Наръчник
Как се научих да програмирам на C++ с помощта на изкуствен интелект

Шенер Юмер, 2401321044

Използван LLM: Aria

Съдържание

Предговор

Здравей, скъпи читателю!

Добре дошъл в твоето пътуване през света на програмирането, пътуване, което започна с любопитство и ще завърши с осъзнаването, че дори изкуственият интелект може да бъде учител.

Тази книга е както проектна работа, така и плод на моя опит в изучаването на `C++` , език, който е едновременно мощен и взискателен. С помощта на изкуствения интелект Aria от Opera GX, аз се научих да разбирам абстрактни концепции, да решавам сложни проблеми и да създавам код, който работи.

На тези страници ще ти покажа уроците, които научих от Aria, и стъпка по стъпка ще те науча как и ти да овладееш тънкостите на езика `C++` . Ще ти докажа, че чрез помощта на изкуственият интелект можете да извлечеш изключително много информация за изучаването на програмни езици и по този начин да се вдъхновиш да се впуснеш в света на програмирането.

Нека това пътуване те вдъхнови да преоткриеш собствения си потенциал и да осъзнаеш, че нищо не е невъзможно!

Глава 1

Въведение

Добре дошли в света на C++ ! Този език за програмиране е истински гигант, който стои в основата на безброй приложения, игри и технологии, които използваме всеки ден. C++ е език, който ви дава мощта да създавате сложни и ефективни програми, да контролирате хардуера на компютъра си и да реализирате най-смелите си идеи.

Но C++ не е за начинаещи. Той е мощен и гъвкав, но е и сложен и изисква задълбочено разбиране.

Какво прави C++ толкова специален?

– Обектно-ориентирано програмиране - C++ е език, който ви позволява да структурирате програмите си около обекти, които съдържат данни и функции. Това е като да създадете симулация на реалния свят в код, където всеки обект е отделен елемент с собствени характеристики и поведение.

– Висока производителност - C++ е известен с ефективността си. Той ви дава пълен контрол над ресурсите на компютъра и ви позволява да създавате приложения, които работят бързо и ефективно.

– Гъвкавост - C++ е гъвкав език, който ви позволява да разработвате разнообразни приложения, от операционни системи и игри до приложения за мобилни устройства.

– Широко разпространен - C++ е широко разпространен език, който се използва от милиони програмисти по целия свят. Това означава, че ще имате лесен достъп до ресурси, общности и поддръжка.

Защо да се учите на C++ ?

– Мощни приложения - C++ ви дава мощта да създавате комплексни и ефективни приложения, които могат да решават трудни задачи.

– Дълбоко разбиране - C++ ви учи да разбирате как работи компютърът и как да управлявате ресурсите му.

– Отворена врата към нови възможности - C++ е отворена врата към широк спектър от професионални възможности.

В тази книга ще ви запознаем с основите на C++ и ще ви покажем как да създавате свои собствени програми. Пригответе се за вълнуващо пътешествие в света на програмирането!

1.1 Първи стъпки в C++

В тази глава ще се запознаем с основите на C++ , като започнем с компилация, компилиране и изпълнение на програми.

1.1.1 Компилация и изпълнение

C++ е компилиран език. Това означава, че кодът, който пишете, трябва да бъде преведен на машинно разбираем език, преди да може да се изпълни.

Компилацията е процес, който превръща изходния код (текстовият файл, който вие пишете) в изпълним файл. Изпълним файл е файл, който може да се изпълни от компютъра.

За да компилирате и изпълните C++ програма, ще ви е необходим компилатор. Компилатор е програма, която превежда изходния код на C++ в изпълним файл.

1.1.2 Как да компилираме и изпълним програма?

Ето стъпките, които трябва да следвате, за да компилирате и изпълните C++ програма:

– Създайте нов текстов файл с разширение .cpp.

– Напишете C++ кода си в този файл.

– Отворете командния ред (или терминал) и отидете до директорията, където е вашият текстов файл.

– Въведете следната команда, за да компилирате програмата:
`g++ име_на_файла.cpp -o име_на_изпълним_файл`

– Въведете следната команда, за да изпълните програмата:
`./име_на_изпълним_файл`

Пример:

Ако вашият файл се казва `hello.cpp` и искате да създадете изпълним файл `hello`, тогава трябва да въведете следните команди:

```
g++ hello.cpp -o hello
./hello
```

1.1.3 Имплементиране на български език в C++

За да използваме български език в кода на `C++` и в конзолата, трябва първо да зададем локализацията на проекта:

```
#include <locale>

int main() {
    setlocale(LC_ALL, "Bulgarian");
        // Локализация на кирилицата в проекта

    return 0;
}
```

`#include <locale>` - Тази линия включва библиотеката `locale`, която ни дава достъп до функции за локализация на езици.

`setlocale(LC_ALL, "Bulgarian");` - Стандартна функция за локализация на български език.

1.1.4 Първата ни програма

Ето пример за проста `C++` програма, която извежда текст на екрана:

```
#include <iostream>
#include <locale>

int main() {
    setlocale(LC_ALL, "Bulgarian");

    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

`#include <iostream>` - Тази линия включва библиотеката `iostream`, която ни дава достъп до функции за вход и изход.

`int main()` - Тази линия дефинира главната функция на програмата. Всички C++ програми трябва да имат главна функция.

`std::cout << "Hello, world!" << std::endl;` - Тази линия извежда текста `"Hello, world!"` на екрана.

`return 0;` - Тази линия завършва изпълнението на програмата.

1.2 Ключови елементи на синтаксиса и семантиката на C++

1.2.1 Ключови думи

Ключовите думи в C++ са резервирани думи, които имат специално значение за компилатора. Те не могат да се използват като имена на променливи, функции или други идентификатори.

Ключова дума	Описание
`int, float, double, char, bool, void, auto, const, constexpr, decltype`	Типове данни
`if, else, else if, switch, case, default, break, continue, goto`	Условни оператори
`for, while, do while, break, continue`	Цикли
`return, sizeof, new, delete, nullptr, this`	Оператори
`namespace, using, struct, class, enum, union, template, typename, friend, operator`	Организъм на кода
`public, private, protected, static, virtual, override, final, explicit`	Модификатори за достъп
`inline, extern, volatile, mutable, register`	Модификатори за компилация
`try, catch, throw, noexcept`	Обработка на изключения

Ключовите думи се използват в C++ код, за да се определят типове данни, структури, класове, функции и други елементи на програмата.

Пример:

```cpp
int main() {
    int a = 10; // int е ключова дума за дефиниране на целочислена променлива
    if (a > 5) { // if е ключова дума за условен оператор
        std::cout << "a is greater than 5" << std::endl;
    }
    return 0; // return е ключова дума за връщане на статус
}
```

1.2.2 Специални символи

В C++ езикът, освен букви, цифри и ключови думи, има и специални символи, които имат специално значение за компилатора. Тези символи се използват за определяне на оператори, разделителни символи, коментари и други елементи на кода.

Оператори

Операторите са специални символи, които извършват операции върху операнди.

Аритметика

`+` (събиране): `a + b`

`-` (изваждане): `a - b`

`*` (умножение): `a * b`

`/` (деление): `a / b`

`%` (остатък от деление): `a % b`

`++` (увеличаване): `a++`

`--` (намаляване): `a--`

Условни

`==` (равенство): `a == b`

`!=` (неравенство): `a != b`

`>` (по-голямо): `a > b`

`<` (по-малко): `a < b`

`>=` (по-голямо или равно): `a >= b`

`<=` (по-малко или равно): `a <= b`

`&&` (логическо И): `a && b`

`||` (логическо ИЛИ): `a || b`

`!` (логическо НЕ): `!a`

`?:` (тернарен условен оператор): `a ? b : c`

Битова аритметика

`&` (битово И): `a & b`

`|` (битово ИЛИ): `a | b`

`^` (битово ИЗКЛ. ИЛИ): `a ^ b`

`~` (битово НЕ): `~a`

`<<` (ляво битово изместване): `a << b`

`>>` (дясно битово изместване): `a >> b`

Разделителни символи

Разделителните символи се използват за разделяне на различни части на `C++` кода.

Пример:

`;` (точка и запетая): `int a = 10;`

`,` (запетая): `int a = 10, b = 20;`

`:` (двоеточие): `switch (a) { case 1: ...; }`

`::` (обхват на име): `std::cout`

`.` (член на клас): `a.b`

`->` (член на указател): `a->b`

`[]` (индексиране): `a[b]`

`()` (извикване на функция): `a()`

`{}` (блокове код): `{ ... }`

Коментари

Коментарите са текст, който се игнорира от компилатора. Те се използват за обяснение на кода, добавяне на документация или деактивиране на част от кода.

Пример:

- `//` (едноредов коментар):

```
// This is a single line comment.
```

- `/* ... */` (многоредов коментар):

```
/*
 * This is a multiline comment.
 * It can extend on multiple lines.
 */
```

1.2.3 Идентификатори

Идентификаторите в `C++` са имена, които се използват за означаване на променливи, функции, класове, структури, изброявания, области на имена и други елементи на програмата.

Правила за идентификатори

Идентификаторите могат да се състоят от букви, цифри, подчертаване (`_`). Първият символ на идентификатора не може да бъде цифра. Ключовите думи не могат да се използват като идентификатори. Чувствителност към регистъра: `myVariable` и `MyVariable` са различни идентификатори.

Примери за идентификатори

- Променлива: `age, firstName, totalScore`
- Функция: `calculateArea, printMessage, sortArray`
- Клас: `Person, Car, Database`
- Структура: `Point, Date, Time`
- Изброяване: `Color, Status, Direction`
- Област на имена: `std, myNamespace, utils`

Препоръки за идентификатори

Използвайте описателни имена, които отразяват целта на идентификатора. Използвайте `camelCase` или `snake_case` за по-добра четимост. Избягвайте къси и неясни имена. Не използвайте резервирани думи като идентификатори.

Пример за код с идентификатори:

```cpp
#include <iostream>

using namespace std; // Дефиниране на областта на имената "std"

int main() {
    // Дефиниране на променлива с име "age"
    int age = 25;

    // Дефиниране на функция с име "printMessage"
    void printMessage(string message) {
    cout << message << endl;
    }

