

Змінної a присвоюється значення 1, змінної b - значення 3. Потім викликається функція max з параметрами 1 і 3, і її результат присвоюється змінної m.

Як ми вже відзначали, присвоювання - необов'язкова операція в операторі-вираженні. Наступні оператори теж цілком коректні:

```
x + y - 12;  // скласти значення x і y і
              // потім відняти 12
func (d, 12, x) // викликати функцію func з
              // заданими параметрами
```

**Оголошення імен**

Ці оператори оголошують імена, тобто роблять їх відомими програмі. Всі ідентифікатори або імена, які використовуються в програмі на мові Сі ++, повинні бути оголошені.

Оператор оголошення складається з назви і типу:
int x;  // оголосити цілу змінну x
double f;  // оголосити змінну f типу
// double
const float pi = 3.1415;
// оголосити константу pi типу float
// зі значенням 3.1415
Оператор оголошення закінчується крапкою з комою.

Над об'єктами в мові С++ можуть виконуватися різні операції:

- операції присвоювання;
- операції відношення;
- арифметичні;
- логічні;
- зсувні операції.

Операції можуть бути бінарними або унарними. Бінарні операції виконуються над двома об'єктами, унарні – над одним.

Основні операції відношення:

- == еквівалентно – перевірка на рівність;
- != не дорівнює – перевірка на нерівність;
- < менше;
- > більше;
- <= менше або дорівнює;
- >= більше або дорівнює.

Результатом цих операцій є 1 біт, значення якого дорівнює 1, якщо результат виконання операції – істина, і дорівнює 0, якщо результат виконання операції – хибність.

Основні бінарні операції, розташовані в порядку зменшення пріоритету:

- * – множення;
- / – ділення;
- + – додавання;
- - – віднімання;
- % – залишок від цілочисельного ділення.

Основні унарні операції:

- ++ – ікрементування (збільшення на 1);
- -- – декрементування (зменшення на 1);
- - – зміна знака.

Якщо операція ++ або -- записана <u>до</u> імені змінної, то спочатку відбувається зміна значення змінної на 1, а потім це значення використовується для виконання наступних операцій.

Якщо операція ++ або -- розташована <u>після</u> змінної, то спочатку виконується операція, а потім значення змінної змінюється на 1.

Приклад.
c = a * ++ b; // a = 8,в=2      c=24,в=3
c = a * b ++; // a = 8, в=2     c=17

Бінарні арифметичні операції можуть бути об'єднані з операцією присвоювання:

- об'єкт * = вираз; // об'єкт = об'єкт * вираз
- об'єкт / = вираз; // об'єкт = об'єкт / вираз
- об'єкт + = вираз; // об'єкт = об'єкт + вираз
- об'єкт - = вираз; // об'єкт = об'єкт - вираз
- об'єкт% = вираз; // об'єкт = об'єкт % вираз.

Приклад.
x*=a; // x=x*a;

Логічні операції діляться на дві групи:

- умовні;

- побітові.

Умовні логічні операції найчастіше використовуються в операціях перевірки умови if і можуть виконуватися над будь-якими об'єктами.

Результат умовної логічної операції:

- 1 якщо вираз - істина;
- 0 якщо вираз - помилка.

Взагалі, всі значення, відмінні від нуля, інтерпретуються умовними логічними операціями як істина.

Основні умовні логічні операції:

- && – І логічне (бінарна) ;
- || – АБО логічне (бінарна);
- ! – НІ (унарна) логічне заперечення.

Побітові_логічні_операції оперують з бітами, кожен з яких може приймати тільки два значення: 0 або 1.

Основні побітові логічні операції в мові С:

- & кон'юнкція (логічне І) – бінарна операція, результат якої дорівнює 1 тільки коли обидва операнди дорівнюють одиниці;
- | диз'юнкція (логічне АБО) – бінарна операція, результат якої дорівнює 1, коли хоча б один з операндів дорівнює 1;
- ~ інверсія (логічне НЕ) – унарна операція, результат якої дорівнює 0 якщо операнд дорівнює 1, і дорівнює 1, якщо операнд нульовий;
- ^ виключення АБО – бінарна операція, результат якої дорівнює 1, якщо тільки один з двох операндів дорівнює 1 (в загальному випадку якщо у вхідному наборі операндів непарне число одиниць).

Результати виконання логічних операцій для кожного біта

| a | b | a & b | a \| b | ~a | a ^ b |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

Наприклад:

```
unsigned char a = 14; // a = 0000 1110
unsigned char b = 9; // b = 0000 1001
unsigned char c, d, e, f;
c = a & b; // c = 8 = 0000 1000
d = a | b; // d = 15 = 0000 1111
e = ~a; // e = 241 = 1111 0001
f = a ^ b; // f = 7 = 0000 0111
```

unsigned char - беззнаковий тип даних, що займає 1 байт. Те ж, що і byte. Діапазон від 0 до 255.

**Операції арифметичного зсуву** застосовуються в цілочисельній арифметиці і позначаються як:

- >> – зсув вправо;
- << – зсув вліво.

Загальний синтаксис здійснення операції зсуву:

об'єкт = вираз зсув Кількість Розрядів;
Наприклад:

```
unsigned char a=6; // a = 0000 0110
unsigned char b; b = a >> 1; // b = 0000 0110 >> 1 = 0000 0011 = 3.
```

Арифметичний зсув цілого числа вправо >> на 1 розряд відповідає поділу числа на 2.
Арифметичний зсув цілого числа вліво << на 1 розряд відповідає множенню числа на .

## 7. Поняття алгоритму. Класифікація алгоритмів

Процес створення програмного продукту можна розбити на наступні етапи:

- постановка задачі;
- формалізація (математична постановка завдання);
- алгоритмізація завдання;
- програмування завдання;
- відлагодження програми;
- рішення задачі на комп'ютері і аналіз результатів

Алгоритм - це послідовність арифметичних і логічних дій над змінними різних типів, які призводять до однозначного рішення задачі при зміні вхідних даних в досить широких межах.

**Способи опису алгоритмів:**

- мовний опис алгоритму;
- опис алгоритму у вигляді формул;
- таблична форма опису алгоритму, на псевдокоді (рідко);
- опис алгоритму у вигляді блок-схеми;
- Опис алгоритму на алгоритмічній мові (програмування).

Правила оформлення схем алгоритмів регламентуються ДСТУ 2938-94, графічні символи регламентуються ДСТУ 2940-94, ДСТУ 2941-94.

Блок-схема - опис структури алгоритму за допомогою геометричних фігур з лініями-зв'язками, які показують порядок виконання окремих інструкцій.

| Геометричне подання | Призначення оператора | Геометричне подання | Призначення оператора |
|---|---|---|---|
| | Процес (арифметичний оператор) | | Розв'язок (умовний оператор) |
| | Дані (оператори введення-виведення) | | Початок-кінець (оператор зупинки або початок блок-схеми) |
| | Документ (друк на бумагу) | | Монітор (виведення на монітор) |
| | Сортування | | Збереження даних |
| | Пам'ять з послідовним доступом | | Пам'ять з прямим доступом |

**Приклад** опису алгоритму на псевдокоді.
Початок

- Введення чисел: Z, X
- Якщо Z> X то Висновок Z
- Інакше висновок X

Кінець

**Приклад** табличного опису алгоритму. Обчислити прощу круга з радіусом R

| R, см | 3,14*R, см | 3,14*R*R, $см^2$ |
|---|---|---|
| | 6,28 | 12,56 |

За характером зв'язків між символами розрізняють алгоритми лінійної, розгалужується і циклічної структури.

**Лінійний алгоритм** - це алгоритм, в якому операції виконуються послідовно.

**Алгоритм з розгалудженням** - це алгоритм, в якому послідовність виконання операцій залежить від певних умов.

**Циклічний алгоритм** - це алгоритм, в якому багато разів виконуються одні й ті ж дії, наприклад з метою багаторазового виконання обчислень по одним і тим же залежностей при різних значеннях вхідних в них змінних. Багаторазово повторювані частина циклічного алгоритму називається тілом циклу.

Змінні, які змінюють своє значення при кожному виході на повторення тіла циклу, називаються параметрами циклу. Параметр циклу, який зберігається в одній і тій же комірці пам'яті, називається простою змінною.

Параметр циклу, який є елементом масиву, називається індексною змінною.

При використанні простої змінної вона є параметром циклу.

При використанні змінної з індексом параметром циклу є її індекс.

В одному циклі може бути кілька параметрів.

Розрізняють цикли: арифметичні або регулярні; ітераційні; вкладені.

Регулярні цикли (з відомим числом повторень) закінчуються за умови досягнення параметром циклу свого кінцевого значення. Регулярні цикли називають ще циклами з лічильниками.

Ітераційні цикли (з невідомим числом повторень) закінчуються по досягненню певного проміжного або кінцевого результату.

## 8. Умовні оператори та оператор switch-case

**Оператори управління**

Оператори управління визначають, в якій послідовності виконується програма. Якби їх не було, оператори програми завжди виконувалися б послідовно, в тому порядку, в якому вони записані.

**Умовні оператори**

Умовні оператори дозволяють вибрати один з варіантів виконання дій в залежності від будь-яких умов. Умова - це логічний вираз, тобто вираз, результатом якого є логічне значення true (істина) або false (брехня).

Оператор if вибирає один з двох варіантів обчислення.

```
if (умова)
    оператор1
else
    оператор2
```

Якщо умова істинна, виконується оператор1, якщо помилково, то виконується оператор2.

```
if (x> y)
    a = x;
else
    a = y;
```

В даному прикладі змінної a присвоюється значення максимуму з двох величин x і y.

Конструкція else необов'язкова. Можна записати:

```
if (x <0)
    x = -x;
```

```
abs = x;
```

В даному прикладі оператор x = -x; виконується тільки в тому випадку, якщо значення змінної x було негативним. Присвоєння змінної abs виконується в будь-якому випадку. Таким чином, наведений фрагмент програми змінить значення змінної x на його абсолютне значення і привласнить змінною abs нове значення x.

Якщо в разі істинності умови необхідно виконати декілька операторів, їх можна зробити висновок в фігурні дужки:

```
if (x <0) {
    x = -x;
    cout << "Змінити значення x на
       протилежне за знаком ";
 }
abs = x
```

Тепер якщо x негативно, то не тільки його значення зміниться на протилежне, але і буде виведено відповідне повідомлення. Фактично, укладаючи кілька операторів в фігурні дужки, ми зробили з них один складний оператор або блок. Прийом укладення кількох операторів в блок працює всюди, де потрібно помістити кілька операторів замість одного.

Умовний оператор можна розширити для перевірки декількох умов:

```
if (x <0)
    cout << "Негативна величина";
else if (x> 0)
    cout << "Позитивна величина";
else
    cout << "Нуль";
```

Конструкцій else if може бути кілька. Хоча будь-які комбінації умов можна виразити за допомогою оператора if, досить часто запис стає незручною і заплутаною.

Оператор вибору switch використовується, коли для кожного з декількох можливих значень виразу потрібно виконати певні дії. Наприклад, припустимо, що в змінної code зберігається ціле число від 0 до 2, і нам потрібно виконати різні дії в залежності від її значення:

```
switch (code) {
case 0:
    cout << "код нуль";
    x = x + 1;
    break;
case 1:
    cout << "код один";
    y = y + 1;
    break;
case 2:
    cout << "код два";
    z = z + 1;
    break;
default:
    cout << "необроблюваних значення";
 }
```

Залежно від значення code управління передається на одну з міток case. Виконання оператора закінчується після досягнення якого оператора break, або кінця оператора switch. Таким чином, якщо code дорівнює 1, виводиться "код один", а потім змінна y збільшується на одиницю. Якби після цього не стояв оператор break, то управління "провалилося" б далі, була б виведена фраза "код два", і змінна z теж збільшилася б на одиницю.

Якщо значення перемикача не збігається ні з одним зі значень міток case, то виконуються оператори, записані після мітки default. Мітка default може бути опущена, що еквівалентно записи:

```
default:
   ;  // порожній оператор, який не виконує
      // ніяких дій
```

Очевидно, що наведений приклад можна переписати за допомогою оператора if:

```
if (code == 0) {
     cout << "код нуль";
     x = x + 1;
                    }
else if (code == 1) {
     cout << "код один";
     y = y + 1;
                              } else if (code == 2) { cout << "код два";
     z = z + 1; }
else { cout << "необроблюваних значення";
       }
```

Мабуть, запис за допомогою оператора перемикання switch більш наочна. Особливо часто перемикач використовується, коли значення виразу має тип набору.

**9. Введення/виведення та їх організація. Консольний та файловий вивід**

У програмі C++ (і C ), як і в програмах написаних на інших мовах програмування особливе і важливе місце займають оператори введення / виведення.

Стандартна мова C містить цілу групу операторів введення і операторів виведення в стандартний потік, в файли і рядки. Ці функції дозволяють вводити і виводити дані згідно з форматом, вказаним програмістом.

В об'єктно-орієнтованій мові C ++ включено ціле сімейство класів, яке дозволяє в програмі створювати об'єкти, що виконують операції введення / виведення.

Найпростіші оператори введення з клавіатури і виведення на екран - це об'єкти потокового введення / виводу cin і cout.

Ці об'єкти готові до використання відразу після підключення відповідного заголовку iostream.

Для використання цих операторів необхідно підключити заголовний файл:

```
# include <iostream>
```

і надалі перераховувати змінні в потоці введення або виведення відповідно.

При введенні-виведенні потоку всі дані розглядаються як потік окремих байтів.

Для користувача потік — це файл на диску або фізичний пристрій, наприклад, дисплей чи клавіатура, або пристрій для друку, з якого чи на який направляється потік даних. Операції введення-виведення для потоку дозволяють обробляти дані різних розмірів і форматів від одиночного символу до великих структур даних. Програміст може використовувати функції бібліотеки, розробляти власні і включати їх у бібліотеку. Для доступу до бібліотеки цих класів треба включити в програму відповідні заголовні файли.

Бібліотеки мови C++ підтримує три рівня введення-виведення даних:

- введення-виведення потоку;
- введення-виведення нижнього рівня;
- введення-виведення для консолі і порту.

У мові C++ існує декілька бібліотек, які містять засоби введення-виведення, наприклад: **stdio, iostream.**

Найчастіше застосовують потокове введення-виведення даних, операції якого включені до складу класів **istream** або **iostream**. Доступ до бібліотеки цих класів здійснюється за допомогою використання у програмі директиви компілятора

**#include <iostream>**.

Для потокового введення даних вказується операція «>>» («читати з»). Це перевантажена операція, визначена для всіх простих типів даних і покажчика на char. Стандартним потоком введення є cin.

Формат запису операції введення має вигляд:

cin [>> values]; де values — змінна.

Так, для введення значень змінних **x і y** можна записати:

**cin >> x >> y;**

Кожна операція «>>» передбачає введення одного значення. При такому введенні даних необхідно дотримуватись конкретних вимог:

для послідовного введення декількох чисел їх слід розділяти символом пропуску («  ») або Enter (дані типу char розділяти пропуском необов’язково);

якщо послідовно вводиться символ і число (або навпаки), пропуск треба записувати тільки в тому випадку, коли символ (типу char) є цифрою;

потік введення ігнорує пропуски;

для введення великої кількості даних одним оператором їх можна розташовувати в декількох рядках (використовуючи Enter);

операція введення з потоку припиняє свою роботу тоді, коли всі включені до нього змінні одержують значення. Наприклад, для операції введення x і y, що вказана вище, можна ввести значення x та y таким чином:

2.34  789

або  2.345

789.

**Для потокового виведення даних** необхідна операція **«<<»** («записати в»), що використовується разом з ім’ям вихідного потоку **cout**.

Наприклад, вираз **cout << x;** означає виведення значення змінної **x** (або запис у потік). Ця операція виконує необхідну функцію перетворення даних у потік байтів.

Формат запису операції виведення представляється як:

**cout << data [<< data1];**,де **data, data1** — це змінні, константи, вирази тощо.

Потокова операція виведення може мати вигляд: **cout << “y =” << x + a – sin(x) << “\n“;.**

Символ переведення на наступний рядок записується як рядкова константа, тобто **“\n”**,

Система введення / виведення С ++ включає функції для зміни параметрів потоку (функції форматування).

Це спеціальні функції, так звані маніпулятори (manipulators), які можуть включатися в вирази введення/виведення.

Маніпуляторами називають спеціальні функції, що дозволяють змінювати стан потоку і що використовуються спільно з операціями витягання і вставки в одному операторові введення або виведення даних. Відмінність маніпуляторів від звичайних функцій полягає в тому, що їх імена можна використати в якості правого операнда при виконанні форматованого обміну за допомогою операцій << і >>. В якості найлівішого операнда вираження з маніпуляторами і операторами обміну завжди використовується ім'я потоку.

Стандартні маніпулятори введення-виводу діляться на дві групи: маніпулятори з параметрами і маніпулятори без параметрів. Маніпулятори з параметрами оголошені у файлі iomanip.h, а без параметрів — у файлі iostream.h. У таблиці приведені стандартні маніпулятори без параметрів.

| Маніпулятор | Дія в потоці |
|---|---|
| *dec* | Перетворення в десяткове представлення |
| *hex* | Перетворення в шістнадцятиричне представлення |
| *oct* | Перетворення у вісімкове представлення |
| *endl* | Вставка символу нового рядка і выгружение буфера |
| *ends* | Вставка в потік нульової ознаки кінця рядка |
| *flush* | Выгружение буфера вихідного потоку |
| *ws* | Витягання і ігнорування пробільних символів * |
| *showbase* | Вставка ознаки системи числення * |
| *noshowbase* | Вилучення ознаки системи числення * |
| *skipws* | Пропуск пробільних символів при введенні * |
| *noskipws* | Відміна пропуску пробільних символів при введенні * |
| *uppercase* | Використання символів верхнього регістра при виведенні чисел * |
| *Nouppercase* | Відміна використання символів верхнього регістра при виведенні чисел * |
| *internal* | Вставка заповнювачів між знаком і модулем числа, що виводиться * |
| *left* - | Вставка заповнювачів після значення в полі виводу * |
| *right* | Вставка заповнювачів перед значенням в полі виводу * |
| *fixed* | Використання для дійсних чисел формату *dddd.dd* * |
| *scientific* | Використання для дійсних чисел формату *i.dddd e dd* * |
| *boolalpha* | Виведення даних типу *bool* в символічному виді * |
| *noboolalfa* | Виведення даних типу *bool* як цілих чисел * |

Зірочкою відмічені маніпулятори, відсутні в ранніх версіях C++. Відмітимо, що маніпулятори без параметрів управляють прапорами форматування потоку і буфером потоку.