    // Извикване на функцията "printMessage"
    printMessage("Hello, world!");

    return 0;
}
```

1.2.4 Литерали

Литералите в C++ са константни стойности, които се използват за представяне на данни в програмата. Те са директни представяния на данни, които се компилират директно в код.

Видове литерали

C++ поддържа различни видове литерали, в зависимост от типа на данните:

- Числови литерали:
  - Цялочислени литерали:

    Десетични: `10, 25, -15`

    Осмоични: `012, 037` (започват с `0`)

    Шестнадесетични: `0x1A, 0x2F` (започват с `0x`)

  - Дробни литерали: `3.14, 1.5e-2` (експоненциална нотация)

- Символни литерали:
  - Обикновени: `'a', 'B', '%'`
  - Escape последователности: `'\n', '\t', '\''`

- Текстови литерали:
  - Обикновени: `"Здравей, свят!` `"Hello, world!"`
  - Raw string литерали: `R"(C:\Users\MyUser\Documents)"` (за запазване на escape последователности)

- Булеви литерали: `true, false`

- Указателни литерали: `nullptr` (за празен указател)

Пример за код с литерали

```cpp
#include <iostream>

using namespace std;

int main() {
    // Цялочислени литерали
    int age = 25;
    int octalNumber = 012;
    int hexNumber = 0x1A;

    // Дробни литерали
    double pi = 3.14;
    double smallNumber = 1.5e-2;

    // Символни литерали
    char character = 'A';
    char newline = '\n';

    // Текстови литерали
    string message = "Hello, world!";
    string path = R"(C:\Users\MyUser\Documents)";

    // Булеви литерали
    bool isTrue = true;
    bool isFalse = false;
```

```
26
27      // Указателни литерали
28      int* ptr = nullptr;
29
30      return 0;
31  }
```

1.2.5 Променливи и константи

Променливи

Променливите в C++ са имена, които се използват за съхраняване на данни в паметта. Тези данни могат да бъдат променяни по време на изпълнението на програмата.

1. Дефиниране на променливи:

```
<datatype> <variable_name>;
```

Пример:

```
int age; // Дефиниране на променлива от тип "int" с име "age"
double price; // Дефиниране на променлива от тип "double" с име "price"
string name; // Дефиниране на променлива от тип "string" с име "name"
```

2. Инициализиране на променливи
   Инициализирането на променлива е процесът на присвояване на начална стойност при дефинирането.

Пример:

```
int age = 25; // Инициализиране на променливата "age" със стойност 25
// Инициализиране на променливата "price" със стойност 19.99
double price = 19.99;
// Инициализиране на променливата "name" със стойност "Ivan"
string name = "Ivan";
```

3. Използване на променливи
   След дефинирането и инициализирането, променливите могат да се използват в програмата.

Пример:

```
int age = 25;
// Извеждане на стойността на променливата "age"
cout << "Your age is: " << age << endl;
```

Константи

Константите в C++ са имена, които се използват за съхраняване на данни в паметта, но стойностите им не могат да се променят по време на изпълнението на програмата.

1. Дефиниране на константи
   За да се дефинира константа, се използва ключовата дума `const`:

```
const <datatype> <NAME_OF_CONSTANT> = <value>;
```

Пример:

```
// Дефиниране на константа от тип "int" с име "MAX_AGE" със стойност 120
const int MAX_AGE = 120;
// Дефиниране на константа от тип "double" с име "PI" със стойност 3.14159
const double PI = 3.14159;
// Дефиниране на константа от тип "string" с име "GREETING"
// със стойност "Greetings!"
const string GREETING = "Greetings!";
```

2. Използване на константи
   Константите могат да се използват в програмата по същия начин като променливите.

Пример:

```
const int MAX_AGE = 120;
int age = 25;
if (age > MAX_AGE) {
    cout << "Invalid age!" << endl;
}
```

1.2.6 Подпрограми (Функции)

Функциите в C++ са блокове от код, които изпълняват конкретна задача. Те могат да приемат аргументи и връщат резултат.

Функции с тип

Функциите с тип връщат резултат от конкретен тип.

Пример:

```
int sum(int a, int b) {
    return a + b;
}
```

В този пример функцията `sum` приема два целочислени аргумента (`a` и `b`) и връща цяло число (`int`), което е сумата на двата аргумента.

Void функции

`Void` функциите не връщат резултат. Те се използват за изпълнение на действия, които не връщат стойност.

Пример:

```
void printHello() {
    cout << "Hello, world!" << endl;
}
```

В този пример функцията `printHello` не връща резултат. Тя просто извежда текст на конзолата.

Аргументи на функцията

Аргументите на функцията са стойности, които се предават на функцията при викането ѝ.

Пример:

```
int sum(int a, int b) {
    return a + b;
}

int main() {
    int result = sum(10, 20); // Предаване на аргументите 10 и 20
    cout << result << endl; // Извеждане на резултата (30)
    return 0;
}
```

В този пример функцията `sum` приема два целочислени аргумента (`a` и `b`). При викането ѝ в `main` функцията, се предават стойностите `10` и `20` за `a` и `b` съответно.

Предаване по стойност

При предаване по стойност, на функцията се предава копия на аргументите.

Пример:

```
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 10;
    int y = 20;
    swap(x, y); // Предаване по стойност
    cout << x << " " << y << endl; // Извеждане на "10 20"
    return 0;
}
```

В този пример, функцията `swap` не модифицира оригиналните стойности на `x` и `y`, защото работи с копия.

Предаване по препратка

При предаване по препратка, на функцията се предава адреса на аргументите.

Пример:

```
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 10;
    int y = 20;
    swap(x, y); // Предаване по препратка (директен адрес)
    cout << x << " " << y << endl; // Извеждане на "20 10"
    return 0;
}
```

В този пример, функцията swap модифицира оригиналните стойности на `x` и `y`, защото работи с адресите им.

Рекурсия

Рекурсията е техника, при която функция се вика сама себе си.

Пример:

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

В този пример функцията `factorial` изчислява факториела на число чрез рекурсивната формула $n! = n \cdot (n-1)!$.

Важно: Рекурсията трябва да има базов случай, който прекратява рекурсивните викания. В противен случай програмата изпада в безкраен цикъл, докато не се стигне до момента, в който паметта, нужна за запазване на променливите и информацията, надвиши разпределената за процеса памет в стека (Stack overflow).

Lambda функции

Lambda функциите са анонимни функции, които могат да се дефинират и използват в една линия код.

Пример:

```
auto sum = [](int a, int b) {
    return a + b;
};
int result = sum(10, 20); // Извикване на lambda функцията
```

В този пример lambda функцията `sum` приема два целочислени аргумента (`a` и `b`) и връща цяло число (`int`), което е сумата на двата аргумента.

1.2.7 Атрибути

Атрибутите в `C++` са специални ключови думи, които модифицират поведението на променливи, функции, класове и други елементи на кода. Те определят важни характеристики, като обхват, вид, жизнен цикъл, достъп и други.

Const

Атрибутът `const` определя, че стойността на променливата не може да се променя след инициализирането ѝ. `Const` гарантира, че стойността няма да се промени неволно, подобрявайки безопасността на кода.

Пример:

```
const int PI = 3.14159;
```

В този пример `PI` е константа с стойност `3.14159`.

Static

Атрибутът `static` определя, че променливата е статична. Статичните променливи съществуват само в рамките на файла, в който са дефинирани. Инициализират се само веднъж при първото викане на файла и живеят цял живот на програмата. Споделят се между всички функции в файла.

Пример:

```
static int count = 0;
```

В този пример `count` е статична променлива.

Extern

Атрибутът `extern` определя, че променливата е дефинирана в друг файл. Атрибутът `extern` не инициализира променливата, само указва, че тя съществува някъде другe.

Пример:

```
extern int count;
```

В този пример `count` е променлива, която е дефинирана в друг файл.

Volatile

Атрибутът `volatile` указва, че стойността на променливата може да се променя външно, без да се вижда от компилатора. Атрибутът `volatile` предотвратява компилатора да оптимизира кода, който работи с променливата. Използва се за променливи, чиято стойност може да се промени от външни фактори, като прекъсвания, таймери или други процеси.

Пример:

```
volatile int counter;
```

В този пример `counter` е променлива, чиято стойност може да се променя от външен код.

Register

Атрибутът `register` препоръчва на компилатора да съхранява променливата в регистър на процесора. Атрибутът `register` е само препоръка. Не гарантира, че компилаторът ще съхрани променливата в регистър. Използва се за често използвани променливи, за да се подобри ефективността.

Пример:

```
register int i;
```

В този пример `i` е променлива, която компилаторът може да съхрани в регистър.

Auto

Атрибутът `auto` позволява на компилатора да определи типа на променливата автоматично. `Auto` улеснява кода, когато типът на променливата е ясен от инициализацията.

Пример:

```
auto x = 10;
```

В този пример `x` е променлива с тип `int`, определен автоматично от компилатора.

1.3 Стандартен конзолен I/O в C++

В тази част ще разгледаме как се работи със системната конзола за стандартно въвеждане и извеждане на информация в `C++` и как да манипулираме получените данни от конзолата, а именно да обработваме информация от `string` данните.

1.3.1 Работа със системната конзола

В `C++` за стандартно въвеждане и извеждане на данни се използва пакетът `<iostream>`. Той предоставя обекти, които улесняват взаимодействието с потребителя и файловете.

- `std::cin` - Стандартен входен поток (конзола).
- `std::cout` - Стандартен изходен поток (конзола).
- `std::cerr` - Стандартен изходен поток за грешки (конзола).
- `std::endl` - `'\n'` (Нов ред).

Операторите `>>` и `<<` са специални оператори, които улесняват взаимодействието с `cin` и `cout`.

- `>>` - Операторът за въвеждане.
- `<<` - Операторът за извеждане.

Пример:

```cpp
#include <iostream>

int main() {
    int number;
    std::cout << "Insert a number: "; // Извеждане на текст на конзолата
    std::cin >> number; // Въвеждане на число от конзолата
    // Извеждане на числото на конзолата
    std::cout << "The inserted number is: " << number << std::endl;
    return 0;
}
```

Пакет `<iomanip>`

Пакетът `<iomanip>` предоставя методи за манипулиране на входящите и изходящите данни.

Пример:

```cpp
#include <iostream>
#include <iomanip>

int main() {
    double number = 123.456789;
    std::cout << "The number is: "
        << std::setprecision(4) << number << std::endl;
    std::cout << "The number is: "
        << std::setw(10) << number << std::endl;
    return 0;
}
```

В този пример:
`std::setprecision(4)` - ограничава точните цифри на числото до 4.
`std::setw(10)` - задава ширина на полето за извеждане на числото на 10 символа.

Резултат:

```
The number is: 123.5
The number is:      123.5
```

Допълнителни методи

Пакетът `<iomanip>` предлага много други методи за форматиране на входящите и изходящите данни.

Някои от тях са:

- `std::fixed` - Извежда числа с фиксирана точка.
- `std::scientific` - Извежда числа в научен запис.
- `std::left` - Подравнява текста вляво.
- `std::right` - Подравнява текста вдясно.

1.3.2 Работа със `string` данни

В `C++` класът `std::string` предоставя мощен инструмент за работа с текстови низове. Той осигурява множество функции за манипулиране, сравняване, търсене и модифициране на текстови низове.

Инициализиране на `string`

Има няколко начина за инициализиране на string обект:

Празен `string`:

```
std::string text; // Инициализира празен string
```

С текстова константа:

```
// Инициализира string с текстова константа
std::string text = "Hello, world!";
```

С друг `string`:

```
std::string text1 = "Hello, ";
// Инициализира string с конкатенация на други string-ове
std::string text2 = text1 + "world!";
```

С масив от символи:

```
char text[] = "Hello, world!"; // string в стил C
// Инициализира string в стил C++ с масив от символи
std::string textString(text);
```

Основни функции

`getline(istream& is, string& str, char delim)`	Въвежда редове от стандартен вход (`cin`) в `string` обект. `delim` определя разделителя на редовете. По подразбиране разделителят е `'\n'` (нов ред).
`length()`	Връща дължината на текстовия низ.
`at(size_t pos)`	Връща символа на позиция `pos` в текстовия низ.
`append(const string& str)  "+"`	Добавя текстов низ `str` към края на текстовия низ.
`compare(const string& str)  "=="`	Сравнява текстовия низ с `str`. Връща `0`, ако низовете са равни; `<0`, ако текстовият низ е по-къс от `str`; `>0`, ако текстовият низ е по-дълъг от `str`.
`substr(size_t pos, size_t len)`	Връща подниз от текстовия низ, започвайки от позиция `pos` и с дължина `len`.
`find(const string& str, size_t pos)`	Търси подниз `str` в текстовия низ, започвайки от позиция `pos`. Връща позицията на първото намиране на `str`, ако е намерен, иначе връща `string::npos`.
`replace(size_t pos, size_t len, const string& str)`	Заменя подниз от текстовия низ, започвайки от позиция `pos` и с дължина `len` с `str`.
`insert(size_t pos,const string& str)`	Вмъква `str` в текстовия низ на позиция `pos`.
`erase(size_t pos, size_t len)`	Премахва подниз от текстовия низ, започвайки от позиция `pos` и с дължина `len`.

Примери:

```cpp
#include <iostream>
#include <string>

using namespace std;

int main() {
    string text = "Hello, world!";

    // Въвеждане на текст от конзолата
    cout << "Insert text: ";
    getline(cin, text);

    // Извеждане на дължината на текста
    cout << "The length of the text is: " << text.length() << endl;

    // Извеждане на символа на позиция 5
    cout << "The character at position 5 is: "
         << text.at(5) << endl;

    // Добавяне на текст към края
    text.append(" How are you?");
    cout << "The text is: " << text << endl;

    // Сравняване на два текста
    string otherText = "Hello, world!";
    if (text.compare(otherText) == 0) {
        cout << "The texts are equal." << endl;
    } else {
        cout << "The texts are not equal." << endl;
    }

    // Извеждане на подниз
    cout << "The substring from position 7 to the end is: "
         << text.substr(7) << endl;

    // Търсене на подниз
    size_t pos = text.find("world");
    if (pos != string::npos) {
        cout << "The substring \"world\" was found at position: "
             << pos << endl;
    } else {
        cout << "The substring \"world\" wasn't found." << endl;
    }
```

```
44
45      // Замяна на подниз
46      text.replace(7, 5, "universe");
47      cout << "The text is: " << text << endl;
48
49      // Вмъкване на текст
50      text.insert(7, "beautiful ");
51      cout << "The text is: " << text << endl;
52
53      // Премахване на текст
54      text.erase(7, 10);
55      cout << "The text is: " << text << endl;
56
57      return 0;
58  }
```

Резултат: (Примерен входен текст: Hello, world!)

```
1  Insert text: >>Hello, world!
2  The length of the text is: 13
3  The character at position 5 is: ,
4  The text is: Hello, world! How are you?
5  The texts are not equal.
6  The substring from position 7 to the end is: world! How are you?
7  The substring "world" was found at position: 7
8  The text is: Hello, universe! How are you?
9  The text is: Hello, beautiful universe! How are you?
10 The text is: Hello, universe! How are you?
```

1.4 Структуриране на C++ програма

1.4.1 Директиви към предпроцесора

Предпроцесорът е ключов компонент в C++ компилационния процес, който обработва кода преди да бъде компилиран. Той изпълнява специални инструкции, наречени директиви, които променят структурата на кода преди да бъде предаден на компилатора.

Най-често използваните директиви към предпроцесора в C++ са:

#include

`#include` е най-често използваната директива. Тя включва съдържанието на друг файл в текущия файл. Това е ключово за организирането на C++ код в

отделни файлове, например за дефиниране на функции или класове в отделни `.h` файлове.

Стандартни файлове: `#include` се използва за включване на стандартни библиотеки, например:

```
// Включва стандартния header файл за входящи и изходящи операции
#include <iostream>
```

Потребителски файлове: `#include` се използва за включване на файлове, създадени от програмиста, например:

```
#include "my_functions.h" // Включва файл с дефиниции на функции
```

#define

`#define` се използва за дефиниране на константи и макроси. Константите са променливи, чиято стойност не може да бъде променена след дефинирането им. Макросите са блокове код, които се заменят с дефинираното съдържание по време на предпроцесорната обработка.

Дефиниране на константи:

```
#define PI 3.14159 // Дефинира константа PI
#define MAX\_SIZE 100 // Дефинира константа MAX_SIZE
```

Дефиниране на макроси:

```
#define SQUARE(x) (x * x) // Дефинира макрос за изчисляване на квадрат
// Дефинира макрос за извеждане на съобщение
#define PRINT_MESSAGE(msg) std::cout << msg << std::endl;
```

#undef

`#undef` се използва за премахване на предишно определение на константа или макрос.

```
#undef PI // Премахва определението на PI
```

#ifdef, #ifndef, #else, #endif

Тези директиви се използват за условно компилиране на код. Тоест, определени части от кода се компилират само ако е изпълнено определено условие.

- `#if` - Позволява условно компилиране на код въз основа на резултата от препроцесорно условие. Това условие може да бъде дефинирана константа (`#if defined(MY_CONSTANT)`), макрос (`#if SQUARE(5) == 25`), оператори за сравнение (`#if 10 > 5`), логически оператори (`#if defined(DEBUG) && defined(RELEASE)`).
- `#ifdef` - Проверява дали константа или макрос е дефиниран.
- `#ifndef` - Проверява дали константа или макрос не е дефиниран.
- `#else` - Изпълнява се, ако условието в `#if`, `#ifdef` или `#ifndef` не е изпълнено.
- `#endif` - Завършва блока, дефиниран от `#if`, `#ifdef`, `#ifndef` или `#else`.

Пример:

```
#define SIX 6

#if defined(SIX)
    #pragma message("The constant SIX is defined.")
#endif

#if (SIX > 5)
    #pragma message("6 is greater than 5.")
#endif

#undef SIX

#ifdef SIX
    // Този ред ще се компилира само ако е дефинирана константата SIX
    #pragma message("The constant SIX is defined.")
#else
    // Този ред ще се компилира само ако константата SIX не е дефинирана
    #pragma message("The constant SIX is not defined.")
#endif
```

Резултат:

```
>| The constant SIX is defined.
>| 6 is greater than 5.
>| The constant SIX is not defined.
```

#pragma

`#pragma` е директива, която предоставя инструкции на компилатора. Тези инструкции са специфични за компилатора и могат да се различават между различните компилатори.

`#pragma message(STRING)` - Извежда съобщения в конзолата по време на компилирането. Това е полезно при дебъгване на компилаторни проблеми.

```
// Изписва съобщението "Hello, world!" редом с другите съобщения от компилатора
#pragma message("Hello, world!");
```

`#pragma once` - Предотвратява многократно включване на header файл. Това е полезно за предотвратяване на грешки при компилация, когато един и същ header файл се включва многократно.