Приклад.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
  int n; char s [8];
  cout << "vvedite n:";
  cin >> n;
  cout << "znachenie n ravno: " << n << endl;
  cout << setprecision (2) << 1000.243 << endl;
  cout << "vvedite stroky" << endl;
  cin >> ws >> s;
cout << "stroka:";
cout <<" "<< s;
return 0;
}
```

C ++ підтримує весь набір функцій введення / виводу файлової системи ANSI C, яка є стандартом для C та C++

Отже, якщо переносити код C на C ++, то не буде потрібно ніяких змін в процедурах введення / виведення

C ++ також визначає свою власну об'єктно-орієнтовану систему введення / виведення, що містить як функції, так і оператори, повністю дублюючі можливості системи введення / виведення ANSI C

При створенні коду C ++ слід використовувати систему введення / виведення C ++. Навіть тим, хто пише в основному програми на C ++, все одно слід познайомитися з системою введення / виводу ANSI C з декількох причин:

- Існує неписане правило, що будь-який програміст на C ++ повинен бути також програмістом на С. Професійні горизонти того, хто не знає, як використовувати систему введення / виведення С, будуть обмежені.
- Протягом декількох років С і С ++ будуть співіснувати разом . Багато програмістів будуть створювати гібриди програм на С і С ++. Наприклад, для заміни функцій введення / виведення C на функції C ++, слід знати, як діють обидві системи введення / виведення.
- Розуміння основних принципів, покладених в основу системи введення / виведення ANSI C, допоможе зрозуміти об'єктно-орієнтовану систему введення / виведення C ++. (Вони обидві використовують загальні концепції.)
- У деяких ситуаціях може бути легше використовувати НЕОБ'ЄКТНо-орієнтований підхід С до введення / виведення, ніж підходу, запропонованого С ++.

**Система введення / виведення С проти С ++**

Стандартна бібліотека С містить дві функції, які виконують форматоване введення і виведення стандартних типів даних: printf () і scanf ().

Термін форматований означає, що ці
функції можуть, читати або писати дані в різних форматах, якими можна управляти.

Функція printf () використовується для виведення даних на консоль, scanf () - для читання даних з консолі.

Як printf(), так і scanf() можуть працювати з будь-якими стандартними типами, включаючи символи, рядки і числа.

**printf ()**

Функція printf () має наступний формат:

printf (форматная_строка, ...);

Перший аргумент форматная_строка визначає спосіб виведення наступних аргументів. Він часто називається форматним рядком і містить два типи елементів: символи, що виводяться на екран, і специфікатор формату, що визначають спосіб виведення аргументів, які перелічені за форматним рядком.

Специфікатори формату починаються зі знака відсоток, за яким записується код формату.

Специфікатори формату наведені в таблиці.

| Назва | Пояснення |
|---|---|
| %c | символ |
| %d | ціле десяткове число |
| %i | ціле десяткове число |
| %e | десяткове число у вигляді x.xx e+xx |
| %E | десяткове число у вигляді x.xx E+xx |
| %f | десяткове число з плаваючею комою xx.xxxx |
| %F | десяткове число з плаваючею комою xx.xxxx |
| %g | %f або %e, що коротше |
| %G | %F або %E, що коротше |
| %o | вісімкове число |
| %s | строчка символів |
| %u | беззнакове десяткове число |

У специфікаторі формату, після символу% може бути вказана точність (число цифр після коми).

Точність задається наступним чином:

% .n<код формата> .

де n – число цифр після коми, а <код формата> – один з кодів наведених вище.

Наприклад, якщо у нас є змінна x = 10.3563 типу float і ми хочемо вивести її значення з точністю до 3-х цифр після коми, то ми повинні написати: printf ( "Змінна x =% .3f", x);

Результат: Змінна x = 10.356

Ви також можете вказати мінімальну ширину поля відведеного для друку.

Якщо рядок або число більше зазначеної ширини поля, то рядок або число друкується повністю.

Наприклад, якщо ви напишите: printf ( "% 5d", 20);

- результат буде

20

Зверніть увагу на те, що число 20 друкується не з самого початку рядка.

Якщо ви хочете щоб невикористані місця поля заповнювалися нулями, то потрібно поставити перед шириною поля символ 0.

Наприклад: printf ( "% 05d", 20);

Результат буде 00020.

Крім специфікаторів формату даних в керуючому рядку можуть перебувати керуючі символи

| Назва | Пояснення |
|---|---|
| \b | Пробіл |
| \f | Нова сторінка, переклад сторінки |
| \n | Новий рядок, новий рядок |
| \r | Повернення каретки |
| \t | Горизонтальна табуляція |
| \v | Вертикальна табуляція |
| \" | Лапки |
| \' | Апостроф |
| \\ | Зворотна коса риска |
| \0 | Нульовий символ, нульовий байт |

Функція scanf () – функція форматованого введення.

З її допомогою ви можете вводити дані із стандартного пристрою введення (клавіатури).

Даними, що вводяться, можуть бути цілі числа, числа з плаваючою комою, символи, рядки і покажчики.

Для використання функції scanf () необхідно підключити файл stdio.h:

Формат функції

scanf (керуючий рядок);

Функція повертає число змінних, яким було присвоєно значення.

Керуючий рядок містить три види символів: специфікатор формату, пробіли та інші символи.

При введенні рядка за допомогою функції scanf() (специфікатор формату% s), рядок вводитися до першого пробілу:

```
char str [80]; // масив на 80 символів
scanf ( "% s", str);
```

**Функції getchar, putchar, gets, puts**

Найпростішою з стандартних функцій вводу / виводу на консоль є getchar (), що читає символ з клавіатури, і putchar () друкуюча символ на екран в поточній позиції курсору.

Функції gets () і puts () дозволяють читати і виводити рядки на консоль.

Якщо ви хочете вводити рядки з пробілами, то потрібно скористатися функцією gets ();

За допомогою функції gets() ви зможете вводити повноцінні рядки.

Функція gets() читає символи з клавіатури до появи символу нового рядка (\ n).

Сам символ нового рядка з'являється, коли ви натискаєте кнопку enter.

**Приклад**

```
#include <stdio.h>
void main (void)
{   float x, y, z;
   x = 10.5;   y = 130.67;   z = 54;
   printf ( "Координати об'єкта: x:%. 2f, y:%. 2f, z:%. 2f", x, y, z);}
```

Результат роботи програми:

Координати об'єкта: x: 10.50, y: 130.67, z: 54.00

## 10. Поняття циклу. Оператори циклу. Цикл while, do while

Циклічний алгоритм - це алгоритм, в якому багато разів виконуються одні й ті ж дії, наприклад з метою багаторазового виконання обчислень по одним і тим же залежностей при різних значеннях змінних.

Багаторазово повторювані частина циклічного алгоритму називається тілом циклу.

Змінні, які змінюють своє значення при кожному виході на повторення тіла циклу, називаються параметрами циклу.

Параметр циклу, який зберігається в одній і тій же комірці пам'яті, називається простою змінною.

Параметр циклу, який є елементом масиву, називається індексною змінною.

При використанні простої змінної вона є параметром циклу. При використанні змінної з індексом параметром циклу є її індекс. В одному циклі може бути кілька параметрів.

Розрізняють цикли:

- арифметичні або регулярні;
- ітераційні;
- вкладені.

Регулярні цикли (з відомим числом повторень) закінчуються за умови досягнення параметром циклу свого кінцевого значення. Регулярні цикли називають ще циклами з лічильниками.

Ітераційні цикли (з невідомим числом повторень) закінчуються по досягненню певного проміжного або кінцевого результату.

**Цикл while (цикл с передумовою)**

Оператор циклу while або цикл while — це цикл, що повторює одну й ту ж дію, поки умова продовження циклу while залишається істинною.

```
// форма запису цикла while
while (/*умова продовження циклу while*/)
{
/*блок операторів*/;
/*керування умовою*/;
}
```

**Приклад.** Розглянемо застосування циклу while на прикладі руху автомобіля. Поки швидкість руху автомобіля менше 60 км/год, продовжуватиме нарощувати швидкість.

```
#include <iostream>
using namespace std;
int main()
{
   int speed = 5, count = 1;
   while (speed < 60)
   {
      speed += 10; // збільшення швидкості
      cout << count <<"-speed = " << speed << endl;
      count++; // подсчёт повторений цикла
```

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

Результат роботи програми

```
1-speed = 15
2-speed = 25
3-speed = 35
4-speed = 45
5-speed = 55
6-speed = 65
Для продолжения нажмите любую клавишу . . .
```

**Цикл do while (цикл з постумовою)**

Форма запису оператора циклу do while:

```
do // початок циклу do while
{
/*блок операторів*/;
}
while (/*умова виконання тіла циклу*/); // кінець циклу do while
```

Цикл do while відрізняється від циклу while тим, що в do while спочатку виконується тіло циклу, а потім перевіряється умова продовження циклу. Через таку особливість do while називають циклом з постумовою. Таким чином, якщо умова do while свідомо хибна, то хоча б один раз блок операторів у тілі циклу do while виконається.

**Приклад.** Обчислити суму чисел в заданому інтервалі.

Інтервал вказаний від -6 до 10 включно,[-6; 10]. Програма підсумовує всі цілі числа із заданого інтервалу.

Сума формується так: -6 -5 -4 -3 -2 -1 + 0 +1 +2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 34.

Таким чином, циклічно виконується дія підсумовування всіх цілих чисел, із зазначеного користувачем інтервалу.

```
#include "stdafx.h"
#include <iostream>
using namespace std;

int main()
{
    cout << "Enter the first limit: "; // начальное значение из интервала
    int first_limit;
    cin >> first_limit;
    cout << "Enter the second limit: "; // конечное значение из интервала
    int second_limit;
    cin >> second_limit;
    int sum = 0, count = first_limit;
    do
    {
     sum += count; // наращивание суммы
     count++; // инкремент начального значения из задаваемого интервала
    } while (count <= second_limit); // конец цикла do while
    cout << "sum = " << sum << endl; // печать суммы
    system("pause");
    return 0;
}
```

## 11. Масиви

Масив - це структура даних, представлена у вигляді групи комірок одного типу, об'єднаних під одним єдиним ім'ям. Масиви використовуються для обробки великої кількості однотипних даних. Ім'я масиву є покажчиком. Окремий осередок даних масиву називається елементом масиву. Елементи масиву можуть бути будь-якого типу.

Масиви можуть мати як одне, так і більше вимірювань.

Залежно від кількості вимірювань масиви поділяються на одномірні масиви, двомірні масиви, тримірні масиви і так далі до n-мірного масиву.

Найчастіше в програмуванні використовуються одномірні й двомірні масиви.

Приклад одномірного масиву з елементами типу int.

Одномірний масив – це масив з одним параметром, який характеризує кількість елементів масиву. Фактично одномірний масив – це масив, у якого може бути тільки один рядок, і n-а кількість стовпців. Стовпці в одновимірному масиві – це елементи масиву. На рисунку показана структура цілочисельного одномірного масиву **a**. Розмір цього масиву – 16.

Максимальний індекс одномірного масиву **a** дорівнює 15, але розмір масиву 16 (елементів або комірок, тому що нумерація осередків масиву завжди починається з **0**.

Індекс елемента або комірки – це ціле невід'ємне число, за яким можна звертатися до кожного елемента масиву і виконувати над ним будь-які дії.

Індекс елемента масиву – це його адреса в масиві.

Приклад. Масив з іменем А складається з 10 цілих чисел. Записати число 5 в перший і останній елементи масиву.

```
// опис масиву А
 int A[10];
A[0] = 5; // перший елемент масиву
A[9] = 5; // останній елемент масиву
```

Синтаксис оголошення одномірного масиву в С ++:

/ * Тип даних * / / * ім'я одномірного масиву * / [/ * розмірність одномірного масиву * /];

**Приклади** оголошення одномірного масиву:

**1)** int a [16];

де, int – цілочисельний тип даних; а – ім'я одновимірного масиву; 16 – розмір одновимірного масиву, 16 комірок.

**2)** int mas [10], a [16];

Оголошено два одномірних масива **mas** і **a** розмірами 10 і 16 відповідно. При такому способі оголошення всі масиви матимуть однаковий тип даних, в нашому випадку – int.

**3)** Масиви можуть бути ініційовані при оголошенні:

int a [16] = {5, -12, -12, 9, 10, 0, -9, -12, -1, 23, 65, 64, 11, 43, 39, -15};

– ініціалізація одномірного масиву

Ініціалізація одномірного масиву виконується в фігурних дужках після знаку = , кожен елемент масиву відокремлюється від попереднього комою.

**4)** int a[] = {5, -12, -12,9,10,0, -9, -12, -1,23,65,64,11,43,39, -15}; –

ініціалізації масиву без визначення його розміру.

В даному випадку компілятор сам визначить розмір одномірного масиву. Розмір масиву можна не вказувати тільки при його ініціалізації, при звичайному оголошенні масиву обов'язково потрібно вказувати розмір масиву.

**Приклад.** Програма виводу елементів одномірного масиву на екран

```
#include <iostream>
using namespace std;
int main ()
{
cout << "obrabotka massiva" << endl;
int array1 [16] = {5,  -12,  -12, 9, 10, 0,  -9,  -12,  -1, 23, 65, 64, 11, 43, 39,  -15};
// оголошення та ініціалізація одновимірного масиву
cout<<"  indeks"  <<"\   t "  <<" element  massiva"<<endl;
//  друк заголовків
for (int counter = 0; counter <16; counter ++) // початок циклу
{// вивід на екран індексу осередку масиву, а потім вмісту цього осередку, в нашому випадку - це ціле число
cout<<"array1 [" << counter << "]" << "\  t     " <<array1 [counter]<<endl;   }
system ("pause");
return 0;}
```

Результат роботи програми: вивід елементів масиву на екран.

**Одномірні масиви**

Ви вже знаєте, що змінна - це комірка в пам'яті комп'ютера, де може зберігатися одне єдине значення.

Масив - це область пам'яті, де можуть послідовно зберігатися кілька значень.

Візьмемо групу студентів з десяти чоловік. У кожного з них є прізвище. Створювати окрему змінну для кожного студента - не раціонально.

Створимо масив, в якому будуть зберігатися прізвища всіх студентів.

std::string students[10] = {"Ivanov", "Petrov", "Sidorov", "Ahmetov", "Erohsin", "Uhin", "Avdeev", "Dizel", "Kartohin", "Hyb"};

```
#include <iostream>
#include <string>
int main()
{
std::string students[10] = {"Ivanov", "Petrov", "Sidorov", "Ahmetov", "Erohsin", "Uhin", "Avdeev", "Dizel", "Kartohin", "Hyb"};
std::cout << students << std::endl;
// Пытаемся вывести весь массив непосредственно
return 0;
}
```

Результат: 0x7e3d59b38910 (адреса комірки пам'яті, де починається запис масиву).

Ім'я масиву є покажчиком, тобто є адресою комірки пам'яті, в якій записується перший елемент масиву.

Справа в тому, що при створенні змінної, їй виділяється певне місце в пам'яті. Якщо ми оголошуємо змінну типу int, то на машинному рівні вона описується двома параметрами - її адресою та розміром збережених даних.

Масиви в пам'яті зберігаються таким же чином. Масив типу int з 10 елементів описується за допомогою адреси його першого елемента і кількості байт, яке може вмістити цей масив. Якщо для зберігання одного цілого числа виділяється 4 байта, то для масиву з десяти цілих чисел буде виділено 40 байт.

При кожному наступному запуску коду мі будемо отримувати різні адреси пам'яті. Цей факт відображає технологію “ рандомізації адресного простору ”, яка  застосовується в більшості операційних систем

**Приклад.** Виведення елементів масиву через цикл

```
#include <iostream>
#include <string>
int main()
{
```

```
std::string students[10] = {"Ivanov", "Petrov", "Sidorov", "Ahmetov", "Erohsin", "Uhin",
"Avdeev", "Dizel", "Kartohin", "Hyb"};
for (int i = 0; i < 10; i++)
{ std::cout << students[i] << std::endl; }
return 0;
}
```

**Двомірні масиви**

Двомірний масив – це звичайна таблиця розміром m на n де,

m – кількість рядків двовимірного масиву;

n – кількість стовпців двовимірного масиву;

m* n – кількість елементів масиву

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| A[0][0] | A[1][0] | A[2][0] | A[3][0] | | | | | A[n-1][0] |
| A[0][1] | A[1][1] | A[2][1] | | | | | | A[n-1][1] |
| A[0][2] | A[1][2] | A[2][2] | | | | | | A[n-1][2] |
| A[0][3] | A[1][3] | | | | | | | |
| A[0][4] | | | | | | | | |
| | | | | | | | | |
| A[0][m-1] | A[1][m-1] | A[2][m-1] | | | | | | A[n-1][m-1] |

**Синтаксис оголошення двомірного масиву** має такий вигляд:

/ * Тип даних * / / * ім'я масиву * / [/ * кількість рядків * /] [/ * кількість стовпців * /];

В оголошенні двомірного масиву, також як і в оголошенні одномірного масиву, в першу чергу, потрібно вказати:

- тип даних (тип елементів масиву, тип масиву);
- ім'я масиву.

Двомірні масиви з числовими елементами іноді називають **матрицями.**

**Приклад** оголошення двомірного масиву:

int a [5] [3];

a – ім'я цілочисельного масиву.

Число в перших квадратних дужках вказує кількість рядків двомірного масиву, в даному випадку їх 5.

Число в других квадратних дужках вказує кількість стовпців двомірного масиву, в даному випадку їх 3.

**Ініціалізація двомірного** масиву при оголошенні:

int a [5] [3] = {{4, 7, 8}, {9, 66, -1}, {5, -5, 0}, {3, -3, 30}, {1, 1, 1}};

В даному масиві 5 рядків, 3 стовпці.

**Адресою елементу (або комірки)** двомірного масиву є ім'я масиву, номер рядка та номер стовпчика.

**Приклад.** Масив задано за допомогою таблиці

| | | |
|---|---|---|
| 4 | 7 | 8 |
| 9 | 66 | -1 |
| 5 | -5 | 0 |
| 3 | -3 | 30 |
| 1 | 1 | 1 |

за адресою a[0][0] зберігається 4;

за адресою a[2][1] зберігається -5;

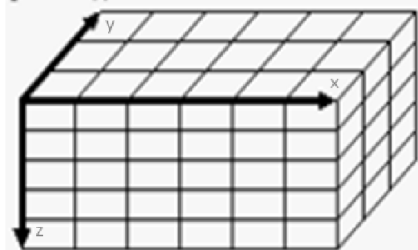
за адресою a[3][2] зберігається 30.

**Приклад.** Вивести на консоль двомірний масив по строках

```
#include <iostream>
int main()
{
int a[2][3] = {1,2,3,4,5,6};
std::cout<<a[0][0]<<' '<<a[0][1]<<' '<<a[0][2]<<std::endl;
std::cout<<a[1][0]<<' '<<a[1][1]<<' '<<a[1][2];
return 0;
}
```

**Трьохмірні масиви, багатомірні масиви**

Оголошення багатомірних масивів повністю співпадає з оголошенням одно- та двомірних масивів, тільки замість одного або двох розмірів вказується декілька. Тип даних, які зберігаються в багатомірних масивах може бути будь-яким.




На рисунку x – перший індекс, y – другий індекс, z – третій індекс масиву.

**Приклад** оголошення трьохмірного масиву.