```
#pragma once // Предотвратява многократно включване на header файл
```

#error

`#error` е директива, която генерира грешка по време на компилация. Това е полезно за сигнализиране на грешки, които не могат да бъдат открити от компилатора.

Пример:

```
#if !defined(MY_CONSTANT)
    #error "The constant MY_CONSTANT is not defined!"
#endif
```

В този пример, ако константата `MY_CONSTANT` не е дефинирана, компилаторът ще генерира грешка с текст `"The constant MY_CONSTANT is not defined!"`.

1.4.2 Области на имената

Областите на имената (namespaces) в C++ са механизъм за организиране на код в логически групи. Те са особено полезни за:

Избягване на конфликти на имена: В големи проекти, с множество модули и библиотеки, е възможно да се използват едни и същи имена за различни променливи, функции, класове и т.н. Областите на имената позволяват да се групират тези елементи, като им се дава уникално име за всяка група.

Подобрена четливост: Те структурират кода, правейки го по-лесен за четене и разбиране.

По-лесно управление на зависимостта: Могат да се използват за управление на зависимостта между различни части от кода.

Дефиниране на области на имената

В C++ се дефинират области на имената с ключовата дума `namespace`. Синтаксисът е следният:

```
namespace <name_of_namespace> {
    // Дефиниции на променливи, функции, класове, т.н.
}
```

Пример:

```
namespace MyNamespace {
    int x = 10;
    void printX() {
        std::cout << "x = " << x << std::endl;
    }
}
```

Използване на области на имената

За да се достъпи до елемент в област на имената, се използва операторът за обхват (`::`).

Пример:

```
int main() {
    MyNamespace::printX(); // Извиква функцията printX() от MyNamespace
    // Достъп до променливата x от MyNamespace
    std::cout << MyNamespace::x << std::endl;
    return 0;
}
```

Резултат:

```
x = 10
10
```

Могат да се дефинират области на имената, вложени една в друга.

Ключовата дума `using` може да се използва за импортиране на всички елементи от дадена област на имената. Може да се използва за дефиниране на алиас за елемент от област на имената.

Пример за `using`:

```
using namespace MyNamespace; // Импортиране на всички елементи от
   MyNamespace

int main() {
    printX(); // Извиква функцията printX() от MyNamespace
    // Достъп до променливата x от MyNamespace
    std::cout << x << std::endl;
    return 0;
}
```

Резултат:

```
x = 10
10
```

Глава 2

Типове данни

В C++ типовете данни определят вида на данните, които можем да съхраняваме в променливи. Те ни казват каква информация може да се съхранява, как се интерпретира и какви операции могат да се извършват с нея.

2.1 Типове данни за цели и реални числа

В C++ имаме няколко типа данни, които се използват за представяне на цели и реални числа:

Целочислени типове:

- `int`: Най-често използваният тип за цели числа. Обикновено заема 4 байта в паметта и може да съхранява числа в диапазона от -2,147,483,648 до 2,147,483,647.
- `short`: Целочислени числа с по-малка големина, обикновено 2 байта. Диапазонът е от -32,768 до 32,767.
- `long`: Целочислени числа с по-голяма големина, обикновено 4 байта. Диапазонът е от -2,147,483,648 до 2,147,483,647.
- `long long`: Целочислени числа с още по-голяма големина, обикновено 8 байта. Диапазонът е от -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807.
- `unsigned int`: Неотрицателни цели числа. Диапазонът е (обикновено) от 0 до 4,294,967,295.
- `unsigned short`: Неотрицателни цели числа с по-малка големина. Диапазонът е от 0 до 65,535.

– `unsigned long`: Неотрицателни цели числа с по-голяма големина. Диапазонът е от 0 до 4,294,967,295.

– `unsigned long long`: Неотрицателни цели числа с още по-голяма големина. Диапазонът е от 0 до 18,446,744,073,709,551,615.

Пример:

```
int age = 25;
short year = 2023;
long population = 8000000;
long long bigNumber = 999999999999999999;
unsigned int counter = 0;
unsigned short port = 80;
unsigned long fileSize = 1024 * 1024 * 1024;
unsigned long long veryBigNumber = 18446744073709551615;
```

Типове с плаваща запетая:

– `float`: Числа с плаваща запетая с по-ниска прецизност, обикновено 4 байта.

– `double`: Числа с плаваща запетая с по-висока прецизност, обикновено 8 байта.

– `long double`: Числа с плаваща запетая с още по-висока прецизност, обикновено 16 байта.

Пример:

```
float temperature = 25.5;
double pi = 3.141592653589793;
long double veryPreciseNumber = 3.14159265358979323846264338327950;
```

2.2 Типове данни за символ и низ

В C++ имаме два основни типа данни, които се използват за представяне на текст:

Символ (`char`)

Използва се за съхраняване на единичен символ, като буква, цифра или специален знак. Заема 1 байт в паметта.

Пример:

```
char letter = 'A';
char digit = '7';
char specialSymbol = '\%';
```

Низ (като масив от символи)

В `C++` низовете могат да се представят и като масиви от символи. Това е постарият начин за работа с текст в `C++` (произлизащ от низовете в `C`).

Пример:

```
char greeting[] = "Greetings!";
```

Този код дефинира масив от символи `greeting`, който съдържа низа `"Greetings"`. Важно е да се отбележи, че масивът `greeting` трябва да е с достатъчно голям размер, за да събере всички символи от низа, плюс един допълнителен символ `'\0'` (null terminator), който маркира края на низа.

За повече информация за масиви, виж (4.1).

Низ (`string`)

Използва се за съхраняване на поредица от символи, т.е. текст. Представлява обект от класа `std::string`, дефиниран в заглавния файл `<string>`.

Пример:

```
std::string greeting = "Greetings!";
```

За повече информация за работа със `string` данни, виж (1.3.2).

Извличане на числа от `string` данни

За да извлечем число от `string`, можем да използваме функциите `stoi()`, `stol()`, `stoll()`, `stof()`, `stod()`, `stold()`, които се намират в заглавния файл `<string>`.

Пример:

```
#include <iostream>
#include <string>

int main() {
    std::string numberString = "12345";
    int numberInt = std::stoi(numberString);
    std::cout << "The number is: " << numberInt << std::endl;

    std::string realNumberString = "3.14159";
    double realNumberDouble = std::stod(realNumberString);
    std::cout << "The real number is: "
        << realNumberDouble << std::endl;

    return 0;
}
```

Резултат:

```
The number is: 12345
The real number is: 3.14159
```

Важно: Функциите `stoi()`, `stol()`, `stoll()`, `stof()`, `stod()`, `stold()` хвърлят `Exception` (виж 3.4) от тип `std::invalid_argument`, ако низът не е валидно число.

2.3 Логически данни

В `C++` имаме логически тип данни (`bool`), който може да приема само две стойности:

`true`: Истина

`false`: Неистина

Пример:

```
bool isSunny = true;
bool isRainy = false;
```

Логическите данни се използват за представяне на условия, които могат да бъдат истинни или неистинни. В C++ `true` се представя като `1` (или всичко различно от `0`), а `false` се представя като `0`. Логическите операции, като например `&&` (И), `||` (ИЛИ), `!` (НЕ), се използват за комбиниране на логически условия.

Пример:

```
bool isSunny = true;
bool isWarm = true;

bool isPerfectDay = isSunny && isWarm; // true

bool isNotPerfectDay = !isPerfectDay; // false
```

Използване на логически данни

Логическите данни се използват широко в C++ за:

- Условни оператори (`if`, `else if`, `else`): За да се изпълнява код само ако дадено условие е истинно.
- Цикли (`for`, `while`, `do while`): За да се изпълнява код, докато дадено условие е истинно.
- Функции: За да се връщат логически стойности.

За условните оператори и циклите ще научим в следващата глава.

Глава 3

Конструкции за поточен контрол

В C++ протичането на програмата може да се контролира чрез различни конструкции, които определят дали дадени части от кода ще бъдат изпълнени или не. Тези конструкции са съответно за разклонение в потока, за цикли, за прекъсване и за управление на изключения.

3.1 Конструкции за разклонение

Конструкциите за разклонение ни позволяват да изпълняваме различни части от код в зависимост от резултата от дадено условие. В C++ имаме 3 основни конструкции за разклонение и 2 алтернативни варианта (частни случаи):

I. `if` оператор

Използва се за изпълнение на код, само ако дадено условие е истинно.

Синтаксис:

```
if (condition) {
    // Код, който се изпълнява, ако условието е истинно
}
// или
if (condition) /* Код за изпълнение */;
```

Пример:

```
int age = 25;

if (age >= 18) {
    std::cout << "You're old enough." << std::endl;
}
```

II. `if-else` оператор

Използва се за изпълнение на един от два кода, в зависимост от резултата от дадено условие.

Синтаксис:

```
if (condition) {
    // Код, който се изпълнява, ако условието е истинно
} else {
    // Код, който се изпълнява, ако условието е неистинно
}
```

Пример:

```
int age = 15;

if (age >= 18) {
    std::cout << "You're old enough." << std::endl;
} else {
    std::cout << "You're not old enough." << std::endl;
}
```

Тернарен оператор (`?:`)

Представлява по-кратка форма за писане на `if-else` конструкция, когато трябва да запишем стойност на променлива, в зависимост от някакво условие.

Синтаксис:

```
<type> <var_name> = <conditon> ? <val_if_true> : <val_if_false>;
```

Пример:

```
int age = 25;

// ‘if-else‘ в случая е по-обемен
if (age >= 18) {
    std::cout << "Adult" << std::endl;
} else {
    std::cout << "Minor" << std::endl;
}

// С тернарен оператор кода става сравнително по-компактен
std::cout << ((age >= 18) ? "Adult" : "Minor") << std::endl;
```

III. `else-if` оператор

Използва се за изпълнение на един от няколко кода, в зависимост от резултата от няколко условия.

Синтаксис:

```
if (condition_1) {
    // Код, който се изпълнява, ако condition_1 е истинно
} else if (condition_2) {
    // Код, който се изпълнява, ако condition_2 е истинно
} else {
    // Код, който се изпълнява, ако никое от условията не е истинно
}
```

Пример:

```
int grade = 85;

if (grade >= 90) {
    std::cout << "A" << std::endl;
} else if (grade >= 80) {
    std::cout << "B" << std::endl;
} else if (grade >= 70) {
    std::cout << "C" << std::endl;
} else {
    std::cout << "F" << std::endl;
}
```

Резултат:

```
B
```

`switch` оператор

Използва се за проверка на стойността на дадена променлива и изпълнение на код, съответстващ на тази стойност.

Синтаксис:

```
switch (variable) {
    case value_1:
        // Код, който се изпълнява, ако изразът е равен на value_1
        break;
    case value_2:
        // Код, който се изпълнява, ако изразът е равен на value_2
        break;
    default:
        // Код, който се изпълнява, ако изразът
        // не е равен на никоя от стойностите
}
```

Пример:

```
int day = 3;

switch (day) {
    case 1:
        std::cout << "Monday" << std::endl;
        break;
    case 2:
        std::cout << "Tuesday" << std::endl;
        break;
    case 3:
        std::cout << "Wednesday" << std::endl;
        break;
    default:
        std::cout << "Invalid day" << std::endl;
}
```

Резултат:

```
Wednesday
```

3.2 Конструкции за цикъл

Конструкциите за цикли ни позволяват да изпълняваме даден код многократно, докато дадено условие е истинно. В C++ имаме 3 основни конструкции за цикли и 1 алтернативен вариант (частен случай):

I. `while` цикъл

Използва се за изпълнение на код докато дадено условие е истинно.

Синтаксис:

```
while (condition) {
    // Код, който се изпълнява многократно
}
```

Условие: Проверява се преди всяко повторение на цикъла. Ако условието е истинно, цикълът се изпълнява.

Пример:

```
int i = 0;

while (i < 5) {
    std::cout << i << std::endl;
    i++;
}
```

`i++`: Увеличава стойността на `i` с `1` след всяко повторение на цикъла.

Резултат:

```
0
1
2
3
4
```

Важно: Циклите винаги трябва да имат стъпка, в която да се подновява условието, за да се гарантира, че няма да изпадне в безкраен цикъл или изобщо да не се осъществи.

II. `do-while` цикъл

Използва се за изпълнение на код поне веднъж и след това докато дадено условие е истинно.

Синтаксис:

```
do {
    // Код, който се изпълнява многократно
} while (condition);
```

Условие: Проверява се след всяко повторение на цикъла. Ако условието е истинно, цикълът се изпълнява отново.

Пример:

```
int i = 0;

do {
    std::cout << i << std::endl;
    i++;
} while (i < 5);
```

III. `for` цикъл

Използва се за изпълнение на код за определен брой пъти.

Синтаксис:

```
for (initialization; condition; updation) {
    // Код, който се изпълнява многократно
}
```

Инициализация: Изпълнява се веднъж в началото на цикъла. Обикновено се използва за дефиниране на променлива, която ще се използва за броене на повторенията.

Условие: Проверява се преди всяко повторение на цикъла. Ако условието е истинно, цикълът се изпълнява.

Стъпка: Изпълнява се след всяко повторение на цикъла. Обикновено се използва за промяна на стойността на променливата, използвана за броене.

Пример:

```
for (int i = 0; i < 5; i++) {
    std::cout << i << std::endl;
}
```

`for-each` цикъл

Тази конструкция е частен случай на `for` цикъла. Използва се за итериране през елементите на контейнер (например масив, вектор, списък).

Синтаксис:

```
for (<type> <element> : <container>) {
    // Код, който се изпълнява за всеки елемент
}
```

Тип елемент: Определя типа на елементите в контейнера.
Контейнер: Определя контейнера от елементи, през който ще се итерира.

Пример:

```
int numbers[] = {0, 1, 2, 3, 4}; // Масив от числа

for (int number : numbers) {
    std::cout << number << std::endl;
}
```

Всяко повторение на цикъла, променливата `number` ще приема стойността на следващия елемент в масива `numbers`.

3.3 Конструкции за прекъсване

Конструкциите за прекъсване ни позволяват да прекъснем нормалното изпълнение на цикъл или функция. В `C++` имаме четири основни конструкции за прекъсване:

break

Използва се за прекъсване на цикъл и излизане от него. Също се ползва и при `switch-case` конструкцията (Виж (3.1)).

Пример:

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break; // Прекъсва цикъла, когато i е 5
    }
    std::cout << i << std::endl;
}
```

Когато `i` е равно на `5`, `break` прекъсва цикъла и изпълнението продължава от кода след цикъл.

Резултат:

```
0
1
2
3
4
```

continue

Използва се за прескачане на текущото повторение на цикъл и преминаване към следващото.

Пример:

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
        continue; // Прескача четните числа
    }
    std::cout << i << std::endl;
}
```

Когато `i` е четно число, `continue` прескача останалата част от кода в цикъла и преминава към следващото повторение.

Резултат:

```
1
3
5
7
9
```

return

Използва се за прекъсване на функция и връщане на стойност. Ако функцията е от тип `void`, `return` просто прекъсва функцията.

Пример:

```
int sum(int a, int b) {
    if (a < 0 || b < 0) {
        return -1; // Връща -1, ако някое от числата е отрицателно
    }
    return a + b;
}
```

Когато `a` или `b` е отрицателно, `return` прекъсва функцията `sum` и връща `-1`.

goto

Примитивна ключова дума. Използва се за неусловен преход към друга част от кода. Може да доведе до нечетлив код или да пречупи интегритета на кода. Да се използва само при изрична необходимост.

Синтаксис:

```
<label_a>:
...
goto <label_a>;

// или

goto <label_b>;
...
<label_b>:
```

Пример:

```
int main() {
    for (int i = 0; i < 10; i++) {
        if (i == 5) {
            goto end; // Прескача останалата част от цикъла
        }
        std::cout << i << std::endl;
    }

    std::cout << "The code will never go through this message"
        << std::endl;

end:
    std::cout << "End of loop" << std::endl;
    return 0;
}
```

Когато `i` е равно на `5`, `goto` прескача останалата част от кода и преминава към кода след етикета `end`.

3.4 Изключения (Exceptions) и работа с тях

Изключенията са непредвидени събития, които възникват по време на изпълнението на програмата и могат да нарушат нормалния й ход. В `C++` можем да улавяме тези изключения и да обработваме грешките, за да предотвратим сривове на програмата.

Как се хвърлят изключения?

Използваме ключовата дума `throw` за хвърляне на изключение. Изключението може да бъде всякакъв тип данни, но най-често се използват класове, които описват типа на грешката.

Синтаксис:

```
throw <exception_type>(<exception_message>);
```

Пример:

```
#include <iostream>
#include <stdexcept> // Включва библиотеката за основните изключения

int main() {
    int a, b;
    std::cout << "Insert two numbers: ";
    std::cin >> a >> b;

    if (b == 0) {
        // Хвърляне на изключение
        throw std::runtime_error("Division by 0!");
    }

    int result = a / b;
    std::cout << "The result is: " << result << std::endl;

    return 0;
}
```

В този пример, ако потребителят въведе `0` за `b`, се хвърля изключение от тип `std::runtime_error`.

Как се улавят изключения?

Използваме контролната конструкция `try-catch`, с основната цел да улавя изключения и да ги обработва без да прекъсва протичането на програмата.

Синтаксис:

```
try {
    // Код, който може да хвърли изключение
} catch (<exception_type> <exception_name>)  {
    // Улавя изключението <exception_name> за обработка
}
```

- `try` блокът обгръща кода, който може да хвърли изключение.
- `catch` блокът улавя изключението, ако се хвърли такова.

Пример:

```
#include <iostream>
#include <string>

int main() {
    std::string str = "Definitely not a number";
    int num;

    try {
        num = stoi(str);
    } catch (std::invalid_argument e) {
        std::cerr << "\"" << str << "\" is NaN!" << std::endl;
    }

    return 0;
}
```

В примера нарочно се подава като аргумент на функцията `std::stoi` низ, който не може да се конвертира в цяло число, за да се улови изключението за невалиден параметър, да се обработи изключението и да се изведе формално като грешка, без да се прекъсва протичането на програмата.

Резултат:

```
>! "Definitely not a number" is NaN!
```

Best practices:

- Да не се вкарва целият код в `try-catch` конструкция, а само частта, в която може да се изхвърли изключение.
- Да не се ползва `try-catch` конструкцията като основен начин за обработване на грешки. Конструкцията е предназначена за изключения.
- Да се ползва само при необходимост.

Глава 4

Съставни типове данни

Съставните типове данни са производни от примитивните типове данни или изобщо нови конструкции, които ни позволяват да групираме данни от различни типове под едно име, създавайки структурирани обекти от данни.

4.1 Масиви

Масивите са структурирани данни, които ни позволяват да съхраняваме множество елементи от един и същ тип под едно име. В C++ имаме два вида масиви: статични и динамични.

4.1.1 Статични масиви

Статичните масиви се декларират с фиксиран размер по време на компилация. Съществуват през целия живот на програмата и са статично разпределени в стека. Те са по-прости за използване, но са ограничени по размер, защото не можем да променяме размера на статичния масив след декларацията му.

Дефиниция:

```
// Инициализация с основни стойности (обикновено 0-ли за числени данни)
<type> <name>[<length>];
<type> <name>[<length>] = {};

// Инициализация чрез масивен литерал
<type> <name>[<length>] = {<el_1> , ... , <el_n>};
<type> <name>[] = {<el_1> , ... , <el_n>};
```

Пример:

```
int numbers[5]; // Масив от 5 цели числа
```

Декларацията `int numbers[5]` създава масив `numbers`, който може да съхранява 5 цели числа.

Важно: Индексите на елементите започват от `0` и стигат до (`n - 1`) (В случая до `4`).

Пример за промяна и достъп до елемент:

```
numbers[0] = 10; // Присвоява стойност 10 на първия елемент
// Извежда стойността на първия елемент
std::cout << numbers[0] << std::endl;
// Извежда стойността на третия елемент
std::cout << numbers[2] << std::endl;
```

Резултат:

```
10
0
```

4.1.2 Динамични масиви

Динамичните масиви се декларират с размер, който може да се променя по време на изпълнение. Те са по-гъвкави, но изискват по-внимателно управление на паметта, защото са динамично разпределени в хийпа (Виж (5.4))

Декларацията на динамичните масиви се осъществява чрез указатели (Виж (5.1)), сочещи към първия елемент от разпределена в хийпа памет, които трябва да се освободят след приключване на работата с масива.

Дефиниция:

```
// Без инициализация
<type> *<name>;
<type> *<name> = nullptr;

// Инициализация
<type> *<name> = new <type>[<length>];
...
delete[] <name>; // Освобождаване на разпределената памет!
```

Пример:

```
int *numbers = new int[5]; // Динамичен масив от 5 цели числа
...
delete[] numbers;
```

Декларацията `int *numbers = new int[5]` създава динамичен масив `numbers`, който може да съхранява 5 цели числа.

`new int[5]` е оператор, който разпределя памет за 5 цели числа на хийпа.

`numbers` е указател към първия елемент на масива.

`delete[] numbers` освобождава разпределената за масива памет.

Пример за промяна на и достъп до елемент:

```
numbers[0] = 10; // Присвоява стойност 10 на първия елемент
// Извежда стойността на първия елемент
std::cout << numbers[0] << std::endl;
// Извежда стойността на третия елемент
std::cout << numbers[2] << std::endl;