```
const unsigned int DIM1 = 3;
const unsigned int DIM2 = 5;
const unsigned int DIM3 = 2;
int ary[DIM1][DIM2][DIM3];
```

В прикладі оголошується трьомірний масив, у якому зберігаються три матриці, кожна з яких складається з 5 строк по 2 значення типу int в кожній.

## 12. Покажчик. Адреса

Покажчик – це змінна, що містить адресу об'єкта. Покажчик не несе інформації про вміст об'єкту, а містить відомості про те, де розміщений об'єкт.

Пам'ять комп'ютера можна представити у вигляді послідовності пронумерованих однобайтових комірок, з якими можна працювати окремо або блоками.

Кожна змінна в пам'яті має свою адресу – номер першої комірки, де вона розташована, а також своє значення.

В свою чергу, покажчик – це теж змінна, яка розміщується в пам'яті. Вона теж має адресу, а її значення є адресою деякої іншої змінної.

Покажчики — це змінні, котрі містять адресу пам’яті, роз-поділеної для об’єкта відповідного типу. Усі змін-ні, розглянуті до цього, зберігали якісь значення (дані). Ці дані могли бути різних типів: символьного, цілого, дійсного тощо.

Покажчик, як і будь-яка змінна, повинен бути оголошений.

Загальна форма оголошення покажчика:

тип *ім’я;

де тип — тип значень, на який вказує покажчик (або тип змінної, адресу якої він містить);

ім’я — ім’я змінної-покажчика;

«*» — операція над типом, що читається «покажчик на тип».

Наприклад:

int *pn – покажчик на ціле значення;

float *pf1, *pf2; — два покажчики на дійсні значення.

Покажчики не прив’язують дані до якого-небудь визначе-ного імені змінної і можуть містити адреси будь-якого неімено-ваного значення.

Існує адресна константа NULL, що означає порожню адресу.

Для роботи з покажчиками в Сі++ визначені дві операції:

«&» — операція взяття адреси («адреса значення»);

«*» — операція розіменування («значення за адресою»).

Операція взяття адреси «&» застосовується разом зі змін-ною і повертає адресу цієї змінної. Операція розіменування «*» використовується разом з покажчиками і вилучає значення, на яке вказує змінна-покажчик, розташована безпосередньо після символа «*».

Наприклад:

char c; // змінна

char * p; // покажчик

p = & c; // p = адреса c.

Оголошення покажчиків можна здійснити одним з таких способів:

<тип> *ptr;

<тип> *ptr = <змінна-покажчик>;

<тип> *ptr = &<ім’я змінної>;.

Наприклад:

int *ptx, b; float y; — оголошені змінна-покажчик ptx та змінні b і y;

float *sp = &y; — покажчику sp присвоюється адреса змінної y;

float *p = sp; — покажчику p присвоюється значення (адреса зна-чення), яке міститься в змінній sp, тобто адреса змінної y.

При оголошенні покажчиків символ «*» може знаходитися перед ім’ям покажчика або відразу після оголошення типу по-кажчика і поширювати свою дію тільки на одну змінну-покаж-чик:

long *pt;   long*Uk;   int *ki, x, h; — оголошення описів.

За потреби для опису покажчика на комірку довільного типу замість ідентифікатора типу записується слово void, а саме:

void *p, *pt; — опис двох покажчиків на довільний тип даних.

Перед використанням покажчика у програмі його обо-в’язково необхідно ініціювати, іншими словами, необхідно присвоїти адресу якого-небудь даного, інакше можуть бути не-передбачені результати.

Розглянемо фрагмент програми:

int *p, *p1; //оголошені два покажчики на комірку пам’яті типу int;

int x = 12, y = 5, m[7]; //оголошені змінні x, y і масив m;

p = &y;      // p (&y); — покажчику p присвоєна адреса змінної y.

Якщо для цього фрагмента програми записати оператор ви-ведення у вигляді

cout << “Адреса p ” << p << “Значення за адресою = ” << *p;,

то виведеться адреса комірки пам’яті, де записана змінна y і зна-чення цієї змінної (тобто 5).

Використовуючи запис «x = *p;» , одержимо x = 5, тому що «*p = y = 5;».

### 13. Арифметичні і логічні операції

Над об'єктами в мові C++ можуть виконуватися різні операції:

- операції присвоювання;
- операції відношення;
- арифметичні;
- логічні;
- зсувні операції.

Операції можуть бути бінарними або унарними.

Бінарні операції виконуються над двома об'єктами, унарні – над одним.

Операція присвоювання позначається символом “=“ і виконується в 2 етапи:
- обчислюється вираз в правій частині;
- результат присвоюється операнду в лівій частині:

Формат: Об'єкт = вираз;

Приклад.

a=4;

b = a + 2; // змінній b присвоюється значення 6, обчислене в правій частині.

Приклад перетворення типів даних за допомогою операції присвоювання:

double res = (double)13 / 7;

Основні операції відношення:

== еквівалентно – перевірка на рівність;

!= не дорівнює – перевірка на нерівність;

< менше;

> більше;

<= менше або дорівнює;

>= більше або дорівнює.

Результатом цих операцій є 1 біт, значення якого дорівнює 1, якщо результат виконання операції – істина, і дорівнює 0, якщо результат виконання операції – хибність.

Основні бінарні операції, розташовані в порядку зменшення пріоритету:

* – множення;

/ – ділення;

+ – додавання;

- – віднімання;

% – залишок від цілочисельного ділення.

Основні унарні операції:

++ – ікрементування (збільшення на 1);

-- – декрементування (зменшення на 1);

- – зміна знаку.

Якщо операція ++ або -- записана до імені змінної, то спочатку відбувається зміна значення змінної на 1, а потім це значення використовується для виконання наступних операцій.

Якщо операція ++ або -- розташована після змінної, то спочатку виконується операція, а потім значення змінної змінюється на 1.

Приклад.

c = a * ++ b; // a = 8,в=2 c=24,в=3

c = a * b ++; // a = 8, в=2 c=17

Бінарні операції можуть бути записані в програмі за правилом скорочення.

Бінарні арифметичні операції можуть бути об'єднані з операцією присвоювання:
- об'єкт * = вираз; // об'єкт = об'єкт * вираз
- об'єкт / = вираз; // об'єкт = об'єкт / вираз

об'єкт + = вираз; // об'єкт = об'єкт + вираз

об'єкт - = вираз; // об'єкт = об'єкт - вираз

об'єкт% = вираз; // об'єкт = % вираз.

Приклад.

x*=a; // x=x*a;

Логічні операції діляться на дві групи:

умовні;

побітові.

Основні умовні логічні операції:

&& – І логічне (бінарна) ;

|| – АБО логічне (бінарна);

! – НІ (унарна) логічне заперечення.

Умовні логічні операції найчастіше використовуються в операціях перевірки умови if і можуть виконуватися над будь-якими об'єктами.

Результат умовної логічної операції:

1 якщо вираз - істина;

0 якщо вираз - помилка.

Взагалі, всі значення, відмінні від нуля, інтерпретуються умовними логічними операціями як істина.

Побітові логічні операції оперують з бітами, кожен з яких може приймати тільки два значення: 0 або 1.

Основні побітові логічні операції в мові Сі:

& кон'юнкція (логічне І) – бінарна операція, результат якої дорівнює 1 тільки коли обидва операнди дорівнюють одиниці;

| диз'юнкція (логічне АБО) – бінарна операція, результат якої дорівнює 1, коли хоча б один з операндів дорівнює 1;

~ інверсія (логічне НЕ) – унарна операція, результат якої дорівнює 0 якщо операнд дорівнює 1, і дорівнює 1, якщо операнд нульовий;

^ виключення АБО – бінарна операція, результат якої дорівнює 1, якщо тільки один з двох операндів дорівнює 1 (в загальному випадку якщо у вхідному наборі операндів непарне число одиниць).

Результати виконання логічних операцій для кожного біта:

| a | b | a & b | a \| b | ~a | a ^ b |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

Приклад:

```
unsigned char a = 14; // a = 0000 1110
unsigned char b = 9; // b = 0000 1001
unsigned char c, d, e, f;
c = a & b; // c = 8 = 0000 1000
d = a | b; // d = 15 = 0000 1111
e = ~a; // e = 241 = 1111 0001
f = a ^ b; // f = 7 = 0000 0111
```

unsigned char - беззнаковий тип даних, що займає 1 байт.

Діапазон від 0 до 255.

Операції арифметичного зсуву застосовуються в цілочисельній арифметиці і позначаються як:

1. >> – зсув вправо;
2. << – зсув вліво.

Загальний синтаксис здійснення операції зсуву:

об'єкт = вираз зсув Кількість Розрядів;

Наприклад:

```
unsigned char a=6; // a = 0000 0110
unsigned char b;
b = a >> 1; // b = 0000 0110 >> 1 = 0000 0011 = 3.
```

Арифметичний зсув цілого числа вправо >> на 1 розряд відповідає поділу числа на 2.

Арифметичний зсув цілого числа вліво << на 1 розряд відповідає множенню числа на

## 14. Оператор керування

До операторів керування відносять оператор безумовного переходу goto та оператори виходу з циклу.

Безумовна передача керування на мітку проводиться за допомогою оператора goto.

Синтаксис оператора безумовного перехода goto має вигляд:

goto мітка;

мітка: оператор;

де мітка — ім’я, після якого ставляться дві крапки, область дії мітки — функція, в якій вона визначена.

Мітка являє собою ідентифікатор з розташованим за ним символом двокрапки (:). Мітками позначають будь-який оператор, на який надалі повинен бути здійснен безумовний перехід.

Слід зауважити, що використання оператора безумовного переходу goto вважається негарним стилем програмування.

Мова С++ передбачає такі засоби виходу з циклу:

1. оператор **break** приводить до виходу з циклу (або з операторів if та switch), у якому він міститься, і переходу до наступного оператора програми
2. оператор **continue** здійснює пропуск усіх операторів, що залишилися до кінця циклу, та реалізує перехід до початку наступної ітерації;
3. функція exit() реалізує вихід з циклу з завершенням програми. Аргумент цієї функції — константа цілого типу, що дорівнює 0 у випадку сприятливого завершення циклу і відмінна від 0 - у протилежному випадку. При використанні цієї функції слід включити в код програми директиву #include <stdlib.h>.

## 15. Оператори циклу while і do while

Циклічний алгоритм - це алгоритм, в якому багато разів виконуються одні й ті ж дії, наприклад з метою багаторазового виконання обчислень по одним і тим же залежностей при різних значеннях змінних.

Багаторазово повторювані частина циклічного алгоритму називається тілом циклу.

Змінні, які змінюють своє значення при кожному виході на повторення тіла циклу, називаються параметрами циклу.

Параметр циклу, який зберігається в одній і тій же комірці пам'яті, називається простою змінною.

Параметр циклу, який є елементом масиву, називається індексною змінною.

При використанні простої змінної вона є параметром циклу.

При використанні змінної з індексом параметром циклу є її індекс.

В одному циклі може бути кілька параметрів.

Розрізняють цикли:

арифметичні або регулярні;

ітераційні;

вкладені.

Регулярні цикли (з відомим числом повторень) закінчуються за умови досягнення параметром циклу свого кінцевого значення. Регулярні цикли називають ще циклами з лічильниками.

Ітераційні цикли (з невідомим числом повторень) закінчуються по досягненню певного проміжного або кінцевого результату.

Оператор циклу while або цикл while — це цикл, що повторює одну й ту ж дію, поки умова продовження циклу while залишається істинною.

Форма запису цикла while

while (/*умова продовження циклу while*/)

{

```
/*блок операторів*/;
/*керування умовою*/;
};
```

**Приклад.** Розглянемо застосування циклу while на прикладі руху автомобіля. До того часу, поки швидкість руху автомобіля буде менше 60 км/год, продовжувати нарощувати швидкість на 10 км/год.

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

int main()
{
   int speed = 5, count = 1;
   while (speed < 60)
   {
      speed += 10; // збільшення швидкості
      cout << count <<"-speed = " << speed << endl;
      count=count+1; // подсчёт повторений цикла
   }
   system("pause");
   return 0;
}
```

Результат роботи програми:

```
1-speed = 15
2-speed = 25
3-speed = 35
4-speed = 45
5-speed = 55
6-speed = 65
Для продолжения нажмите любую клавишу . . .
```

Форма записи оператора цикла do while:

```
do // початок циклу do while
{
/*блок операторів*/;
}
while (/**/); // закінчення циклу do while
```

Цикл do while відрізняється від циклу while тим, що в do while спочатку виконується тіло циклу, а потім перевіряється умова продовження циклу. Через таку особливість do while називають циклом з постумовою. Таким чином, якщо умова do while свідомо хибна, то хоча б один раз блок операторів у тілі циклу do while виконається.

Приклад. Обчислити суму чисел в заданому інтервалі.

Інтервал вказаний від -6 до 10 включно,[-6; 10]. Програма підсумовує всі цілі числа із заданого інтервалу.

Сума формується так: -6 -5 -4 -3 -2 -1 + 0 +1 +2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 34.

Таким чином, циклічно виконується дія підсумовування всіх цілих чисел, із зазначеного користувачем інтервалу.

```
#include "stdafx.h"
#include <iostream>
using namespace std;

int main()
{
   cout << "Enter the first limit: "; // початкове значення з інтервалу
   int first_limit;
   cin >> first_limit;
```

```
    cout << "Enter the second limit: "; // кінцеве значення з інтервалу
    int second_limit;
    cin >> second_limit;
    int sum = 0, count = first_limit;
    do
    {
     sum += count; // приріст суми
     count++; // інкремент початкового значення із інтервалу
    } while (count <= second_limit); // кінець циклу do while
    cout << "sum = " << sum << endl; // друк суми
    system("pause");
    return 0;

}
```

Результат роботи програми:

```
Enter the first limit: -6
Enter the second limit: 10
sum = 34
Для продолжения нажмите любую клавишу . .
```

## 16. Сортування даних. Алгоритми сортування

Пошук – обробка деякої безлічі даних з метою виявлення підмножини даних, яка відповідає критеріям пошуку.

Всі алгоритми пошуку поділяються на:
- пошук в неврегульованій безлічі даних;
- пошук в упорядкованій безлічі даних.

Впорядкованість – наявність певним чином встановленних взаємозв΄язків між даними.

Сортування – впорядкування (перестановка) елементів в множині даних по певному критерію. Найчастіше в якості критерію використовується деяке числове поле, яке називають ключовим.

Приклад.

Сортування за зменшенням - впорядкування елементів по ключовому полю,яке передбачає, що ключове поле кожного наступного елемента не більше попереднього.

Сортування за зростанням.

Якщо ключове поле кожного наступного елемента не менше попереднього, то говорять про сортування за зростанням.

Мета сортування — полегшити подальший пошук елементів в відсортованій безлічі даних.

Всі алгоритми сортування діляться на:
- алгоритми внутрішнього сортування (сортування масивів);
- алгоритми зовнішньої сортування (сортування файлів).

Масиви зазвичай розташовуються в оперативній пам'яті, для якої є
швидкий довільний доступ. Основним критерієм, який пред'являється до алгоритмів сортування масивів, є мінімізація оперативної пам'яті,. Перестановку елементів потрібно виконувати на тому ж місці оперативної пам'яті, де вони знаходяться.

Методи, які пересилають елементи з масиву А в масив В, не представляють інтересу.

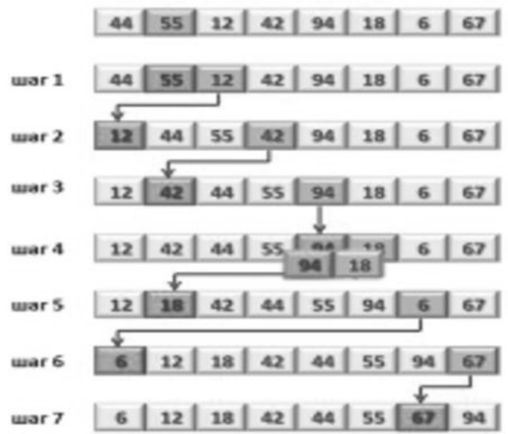
Методи сортування масивів можна розбити на три класи:
- сортування включеннями;
- сортування вибором;
- сортування обміном.

**Сортування прямими включеннями**

Елементи масиву умовно поділяються на готову послідовність

a1 , a2 ,….ai-1 і вхідну послідовність ai ,ai+1,…..a n

На кожному кроці і-й елемент поміщається на відповідне місце в готову послідовність




Приклад.

```
#include <iostream>
// Функція сортування прямими включеннями
void inclusionSort (int * num, int size)
{  // Для всіх елементів крім початкового
for (int i = 1; i <size; i ++)
{  int value = num [i]; // запам'ятовуємо значення елемента
int index = i; // і його індекс
while ((index> 0) && (num [index - 1]> value))
{// зміщуємо інші елементи до кінця масиву поки вони менше index
num [index] = num [index - 1];
index--;  }// зміщуємо перегляд до початку масиву
num [index] = value;   }} // розглянутий елемент поміщаємо на місце,
що звільнилося
```

**Сортування прямим вибором**

При сортуванні прямим вибором для пошуку одного елемента з найменшим ключем проглядаються всі елементи вхідної послідовності і знайдений елемент поміщається як черговий елемент в кінець готової послідовності.

Сортування прямим вибором краще алгоритму прямого включення, однак, якщо дані на початку впорядковані або майже впорядковані, пряме включення виявиться більш швидким.

Метод сортування прямим вибором заснований на наступних правилах:

- вибирається елемент з найменшим ключем;
- він міняється місцями з першим елементом a0.

Потім ці операції повторюються з a1-шими елементами, a2

елементами і так далі до тих пір, поки не залишиться один, елемент з найбільшим ключем.

Приклад.

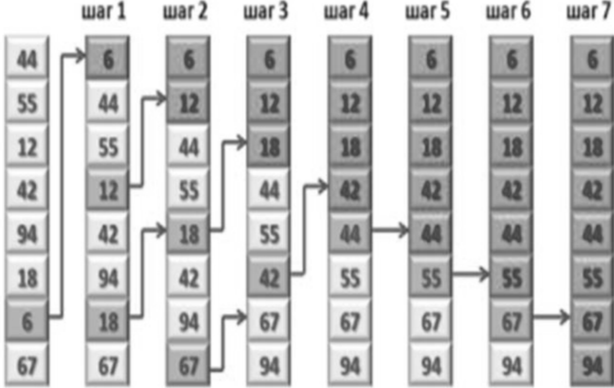
```
#include <iostream>
// Функція сортування прямим вибором

void selectionSort (int * num, int size)
{  int min, temp; // для пошуку мінімального елемента і для обміну
for (int i = 0; i <size - 1; i ++)  {
min = i; // запам'ятовуємо індекс поточного елемента
// шукаємо мінімальний елемент щоб помістити на місце i-ого
for (int j = i + 1; j <size; j ++) // для інших елементів після i-ого
{  if (num [j] <num [min]) // якщо елемент менше мінімального,
min = j; }// запам'ятовуємо його індекс в min
temp = num [i]; // міняємо місцями i-ий і мінімальний елементи
num [i] = num [min];
num [min] = temp;  }}
int main ()
{  int a [10]; // Оголошуємо масив з 10 елементів
// Вводимо значення елементів масиву
for (int i = 0; i <10; i ++)
{  printf ( "a [% d] =", i);
scanf ( "% d", & a [i]);  }
selectionSort (a, 10); // викликаємо функцію сортування
// Виводимо відсортовані елементи масиву
for (int i = 0; i <10; i ++)
printf ( "% d", a [i]);
getchar (); getchar ();
return 0;}
```

**Сортування прямим обміном (метод «бульбашки»)**

Алгоритм сортування прямим обміном заснований на принципі порівняння і обміну пари сусідніх елементів до тих пір, поки не будуть відсортовані всі елементи. Як і в методі прямого вибору, відбуваються проходи по масиву, зрушуючи кожного разу найменший елемент залишилася послідовності до початку масиву.

Якщо розглядати масиви як вертикальні, а не горизонтальні побудови, то елементи можна інтерпретувати як бульбашки в банці з водою, причому вага кожного відповідає його ключу. В цьому випадку при кожному проході один пухирець як би піднімається до рівня, відповідного його вазі.

Такий метод відомий під ім'ям «бульбашкове сортування».




Приклад.

```
#include <iostream>
// Функція сортування прямим обміном (метод "бульбашки")
void bubbleSort (int * num, int size)
{  // Для всіх елементів
for (int i = 0; i <size - 1; i ++)
{  for (int j = (size - 1); j> i; j--) // для всіх елементів після i-ого
{  if (num [j - 1]> num [j]) // якщо поточний елемент менше
попереднього
{    int temp = num [j - 1]; // міняємо їх місцями
num [j - 1] = num [j];
num [j] = temp;  }  }  }  }
int main ()
{  int a [10]; // Оголошуємо масив з 10 елементів
// Вводимо значення елементів масиву
for (int i = 0; i <10; i ++)
{   printf ( "a [% d] =", i);
scanf ( "% d", & a [i]);  }
bubbleSort (a, 10); // викликаємо функцію сортування
// Виводимо відсортовані елементи масиву
for (int i = 0; i <10; i ++)
printf ( "% d", a [i]);
getchar (); getchar ();
return 0;}
```

**Сортування за допомогою дерева**

Сортування за допомогою дерева здійснюється на основі бінарного дерева пошуку. Бінарне (двійкове) дерево пошуку – це бінарне дерево, для якого виконуються наступні додаткові умови (властивості дерева пошуку):

- обидва піддерева – ліве і праве, є двійковими деревами пошуку;
- у всіх вузлів лівого піддерева довільного вузла X значення джерел інформації менше, ніж значення ключа даних самого вузла X;
- у всіх вузлів правого піддерева довільного вузла X значення ключів даних не менше, ніж значення ключа даних вузла X.

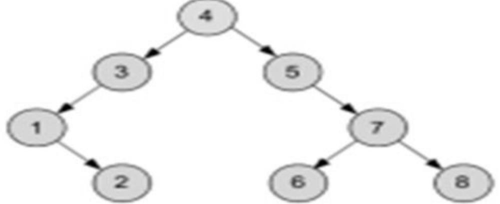
Дані в кожному вузлі повинні володіти ключами, на яких визначена операція порівняння менше.

Для сортування за допомогою дерева вихідна послідовність представляється у вигляді структури даних "дерево".

Наприклад, вихідна послідовність має вигляд: 4, 3, 5, 1, 7, 8, 6, 2

Коренем дерева буде початковий елемент послідовності. Далі всі елементи, менші кореневого, розташовуються в лівому піддереві, всі елементи, які більші

кореневого, розташовуються в правому піддереві. Причому це правило повинне дотримуватися на кожному рівні.
Після того, як всі елементи розміщені в структурі "дерево", необхідно вивести їх, використовуючи інфіксну форму обходу




## 17. Функції в С++. Прототипи функцій. Перевантаження функцій

Прості програми - алгоритм програми знаходився в головній функції, причому інших функцій в програмі не було.

Великі програми - програма складатиметься з окремих фрагментів коду, під окремим фрагментом коду розуміється функція.

Окремий фрагмент коду – це функція,робота якої не залежить від роботи іншої.

Тобто алгоритм в кожній функції функціонально достатній і не залежить від інших алгоритмів програми. Одного разу написавши функцію, її можна буде з легкістю переносити в інші програми.

Функція – це частина програми, яка має такі властивості чи ознаки:

- є логічно самостійною частиною програми;
- має ім’я, на основі якого здійснюється виклик функції (виконання функції). Ім’я функції підпорядковується правилам задавання імен ідентифікаторів мови С++;
- може містити список параметрів, які передаються їй для обробки або використання. Якщо функція не містить список параметрів, то така функція називається функцією без параметрів;
- може повертати (не обов’язково) деяке значення. У випадку, якщо функція не повертає ніякого значення, тоді в якості типу функції (перед ім'ям) вказується ключове слово void;
- має власний програмний код, який береться у фігурні дужки { } і вирішує власну задачу. Програмний код функції, реалізований в фігурних дужках, називається “тіло функції”.

**Використання стандартних функцій**

Для того, щоб скористатися функцією зі стандартного заголовного файлу С++ необхідно виконати дві дії:

1) підключити необхідний заголовний файл ( оголошується передпроцесорна директива #include, після чого всередині знаків <> пишеться ім'я заголовного файлу)

Приклад: #include <cmath> ;

2) здійснити виклик стандартної функцію в коді

Приклад: float power = pow(3.14,2) – виклик функції зведення в ступінь .

Завжди після імені функції ставляться круглі дужки, усередині яких функції передаються аргументи, і якщо аргументів кілька, то вони відокремлюються один від одного комами.

Крім виклику функцій зі стандартних заголовків файлів, в мові програмування С передбачена можливість створення власних функцій.

В мові програмування С є **два типи** функцій:

- функції, які не повертають значень;
- функції, які повертають значення.

Функції, що не повертають значення, завершивши свою роботу, ніякої відповіді програмі не дають.

Функції, які повертають значення, по завершенню своєї роботи повертають певний результат. Такі функції можуть повертати значення будь-якого типу даних.

Загальна форма опису функції виглядає наступним чином:

тип ім'я_функції(список_параметрів або void)

{ тіло_функції [return] (вираз); }

де тип – тип значення, яке повертає функція. Якщо поле "тип" містить ключове слово void, то функція не повертає ніякого значення;

– ім'я_функції – це безпосередньо ім'я функції. За цим іменем відбувається виклик програмного коду, реалізованого в тілі_функції. Крім того, ім'я_функції є покажчиком на цю функцію. Значенням покажчика - це адреса точки входу в функцію;

– список_параметрів – параметри, які передаються в функцію. Функція може отримувати будь-яку кількість параметрів. Якщо описується функція без параметрів, то в дужках вказується слово void;

– тіло_функції – набір операторів програмного коду, що реалізують алгоритм обчислення всередині функції;

– return (вираз) – ключове слово return вказує, що функція повертає значення задане в (вираз). Слово return може зустрічатись в декількох місцях тіла функції залежно від алгоритму (повторюватись).

В С++ існує 3 способи передачі параметрів у функцію:

– передача параметру за значенням (Call-By-Value). Це є проста передача копій змінних в функцію.

```
void func1 ( int arg );
```

– передача параметру за адресою змінної. У цьому випадку функції в якості параметрів передаються не копії змінних, а копії адрес змінних, тобто покажчик на змінну. Використовуючи цей покажчик функція здійснює доступ до потрібних комірок пам'яті, де розташована передана змінна і може змінювати її значення;

```
void func2 ( int * arg );
```

– передача параметру за посиланням (Call-By-Reference). Передається посилання (покажчик) на об'єкт (змінну), що дозволяє синтаксично використовувати це посилання як покажчик і як значення.

```
void func3 ( int & arg );
```

**Приклад.** Описується функція, що отримує три параметри. Перший параметр (x) передається за значенням. Другий параметр (y) передається за адресою (як покажчик). Третій параметр (c) передається за посиланням.

```
void MyFunction(int x, int* y, int & c)
{ x = 8; // значення параметра змінюється тільки в межах тіла функції
*y = 8; // значення параметра змінюється також за межами функції
c = 8; // значення параметра змінюється також за межами функції
return; }
```

Демонстрація виклику функції MyFunction() з іншого програмного коду:

```
int a, b, c;
a = b = c = 5;
// виклик функції MyFunction()
 // параметр a передається за значенням a->x
 // параметр b передається за адресою b->y
 // параметр c передається за посиланням c->z
MyFunction(a, *b, & c); // на виході a = 5; b = 8; c = 8;
```

Значення змінної **a** не змінилось, тому що змінна a передавалась у функцію з передачею значення.

Значення змінної **b** після виклику функції змінилось, тому що в функцію передавалось значення адреси змінної b.

Значення **c** після виклику функції змінилось. Посилання є адресою об'єкту в пам'яті. З допомогою цієї адреси можна мати доступ до значення об'єкта.

Формальні параметри – це змінні, що приймають значення аргументів (параметрів) функції.

При виклику функції, що має аргументи, компілятор здійснює копіювання копій формальних аргументів в стек.

При виклику функції з програмного коду фігурує фактичний параметр.

При виклику функції фактичні параметри копіюються в спеціальні комірки пам'яті в стеку (стек – частина пам'яті). Ці комірки пам'яті відведені для формальних параметрів. Таким чином, формальні параметри (через використання стеку) отримують значення фактичних параметрів.

Область видимості формальних параметрів функції визначається межами тіла функції, в якій вони описані ( в межах фігурних дужок { }).

Прототипом функції називається оголошення функції, що не містить тіла функції, але вказує ім'я функції, арність (кількість аргументів), типи аргументів і тип даних, що повертається. У той час як визначення функції описує, що саме робить функція, прототип функції може сприйматися як опис її інтерфейсу.

Приклад.

double new_style(int a, double *x); /* прототип функции */

У прототипі імена аргументів є необов'язковими, тим не менш, необхідно вказувати тип разом з усіма модифікаторами (наприклад, покажчик чи це або константний аргумент).

**double alt_style (int, double *);** / * Альтернативна форма прототипу * /

Прототип використовується компілятором для правильного виклику функції. Для цього компілятор спочатку аналізує ім'я викликається функції і шукає в файлі її прототип. Потім перевіряються аргументи, які передаються в функцію і повертає значення.

Якщо немає прототипу, то в кожен вихідний файл необхідно включити повний опис функції. Компілятор буде інтерпретувати це як перевизначення.

Якщо ж ми використовуємо прототип, то ми можемо включати цей прототип в стільки вихідних файлів, скільки нам необхідно.

Краще опис функції включити в окремий вихідний файл. Після цього треба скомпілювати цей файл і отримати об'єктний файл. Прототип слід помісити в заголовки і включати його директивою #include в ті вихідні файли, в яких присутній виклик функції.

**Перевантаження функції**

Під перевантаженням функції розуміється визначення декількох функцій (дві або більше) з однаковим ім'ям, але різними параметрами.

Перевантаження функцій потрібно для того, щоб уникнути дублювання імен функцій, які виконують подібні дії, але з різною програмної логікою.

Наприклад, розглянемо функцію areaRectangle () яка обчислює площу прямокутника.

**float areaRectangle** (float a, float b)

{Return a * b; }

Отже, це функція з двома параметрами типу float, причому аргументи функції повинні бути в сантиметрах, повертається значення типу float - теж в сантиметрах.

Припустимо, що наші вихідні дані (сторони прямокутника) задані в метрах і сантиметрах, наприклад: a = 2м 35 см; b = 1м 86 см.

В такому випадку, зручно було б використовувати функцію з чотирма параметрами. Тобто, кожна довжина сторін прямокутника передається в функцію за двома параметрами: метри й сантиметри.

**float areaRectangle (float a_am, float a_sm, float b_m, float b_sm)**

// функція, обчислює площу прямокутника з 4-ма параметрами a (м) a (см) b (м) b (cм)

**{Return (a_m * 100 + a_sm) * (b_m * 100 + b_sm);}**

Виклик перевантажених функцій нічим не відрізняється від виклику звичайних функцій.

Наприклад:

**areaRectangle (32, 43);** // буде викликана функція, яка обчислює площу прямокутника з двома параметрами a (см) і b (див)

**areaRectangle (4, 43, 2, 12);** // буде викликана функція, обчислює площу прямокутника з 4-ма параметрами a (м) a (см) b (м) b (см).

Компілятор самостійно вибере потрібну функцію, аналізуючи тільки сигнатури перевантажених функцій.

## 18. Функції зі змінною кількістю параметрів. Структури. Об'єднання

У С++ крім функцій з фіксованим числом параметрів є можливість також використовувати функції зі змінною кількістю параметрів (варіадичних), під якими розуміють функції, в які можна передавати дані, не описуючи їх у прототипі й, відповідно, у заголовку опису функції.

Приклад. Функції scanf та printf .

```
#include <stdio.h> /заголовочний файл стандарту в/в в Сі
/ int printf(const char *format, ...); формат функції printf
…
printf("Hi %c %d %s", 'c', 10, "there!");
"Hi c 10 there!"
```

Формат оголошення функції зі змінною кількістю параметрів має такий вигляд:

**[тип] ім'я_функції(параметр1, параметр2, ...);**

де **параметр1, параметр2** - постійна частина списку параметрів, а еліпсис **…** - змінну частину, яка означає довільну кількість параметрів.

У С/С++ фактичні параметри записуються в стек з кінця списку аргументів, причому ці дані поміщаються в стек відповідно до типу, який використовується при виклику функції.

Зрозуміло, що маючи вказівник на початок (вершина стеку) і знаючи тип даних аргументів, ми можемо послідовно перебрати всі аргументи зі стеку (поки стек не стане порожнім), навіть не знаючи наперед їх кількість.

При передачі у функцію додаткових аргументів (тобто аргументів, що відповідають змінній частині списку формальних параметрів) відсутня інформація про їх тип даних на етапі компіляції програми.

Уся відповідальність за правильну обробку типів таких аргументів покладається на програміста. **Якщо передати функції зі змінним числом параметрів параметри не того типу, обробка якого запрограмована, наслідки будуть непередбачувані і, скоріше за все, сумні, адже компілятор у цьому випадку не перевірятиме ні кількість, ні типи параметрів**.

Видобути аргументи зі стеку у цьому випадку можна лише через вказівники. Причому тут існує два способи завдання довжини змінного списку параметрів:

- в одному з постійних аргументів вказувати їх кількість;
- або ж додати у кінець списку аргумент з унікальним значенням, який буде слугувати ознакою кінця списку.

Наприклад:

func1(5,xl,x2,x3,x4,x5); - тут зазначене число аргументів - 5;

func2( xl,x2,x3,x4,0); - тут зазначена ознака кінця списку - 0.

У стандарт мови С++ входять макроси для роботи зі списками параметрів змінної довжини.

Ці макроси визначені у файлі **cstdarg** (**stdarg.h**). При їх використанні також доводиться вказувати у списку постійний параметр, оголосити та встановити на нього вказівник та переміщувати цей вказівник по списку. В кінці списку має бути NULL. Макроси мають наступний формат:

**void va_start(va_list prm, останній_фіксований_параметр);**
**тип va_arg(va_list prm, тип);**
**void va_end(va_list prm);**

Макрос va_start встановлює вказівник типу va_list на останній фіксований параметр.

Макрос va_arg переміщає вказівник на наступний параметр. Синтаксис наступний: va_arg(вказівник_типу_va_list, тип_чергового_параметра). У момент написання програми програміст повинен знати тип кожного з параметрів.

Макрос va_end встановлює вказівник в NULL.

Приклад.

```
#include <iostream>
        #include <cstdarg>
int sum (int k, ...)
{
   int result = 0;
   va_list args;
           va_start(args, k);
   for (int i = 0; i < k; ++i)
   {
      result += va_arg(args, int);
           }
   va_end(args);
   return result;
        }
int main(void)
{
   printf("%d \n", sum(4, 1, 2, 3, 4));
   printf("%d \n", sum(5, 12, 21, 13, 4, 5));
   return 0;
        }
```

**Складні типи даних**

Тип Об‘єкта може бути:

- базовим типом даних, прийнятим в мові С++;
- Структурою;
- Об’єднанням;
- Класом (будемо вивчати пізніше).

Базові типи даних в програмі:

- Змінні (чисельні, строкові , покажчики та ін.)
- Масиви (статичні, динамічні)

Складні типи даних в програмі:

- Переліки
- Структури
- Об'єднання
- Бітові поля
- Масиви Структур та Об'єднань.

**Структура**

Нам відомо, що Структура – це об'єднання кількох об'єктів, можливо, різного типу під одним ім'ям. В якості об'єктів можуть виступати змінні, масиви, покажчики та інші структури.

Приклад оголошення структури типу date:

```
struct date
```

```
{int day; // 4 байта
char * month; // 4 байта
int year; // 4 байта };
```

Поля структури розташовуються в пам'яті в тому порядку, в якому вони оголошені.

У зазначеному прикладі структура date займає в пам'яті 12 байт, кожне поле займає по 4 байта.

Як і будь-який інший тип даних Структуру треба оголосити та ініціалізувати.

**Об'єднання**

**На відмінність від структури Об'єднання -** це складний тип даних, що дозволяє розміщувати в одному і тому ж місці оперативної пам'яті дані різних типів.

**Об'єднання** – це групування змінних, які займають одну й ту ж область пам'яті.

Зовнішні атрибути **Об'єднання** в програмі не відрізняються від атрибутів **Структур.** Фактично **Об'єднання** оголошуються так же як структура, тільки замість слова **struct** використовується ключеве слово **union.**

Загальна форма оголошення **Об'єднання**:

**union І'мя_типу_Об'єднання**

{тип І'мя_Об'єкта_1;

тип І'мя_Об'єкта_2;

. . .

тип І'мя_Об'єкта_n; };

де **І'мя_типу_Об'єднання** – безпосередньо ім'я новостворюваного типу;

І'мя_Об'єкта_1,… І'мя_Об'єкта_n – змінні, що є полями об'єднання.

Ці змінні можуть бути різних типів;

тип – тип Об'єкта, що є полем об'єднання.

Розмір оперативної пам'яті, необхідний для зберігання об'єднань, визначається розміром пам'яті, необхідним для розміщення даних того типу, який вимагає максимальної кількості байт.

Коли використовується елемент меншої довжини, ніж найдовший елемент об'єднання, то цей елемент використовує тільки частину відведеної пам'яті.

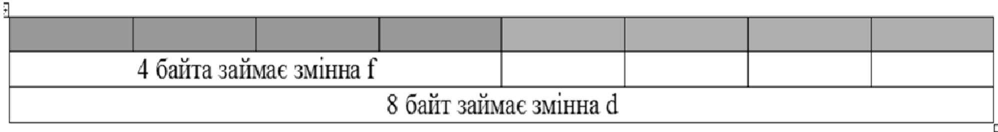
Всі елементи об'єднання зберігаються в одній і тій же області пам'яті, починаючи з однієї адреси.

Доступ до полів об'єднання здійснюється так само, як і для структури:

за допомогою символу ' **.** ' (приклад **Fl.f**) .

**Приклад** оголошення типу "об'єднання Floats" .

```
union Floats
{ float f; // f займає 4 байти
double d; // d займає 8 байт };
```




**Приклад** використання типу Floats в програмному коді.

```
Floats Fl; // Задаємо змінну Fl типу Floats.
int d;
Fl.f = 20.5; // Fl.f – звернення до поля f
Fl.d = -100.35; // Fl.d – звернення до поля d
d = sizeof(Fl); // d = 8
```

**Функція sizeof()** повертає довжину в байтах, необхідних для зберігання заданого типу даних.

Приклад.

```
#include <iostream>
union MyUnion // оголошуємо змінну типу Об'єднання з іменем MyUnion
{
```

```
short x;// поле x типу short
int y;// поле y типу int
} ;
int main ()
{
using namespace std;
MyUnion u;// Оголошуємо об‘єкт u типу MyUnion
u.x = 200;//записуємо в u.x значення 200
u.y = 111;//записуємо в u.y значення 111
cout << u.x;//в u.x  було записано 111, оскільки все пишеться в одну комірку пам'яті
}
```

**Результат роботи програми: 111**

## 19. Переліки. Генератор випадкових чисел

Переліки (перерахування) - це набір іменованих цілочисельних констант, що визначає всі допустимі значення, які може приймати змінна.

Кожне перерахування – це окремий тип даних.

Наприклад, в якості перерахування монет в Сполучених Штатах використовуються:

*один цент, п'ять центів, десять центів, двадцять п'ять центів, півдолару, долар*

Перерахування визначаються за допомогою ключового слова enum, яке вказує на початок даних цього типу.

Стандартний вид оголошення перерахуванняу наступний:

**enum ярлик {список перерахування} список змінних;**

**enum ярлик {список перерахування};**

Як **ім'я перерахування - ярлик**, так і список змінних необов'язкові, але один з них повинен бути присутнім.

Список перерахування - це розділений комами список ідентифікаторів. Ярлик використовується для оголошення змінних даного типу.

Наступний фрагмент визначає перерахування coin і оголошує змінну money цього типу:

**enum coin {penny, nickel, dime, quarter, half_dollar, dollar,};**

**enum coin money;**

Маючи дане визначення і оголошення, наступний тип привласнення абсолютно коректний:

```
money = dime;
if (money == quarter) printf ( "is a quarter \ n");
```

В перерахуваннях кожному символу ставиться у відповідність цілочисельне значення і тому перерахування можна використовуватися в будь-яких цілочисельних виразах.

**Наприклад:**

printf ( "The value of quarter is% d", quarter); абсолютно коректно.

Якщо явно не проводити ініціалізацію, то перший символ буде 0, другий - 1 і так далі.

Отже: printf ( "% d% d", penny, dime);

виводить 0 2 на екран.

Можна визначити значення одного або декількох символів, використовуючи ініціалізатор.

Це робиться шляхом запису за символом знака = і цілочисельного значення. При використанні ініціалізатора, символи, які розміщені за ініціалізованим значенням, отримують значення більше ніж вказане перед цим.

Наприклад, в наступному оголошенні quarter отримує значення 100.

**enum coin (penny, nickel, dime, quarter = 100, half_dollar, dollar);**

Тепер символи отримують наступні значення:

penny 0 nickel 1 dime 2 quarter 100 Half dollar 101 dollar 102

Використовуючи ініціалізацію, більш одного елемента перерахування можуть мати одне і теж значення.

**Генератор випадкових чисел в C++**

Випадкові числа в мові програмування C++ можуть бути згенеровані функцією Rand () із стандартної бібліотеки C + +.

Вони знаходяться в бібліотечном файлі cstdlib, тому щоб їх застосовувати в програмі, необхідно підключити цей бібліотечний файл: #include <cstdlib> або #include <stdlib.h> (для старих компіляторів).

Функція rand() генерує числа в діапазоні від 0 до RAND_MAX.

RAND_MAX - це константа, визначена в бібліотеці <cstdlib>. Для MVS RAND_MAX = 32767, воно може бути і більше, залежно від компілятора.

Програма, яка використовує генератор випадкових чисел rand():

```
#include "stdlib.h"
#include <iostream>
using namespace std;

int main()
{
    cout << "RAND_MAX = " << RAND_MAX << endl; // константа, хранящая максимальный предел из интервала случайных чисел
    cout << "random number = " << rand() << endl; // запуск генератора случайных чисел
    system("pause");
    return 0;
}
```

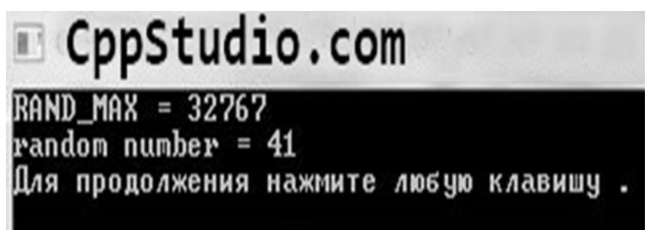
В рядку cout << "RAND_MAX = " << RAND_MAX << endl;

значення константи RAND_MAX виштовхується в потік виведення, так ми зможемо подивитися максимальне значення з інтервалу випадкових чисел.

В рядку cout << "random number = " << rand() << endl;

запускається генератор випадкових чисел rand(), причому він згенерує випадкове число один раз, при першому запуску програми. Надалі, скільки б Ви не запускали цю програму, згенероване число залишиться тим самим.

Результат:




Формула генерації випадкових чисел за заданим діапазоном

**random_number = firs_value + rand() % last_value;**

// де firs_value – мінімальне число із потрібного діапазону

// last_value - ширина вибірки

Приклад. **rand() % 3 +1** // діапазон від 1 до 3 включно

Число 3 є коефіцієнтом, що масштабується. Тобто яке б не видав число генератор випадкових чисел rand() запис rand() % 3 в результаті видасть число з діапазону від 0 до 2. Для того щоб змістити діапазон, ми додаємо одиницю, тоді діапазон зміниться на такий — від 1 до 3 включно.

**Функція srand()**

Особливість функції rand() - при повторному запуску програми друкуються ті самі числа. Ця особливість потрібна для того, щоб можна було правильно налагодити програму, що розробляється.

Коли потрібно, щоб при кожному виконанні програми генерувалися випадкові числа

використовують функцію srand () зі стандартної бібліотеки С++.

Функція srand() отримавши цілий позитивний аргумент типу unsigned або unsigned int (без знакове ціле) виконує рандомізацію, таким чином, щоб при кожному запуску програми функція srand() генерувала випадкові числа.

Приклад.

```
#include "stdlib.h"
#include <iostream>
using namespace std;
int main()
{
   unsigned rand_value = 11;
   srand(rand_value); // рандомізація
   cout << "rand_value = " << rand_value << endl;
   cout << "1-random number = " << 1 + rand() % 10 << endl; // перший запуск генератора випадкових чисел
   cout << "2-random number = " << 1 + rand() % 10 << endl; // другий запуск генератора випадкових чисел
   system("pause");
   return 0;
}
```

Результат № 1:

```
rand_value = 11
1-random number = 5
2-random number = 9
Для продолжения нажмите любую клавишу . . .
```

Результат № 2:

```
rand_value = 5
1-random number = 5
2-random number = 4
Для продолжения нажмите любую клавишу .
```

**Використання функції time() для автоматичної рандомізації**

```
// автоматична рандомізація
srand( time(0) );
```

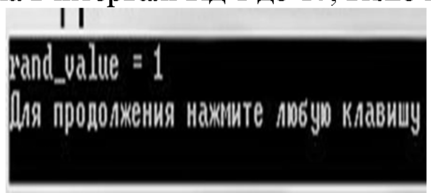
Щоб використовувати функцію time(), необхідно підключити заголовний файл <ctime>.

Приклад.

```
#include "stdafx.h"
#include <iostream>
#include <ctime>
using namespace std;
int main(int argc, char* argv[])
{
   srand( time( 0 ) ); // автоматическая рандомизация
   cout << "rand_value = " << 1 + rand() % 10 << endl;
   system("pause");
```

```
return 0;
}
```

Результат роботи програми: при кожному запуску програми будуть генеруватися випадкові числа в інтервалі від 1 до 10, включно




## 20. Класи в С++

Класи і об'єкти в С ++ є основними концепціями об'єктно-орієнтованого програмування – ООП.

Об'єктно-орієнтоване програмування –
розширення структурного програмування.

Основна відмінність мови програмування С ++ від С
полягає в тому, що в С немає класів, а отже мова С не підтримує ООП на відміну від С ++.

Класи – це певний опис, схема, креслення, за якими створюються об'єкти.

Для створення об'єкта в ООП необхідно спочатку
скласти креслення, тобто класи.

Класи мають свої функції, які називаються методами класу.

**Класи в С ++ –** це абстракція, яка описує методи та властивості ще не існуючих об'єктів.

**Об'єкти –** конкретне уявлення, яке має свої властивості і методи. Створені об'єкти на основі одного класу називаються екземплярами цього класу. Ці об'єкти можуть мати різну поведінку, властивості, але все одно будуть об'єктами одного класу.

В ООП існує три основних принципи побудови класів:

**- Інкапсуляція –** це властивість, що дозволяє об'єднати в класі властивості і методи, що працюють з ними і приховати деталі реалізації від користувача. Розміщення описів змінних і функції з їх обробки, тобто властивостей та методів в одному класі називається інкапсуляцією.

**- Успадкування –** це властивість, що дозволяє створити новий клас-нащадок на основі вже існуючого, при цьому всі характеристики батьківського класу присвоюються класу-нащадку. Ще його називають – похідний клас.

**- Поліморфізм –** властивість класів, що дозволяє використовувати об'єкти класів з однаковим інтерфейсом без інформації про тип і внутрішню структуру об'єкта.

```
class ім’я класу>: список класів-батьків>
{
public:        // доступно всім
<дані, методи, властивості, події>
protected: // доступно тільки нащадкам
<дані, методи, властивості, події>
private:   // доступно тільки в середині класу
<дані, методи, властивості, події>
} <список змінних>;
```

**Методи** класу - це його функції.

**Властивості** класу - його змінні.

Всі властивості і методи класів мають права доступу. За замовчуванням, весь вміст класу є доступним для читання і запису тільки для нього самого.

Для того, щоб дозволити доступ до даних класу ззовні, використовують модифікатор доступу **public**. Всі функції і змінні, які знаходяться після модифікатора **public**, стають доступними з усіх частин програми.

Закриті дані класу розміщуються після модифікатора доступу **private**. Якщо відсутній модифікатор public, то всі функції і змінні, за замовчуванням є закритими.

Зазвичай, приватними роблять всі властивості класу, а публічними - його методи. Всі дії із закритими властивостями класу реалізуються через його методи.

Такий підхід називається **абстракцією даних або відділенням даних від логіки** – один з фундаментальних принципів об'єктно-орієнтованого програмування.

Наприклад, якщо хтось інший захоче використовувати наш клас в своєму коді, йому не обов'язково знати, як саме підраховується функція. Він просто буде використовувати цю функцію без знання алгоритму її роботи.

Крім того, відділення даних від логіки є гарним тоном в програмувані.

**Структура оголошення класів:**

```
// оголошення класів в С ++
class / * ім'я класу * /
{
private:
/ * Список властивостей і методів для використання всередині класу * /
public:
/ * Список методів доступних інших функцій і об'єктів програми * /
protected:
/ * Список засобів, доступних при спадкуванні * /
};
```

Оголошення класу починається з зарезервованого ключового слова **class**, після якого пишеться ім'я класу. У фігурних дужках оголошується тіло класу, причому після дужки } обов'язково потрібно ставити крапку з комою.

У тілі класу оголошуються три мітки специфікації доступу, після кожної мітки потрібно обов'язково ставити двокрапку:

- Специфікатор доступу **private**. Всі методи і властивості класу, оголошені після специфікатор доступу private будуть доступні тільки усередині класу.
- Специфікатор доступу **public**. Всі методи і властивості класу, оголошені після специфікатор доступу public будуть доступні іншим функціям і об'єктам в програмі.
- Специфікатор доступу **protected.** Всі методи і властивості класу, оголошені після специфікатор доступу **protected** доступні при спадкуванні.

При оголошенні класу, не обов'язково оголошувати три специфікатори доступу, і не обов'язково їх оголошувати в такому порядку.

Оголосимо найпростіший клас, в якому буде оголошена одна функція, що друкує повідомлення.

```
#include <iostream>
using namespace std;
// початок оголошення класу
class Acit // ім'я класу
{public: // специфікатор доступу
void message() // функція (метод класу) виводить повідомлення на екран
{cout <<"Classes and Objects in C + +";
};
}; // кінець оголошення класу Acit
int main ()
{Acit objMessage; // оголошення об'єкту
objMessage.message(); // виклик функції класу message
system ("pause");
return 0;}
```

```
Classes and Objects in C + +
```

Результат роботи програми:

В програмі визначено клас з ім'ям **Acit**. Функція (метод класу) **message** виводить повідомлення на екран повідомлення ***Classes and Objects in C + +.*** У тілі класу оголошений специфікатор доступу **public**, який дозволяє викликати інші функції (методи) класу, оголошені після public. Ось саме тому в головній функції ми змогли викликати функцію message.

Ім'я класу прийнято починати з великої літери, наступні слова в імені також повинні починатися з великої літери.

*Зверніть увагу: елементи класу записуються через крапку після імені об'єкта.*

**Методи класу** – це ті ж функції, тільки оголошені всередині класу, тому все що відноситься до функцій актуально і для методів класу.

**Оголошення класів** виконується аналогічно оголошенню функцій, тобто клас можна оголошувати в окремому файлі або в головному файлі.

В головній функції оголошена змінна objMessage типу Acit,

змінна objMessage – це об'єкт типу клас Acit. Після того як об'єкт класу оголошений, можна скористатися його методами.

Метод – це функція message(). Звертаємося до методу об'єкта objMessage через крапку «.».

В результаті програма виводить на екран текстове повідомлення "Classes and Objects in C + +".

Атрибути об'єкта зберігаються в змінних, оголошених всередині класу, якому належить даний об'єкт.

Причому, оголошення змінних має виконуватися із специфікатором доступу private. Такі змінні називаються елементами даних. Так як елементи даних оголошені в private, то і доступ до них можливо отримати тільки методами класу, зовнішній доступ до елементів даних заборонений.

Тому зазвичай оголошують в класах спеціальні методи – так звані set і get функції, за допомогою яких можна маніпулювати елементами даних:

**set-функції** ініціалізують елементи даних,

**get-функції** дозволяють переглянути значення елементів даних.

Доопрацюємо клас Acit таким чином, щоб в ньому можна було зберігати дату в форматі дд.мм.гг. Реалізуємо відповідно set і get функції

```
#include <iostream>
using namespace std;
class Acit // ім'я класу
{
private: // специфікатор доступу private
int day, // день
month, // місяць
year; // рік
public: // специфікатор доступу public
void message () // функція (метод класу) виводить повідомлення на екран
{
cout << "Classes and Objects in C + + "<<endl;
}
void setDate (int date_day, int date_month, int date_year) // установка дати в форматі дд.мм.гг
{
day = date_day; // ініціалізація день
month = date_month; // ініціалізація місяць
```

```
year = date_year; // ініціалізація рік
}
void getDate () // відобразити поточну дату
{
cout << "Date:" << day << "." << month << "." << year << endl;
}
}; // кінець оголошення класу Acit.
int main (int argc, char * argv [])
{
int day, month, year;
cout <<"DATE"<<endl;
cout <<"day:"; cin>>day;
cout <<"month:"; cin>>month;
cout <<"year:"; cin>>year;
Acit objAcit; // оголошення об'єкту
objAcit.message(); // виклик функції класу message
objAcit.setDate(day, month, year); // ініціалізація дати
objAcit.getDate(); // відобразити дату
system( "pause");
return 0; }
```

**Set - функції і get - функції класів**

Функції set і get називають методами доступу до даних.

Специфікатор доступу private обмежує доступ до змінних, які оголошені після нього і до початку специфікатора public.

Таким чином до змінних day, month, year, можуть отримати доступ тільки методи класу. Функції, які не належать класу, не можуть звертатися до цих змінним.

Елементи даних або методи класу, оголошені після специфікатор доступу private, але до початку наступного специфікатора доступу називаються закритими елементами даних і закритими методами класу.

Тоді, для маніпулювання елементами даних, оголошуються спеціальні функції – get і set.

В клас Acit ми додали два методу setDate() і getDate(), докладно розглянемо кожен метод.

Метод setDate() ініціалізує змінні day, month, year.

Щоб викопристовувати значення закритих даних оголошена функція getDate(), яка повертає значення трьох змінних day, month, year у вигляді дати.

В main(), як і завжди, створюємо об'єкт класу, і через об'єкт викликаємо його методи.

Якби елементи даних були оголошені після специфікатора public ми б змогли до них звернутися так само, як і до методів класу.

В описі класу може бути оголошено тільки прототипи функцій, а їх реалізацію наведено за межами опису. Але якщо функція складається з декількох операторів, її можна розмістити всередині класу.

У випадку, коли функції-методи описано за межами класу, їхні заголовки повинні складатися з імені класу, операції розширення області класу «::», імені функції та її формальних аргументів, якщо вони є, а далі йтиме зміст функції.

Правила створення розділів класу:

- розділи можуть з'являтися у будь-якому порядку і декілька разів;
- якщо не оголошено жодного розділу, компілятор за замовчуванням оголошує усі елементи закритими;
- розміщати дані-властивості класу у відкритому розділі можна тільки за необхідності. Дані-властивості класу звичайно розміщують у закритому, або захищеному розділі, щоб до них мали доступ функції-методи класу, а також функції класів-нащадків;

– для зміни значень даних (властивостей) слід використовувати функції-члени класу.

### 21. Конструктор. Деструктор

**Конструктор** – спеціальний метод класу, який виконує початкову ініціалізацію елементів даних, при чому ім'я конструктора обов'язково має збігатися з ім'ям класу. Важливою відмінністю конструктора від інших функцій є те, що він не повертає значень взагалі ніяких, в тому числі і void.

У будь-якому класі повинен бути конструктор, навіть якщо явно конструктор не оголошений , то компілятор надає конструктор за замовчуванням, без параметрів.

Доопрацюємо клас Acit, додавши до нього конструктор.

**Приклад 1.**

Задача. Конструктор має три параметри, через які він отримує інформацію про дату, в тілі конструктора викликається set – функція для установки дати.

Можна було реалізувати початкову ініціалізацію елементів даних класу і без set – функції, але так як ця функція була передбачена, то правильніше буде використовувати саме цю функцію.

Після main() оголошуємо об'єкт класу, причому після імені об'єкта в круглих дужках передаємо три аргументи.

Програма, в який реалізована поставлена задача розміщена на наступних трьох слайдах

```
#include <iostream>
using namespace std;
class Acit // ім’я класу
{
private: // специфікатор доступу private
int day, //день
month, // місяць
year; // рік
public: // специфікатор доступу public
Acit (int date_day, int date_month, int date_year) // конструктор класу
{
setDate (date_day, date_month, date_year); // виклик функції установки дати
}
void message () // функція виводить повідомлення на екран
{ cout<<"Classes and Objects in C + +"<<endl;
}
void setDate (int date_day, int date_month, int date_year)
// установка дати в форматі дд.мм.гг
{ day = date_day; // ініціалізація день
month = date_month; // ініціалізація місяць
year = date_year; // ініціалізація рік
}
void getDate () // відобразити поточну дату
{ cout << “\nDATA:" << day << "." << month << "." << year << endl;
}
}; // кінець оголошення класу Acit
int main ()
{
Acit objAcit (30,04,2020); // оголошення об'єкту і ініціалізація елементів даних
objAcit.message (); // виклик функції message
objAcit.getDate (); // відобразити дату
```

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

Разультат роботи програми: вивід на екран повідомлення та дати

```
Classes and Objects in C + +

DATE:30.4.2020

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

**Конструктор**

Коли всі члени класу (або структури) є відкритими, то ми можемо ініціалізувати клас (або структуру) напряму, використовуючи список ініціалізаторів або uniform-ініціалізацію (в C++11).

Приклад 2. Способи ініціалізації даних області public

```
class Boo
{
public:
   int m_a;
   int m_b;
};
int main()
{
   Boo boo1 = { 7, 8 }; // список ініціалізаторів
   Boo boo2 { 9, 10 }; // uniform-ініціалізація (C++11)

   return 0;
}
```

Конструктор — це особливий тип методу класу, який автоматично викликається при створенні об'єкта цього ж класу.

Конструктори зазвичай використовуються для ініціалізації змінних-членів класу значеннями, які надані за замовчуванням/користувачем, або для виконання будь-яких кроків налаштування, необхідних для використовуваного класу (наприклад, відкрити певний файл або базу даних).

На відміну від звичайних методів, конструктори мають певні правила щодо своїх імен:

конструктори завжди повинні мати те ж ім'я, що і клас (враховуючи верхній і нижній регістри);

конструктори не мають типу повернення (навіть void).

Зверніть увагу, конструктори призначені тільки для виконання **ініціалізації**. Не слід намагатися викликати конструктор для повторної ініціалізації існуючого об'єкта. Хоча це може скомпілюватися без помилок, результати можуть вийти несподівані (компілятор створить тимчасовий об'єкт, а потім видалить його).

Конструктор, який не має параметрів (або містить параметри, всі з яких мають значення за замовчуванням), називається **конструктором за замовчуванням.** Він викликається, якщо користувачем не вказані значення для ініціалізації.

Приклад 3. Явно описано конструктор за замовчуванням

```
#include <iostream>
class Fraction
{
private:
   int m_numerator;
   int m_denominator;
public:
   Fraction() // конструктор за замовчуванням
```

```
    {
        m_numerator = 0;
        m_denominator = 1;
    }
    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
```

// static_cast створено для виконання всіх видів перетворень, які дозволені компілятором. Синтаксис: static_cast < type_to > ( object_from ).

```
};
```

Приклад 3. Продовження

```
int main()
{
    Fraction drob; // оскільки аргументи відсутні, то викликається конструктор за замовчуванням Fraction()
    std::cout << drob.getNumerator() << "/" << drob.getDenominator() << '\n';
    return 0;
}
```

Цей клас містить дріб у вигляді окремих значень типу int. Конструктор за замовчуванням називається Fraction (як і клас). Оскільки ми створили об'єкт класу Fraction без аргументів, то конструктор за замовчуванням спрацював відразу ж після виділення пам'яті для об'єкта, і ініціалізував наш об'єкт.

Результат виконання програми: 0/1

Зверніть увагу, наш чисельник (m_numerator) і знаменник (m_denominator) були ініціалізовані значеннями, які ми вказали в конструкторі за замовчуванням! Це настільки корисна особливість, що майже кожен клас має свій конструктор за замовчуванням. Без нього значеннями нашого чисельника і знаменника було б сміття до тих пір, поки ми явно не присвоїли б їм нормальні значення.

Конструктори також можуть бути оголошені з параметрами. Ось приклад конструктора, який має два цілочисельних параметри, які використовуються для ініціалізації чисельника і знаменника:

Приклад 4.

```
#include <cassert>
class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    Fraction() // конструктор за замовчуванням
    {
        m_numerator = 0;
        m_denominator = 1;
    }
```

Приклад 4. Продовження

```
// Конструктор з двома параметрами, один з яких має значення за замовчуванням
    Fraction(int numerator, int denominator=1)
    {
        assert(denominator != 0);
        m_numerator = numerator;
        m_denominator = denominator;
    }
```

```
    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
};
```

Ці два конструктора можуть мирно співіснувати в одному класі завдяки перевантаженню функцій.

Можна визначити будь-яку кількість конструкторів до тих пір, поки у них будуть унікальні параметри (враховується їх кількість і тип).

Якщо клас не має конструкторів, то мова C++ автоматично згенерує для вашого класу відкритий конструктор за замовчуванням.

```
class Date
{
private:
    int m_day = 12;
    int m_month = 1;
    int m_year = 2018;
};
```

У цього класу немає конструктора, тому компілятор згенерує наступний конструктор:

```
class Date
{
private:
    int m_day = 12;
    int m_month = 1;
    int m_year = 2018;
public:
    Date() // конструктор, що неявно генерується
    {
    }
};
```

Цей конструктор дозволяє створювати об’єкти класу, але не виконує їх ініціалізацію і не присвоює значення членам класу.

Приклад 5. Хоча ви не можете побачити неявно згенерований конструктор, але його існування можна довести:

```
class Date
{
private:
    int m_day = 12;
    int m_month = 1;
    int m_year = 2018;
    // Не було надано конструктора, тому C++ автоматично створить відкритий конструктор за замовчуванням
};

int main()
{
    Date date; // виклик неявного конструктора
    return 0;
}
```

**Деструктор**

Конструктор (від слова побудувати – створювати) - це спеціальний метод класу, який призначений для ініціалізації елементів класу деякими початковими значеннями.

На відміну від конструктора, **деструктор** (від слова самознищення – руйнувати) – спеціальний метод класу, який служить для знищення елементів класу. Найчастіше його використовують тоді, коли в конструкторі,при створенні об'єкта класу, динамічно була виділена ділянка пам'яті і необхідно цю пам'ять очистити, якщо ці значення вже не потрібні для подальшої роботи програми.

Важливо запам'ятати:

конструктор і деструктор ми завжди оголошуємо в розділі **public**;

при оголошенні конструктора тип даних (значення), що повертається не вказується, в тому числі – void!!!;

у деструктора також немає типу даних для повернення , до того ж деструктором не можна передавати ніяких параметрів;

ім'я класу і конструктора повинно бути ідентично;

ім'я деструктора ідентично імені конструктора, але з приставкою тильда ~ ;

Приклад оголошення конструктору і деструктору в програмі:

```
Acit (int date_day, int date_month, int date_year) // конструктор класу
{
setDate  (date_day,  date_month,  date_year);  //  виклик  функції установки дати
}
~Acit () // це деструктор класу
{
cout << "Тут сработал деструктор" << endl;
}
```

**22. Успадкування**

Одні класи можуть містити інші класи в якості змінних-членів.

За замовчуванням, при створенні зовнішнього класу, для змінних-членів викликатимуться конструктори за замовчуванням. Це станеться до того, як тіло конструктора виконається.

Приклад.

```
#include <iostream>
class A
{
public:
   A() { std::cout << "A\n"; }
};
class B
{
private:
   A m_a; // B містить A, як змінну-член
public:
   B() { std::cout << "B\n"; }
};
int main()
{
   B b;
   return 0;
}
```

Результат виконання програми:

```
options | compilation | executio
A
B
```

При створенні змінної b викликається конструктор B(). Перш ніж тіло конструктора виконається, m_a ініціалізується, викликаючи конструктор за замовчуванням класу A. Таким

чином, виведеться A. Потім керування повернеться назад до конструктора B, і тіло конструктора B почне своє виконання.

В цьому є сенс, тому що конструктор B() може захотіти використати змінну m_a, тому спочатку потрібно ініціалізувати m_a!

**Ключове слово this**

Ключове слово *this представляє покажчик на поточний об'єкт даного класу.

this — ключове слово у деяких об'єктно-орієнтованих мовах програмування, яке використовують у методах екземплярів класу для посилання на об'єкт класу.

Відповідно через це ми можемо звертатися всередині класу до будь-яких його членів.

Приклад.

```
#include <iostream>
class Point
{public:
Point(int x, int y)    {        this->x = x;         this->y = y;    }
void showCoords()
{
std::cout << "Coords x: " << this->x << "\t y: " << y << std::endl;    }
private:
int x;
int y;};
int main()
{    Point p1(20, 50);
p1.showCoords();
return 0;
}
```

У даному випадку визначено клас Point, який визначає крапку на площині. І для зберігання координат цієї крапки у класі визначено змінні x та y.

Для звернення до змінних використовується покажчик this. Причому після this ставиться не точка, а стрілка ->.

Для звернення до членів класу навряд чи знадобиться ключове слово this. Але воно може бути необхідним, якщо параметри функції або змінні, які визначаються всередині функції, називаються так само як і змінні класу.

Наприклад, щоб у конструкторі розмежувати параметри та змінні класу якраз і використовується покажчик this.

За допомогою this можна повертати поточний об'єкт класу:

Приклад 4.

```
#include <iostream>
class Point
{
public:
Point(int x, int y)
{   this->x = x;
this->y = y;
}
void showCoords()
{
std::cout << "Coords x: " << x << "\t y: " << y << std::endl;
}
Point &move(int x, int y)
{  this->x += x;       this->y += y;       return *this;
}
private:
```

```
int x;
int y;
};
```

Приклад 4. Продовження

```
int main()
{
Point p1(20, 50);
p1.move(10, 5).move(10, 10);
p1.showCoords(); // x: 40  y: 65
return 0;
}
```

Тут метод move за допомогою покажчика this повертає посилання на об'єкт поточного класу, здійснюючи умовне переміщення точки.

Тут важливо відзначити повернення не просто об'єкта Point, а посилання на цей об'єкт.

Таким чином, ми можемо по ланцюжку для того самого об'єкта викликати метод move: p1.move(10, 5).move(10);

**Успадкування**

**Успадкування класів** дозволяє створювати похідні класи (класи спадкоємці), взявши за основу всі методи і елементи базового класу (класу батька).

Об'єкти похідного класу вільно можуть використовувати все, що створено і налагоджено в базовому класі.

При цьому, ми можемо в похідний клас дописати необхідний код для удосконалення програми: додати нові елементи, методи і т.д.. Базовий клас залишиться недоторканим.

**Розглянемо простий код програми.**

У цій програмі створені два класи: базовий – FirstClass і похідний від нього – SecondClass.

Код програми розміщено на наступних трьох слайдах.

Приклад.

```
#include <iostream>
using namespace std;
class FirstClass
{protected:
    int value;
public:
FirstClass()
{value = 0;
}
FirstClass(int input)
{value = input;
}
void show_value()
{cout << value << endl;
}
};
        class SecondClass:public FirstClass// клас-нащадок
    { public:
    SecondClass():FirstClass()
    // конструктор класу SecondClass викликає конструктор класу FirstClass
    {
    }
    SecondClass(int inputS):FirstClass(inputS)
    // передає inputS в конструктор с параметром класу FirstClass
    {
```

```
}
void ValueSqr()
// метод, який обчислює value*value. Без специфікатора доступу protected ця функція не змогла б змінити значення value
{value *= value;
}
};
int main()
{
FirstClass F_object(3);// об'єкт  базового класу
cout<<"value F_object = ";
F_object.show_value();
SecondClass S_object(4);// об'єкт класу-нащадка
cout <<"value S_object =";
S_object.show_value();// виклик методу базового класу
S_object.ValueSqr();// виклик методу класа-нащадка, value*value
cout<<" square value S_object =";
S_object.show_value();
//F_object.ValueSqr(); - помилка
// базовый класс не має доступу до методів класу-нащадка
cout<<endl;
return 0;
}
```

Результат роботи програми:

```
value F_object = 3
value S_object =4
square value S_object =16
```

Специфікатор доступу **protected** відрізняється від **private** тим, що дозволяє доступ до елементів базового класу з похідних класів.

Якби елемент **value** перебував у полі **private**, то доступ до нього був би закритий і ми б не могли змінити його значення через об'єкт класу **SecondClass**, використовуючи функцію ValueSqr().

Якщо ви створюєте клас, який надалі плануєте використовувати, як базовий, то оголошуйте в ньому поле **protected** замість **private.** Інакше об'єкти похідного класу не зможуть звертатися до елементів базового.

## 22. Функції роботи з кирилицею, датою і часом

Як використовувати кирилицю в програмах С++?

Проста відповідь:

Щоб використовувати кирилицю в програмах на мові С++, вам необхідно підключити заголовковий файл Windows.h:

```
#include <Windows.h>
```

І прописати два наступних рядки в функції main():

```
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
```

Якщо програма під час виведення на консоль використовує кирилицю, ми можемо зіткнутися з ситуацією, коли замість кириличних символів відображаються незрозумілі знаки. Особливо це актуально для Windows. І в цьому випадку необхідно явно задати поточну локаль (культуру) для виведення символів. У мові С++ можна зробити з допомогою вбудованої функції setlocale().

```
setlocale(LC_ALL, "");
```

Як кодування текстового файлу в цьому випадку має використовуватися кодування **ANSI** (American National Standards Institute) або Windows-1251, але не UTF-8.

На деяких платформах, наприклад Ubuntu, ми можемо не зіткнутися з подібною проблемою. І в цьому випадку виклик функції setlocale просто не вплине.

**Локаль** – це набір параметрів: набір символів, мова користувача, країна, часовий пояс та ін. Локаль необхідна для швидкої настройки призначеного для користувача інтерфейсу, в залежності від географічного положення.

В С ++ є функція **setlocale** (), яка виконує перекодування символів відповідно до необхідного мовою. Ця функція визначена в заголовку **<clocale>.**

Переробимо програму, яка передає повідомлення «Кирилиця в консолі» в командний рядок Windows:

Приклад.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main (int argc, char * argv [])
{   cout << "Кирилиця в консолі \ n"; // передаємо текстове
повідомлення в командний рядок windows
system ("pause");
return 0;}
```

**Результат: незрозуміла послідовність символів.**

Приклад.

```
#include "stdafx.h"
#include <iostream>
#include <clocale>
using namespace std;
int main (int argc, char * argv [])
{   setlocale (LC_CTYPE, "rus"); // виклик функції настройки локалі
cout << "Кирилиця в консолі \ n";
system ("pause");
return 0;}
```

**Результат: повідомлення «Кирилиця в консолі» в командному рядку Windows**

Функція **setlocale**() має два параметри,

перший параметр - тип категорії локалі, в нашому випадку LC_TYPE – набір символів,

другий параметр – значення локалі.

Таким чином, спочатку встановлюємо потрібну локаль, в нашому випадку – "rus", після чого, можна використовувати кирилицю, для виведення повідомлень в консоль.

Замість аргументу "rus" можна також писати "Russian", або залишати порожні подвійні лапки, тоді набір символів буде такою ж як і в ОС.

**Приклад.**

**setlocale (LC_CTYPE, "rus");**

Функція setlocale() працює тільки з потоком виводу.

Якщо ж використовувати потік введення, то там буде все та ж незрозуміла абракадабра.

Переробимо вже розібрану нами програму, таким чином, щоб рядок зберігався в змінну, після чого, виводився на екран:

```
#include "stdafx.h"
#include <iostream>
#include <clocale>
using namespace std;
int main (int argc, char * argv [])
{ setlocale (LC_CTYPE, "rus"); // не працює з потоком введення
char string [20];
cin >> string; // вводимо рядок кирилицею
```

(СТРОКА ЗБЕРЕЖЕТЬСЯ В ЗМІННУ НЕКОРЕКТНО)
cout << "\ n, вивод:" << string << endl; // вивод рядка
system ("pause");
return 0; }

Функція **setlocale()** працює тільки з потоком виводу.

У файлі <windows.h> містяться прототипи функцій **SetConsoleCP()** і **SetConsoleOutputCP()**, які використовуються для введення/виведення кирилиці.

Аргументом цих функцій є ідентифікатор кодової таблиці, потрібна нам таблиця win - **cp 1251**.

Функція **SetConsoleCP ()** встановлює потрібну кодову таблицю, на потік введення, тоді як функція **SetConsoleOutputCP ()** встановлює потрібну кодову таблицю, на потік виведення.

**Функції роботи з датою та часом**

- size_t Інтегральний тип даних
- clock_t Тип даних для представлення тактів. Повертається функцією clock (). Зазвичай визначений як int або long int
- time_t Тип даних часу. Повертається функцією time (). Зазвичай визначений як int або long int.
- struct tm Структура часу. Нелінійне, дискретне календарне уявлення часу

CTime (time.h) — заголовний файл стандартної бібліотеки мови програмування СІ, що містить типи і функції для роботи з датою й часом.

Структура tm має вигляд:

```
struct tm
{Int tm_sec; // секунди після хвилин [0,59]
int tm_min; // хвилини після годин [0,59]
int tm_hour; // години після півночі [0,23]
int tm_mday; // день місяця [1,31]
int tm_mon; // місяць року (січень = 0) [0,11]
int tm_year; // рік (1900 рік = 0)
int tm_wday; // день тижня (НД = 0) [0,6]
int tm_yday; // день року (1 січня = 0) [0,365]
int tm_isdst;}; // прапор переходу на літній час (> 0 - вкл.)
```

**Приклад.**

Зазначемо, що параметр **ltime** вказує на структуру, яка зберігає дані в такому форматі 10:00:00 AM, Jan 2, 1994.

Тоді наступний фрагмент програми виведе «It is now 10 AM».

```
strftime (str, 100, "It is now %H %p", ltime);
printf (str);
```

**Константи:**

**CLOCKS_PER_SEC**

Визначає кількість тактів системного годинника в секунду. Використовується для перерахунку величини, що повертається функцією clock (), в секунди.

**CLK_PER_SEC**

Альтернативне ім'я константи CLOCKS_PER_SEC, що використовується в деяких бібліотеках.

## 24 Препроцесор

**Препроцесор** найкраще розглядати як окрему програму, яка виконується перед компіляцією. При запуску програми, препроцесор переглядає код зверху вниз, файл за файлом, в пошуку директив.

**Директиви** - це спеціальні команди, які починаються з символу # і НЕ закінчуються крапкою з комою. Є кілька типів директив, які ми розглянемо нижче.

Ви вже бачили директиву #include в дії.

Коли ви записуєте ***#include <файл>***, препроцесор копіює вміст файлу, що підключається, в поточний файл відразу після рядка з #include. Це дуже корисно при використанні функцій та складних типів даних в програмі,наприклад, стандартних або попередньо визначених.

Вихідний код програми на С (або С + +) може містити різні інструкції компілятору. Не будучи частиною мови, директиви препроцесора розширюють сферу застосування мови.

Стандарт ANSI C визначає такі директиви препроцесора:

#if
#ifdef
#ifndef
#else
#elif
#endif
#include
#define
#undef

Усі директиви препроцесора починаються зі значка #, і кожна директива має бути у своєму власному рядку.

Директива #include (пер. з анг. "включати") має дві форми:

**#include <filename>**
**#include "filename"**

#include <filename> повідомляє препроцесору шукати файл в системних папках. Частіше за все ви будете використовувати цю форму при підключенні заголовочних файлів із стандартної бібліотеки С ++.

#include "filename" повідомляє препроцесору шукати файл в поточному каталозі (папці) проекту. Якщо його там не виявиться, то препроцесор почне перевіряти системні папки та будь-які інші, які ви вказали в налаштуваннях вашого IDE. Ця форма використовується для підключення заголовків(назв) власних файлів програміста або файлів користувача. *Інтегроване середовище розробки (англ. «IDE» від «Integrated Development Environment») - це програмне забезпечення, яке містить все необхідне для розробки, компіляції, лінкінга і налагодження коду.*

Файли, що підключаються, також можуть мати директиви #include. Якщо це має місце, то говорять про вкладені підключення. Наприклад, наступна програма підключає файл, який, у свою чергу, підключає інший файл:

```
/* файл програми */
#include <stdio.h>
int main(void)
{
#include "one"
return 0;
}
/* файл one, що підключається */
printf("This is from the first include file.\n");
#include "two"
/* файл two, що підключається */
printf("This is from the second include file.\n");
```

Директива **#define** (пер. з анг. "визначити") визначає ідентифікатор і послідовність символів, якою буде заміщатися цей ідентифікатор при його виявленні в тексті програми. Ідентифікатор також називають іменем макросу, а процес заміщення називають підстановкою макросу.

Стандартний вид директиви наступний:

**#define ідентифікатор послідовність_символів**
**або**

**#define і’мя_макросу текст_заміна**

Якщо послідовність_символів не вміщується в одному рядку, то його можна продовжити на наступному рядку, помістивши в кінці рядка зворотний слеш.

**#define LONG_STRING "Тим є дуже довгим" **
**string that is used as an example."**

*Є два основних типи макросів: макроси-функції і макроси-об'єкти.*

*Макроси-функції поводяться як функції і використовуються в тих же цілях. Ми не будемо зараз їх обговорювати, тому що їх використання, як правило, вважається небезпечним, і майже все, що вони можуть зробити, можна здійснити за допомогою простої (лінійної) функції.*

**#define идентифікатор текст_заміна**

Коли препроцесор зустрічає директиву **#define идентифікатор послідовність_символів**, то будь-яка подальша поява ідентифікатора замінюється на послідовність символів.

Директиву такого формату ще називають макросом-об'єктом з текст_заміною.

Ідентифікатор зазвичай пишеться великими літерами з символами підкреслення замість пробілів.

Розглянемо наступний фрагмент коду:

```
#define MY_FAVORITE_NUMBER 9
.
cout << "My favorite number is: " << MY_FAVORITE_NUMBER <<endl;
```

Результат:

My favorite number is: 9

Макроси-об'єкти також можуть бути визначені без послідовність_символів (текст_заміна).

Формат:

#define ідентифікатор

Приклад:

#define USE_YEN

Будь-яка подальша поява ідентифікатора USE_YEN видаляється і замінюється «нічим» (порожнім місцем).

Це може здатися досить марним, однак, це не основне призначення подібних директив. На відміну від макросів-об'єктів з текст_заміна, ця форма макросів вважається корректною.

Програмісти часто використовують великі літери для визначення ідентифікаторів. Ця угода допомагає будь-якій людині, яка читає програму, дізнатися, що вона має справу з макросом.

Дуже часто макроси використовують для визначення чисел, які використовуються в програмі.

Наприклад, програма може визначати масив та мати кілька процедур для роботи з ним. Замість того, щоб жорстко задавати розмірність масиву, краще визначити макрос, що відповідає розмірності масиву, і використовувати його в тих місцях, де це потрібно. Таким чином, якщо необхідно змінити розмір масиву, єдине, що потрібно зробити, це змінити оператор #define і перекомпілювати програму.

Приклад:

```
#define MAX_SIZE 100
/*...*/
float balance [MAX_SIZE];
/*...*/
float temp[MAX_SIZE];
```

Для зміни розмірів обох масивів просто змінимо визначення MAX_SIZE.

Директива **#define** має ще одну властивість: макрос може мати аргументи. При зустрічі такого макросу аргументи макросу заміщатимуться реальними аргументами програми. Такий тип макросу називається макрос типу функція.

Приклад:

```
#include <iostream>
using namespace std;
#define SYM(a,b) a+b
int main(void)
{
int x, y;
x = 10;
y = 20;
cout<<"Symma a+b="<< SYM(x,y);
return 0;
}
```

Існує кілька директив умовної компіляції, що дозволяють змінювати порядок компіляції програми залежно від стану системи. Цей процес називається умовною компіляцією і широко використовується при розробці комерційного програмного забезпечення, що надає та підтримує багато різних версій програми

#if, #endif

#else, #elif

#ifdef та #ifndef

**#if, #endif,** #else

Якщо після #if константний_вираз набуває значення ВЕРНО, то код між #if і #endif компілюється, інакше код пропускається. Директива #endif використовується для позначення кінця блоку #if.

Формат директиви **#if** наступний:

```
#if константний_вираз послідовність операторів
#endif
```

Приклад:

```
#include <stdio.h>
#define MAX 100
int main(void)
{
#if MAX>99
printf("Compiled for array greater than 99.\n");
#endif
return 0;
}
```

Константний вираз після #if обчислюється на етапі компіляції, отже, він має містити раніше ініціалізовані ідентифікатори і константи, а не змінні.

Робота **#else** багато в чому схожа на роботу оператора else - вона надає альтернативний варіант, якщо константний_вираз #if має значення FALSE. Попередній приклад можна розширити так:

```
/* Простий приклад з #if / #else */
#include <stdio.h>
#define MAX 10
int main(void)
{
# if MAX>99
printf("Compiled for array greater than 99.\n");
#else
printf("Compiled for small array.\n");
```

#endif
return 0;
}
#ifdef
#ifndef
Формат
#ifdef имя_макроса последовательность операторов
#endif

Якщо ім'я макросу визначено раніше в операторі #define, послідовність операторів, що стоять між #ifdef і #endif, буде компілюватися.

Директива #ifdef (англ. «If defined» = «якщо визначено») дозволяє препроцесору перевірити, чи було значення раніше #define. Якщо так, то код між #ifdef і #endif скомпілюється. Якщо немає, то код буде проігнорований.

**Формат**
#ifndef ім'я_макросу послідовність операторів
#endif

Якщо ім'я макросу не визначено раніше в операторі #define, послідовність операторів, що стоять між #ifndef і #endif, буде компілюватися.

Директиви препроцесора дозволяють визначити, за яких умов код буде компілюватися, а при яких - ні.

**#elif** означає «інакше якщо» і використовується для побудови драбинки if-else-if з метою визначення різних опцій компіляції. За #elif слід константний вираз. Якщо вираз істинно, то блок коду компілюється та інші вирази не перевіряються. В іншому випадку розглядається наступний блок. Формат **#elif** наступний:

```
#if вираз
послідовність операторів
#elif вираз 1
послідовність операторів
#elif вираз 2
послідовність операторів
#elif вираз 3
послідовність операторів
#elif вираз 4
послідовність операторів
...
#elif вираз N
послідовність операторів
#endif
```

Приклад.

Макрос ACTIVE_COUNTRY використовується для визначення грошової одиниці.

```
#define US 0
#define ENGLAND 1
#define FRANCE 2
#define ACTIVE_COUNTRY US
#if ACTIVE_COUNTRY==US
char currency[] = "dollar";
#elif ACTIVE_COUNTRY==ENGLAND
char currency[] = "pound";
#else
char currency[] = "franc";
#endif
```

Після того, як препроцесор завершить своє виконання, все ідентифікатори (наприклад, визначені за допомогою #define) з цього файлу викидаються.

Це означає, що директиви діють тільки з точки визначення і до кінця файлу, в якому вони визначені.

Директиви, зазначені в одному файлі коду, не впливають на директиви всередині інших файлів цього ж проекту.

**25. Динамічний розподіл пам'яті**

Спосіб збільшення розміру статичного масиву

Якщо розмір виділеної пам'яті не можна задати заздалегідь, то для збільшення розміру масиву при введенні наступного значення необхідно виконати наступні дії:

- виділити блок пам'яті розмірності n + 1 (на 1 більше поточного розміру масиву);
- скопіювати всі значення, що зберігаються в масиві під знов виділену область пам'яті;
- звільнити пам'ять, виділену раніше для зберігання масиву;
- перемістити покажчик початку масиву на початок знову виділеної області пам'яті;
- доповнити масив останнім введеним значенням.

Всі перераховані вище дії (крім останньої) виконує функція:

**void * realloc (void * ptr, size);**

**ptr** – покажчик на блок раніше виділеної пам'яті функціями **malloc(), calloc()** або **realloc()** для переміщення в нове місце. Якщо цей параметр дорівнює NULL, то виділяється новий блок, і функція повертає на нього покажчик.

**size** – новий розмір, в байтах, що виділяється блоку пам'яті. Якщо size = 0, раніше виділена пам'ять звільняється і функція повертає нульовий покажчик, ptr встановлюється в NULL.

Розмір блоку пам'яті, на який посилається параметр ptr змінюється на size байтів. Блок пам'яті може зменшуватися або збільшуватися в розмірі.

Вміст блоку пам'яті зберігається навіть якщо новий блок має менший розмір, ніж старий. Але відкидаються ті дані, які виходять за рамки нового блоку.

Якщо новий блок пам'яті більше старого, то вміст знову виділеної пам'яті буде невизначеним.

Функція realloc() змінює величину виділеної пам'яті, яку вказує ptr, на нову величину, задану параметром newsize.

Величина newsize задається в байтах і може бути більшою або меншою за оригінал.

Повертається покажчик на блок пам'яті, оскільки може виникнути необхідність перемістити блок у разі зростання його розміру.

У такому разі вміст старого блоку копіюється в новий блок і інформація не втрачається.

Якщо вільної пам'яті недостатньо виділення в купі блоку розміром newsize, то повертається **нульовий покажчик.**

Наступна програма виділяє 17 байт пам'яті, копіює рядок “This is 16 chars" в цю область, а потім використовує функцію realloc(), щоб збільшити розмір блоку до 18 байт і помістити в кінці крапку.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(void)
{char *p;
p = (char *) malloc(17);
if(p==0) {printf("Allocation error.");
exit (1);}
strcpy(p, "This is 16 chars");
p = (char *) realloc (p,18);
if(p==0) {printf("Allocation error.");
exit (1);}
```

```
strcat (p, ".");
printf("%s",p);
free(p);
return 0;}
```

Функція exit(), що знаходиться у стандартній бібліотеці stdlib.h, викликає негайне закінчення програми. Функція exit() зупиняє виконання програми та форсує повернення в операційну систему.

void exit (int статус);

Значення статусу повертається до операційної системи.

Для індикації коректності завершення роботи exit() зазвичай викликається з аргументом 0.

**Операція new**

У мові C++ використання функцій malloc() або calloc() не має сенсу.

Операції **new** та **delete** виконують динамічний розподіл та скасування розподілу пам'яті з більш високим пріоритетом,ніж функції malloc і free

Вільна пам'ять у програмах мовою C++ виділяється за допомогою операції **new**.

void * new (size_t);

void delete(void*);

size_t - беззнаковий цілісний тип, визначений у <stddef.h>.

В цьому випадку виділиться пам'ять, достатня для розміщення об'єкта такого типу та в результаті буде повернено покажчик на виділену пам’ять.

Приклад:

int * pi = new int;

Тут виділено пам'ять для об'єкта типу int. Тип значення, що повертається «покажчик на int». Порожній покажчик (0) означає невдале завершення операції. Цей випадок виникає, коли недостатній обсяг або занадто велика фрагментація області пам'яті, що розподіляється. Слід, однак, відзначити, що на відміну від функції malloc оператор new не очищає виділену пам'ять і містить "сміття".

Операція **new** зручніша тим, що вона як параметр отримує тип створюваного об'єкта, а не його розмір та повертає покажчик на заданий тип, не вимагаючи приведення типу.

C++ допускає явну ініціалізацію пам'яті, що виділяється, для об'єкта будь-якого типу

Приклад.

int * pi = new int (100); // *pi == 100 - значення, записане у виділену динамічну пам'ять.

При оголошенні масиву в пам'яті виділяється блок, розмір якого залежить від розмірності масиву та типу його елементів. Розмірності масиву зазначаються у квадратних дужках.

При створенні за допомогою **new** багатовимірних масивів слід зазначати всі розмірності масиву.

Приклад.

mattr = new int[10][10][10]; // допустимо

mattr1 = new int[10][][10]; // не допустимо

Розмірність може бути виразом довільної складності. Операція **new** повертає покажчик на перший елемент масиву.

Приклад.

int i = 200;

char *ps = new char[i*2]; // ps вказує на масив із 400 елементів типу char.

Задача. Виділення та звільнення пам'яті для массиву покажчиків на строки двовимірного масиву з використанням оператора **new.**

Приклад.

```
// Виділення пам'яті для масиву покажчиків
a = new int * [nn];
for (int j = 0; j < nn; j++)
a[j] = new int[mm];//виділення пам'яті для строк двовимірного масиву
```

Коректна робота з покажчиками полягає в тому, щоб виділити пам'ять у динамічній області та встановити покажчик на цю пам'ять.

Слід пам'ятати, якщо цю пам'ять неможливо звільнити явно, вона не звільниться і після закінчення програми (це називається «сміттям»).

До пам'яті, відведеної в динамічній області, немає іншого доступу, крім через покажчик, який її адресує. Тому якщо цьому покажчику буде присвоєно будь-яка інша адреса пам'яті, то та пам'ять, на яку він вказував, буде для програми втрачена і ніколи не звільниться

Приклад.

```
//відводимо пам'ять у динамічній області
int * ia = new int [100];
int * ia2 = new int [100]; // ще відводимо пам'ять
ia = ia2; // ia вказує на ту саму пам'ять, що і ia2, а пам'ять,
//на яку вказував ia до присвоєння недоступна,
// і стає «сміттям»
```

**Операція delete**

Пам'ять, виділена за допомогою операції new, буде зайнята доки програміст явно її не звільнить. Для явного звільнення цієї пам'яті використовується операція delete, яка застосовується до покажчика, що адресує динамічний об'єкт.

Наприклад:

```
int * pi = new int; // пам'ять відведено
delete pi;
// Пам'ять звільнена
```

Тут пам'ять, займана *pi, знову повертається у вільну пам'ять і згодом знову може бути виділено за допомогою new.

Для звільнення пам'яті, відведену під масив необхідно вставляти пару порожніх квадратних дужок між delete та покажчиком:

```
int * parr = new int [100];
delete [] parr;
```

Для звільнення пам'яті, відведеної масиву об'єктів, які є класами, можна використовувати оператор delete без квадратних дужок:

```
delete parr; // аналогічно delete [] parr;
```

Операція delete повинна застосовуватись лише до пам'яті, виділеної за допомогою операції new.

Застосування цього оператора до пам'яті, виділеної за допомогою іншого оператора, призведе до помилки.

Однак, застосування delete до нульового покажчика не вважається помилкою і просто ігнорується.

Особливо небезпечне повторне застосування delete до того самого покажчика.

Існує спеціальна форма оператора new, яка має назву Placement new. Даний оператор не виділяє пам'ять і має своїм аргументом адресу, яка виділена якимось чином (наприклад, для стеку чи через malloc).

Відбувається розміщення (ініціалізація) об'єкта шляхом виклику конструктора і об'єкт створюється в пам'яті за вказаною адресою.

Програміст може розташувати об'єкт у вільній пам'яті за певною адресою. Для цього викликається операція new у такому вигляді:

new (адреса розташування) тип;

Тут адреса розташування має бути покажчиком. Для того, щоб використовувати цей варіант операції new, має бути підключений заголовний файл new.h.

Ця можливість дозволяє програмісту попередньо виділяти пам'ять, яка пізніше буде містити об'єкти, створені за допомогою цієї форми операції new, що може виявитися надійнішим способом їх розміщення.

Висновок

1) для розміщення об'єктів у вільній пам'яті використовується операція new, а для визволення - операція delete.

2) якщо після виклику операції new не вистачає пам'яті для розміщення об'єкта, вона за умовчанням поверне 0.

3) якщо застосувати delete до нульового покажчика, то ця дія проігнорується та помилки не буде.

4) якщо застосувати delete до покажчика на пам'ять, яка вже була звільнена раніше, то програма, швидше за все, зависне.

### 26. Успадкування

Успадкування є найпотужнішим інструментом ООП і застосовується для наступних взаємопов'язаних цілей:

- виключення з програми фрагментів коду , що повторюються;
- спрощення модифікації програми;
- спрощення створення нових програм на основі існуючих.

Крім того, успадкування є єдиною можливістю використовувати об'єкти, вихідний код яких недоступний, але в які потрібно внести зміни.

Синтаксис успадкування

*Ключи доступа* class имя : [private | protected | public] базовый_класс { тело класса };

Наприклад:

```
class A { ... };
class B { ... };
class C { ... };
class D: A, protected B, public C { ... };
```

Повний синтаксис:

```
class Derived : [virtual] [access-specifier] Base
{
  // member list
};
class Derived : [virtual] [access-specifier] Base1,
  [virtual] [access-specifier] Base2, . . .
{
  // member list
};
```

Після тега (імені) класу ставиться двокрапка і список базових специфікацій. Названі в такий спосіб базові класи, ймовірно, були оголошені раніше. Базові специфікації можуть містити опис доступу, який є одним із ключових слів **public, protected** або **private**. Ці специфікатори доступу відображаються перед іменем базового класу та застосовуються лише до базового класу. Ці специфікатори контролюють дозвіл похідного класу щодо використання членів базового класу.

Якщо описувач доступу не вказаний, то за умовчанням застосовується приватний доступ **private**.

Базові специфікації можуть містити ключове слово virtual, що вказує на віртуальне успадкування. Це ключове слово може відображатися до або після опису доступу, якщо такі є. Якщо використовується віртуальне спадкування, базовий клас називається віртуальним базовим класом.

Можна визначити кілька базових класів, розділивши їх комами. Якщо вказано один базовий клас, модель успадкування є **одиничним** успадкуванням. Якщо задано кілька базових класів, модель успадкування називається **множинним** успадкуванням.

У класі-спадкоємці можна описувати нові поля та методи та перевизначати існуючі методи.

Перевизначити методи можна декількома способами:

- якщо будь-який метод у класі-спадкоємці повинен працювати зовсім по-іншому, ніж у батьківському класі, метод описується в спадкоємці наново. При цьому він може мати інший набір аргументів;
- якщо потрібно внести додавання в метод предка, то у відповідному методі спадкоємця поряд з описом додаткових дій виконується виклик методу предка за допомогою операції доступу до області видимості.

Якщо у програмі планується працювати одночасно з різними типами об'єктів ієрархії або планується додавання до ієрархії нових об'єктів, метод оголошується як віртуальний за допомогою ключового слова virtual.

Усі віртуальні методи ієрархії з одним і тим самим ім'ям повинні мати однаковий перелік аргументів.

Іншими словами:

– **private** елементи базового класу у похідному класі недоступні незалежно від ключа. Звернення до них може здійснюватись тільки через методи базового класу.

– елементи **protected** при успадкуванні з приватним ключем стають у похідному класі private, в інших випадках права доступу до них не змінюються.

– доступ до елементів **public** при успадкуванні стає відповідним ключем доступу.

Конструктори не успадковуються, тому похідний клас повинен мати власні конструктори.

**Порядок виклику конструкторів:**

– якщо в конструкторі нащадка явний виклик конструктора предка відсутній, автоматично викликається конструктор предка по замовчуванням.

– для ієрархії, що складається з кількох рівнів, конструктори предків викликаються з самого верхнього рівня. Після цього виконуються конструктори тих елементів класу, які є об'єктами в порядку їх оголошення у класі, а потім виконується конструктор класу.

– у разі кількох предків їхні конструктори викликаються в порядку їх оголошення.

Деструктори не успадковуються.

Якщо деструктор у похідному класі не описаний, він формується автоматично та викликає деструктори всіх базових класів.

У деструкторі похідного класу не потрібно явно викликати деструктори базових класів, це буде зроблено автоматично.

Для ієрархії, що складається з кількох рівнів, деструктори викликаються в порядку, суворо зворотному виклику конструкторів: спочатку викликається деструктор класу, потім деструктори елементів класу, а потім деструктор базового класу.

**Віртуальна функція**

Віртуальна функція в мові C++ - це особливий тип функції, яка, при її виклику, виконує найбільш дочірній метод, який існує між батьківським і дочірніми класами.

Ця властивість ще відома як поліморфізм.

Дочірній метод викликається тоді, коли збігається сигнатура (ім'я, типи параметрів) та тип повернення дочірнього методу із сигнатурою та типом повернення методу батьківського класу. Такі методи називаються перевизначенням (або «перевизначеним методом»).

Щоб зробити функцію віртуальною, необхідно просто вказати ключове слово virtual перед оголошенням функції.

Віртуальні методи

```
virtual void draw(int x, int y, int scale, int position);
monstr *r, *p;
// Створюється об'єкт класу monstr
r = new monstr;
// Створюється об'єкт класу daemon
```

```
p = new daemon;
// Викликається метод monstr::draw
r->draw(1, 1, 1, 1);
// Викликається метод daemon::draw
p->draw(1, 1, 1, 1);
// Обхід механізму віртуальних методів
p-> monstr::draw(1, 1, 1, 1);
```

Опис та використання віртуальних методів

Якщо у предку метод визначено як віртуальний, метод, визначений у нащадку з тим же ім'ям та набором параметрів, автоматично стає віртуальним, а з різним набором параметрів - простим.

Віртуальні методи успадковуються, тобто перевизначати їх у нащадку потрібно тільки при необхідності виконання різних дій.

Не можна змінити права доступу при перевизначенні.

Якщо віртуальний метод перевизначено у нащадку, об'єкти цього класу можуть отримати доступ до методу предка за допомогою операції доступу до області видимості.

Віртуальний метод не може оголошуватись з модифікатором static, але може бути оголошений як дружній.

Якщо в класі вводиться оголошення віртуального методу, він має бути визначений хоча б як суто віртуальний.

Чисто віртуальні методи:

- містить ознаку = 0 замість тіла:

virtual void f(int) = 0;

- повинен перевизначатися у похідному класі.

Клас, що містить хоча б один суто віртуальний метод,називається **абстрактним**:

абстрактний клас не можна використовувати при явному наведенні

типів, для опису типу параметра та типу значення функції, що повертається;

допускається оголошувати покажчики та посилання на абстрактний клас,

якщо при ініціалізації не потрібно створювати тимчасовий об'єкт;

якщо клас, похідний від абстрактного, він також є абстрактним.

**Множинне успадкування**

Множинне успадкування застосовується для того, щоб забезпечити похідний клас властивостями двох чи більше базових.

Найчастіше один із цих класів є основним,інші забезпечують деякі додаткові властивості, тому вони називаються класами підмішування.

За можливостями класи підмішування мають бути віртуальними та створюватися за допомогою конструкторів без параметрів, що дозволяє уникнути багатьох проблем, які виникають, коли у основного класа є загальний предок.

**Абстрактні класи**

Абстрактні класи використовуються як узагальнені концепції, на основі яких можна створювати більш конкретні похідні класи. Неможливо створити об'єкт абстрактного типу класу. Однак можна використовувати покажчики та посилання на абстрактні типи класів.

Можна створити абстрактний клас, оголосивши принаймні одну чисту віртуальну функцію-член. Це віртуальна функція, оголошена синтаксисом чистого описувача (= 0). Класи, похідні від абстрактного класу, мають реалізовувати суто віртуальну функцію; інакше вони також будуть абстрактними.

Розглянемо приклад, поданий у віртуальних функціях. Клас Account створений для того, щоб надавати спільні функції, але об'єкти типу Account мають надто загальний характер для практичного застосування. Це означає, що Account є хорошим кандидатом для абстрактного класу:

Приклад.

```
class Account {
public:
```

```
    Account( double d );   // Constructor.
    virtual double GetBalance();   // Obtain balance.
    virtual void PrintBalance() = 0;   // Pure virtual function.
  private:
    double _balance;
  };
```

Функцію PrintBalance оголошено зі специфікатором суто віртуальної функції pure (= 0).

## 27. Робота з файлами

Для роботи з файлами необхідно підключити заголовний модуль **<fstream>**.

В модулі **<fstream>** реалізовані класи **ifstream** - файлове введення і **ofstream** - файлове виведення, **fstream** – в\в.

Файлове введення / виведення аналогічне стандартному в/в. Єдина відмінність - це те, що введення / виведення буде виконуватись не з клавіатури та на екран, а в файл.

Якщо введення / виведення на стандартні пристрої виконується за допомогою **cin** і **cout**, то для організації файлового введення / виведення досить створити власні **об'єкти**, які можна використовувати аналогічно операторам **cin** і **cout**.

Робота з об'єктами класу fstream аналогічна роботі з cin та cout.

Оголошення та ініціалізація об'єктів введення та виведення:

- в класі iostream оператори cin та cout вже оголошені;
- при роботі з класом fstream об'єкти введення і виведення потрібно оголошувати.

Для роботи з файлами в програмі вводиться об'єкт.

Розглянемо на прикладі.

Необхідно створити текстовий файл і записати в нього рядок «Робота з файлами в С ++».

Для цього необхідно виконати наступні кроки:

- створити об'єкт класу ofstream;
- зв'язати об'єкт з файлом, в який буде здійснюватись запис;
- записати рядок в файл;
- закрити файл.

**ofstream / * ім'я об'єкта * /;** // об'єкт класу ofstream

Приклад. Створемо об'єкт класу ofstream для запису в файл, назвемо об'єкт – fout.

**ofstream fout;**

**Частіше використовується наступна термінологія. Для роботи з файлом створюємо файловий потік.**

**Яким чином здійснюється зв'язок потоку з файлом?**

Щоб створити файловий потік потрібно оголосити екземпляр одного з класів ifstream, ofstream або fstream.

Приклад.

ifstream is; // потік введення

ofstream os; // потік виведення

fstream fs; // потік введення/виведення

Створений потік зв'язується з файлом з допомогою функції open(), яка має різні реалізації у різних класах.

Для відкриття файла в функцію необходідно передати шлях до файлу у вигляді строки.

Функція Open дозволяє задавати режими відкриття.

Список доступних режимів відкриття файла наведені далі:

У класі ifstream функція open() має таку реалізацію

ifstream::open(const char* filename, ios::openmode mode = ios::in);

тут

*filename* – ім'я файлу, який відкривається. Ім'я може бути повним або скороченим, наприклад, “myfile.txt”;

*mode* – спосіб відкриття файлу. Цей параметр приймає значення ios::in, яке описується у функції openmode(). Значення ios::in означає, що файл відкривається тільки для введення.

У класі ofstream функція open() має наступний прототип

ofstream::open(const char* filename, ios::openmode mode = ios::out | ios::trunc);

тут

filename – ім'я файлу, який виводиться;

mode – параметр, що задає спосіб відкриття файлу. У цьому параметрі значення ios::out означає, що файл відкривається для виведення.

Значення ios::trunc означає, що попередній вміст існуючого файлу з таким самим іменем буде видалений, а довжина файлу стане рівною нуль;

| (побітове або) - способи відкриття файлу можна комбінувати між собою за допомогою операції |.

У класі fstream прототип функції open() наступний

fstream::open(const char* filename, ios::openmode mode = ios::in | ios::out);

тут

filename – ім'я файлу;

mode – параметр, що задає спосіб відкриття файлу. Як видно зі значення параметру mode, файл відкривається і для запису і для читання;

| (побітове або) - способи відкриття файлу можна комбінувати між собою за допомогою операції |.

Способи відкриття файлу можуть бути різними. Значення, що описують способи відкриття файлу, реалізовані в функції openmode().

Нижче наведено значення, які можуть повертатись функцією openmode() та пояснення до них.

Способи відкриття файлу можна комбінувати між собою з допомогою операції | (побітове АБО).

Режими відкриття файлу:

**ios::in**: файл відкритий для введення (читання). Може бути встановлений тільки для обє´кта ifstream або fstream

**ios::out**: файл відкритий для виведення (запису). При цьому старі дані будуть видалені. Може бути встановлений тільки для обє´кта ofstream або fstream

**ios::app**: файл відкритий для дозапису. Старі дані не будуть видалені.

**ios::ate**: після відкриття файла переводить покажчик в кінець файлу

**ios::trunc**: означає, що попередній вміст існуючого файлу з таким самим іменем буде видалений, а довжина файлу стане рівною нуль. Якщо відкрити потік виведення з допомогою класу ofstream увесь вміст файлу з таким самим іменем стирається

**ios::binary**: файл відкритий в бінарном режимі Існує два режими відкриття файлу: текстовий та бінарний. За замовчуванням файли відкриваються у текстовому

Приклад 1. Відкриття файлу для виведення (запис у файл)

```
ofstream outFile;
outFile.open("myfile.txt", ios::out);
```

Приклад 2. Відкриття файлу для виведення - спрощений варіант (запис у файл)

```
outFile.open("myfile.txt");
```

Приклад 3. Відкриття файлу для введення (читання з файлу)

```
ifstream inFile;
inFile.open("myfile.txt", ios::in);
```

Приклад 4. Відкриття файлу для введення - спрощений варіант (читання з файлу)

```
inFile.open("myfile.txt");
```

## 28. Функціонали array і vector в C++

Ми докладно розглядали фіксовані та динамічні масиви. Хоча вони дуже корисні і активно використовуються в мові C++, вони також мають свої недоліки:

фіксовані масиви розпадаються в покажчики, втрачаючи інформацію про свою довжину;

у динамічних масивах проблеми можуть виникнути із звільненням пам'яті та зі спробами змінити їх довжину після виділення.

Тому Стандартну бібліотеку C++ додали функціонал, який спрощує процес управління масивами:

**std::array і**

**std::vector.**

Починаючи з версії C++11 використовують std::array.

**array**

- це фіксований масив, який розпадається в покажчик під час передачі у функцію;
- визначається заголовному файлі array, всередині простору імен std.

Оголошення змінної std::array в програмі наступне:

Оголошуємо масив myarray з 4 елементів типу int

```
1 #include <array>
2
3 std::array<int, 4> myarray;
```

Подібно до звичайних фіксованих масивів, довжина std::array повинна бути встановлена під час компіляції. std::array можна ініціалізувати за допомогою списку ініціалізаторів або uniform-ініціалізації:

```
1 std::array<int, 4> myarray = { 8, 6, 4, 1 };
2 std::array<int, 4> myarray2 { 8, 6, 4, 1 };
```

На відміну від стандартних фіксованих масивів, std::array ви не можете пропустити (не вказувати) довжину масиву.

**Функції array C++**

- array  Створює об'єкт масиву.
- assign  (Застаріло. Використовуйте fill.) Замінює всі елементи.
- at  Звертається до елемента у зазначеній позиції.
- back  Звертається до останнього елемента.
- begin  Задає початок керованої послідовності.
- cbegin  Повертає постійний ітератор довільного доступу, що вказує на перший елемент масиву.
- cend  Повертає постійний ітератор довільного доступу, що вказує на передостанню позицію масиву.
- crbegin  Повертає константний ітератор, який вказує на перший елемент зверненому масиві.
- crend  Повертає постійний ітератор, який вказує на останній елемент у зворотному масиві.
- data  Отримує адресу першого елемента.
- empty  Перевіряє наявність елементів.
- end  Задає кінець керованої послідовності.
- fill  Замінює всі елементи вказаним значенням.
- front  Звертається до першого елемента.
- max_size Підраховує кількість елементів.
- rbegin  Задає початок зворотної керованої послідовності.

– rend Задає кінець зворотної керованої послідовності.
– size Підраховує кількість елементів.
– swap Змінює місцями вміст двох контейнерів

std::array - це чудова заміна стандартних фіксованих масивів. Масиви, створені за допомогою std::array, ефективніші, оскільки використовують менше пам'яті. Єдиними недоліками std::array у порівнянні зі стандартними фіксованими масивами є трохи незручний синтаксис і те, що потрібно явно вказувати довжину масиву (компілятор не вираховуватиме її за нас).

Рекомендується використовувати std::array замість стандартних фіксованих масивів у будь-яких нетривіальних задачах.

**Функціонал std::vector в C++**

Аналогічно **array** у Стандартній бібліотеці C++ є і покращена версія динамічних масивів (безпечніша і зручніша) — **std::vector**.

На відміну від std::array, який недалеко відходить від базового функціоналу звичайних фіксованих масивів, std::vector йде в комплекті з додатковими можливостями, які роблять його одним із найкорисніших та універсальних інструментів у мові C++.

Представлений у C++03, std::vector (або просто «вектор») — це той самий динамічний масив, але може сам управляти виділеної собі пам'яттю. Це означає, що ви можете створювати масиви, довжина яких визначається під час виконання, без використання операторів new і delete (явної вказівки виділення та звільнення пам'яті). std::vector знаходиться в заголовному файлі **vector**.

Клас Vector стандартної бібліотеки C++ – це шаблон класу для контейнерів послідовностей. Вектор зберігає елементи заданого типу в лінійному розташуванні та забезпечує швидкий випадковий доступ до будь-якого елемента.

Vector (std::vector<T>) — стандартний шаблон, що реалізує динамічний масив.

Шаблон vector розташований у заголовному файлі <vector>. Як і всі стандартні компоненти, він розташований у просторі назв std. Даний інтерфейс емулює роботу стандартного масиву C (наприклад, швидкий довільний доступ до елементів), а також деякі додаткові можливості, такі як автоматична зміна розміру вектора при вставці або видаленні елементів.

Усі елементи Vector мають належати одному типу. Наприклад, не можна спільно зберігати дані типів char та int в одному екземплярі вектора. Клас vector має стандартний набір методів для доступу до елементів, додавання та видалення елементів, а також отримання кількості елементів, що зберігаються.

Немає необхідності оголошувати розмір при ініціалізації

Тобто, що в неініціалізованому, що в ініціалізованому випадку не потрібно вказувати довжину масивів. Це пов'язано з тим, що std::vector динамічно виділяє пам'ять для елементів масиву запитом.

Оголошення об'єктів array, array2 та array3 через std::vector наступне:

```
#include <vector>
.
std::vector<int> array; // немає необхідності вказувати довжину при ініціалізації
std::vector<int> array2 = { 10, 8, 6, 4, 2, 1 }; // ініціалізація списком
std::vector<int> array3 { 10, 8, 6, 4, 2, 1 }; // uniform-ініціалізація (починаючи з C++11)
```

Змінити довжину стандартного динамічного масиву досить складно.

Змінити довжину std::vector так само просто, як викликати функцію resize():

```
#include <vector>
#include <iostream>
int main()
{
std::vector<int> array { 0, 1, 2 };
array.resize(7); // змінюємо довжину array на 7
std::cout << "The length is: " << array.size() << '\n';
```

```
for (auto const &element: array)
std::cout << element << ' ';
return 0;
}
```

Результат:

The length is: 7

0 1 2 0 0 0 0

Довжину вектора також можна змінити і у зворотний бік (обрізати):

```
#include <vector>
#include <iostream>
int main()
{    std::vector<int> array { 0, 1, 4, 7, 9, 11 };
array.resize(4); // змінюємо довжину array на 4
std::cout << "The length is: " << array.size() << '\n';
for (auto const &element: array)
std::cout << element << ' ';
return 0;}
```

Результат:

The length is: 4 0 1 4 7

**Функції для типу vector в C++**

assign   Видаляє вектор і копіює вказані елементи в порожній вектор.

at   Повертає посилання на елемент у заданому положенні у векторі.

back   Повертає посилання на останній елемент вектора.

begin   Повертає ітератор довільного доступу, що вказує на перший елемент у векторному.

capacity Повертає кількість елементів, які вектор може містити без виділення додаткового простору.

cbegin   Повертає постійний ітератор довільного доступу, що вказує на перший елемент у векторному.

cend   Повертає константний ітератор довільного доступу, що вказує на позицію, що йде за кінцем вектора.

crbegin Повертає константний ітератор, який вказує на перший елемент у зворотному векторі.

crend   Повертає константний ітератор, який вказує на останній елемент у зворотному векторі.

clear   Очищає елементи вектор.

data   Повертає покажчик на перший елемент у векторному.

emplace Вставляє елемент, створений на місці, у зазначене положення вектор.

emplace_back Додає елемент, створений на місці, в кінці вектора.

empty   Перевіряє, чи порожній контейнер вектор.

end   Повертає ітератор довільного доступу, який вказує на кінець вектор.

erase Видаляє елемент або діапазон елементів у векторі із заданих позицій.

front   Повертає посилання на перший елемент у векторному.

get_allocator Повертає об'єкт класу allocator, використовуваний вектор.

insert   Вставляє елемент або кілька елементів у вектор за заданою позиції.

max_size Повертає максимальну довжину вектор.

pop_back Видаляє елемент у кінці вектора.

push_back Додає елемент до кінця вектор.

rbegin   Повертає ітератор, що вказує на перший елемент у зворотному векторі.

rend   Повертає ітератор, який вказує на останній елемент у зворотний вектор.

reserve Резервує мінімальну довжину сховища для векторного об'єкта. Визначає новий розмір вектора.

shrink_to_fit Видаляє зайву ємність.

size	Повертає кількість елементів у векторі.
swap	Змінює місцями елементи двох векторів.

Оскільки змінні типу std::vector можуть самі керувати виділеною пам'яттю (що допомагає запобігти зайвому виділенню пам'яті), відстежують свою довжину і легко її змінюють, то рекомендується використовувати std::vector замість стандартних динамічних масивів.

## 29. Перевантаження операторів

Однак цілком природно спробувати розширити застосування перевантаження операторів на класи, які створені програмістом.

Наприклад, матричні обчислення стають набагато наочніше, якщо сума двох матриць записується як a+b, а не як summa(a,b). Для цього в мові C++ існує механізм перевантаження операторів.

Існують також оператори, заборонені до перевантаження. Зміна їх змісту зруйнувало би логіку програми.

До таких операторів належать
:: (оператор дозволу області видимості),
. (“точка” — оператор доступу до члена класу),
?: (тернарний оператор),
.* (доступ до розіменованого вказівника-члена класу),
sizeof, typeid, static_cast, dynamic_cast, const_cast і reinterpret_cast.

Крім того, не рекомендується перевантажувати логічні оператори && і ||, оскільки на їхні перевантажені версії не поширюється правило скорочених обчислень логічних виразів.

Нагадаємо, що це правило полягає в наступному: якщо на деякому етапі значення усього вираження стає визначеним, подальші обчислення припиняються. Оператор ?: - є ще одним оператором вибору (розгалуження)

Використовується він зазвичай у тих випадках, якщо умова та код, який треба виконати, в результаті перевірки умови дуже прості.

Наприклад, запитати у користувача хоче він продовжити працювати в програмі або хоче вийти з неї.

Формат : УМОВА? КОМАНДА 1: КОМАНДА 2;

Спочатку треба записати необхідну нам умову і за ним поставити знак питання? . Далі, у цьому ж рядку, після знака питання пишемо першу просту команду (код), яка виконуватиметься, якщо умова поверне істину (true). Після цієї команди ставимо двокрапку: і пишемо другу команду (код). Ця друга команда після двокрапки виконається тільки в тому випадку, якщо умова повертає невірно (false).

Синтаксис операторних функцій виглядає в такий спосіб.

**тип_значення_що_повертається *operator* символ_операції(параметри) { ... }**

Наприклад, операторна функція, що перевантажує операцію +, називається operator+().

Необхідно пам'ятати про обмеження, що супроводжують застосування перевантажених операторів.

1. Перевантажені функції не можуть змінити пріоритет операторів.
2. Кількість операндів фіксована: жодного, один чи два.
3. Значення операндів не можна задавати за замовчуванням

Приклад.

```
#include <iostream>
class Counter {
public:
   Counter(int sec)
   {  seconds = sec;}
   void display()
```

```
    {    std::cout << seconds << " seconds" << std::endl; }
int seconds;
                    };
 Counter operator + (Counter c1, Counter c2)
{    return Counter(c1.seconds + c2.seconds);}
int main()
{
    Counter c1(20);
    Counter c2(10);
    Counter c3 = c1 + c2;
    c3.display();   // 30 seconds
    return 0;
}
```

Ця функція перевантажує оператор додавання для типу Counter. Повертає функція також об'єкт Counter, який зберігає загальну кількість секунд. Тобто по суті тут операція додавання зводиться до складання секунд обох об'єктів.

**30. Строкові класи std::string и std::wstring**

Стандартна бібліотека C++ містить багато корисних класів, але одним із найкорисніших – це std::string.

std::string (і std::wstring) — це стандартний (строковий) клас, який дозволяє виконувати операції присвоєння, порівняння та зміни рядків.

Використовуючи такі концепції C++, як конструктори, деструктори та перевантаження операторів, std::string дозволяє створювати та маніпулювати рядками в інтуїтивно зрозумілій формі. Жодного управління пам'яттю, запам'ятовування назв функцій та значно менша ймовірність виникнення помилок/збоїв.

Весь функціонал класу std::string знаходиться в заголовному файлі string:

```
#include <string>
```

Насправді в заголовному файлі є 3 різні рядкові класи. Перший - це шаблон класу з ім'ям basic_string<>, який є батьківським класом

Далі йдуть два різновиди basic_string<>:

```
namespace std
{
    typedef basic_string<char> string;
    typedef basic_string<wchar_t> wstring;
}
```

**Ключове слово typedef**

Ключове слово **typedef** дозволяє програмісту створити псевдонім для будь-якого типу даних та використовувати його замість фактичного імені типу.

Щоб оголосити typedef (використовувати псевдонім типу) — використовують ключове слово typedef разом із типом даних, псевдонім якого створюється.

Наприклад:

```
typedef double time_t;
 // використовуємл time_t в якості псевдонима для типу double
 // Наступні два оголошення еквівалентні
double howMuch;
time_t howMuch;
```

Зазвичай до псевдонімів typedef додають закінчення _t, вказуючи таким чином, що ідентифікатором є тип, а не змінна.

**typedef basic_string<char> string;**

**typedef basic_string<wchar_t> wstring;**

– std::string використовується для стандартних ASCII-рядків (кодування UTF-8),

– std::wstring використовується для Unicode-рядків (кодування UTF-16). Вбудованого класу для рядків UTF-32 немає.

– для std::string та std::wstring весь функціонал реалізований у класі basic_string<>.

– string та wstring мають доступ до цього функціонала безпосередньо. Отже, всі функції, наведені нижче, працюють як з string, і з wstring.

– **UTF**-8 (від англ. Unicode Transformation Format, 8-bit — «формат перетворення Юнікода, 8-біт»)

**Створення та видалення:**

– конструктор - створює або копіює рядок;
– деструктор – знищує рядок
– конструктори для створення рядків із чисел;
– to_string і to_wstring (C++ 11) дозволяють отримати рядок із числа.

**Розмір та ємність:**

– capacity() — повертає кількість символів, які рядок може зберігати без додаткового перевиділення пам'яті;
– empty() — повертає логічне значення, що вказує, чи рядок є порожнім;
– length(), size() - повертають кількість символів у рядку;
– max_size() - повертає максимальний розмір рядка, який може бути виділений;
– reserve() — розширює або зменшує ємність рядка.

**Доступ до елементів:**

– [], at() – доступ до елемента за заданим індексом.

**Зміна:**

– =, assign() - надають нове значення рядку;
– +=, append(), push_back() - додають символи до кінця рядка;
– insert() — вставляє символи у довільний індекс рядка;
– clear() – видаляє всі символи рядка;
– erase() — видаляє символи за довільним індексом рядка;
– replace() — замінює символи довільних індексів рядка на інші символи;
– resize() - розширює або зменшує рядок (видаляє або додає символи в кінці рядка);
– swap() — змінює місця значення двох рядків.

**Введення/ виведення:**

– >>, getline() - зчитують значення із вхідного потоку в рядок;
– << - записує значення рядка у вихідний потік;
– c_str() — конвертує рядок у рядок C-style з нуль-термінатором наприкінці;
– copy() — копіює вміст рядка (який без нуль-термінатора) масив типу char;
– data() — повертає вміст рядка у вигляді масиву типу char, який не закінчується нуль-термінатором.

**Порівняння рядків:**

– ==, != - Порівнюють, чи є два рядки рівними/нерівними (повертають значення типу bool);
– <, <=, >, >= — порівнюють, чи є два рядки менше або більше один одного (повертають значення типу bool);
– compare() — порівнює, чи два рядки є рівними/нерівними (повертає -1, 0 або 1).

**Підрядки та конкатенація:**

– + - з'єднує два рядки;

– substr() — повертає підрядок.

**Пошук:**

– find() - шукає індекс (ітератор)першого символу / підрядка;
– find_first_of() - шукає індекс першого символу з набору символів;
– find_first_not_of() - шукає індекс першого символу НЕ із набору символів;
– find_last_of() - шукає індекс останнього символу з набору символів;
– find_last_not_of ()- шукає індекс останнього символу НЕ із набору символів;
– rfind ()— шукає індекс останнього символу/підстроки.

Ітератор (від англ. iterator — перечислювач) — інтерфейс, що надає доступ до елементів колекції (масиву або контейнера) та навігації за ними.

Строкові класи мають ряд конструкторів та деструкторів, які можна використовувати для створення рядків. Розглянемо кожен із них.

**Створення, знищення та конвертація std::string**

string::string(const string& strString, size_type unIndex)

string::string(const string& strString, size_type unIndex, size_type unLength)

Конструктори, які створюють нові рядки, які складаються з рядка strString (починаючи з індексу unIndex) та кількості символів, зазначених у unLength.

Якщо компілятор зустрічає NULL, копіювання рядка завершується, навіть якщо unLength не було досягнуто.

Якщо unLength не було вказано, всі символи, починаючи з unIndex, будуть використані.

Якщо unIndex більше розміру рядка, то викидається виняток out_of_range.

**string::string(const char *szCString)**

Конструктор, який створює новий рядок з рядка C-style szCString, що передається, аж до нуль-термінатора (але його не включає). Якщо розмір результату перевищує максимальну довжину рядка, генерується виняток length_error.

Попередження: szCString не може бути NULL.

**string::string(const char *szCString, size_type unLength)**

Конструктор, який створює новий рядок з рядка C-style szCString з кількістю символів, вказаних у unLength.

Якщо розмір результату перевищує максимальну довжину рядка, генерується виняток length_error.

**string::~string()**

Деструктор, який знищує рядок та звільняє пам'ять.

Один помітний недолік у класі std::string — відсутність можливості створювати рядки з чисел.

Найпростіший спосіб конвертувати числа в рядки - це використовувати клас **std::ostringstream**, який знаходиться в заголовному файлі **sstream**.

std::ostringstream вже налаштований для прийому різних вхідних даних: символів, чисел, рядків тощо. А за допомогою std::istringstream можна виконувати зворотну конвертацію - виводити рядки (або через оператор виведення >> або через функцію str()).

Наприклад, створимо std::string із різних вхідних даних:

```
#include <iostream>
#include <sstream>
#include <string>
template <typename T>
inline std::string ToString(T tX)
{
   std::ostringstream oStream;
   oStream << tX;
   return oStream.str();
}
```

```
int main()
{
   std::string sFive(ToString(5));
   std::string sSevenPointEight(ToString(7.8));
   std::string sB(ToString('B'));
   std::cout << sFive << std::endl;
   std::cout << sSevenPointEight << std::endl;
   std::cout << sB << std::endl;
}
```

Результат:

```
5
7.8
B
```

## Перелік використаних джерел


1. О. Васильєв. Алгоритми. – Ліра-К, 2022. – 424 с.
2. О. Васильєв. Програмування на C++ в прикладах і задачах – Ліра-К, 2019. – 382 с.
3. Bjarne Stroustrup. Programming - Principles and Practice Using C++, 2nd edition. - https://www.amazon.com/Programming-Principles-Practice-Using-2nd/dp/0321992784
4. C++17 - The Complete Guide by Nicolai M. Josuttis. – http://www.cppstd17.com
5. Основи програмування на С ++ для початківців. - https://purecodecpp.com/uk/
6. Огляд і основи мови програмування C++.- http://www.znannya.org/?view=Cpp_basics
7. Програмування мовою C++. - https://programming.in.ua/programming/c-plus-plus.html
8. Що нового у C++20: можливості та перспективи. - https://dou.ua/lenta/columns/c-plus-plus-perspectives/
9. Microsoft Visual Studio. - https://visualstudio.microsoft.com/ru/students/

Навчальний документ

# КОНСПЕКТ ЛЕКЦІЙ

з дисципліни
«Програмування»

для студентів усіх форми навчання

спеціальність 151 «Автоматизація та комп'ютерно-інтегровані технології»
(шифр і назва спеціальності)

Упорядники СЕЗОНОВА Ірина Костянтинівна

Відповідальний випусковий І.Ш. Невлюдов

*Редактор* Л.П. Гобельовська

Підписано до друку 11.06.2022 р.
Формат 60x84/16. Папір офсетний.
Гарнітура Шкільна. Друк цифровий.
Ум. друк. арк. 5,5. Тираж 100 прим. Зам. № 3205

Видавець та виготовлювач ТОВ «Друкарня Мадрид»

Через ФОП Гобельовська Л.П.

61024, м. Харків, вул. Максиміліанівська, 11 Тел.: (057) 756-53-25

Свідоцтво суб'єкта видавничої справи: Серія ДК № 4399 від 27.08.12

www.madrid.in.ua e-mail: info@madrid.in.ua