// или

*numbers = 10; // Присвоява стойност 10 на първия елемент
// Извежда стойността на първия елемент
std::cout << *numbers << std::endl;

numbers += 2; // Измества адреса на указателя, за да сочи към третия елемент
// Извежда стойността на третия елемент
std::cout << *numbers << std::endl;

delete[] numbers;
```

Важно: Необходимо е да се освободи паметта, заделена за динамичния масив, след като вече не е необходима, използвайки `delete[] numbers`
(Виж защо в (5.4)).

4.1.3 Дължина на масив

Дължината на масив се извежда от следната формула:

```
<type> array[n];
```

```
std::size_t length = sizeof(array) / sizeof(array[0]);
// или
std::size_t length = sizeof(array) / sizeof(<type>);
```

Защо не просто `sizeof(array)`? Функцията `sizeof(<obj>)` Връща бройката байтове, които `<obj>` заема, а не бройката елементи (ако изобщо има такива).

Пример:

```
int arr[10];

// 40 байта данни на масив от цели числа
std::cout << "Array bytes: " << sizeof(arr) << endl;

// 4 байта данни от тип цяло число
std::cout << "int bytes: " << sizeof(arr[0]) << endl;

// 40 байта / 4 байта = 10 цели числа
std::size_t length = sizeof(arr) / sizeof(arr[0]);
std::cout << "Array length: " << length << endl;
```

Резултат:

```
Array bytes: 40
int bytes: 4
Array length: 10
```

4.2 Структури

Структурите (`struct`) са съставни типове данни, които ни позволяват да групираме данни от различни типове под едно име, създавайки структурирани данни. Данните в структурите се наричат членове. Структурите са прости за използване и са подходящи за групиране на данни, които не изискват скриване на данни или контрол на достъпа.

Дефиниция:

```
// Именувана структура, дефинира се извън функциите
struct <Struct_Type> {
    <type_1> <var_1>;
    <type_2> <var_2> = <default_value>;
    <type_3> <var_3> , ... , <var_n>; // Няколко еднотипни данни
    ...
};

// Инстанциализация
struct <Struct_Type> <struct_name>;

// Инициализация
<struct_name>.<var_1> = <value_1>;
<struct_name>.<var_2> = <value_2>;
...

// Инстанциализация + Инициализация
struct <Struct_Type> <struct_name> = { <value_1> , ... , <value_n>
   };

// struct променлива, дефинира се във функция или клас
struct {
    ... // Стандартно дефиниране на данните
} <struct_name>;
```

Пример:

```
struct Person {
    std::string name; // Име (текст)
    int age; // Възраст (цяло число)
    double height; // Ръст (число с плаваща запетая)
};
```

Структурата `Person` дефинира нов тип данни, който съдържа три члена: `name`, `age` и `height`.

Членовете на структурата могат да бъдат от различни типове.

Пример за работа с обект от тип `Person`:

```
Person p1; // Създава обект от тип ‘Person‘

p1.name = "Ivan"; // Присвоява името "Ivan"на члена ‘name‘
p1.age = 25; // Присвоява възрастта 25 на члена ‘age‘
p1.height = 1.80; // Присвоява ръста 1.80 на члена ‘height‘

// или

Person p1 = {"Ivan", 25, 1.80};

std::cout << "The name is: " << p1.name << std::endl;
std::cout << "The age is: " << p1.age << std::endl;
std::cout << "The height is: " << p1.height << std::endl;
```

Резултат:

```
The name is: Ivan
The age is: 25
The height is: 1.8
```

Пример за използване на `struct`:

```
#include <iostream>
#include <string>

using namespace std;

struct Student {
    string name;
    int age;
    int grades[5];
};

int main() {
    Student s1;

    s1.name = "Maria";
    s1.age = 18;
    s1.grades[0] = 5;
    s1.grades[1] = 6;
    s1.grades[2] = 4;
    s1.grades[3] = 5;
    s1.grades[4] = 6;
```

```
23
24      cout << "Name: " << s1.name << endl;
25      cout << "Age: " << s1.age << endl;
26      cout << "Grades: ";
27      for (int i = 0; i < 5; i++) {
28          cout << s1.grades[i] << " ";
29      }
30      cout << endl;
31
32      return 0;
33  }
```

Резултат:

```
1  Name: Maria
2  Age: 18
3  Grades: 5 6 4 5 6
```

4.3 Обединения

Обединенията (`union`) са съставни типове данни, които ни позволяват да съхраняваме различни типове данни в една и съща памет. В даден момент може да се съхранява само една от стойностите на членовете. Достъпът до различните членове се осъществява чрез името на обединението, последвано от името на члена.

Дефиниция:

```
1  union <Union_Name> {
2      <type_1> <var_1>;
3      <type_2> <var_2>;
4      ...
5  };
```

Пример:

```
1  union Data {
2      int i;
3      float f;
4      char c;
5  };
```

`union Data` - Дефинира нов тип данни, наречен Data, който е обединение.
`int i`, `float f`, `char c` - Членовете на обединението, които могат да съхраняват цели числа, числа с плаваща запетая и символи.

Пример за използване на `union`:

```
int main() {
    Data d;

    d.i = 10; // Съхранява цяло число
    std::cout << "d.i = " << d.i << std::endl;

    d.f = 3.14; // Съхранява число с плаваща запетая
    std::cout << "d.f = " << d.f << std::endl;

    d.c = 'A'; // Съхранява символ
    std::cout << "d.c = " << d.c << std::endl;

    return 0;
}
```

Резултат:

```
d.i = 10
d.f = 3.14
d.c = A
```

Важно: Обединенията са небезопасни, тъй като не е възможно да се проследи кой член е активен в даден момент. Не е препоръчително да се използват обединения, освен ако не е абсолютно необходимо. Алтернативи на обединенията са структурите и класовете, които предлагат по-безопасни начини за групиране на данни.

4.4 Изброяване

Изброяванията (`enum`) ни позволяват да дефинираме собствени типове данни, които представляват набор от константи. Тези константи са имена, които представляват числови стойности. Изброяванията ни позволяват да ограничим възможните стойности на променливите. Те улесняват читаемостта на кода, тъй като имена се използват вместо числа.

Пример:

```
enum Color { RED, GREEN, BLUE };
```

`enum Color` - Дефинира нов тип данни, наречен Color, който е изброяване.
`RED`, `GREEN`, `BLUE` - Имената на константите, които са част от изброяването.
По подразбиране константите в изброяването получават числови стойности, започвайки от 0.

Пример за използване на `enum`:

```
int main() {
  Color c = GREEN; // Задава стойността на ‘c‘ на ‘GREEN‘
  std::cout << "c = " << c << std::endl; // Извежда: c = 1

  return 0;
}
```

Изброяванията могат да се инициализират ръчно с числови стойности.

Пример за ръчно инициализиране:

```
enum Weekday { MON = 1, TUE, WED, THU, FRI, SAT, SUN };
```

`MON` е инициализиран с `1`.
`TUE`, `WED`, `THU`, `FRI`, `SAT`, `SUN` са инициализирани с `2`, `3`, `4`, `5`, `6`, `7` съответно.
Изброяванията са полезни за:

– Дефиниране на константи, които са лесни за четене и разбиране.

– Ограничаване на възможните стойности на променливите.

– Управление на типове данни в приложения, където е необходимо да се използват ограничен брой стойности.

4.5 Класове

Класовете са основен градивен елемент в обектно-ориентираното програмиране (ООП) в C++ . Те ни позволяват да дефинираме собствени типове данни, които инкапсулират данни и функции, свързани с тях. По този начин може да се описват обекти от истинския свят, редом с техните аспекти и функционалности.

Основни характеристики на класовете:

- Атрибути: Данните, които се съхраняват в обекти от класа. Те се дефинират като член-данни на класа.
- Методи: Функциите, които оперират с данните на класа, или с други класове и функции. Те се дефинират като член-функции на класа.
- Инкапсулация: Скриване на данните и методите на класа от външния свят.
- Абстракция: Представяне на класа само с неговите публични интерфейси, без да се показват подробности за имплементацията.
- Наследственост: Възможност за създаване на нови подкласове, които наследяват свойствата и методите на съществуващи класове.
- Полиморфизъм: Възможност за различно поведение на методите, в зависимост от типа на обекта, към който се прилагат.

Дефиниция:

```
class <Class_Name> {
<access_specifier>: // private, protected или public
    ... // Променливи

    // Конструктор
    <Class_Name>(list_of_parameters) {
        ... // Дефиниция на обекта
    }

    ... // Функции

    // Деструктор
    ~<Class_Name>() {
        ... // Деконструкция на обекта
    }
}
```

- `class` - Ключова дума за създаване на клас.
- `<access_specifier>` - Ключови думи, определящи достъпа до конкретни променливи, функции, конструктори или деструктори.
  `private:` скрива данните от други класове. `protected:` скрива данните от всички класове, които не унаследяват този клас. `public:` прави данните видими от всички класове.
- Конструктор - Специален метод, който се извиква при създаването на обект от типа на класа. Ако не е дефиниран от потребителя, компилатора го дефинира автоматично по стандарт.
- Деструктор - Специален метод, който се извиква при унищожаването на дадена инстанция на класа. Ако не е дефиниран от потребителя, компилатора го дефинира автоматично по стандарт.

Пример за дефиниране на клас:

```
class Rectangle {
private:
    double width;
    double height;

public:
    // Конструктор
    Rectangle(double w, double h) : width(w), height(h) {}

    // Методи
    double getArea() const {
        return width * height;
    }

    double getPerimeter() const {
        return 2 * (width + height);
    }
};
```

- `class Rectangle` - Дефинира нов тип данни, наречен `Rectangle`.
- `private:` - Секция, която съдържа частните член-данни на класа. Те са недостъпни отвън.

- `width` и `height` - Частните член-данни, които съхраняват ширината и височината на правоъгълника.
- `public:` - Секция, която съдържа публичните член-функции на класа. Те са достъпни отвън.
- `Rectangle(double w, double h)` - Конструктор, който се извиква при създаване на обект от тип `Rectangle`.
- `getArea()` и `getPerimeter()` - Публични член-функции, които връщат площта и периметъра на правоъгълника.

Пример за използване на клас:

```
int main() {
    Rectangle rect1(5.0, 3.0); // Създава обект от тип ‘Rectangle‘

    std::cout << "The area is: " << rect1.getArea() << std::endl;
    std::cout << "The perimeter is: " << rect1.getPerimeter() << std
    ::endl;
    return 0;
}
```

Резултат:

```
The area is: 15
The perimeter is: 16
```

4.6 Колекции

Колекциите в C++ са структури от данни, които ни позволяват да съхраняваме множество елементи от един и същ тип. Тези структури ни дават гъвкавост при обработката на данни, като ни позволяват да добавяме, премахваме, сортираме и търсим елементи по ефективен начин.

4.6.1 Вектор

Векторите в C++ са динамични масиви, които съхраняват елементи от един и същ тип. За разлика от статичните масиви, векторите могат да разширяват размера си динамично, което ги прави гъвкави за работа с променлив брой елементи.

Дефиниция:

```cpp
#include <vector>

std::vector<int> numbers; // Дефинира вектор от цели числа
std::vector<std::string> names; // Дефинира вектор от низове
```

Добавяне на елементи:

```cpp
numbers.push_back(10); // Добавя 10 към края на вектора
numbers.push_back(20); // Добавя 20 към края на вектора
names.push_back("Ivan"); // Добавя "Ivan" към края на вектора
```

Достъп до елементи:

```cpp
int firstNumber = numbers[0]; // Достъп до първия елемент
std::string secondName = names[1]; // Достъп до втория елемент
```

Размер на вектора:

```cpp
int size = numbers.size(); // Връща броя на елементите във вектора
```

Итериране през вектора:

```cpp
for (int i = 0; i < numbers.size(); i++) {
    std::cout << numbers[i] << " ";
}
```

Изтриване на елементи:

```
numbers.erase(numbers.begin() + 1); // Изтрива втория елемент
```

Други полезни методи:

- `clear()`: Изтрива всички елементи от вектора.
- `empty()`: Връща `true`, ако векторът е празен, и `false` в противен случай.
- `front()`: Връща референция към първия елемент.
- `back()`: Връща референция към последния елемент.
- `insert()`: Вмъква елемент на определена позиция.

Предимства на векторите:

– Динамично разширяване: Могат да съхраняват променлив брой елементи.

– Лесна работа: Предлагат удобни методи за добавяне, премахване, достъп и итериране.

– Ефективност: Осигуряват бърз достъп до елементите.

4.6.2 Списък

Списъците в C++ са динамични структури от данни, които съхраняват елементи в свързана верига. Всеки елемент в списъка съдържа данни и посока към следващия елемент.

Дефиниция:

```
#include <list>

std::list<int> numbers; // Дефинира списък от цели числа
std::list<std::string> names; // Дефинира списък от низове
```

Добавяне на елементи:

```
numbers.push_back(10); // Добавя 10 в края на списъка
numbers.push_front(5); // Добавя 5 в началото на списъка
names.push_back("Ivan"); // Добавя "Ivan" в края на списъка
```

Достъп до елементи: Списъците не поддържат индексиране. За достъп до елементи се използват итератори.

```
std::list<int>::iterator it = numbers.begin(); // Итератор към началото
int firstNumber = *it; // Достъп до първия елемент
```

Размер на списъка:

```
int size = numbers.size(); // Връща броя на елементите в списъка
```

Итериране през списък:

```
for (
    std::list<int>::iterator it = numbers.begin();
    it != numbers.end();
    ++it
) {
    std::cout << *it << " ";
}
```

Изтриване на елементи:

```
numbers.erase(numbers.begin()); // Изтрива първия елемент
```

Други полезни методи:

- `clear()`: Изтрива всички елементи от списъка.
- `empty()`: Връща `true`, ако списъкът е празен, и `false` в противен случай.
- `front()`: Връща референция към първия елемент.

- `back()`: Връща референция към последния елемент.
- `insert()`: Вмъква елемент на определена позиция.
- `sort()`: Сортира елементите в списъка.
- `reverse()`: Обръща реда на елементите в списъка.

Предимства на списъците:

– Динамично разширяване: Могат да съхраняват променлив брой елементи.

– Ефективност при вмъкване/изтриване: Позволяват бързо вмъкване и изтриване на елементи на всяка позиция.

– Гъвкавост: Поддържат различни операции, като сортиране, обръщане и т.н.

4.6.3 Стек

Стекът в C++ е линейна структура от данни, която съхранява елементи по правилото LIFO (Last In, First Out) - последният добавен елемент е първият, който се извлича. Може да се представи като купчина предмети, където можем да добавяме предмети отгоре и да изваждаме само отгоре.

Дефиниция:

```cpp
#include <stack>

std::stack<int> numbers; // Дефинира стек от цели числа
std::stack<std::string> names; // Дефинира стек от низове
```

Добавяне на елементи:

```cpp
numbers.push(10); // Добавя 10 в стека
numbers.push(20); // Добавя 20 в стека
names.push("Ivan"); // Добавя "Ivan" в стека
```

Извличане на елементи:

```
// Връща последния елемент (без да го изтрива)
int topNumber = numbers.top();
numbers.pop(); // Изтрива последния елемент
```

Проверка за празен стек:

```
bool isEmpty = numbers.empty(); // Връща true, ако стекът е празен
```

Размер на стека:

```
int size = numbers.size(); // Връща броя на елементите в стека
```

Предимства на стека:

- Ефективност: Операциите за добавяне (`push`) и извличане (`pop`) са бързи.
- Проста реализация: Лесна за разбиране и имплементиране.
- Широко приложение: Използва се в разнообразни алгоритми и структури от данни, например рекурсия, обработка на изрази, обработка на грешки.

4.6.4 Опашка

Опашката в C++ е линейна структура от данни, която съхранява елементи по правилото FIFO (First In, First Out) - първият добавен елемент е първият, който се извлича. Може да се представи като опашка в магазин, където клиентите се обслужват по ред.

Дефиниция:

```
#include <queue>

std::queue<int> numbers; // Дефинира опашка от цели числа
std::queue<std::string> names; // Дефинира опашка от низове
```

Добавяне на елементи:

```
numbers.push(10); // Добавя 10 в опашката
numbers.push(20); // Добавя 20 в опашката
names.push("Ivan"); // Добавя "Ivan" в опашката
```

Извличане на елементи:

```
// Връща първия елемент (без да го изтрива)
int frontNumber = numbers.front();
numbers.pop(); // Изтрива първия елемент
```

Проверка за празна опашка:

```
bool isEmpty = numbers.empty(); // Връща true, ако опашката е празна
```

Размер на опашката:

```
int size = numbers.size(); // Връща броя на елементите в опашката
```

Предимства на опашката:

- Ефективност: Операциите за добавяне (`push`) и извличане (`pop`) са бързи.
- Проста реализация: Лесна за разбиране и имплементиране.
- Широко приложение: Използва се в разнообразни алгоритми и структури от данни, например обработка на задачи, симулация на системи, мрежови протоколи.

4.6.5 Двойнокрайна опашка

Двойнокрайна опашка (Deque, произлиза от "double-ended queue") в `C++` е линейна структура от данни, която позволява добавяне и извличане на елементи както от началото, така и от края. Може да се представи като опашка, която е отворена и от две страни.

Дефиниция:

```cpp
#include <deque>

std::deque<int> numbers; // Дефинира двойнокрайна опашка от цели числа
std::deque<std::string> names; // Дефинира двойнокрайна опашка от низове
```

Добавяне на елементи:

```cpp
numbers.push_front(10); // Добавя 10 в началото на опашката
numbers.push_back(20); // Добавя 20 в края на опашката
names.push_front("Ivan"); // Добавя "Ivan" в началото на опашката
```

Извличане на елементи:

```cpp
// Връща първия елемент (без да го изтрива)
int frontNumber = numbers.front();
numbers.pop_front(); // Изтрива първия елемент

// Връща последния елемент (без да го изтрива)
int backNumber = numbers.back();
numbers.pop_back(); // Изтрива последния елемент
```

Проверка за празна двойнокрайна опашка:

```cpp
bool isEmpty = numbers.empty(); // Връща true, ако опашката е празна
```

Размер на двойнокрайна опашка:

```cpp
int size = numbers.size(); // Връща броя на елементите в опашката
```

Предимства на двойнокрайна опашка:

- Гъвкавост: Позволява добавяне и извличане от две страни, което я прави по-гъвкава от обикновена опашка.
- Ефективност: Операциите за добавяне и извличане са бързи.
- Широко приложение: Използва се в разнообразни алгоритми и структури от данни, например обработка на буфери, реализация на стекове, реализация на опашки.

4.6.6 Множество

Множеството (Set) в C++ е структура от данни, която съхранява елементи в подреден ред, без дубликати. Може да се представи като множество в математиката, където всеки елемент се появява само веднъж.

Дефиниция:

```cpp
#include <set>

std::set<int> numbers; // Дефинира множество от цели числа
std::set<std::string> names; // Дефинира множество от низове
```

Добавяне на елементи:

```cpp
numbers.insert(10); // Добавя 10 в множеството (ако не е вече там)
numbers.insert(20); // Добавя 20 в множеството (ако не е вече там)
names.insert("Ivan"); // Добавя "Ivan" в множеството (ако не е вече там)
```

Проверка за наличие на елемент:

```cpp
bool contains = numbers.count(10); // Връща true, ако 10 е в множеството
```

Изтриване на елемент:

```cpp
numbers.erase(10); // Изтрива 10 от множеството (ако е там)
```

Размер на множеството:

```cpp
int size = numbers.size(); // Връща броя на елементите в множеството
```

Предимства на множеството:

- Подреждане: Елементите са подредени по ключ, което улеснява търсенето.
- Ефективност: Операциите за добавяне, изтриване и търсене са бързи.
- Единственост: Не се допускат дубликати, което гарантира уникалност на елементите.
- Широко приложение: Използва се в разнообразни алгоритми и структури от данни, например проверка за дубликати, търсене на минимален или максимален елемент, реализация на графове.

4.6.7 Карта

Картата (Map) в C++ е структура от данни, която съхранява двойки от ключ и стойност, като ключът е уникален. Може да се представи като речник, където всеки ключ е свързан с единствена стойност.

Дефиниция:

```cpp
#include <map>

// Дефинира карта, където ключовете са цели числа, а стойностите са низове
std::map<int, std::string> names;
// Дефинира карта, където ключовете са низове, а стойностите са цели числа
std::map<std::string, int> ages;
```

Добавяне на елементи:

```cpp
names[10] = "Ivan"; // Добавя двойка (10, "Ivan") в картата
// Добавя двойка (20, "Peter") в картата
names.insert(std::make_pair(20, "Peter"));
ages["Ivan"] = 25; // Добавя двойка ("Ivan" 25) в картата
```

Достъп до стойност по ключ:

```cpp
std::string name = names[10]; // Връща стойността, свързана с ключа 10
int age = ages["Ivan"]; // Връща стойността, свързана с ключа "Ivan"
```

Проверка за наличие на ключ:

```cpp
bool exists = names.count(10); // Връща true, ако ключът 10 е в картата
```

Изтриване на елемент:

```cpp
names.erase(10); // Изтрива двойката, свързана с ключа 10
```

Размер на картата:

```
int size = names.size(); // Връща броя на елементите в картата
```

Предимства на картата:

- Ефективност: Операциите за добавяне, изтриване и търсене са бързи.
- Уникалност: Не се допускат дубликати на ключове, което гарантира единственост.
- Асоциация: Свързва ключове с стойности, което улеснява достъпа до данни.
- Широко приложение: Използва се в разнообразни алгоритми и структури от данни, например реализация на речници, търсене на данни по ключ, обработка на конфигурационни файлове.

4.7 Ключовата дума `typedef`

`typedef` в `C++` е ключова дума, която ви позволява да дефинирате ново име за съществуващ тип данни. Това може да улесни кода ви, като го направи по-четлив и по-лесен за разбиране. Може да се използва също извън функции и класове, т.е. да се дефинират прякори на типове данни извън протичането на програмата. `typedef` не създава нов тип данни, а само дефинира ново име за съществуващ. Не може да се използва за дефиниране на нови класове или структури.

Синтаксис:

```
typedef <type_name> <new_type_name>;
```

Пример:

```
// Дефинира новото име ULLI за типа unsigned long long int
typedef (unsigned long long int) ULLI;
```

Ползи от `typedef`:

- По-четлив код: Използването на ясни имена за типове може да улесни разбирането на кода.
- По-лесно пренаписване: Ако трябва да промените типа на променлива, можете да промените само `typedef` дефиницията.
- По-кратки имена: Можете да създадете кратки имена за дълги типове данни.

Пример: за използване:

```
// Дефинира по-кратко име за типа итератор през списък от цели числа
typedef (std::list<int>::iterator) IntLstIterator;

IntLstIterator it = nullptr;
```

Глава 5

Манипулиране на паметта

В C++ можем да манипулираме паметта директно, използвайки указатели, адреси и адресна аритметика. Това ни дава голяма гъвкавост в управлението на програмното изпълнение и ни позволява да оптимизираме на системно ниво заетостта на паметта, но изисква внимателно боравене, за да се избегнат грешки и проблеми с сигурността.

5.1 Указатели

Указателите са мощен инструмент в C++ , който ви позволява да манипулирате паметта директно. Те са променливи, които съхраняват адреса на друга променлива. С помощта на указатели можем да достъпваме и модифицираме стойността на променливата, на която сочат.

Дефиниране на указател

За да дефинираме указател, използваме символа * пред името на променливата. Типът на указателя трябва да съответства на типа на променливата, чийто адрес ще съхранява.

Синтаксис:

```
<type> *<name>; // Конвенционално избран синтаксис
\\ или
<type>* <name>;
\\ или
<type> * <name>;
```

Пример:

```
// Дефинира указател ptr, който може да съхранява
// адрес на променлива от тип int
int *ptr;

// Дефинира указател dptr, който може да съхранява
// адрес на променлива от тип double
double *dptr;
```

Задаване на адрес на указател

За да зададем адрес на указател, използваме оператора `&` пред името на променливата. Това се нарича директно адресиране на променлива.

Пример:

```
int num = 10;
ptr = &num; // Задава адреса на num на указателя ptr
```

Достъп до стойността на променливата

За да достъпваме стойността на променливата, на която сочи указателят, използваме оператора `*` пред името на указателя. Това се нарича директно дереференциране на указателя.

Пример:

```
// Връща стойността, съхранявана на адреса, на който сочи ptr
int value = *ptr;
```

Модифициране на стойността на променливата

За да модифицираме стойността на променливата, на която сочи указателят, използваме оператора `*` пред името на указателя, последвано от знак за присвояване.

Пример:

```
*ptr = 20; // Променя стойността на променливата, на която сочи ptr, на 20
```

Нулев указател

Указателите могат да бъдат нулеви, което означава, че не сочат към никаква променлива. Нулевият указател се представя с константата `nullptr`.

```
ptr = nullptr; // Променя ptr да не сочи към никъде
```

Използване на указатели

– Предаване на аргументи по адрес: При предаване на аргументи във функция, функцията създава копие на техните стойности и работи съответно с копията, а не с оригиналните аргументи. Указателите ни позволяват да предаваме аргументи на функции по адрес, което ни дава възможност да модифицираме стойността на оригиналната променлива.

– Динамично разпределение на паметта: Указателите се използват за динамично разпределение на паметта, което ни позволява да създаваме променливи по време на изпълнение.

– Работа с масиви: Указателите са ефективен начин за достъп до елементите на масив.

Препоръки за работа с указатели:

– Винаги проверявайте стойността на указателя, преди да достъпвате променливата, на която сочи.

– Бъдете внимателни при използване на адресна аритметика, тъй като може да доведе до грешки в паметта.

5.2 Референции

Референциите в C++ са специален тип променлива, която не е самостоятелна, а е псевдоним на съществуваща променлива. Те са мощен инструмент, който ни позволява да работим с данни по по-ефективен и удобен начин.

Дефиниране на референция

За да дефинираме референция, използваме символа `&` след типа на променливата, следван от името на референцията и знак за присвояване, последван от името на променливата, на която референцията ще е псевдоним.

Синтаксис:

```
<type> &<ref_name> = <var_name>; // Конвенционално избран синтаксис
// или
<type>& <ref_name> = <var_name>;
// или
<type> & <ref_name> = <var_name>;
```

Пример:

```
int num = 10;
int &ref = num; // ref е референция към num
```

В този пример `ref` е псевдоним на `num`. Всяка операция, извършена над `ref`, действа директно върху `num`.

Характеристики на референциите

- Не са самостоятелни: Референциите не съхраняват стойност сами. Те не са променливи в класическия смисъл.
- Псевдоним: Референцията е просто друго име за съществуваща променлива.
- Не могат да бъдат презададени: След като референцията е дефинирана, тя е свързана с променливата си завинаги. Не може да се презададе да сочи към друга променлива.
- Не могат да бъдат нулеви: Референциите винаги трябва да сочат към валидна променлива. Не могат да сочат към невалиден адрес.

Примери за използване на референции

Предаване на аргументи по адрес: Референциите ни позволяват да предаваме аргументи на функции по адрес, без да създаваме нови променливи. Това е по-ефективно и по-ясно от използването на указатели.

Пример:

```
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 10, y = 20;
    swap(x, y);
    cout << "x: " << x << ", y: " << y << endl; // x: 20, y: 10
    return 0;
}
```

Увеличаване на ефективността: Референциите са по-ефективни от указателите, тъй като не създават нови променливи. Те работят директно с оригиналната променлива, без да копират стойността.

5.3 Адресна аритметика

Адресна аритметика е специална форма на аритметика, която се прилага към указатели. Тя ни позволява да манипулираме адресите, съхранявани в указателите, за да достъпваме различни места в паметта.

5.3.1 Основни операции

Събиране

Когато събираме цяло число към указател, адресът, към който сочи, се премества напред с броя байтове, съответстващи на типа на променливата, към която сочи указателят.

Пример:

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr; // ptr сочи към arr[0]
ptr += 2; // ptr сега сочи към arr[2]
```

Изваждане

Когато изваждаме цяло число от указател, адресът, към който сочи, се премества назад с броя байтове, съответстващи на типа на променливата, към която сочи указателят.

Пример:

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr + 3; // ptr сочи към arr[3]
ptr -= 1; // ptr сега сочи към arr[2]
```

Разлика

Можем да извадим два указателя, сочещи към елементи от един и същ масив. Резултатът е разликата в индексите между двамата указателя.

Пример:

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr1 = arr + 2; // ptr1 сочи към arr[2]
int *ptr2 = arr; // ptr2 сочи към arr[0]
int diff = ptr1 - ptr2; // diff е 2
```

Важно забележка

- Адресна аритметика работи само с указатели, сочещи към елементи от един и същ масив.
- Преместването на указател извън границите на масива може да доведат до непредвидени резултати и грешки.

Примери за използване на адресната аритметика

– Обработка на масиви: Адресна аритметика е необходима за ефективна обработка на масиви.

– Работа с динамично разпределена памет: Адресна аритметика се използва за манипулиране на блокове от памет, разпределени динамично.

– Реализация на алгоритми: Адресна аритметика е ключова за реализацията на много алгоритми, като сортиране, търсене и обработка на списъци.

5.4 Динамично и статично разпределяне на паметта

В C++ , паметта, необходима за съхранение на данни, се разпределя по два основни начина: статично и динамично. Всеки от тези методи има своите предимства и недостатъци, което ги прави подходящи за различни ситуации.

5.4.1 Статично разпределяне на паметта

При статичното разпределяне на паметта, компилаторът определя размера и местоположението на паметта, необходима за променливите, преди програмата да започне да се изпълнява. Разпределя се в стека на паметта и съществува през целия жизнен цикъл на програмата. Тази памет е фиксирана и не може да се променя по време на изпълнение.

Характеристики:

– Заделяне по време на компилация: Паметта се заделя автоматично от компилатора, когато програмата се компилира.

– Фиксиран размер: Размерът на паметта, заделена за променливите, е фиксиран и не може да се променя по време на изпълнение.

– Достъпност през целия жизнен цикъл: Променливите, за които е заделена памет статично, са достъпни през целия жизнен цикъл на програмата.

– Лесно управление: Управлението на статично заделена памет е по-лесно, тъй като компилаторът се грижи за нея.

– По-малко гъвкаво: Статичното разпределяне е по-малко гъвкаво, тъй като размерът на паметта не може да се променя по време на изпълнение.

Примери:

- Локални променливи: Локалните променливи, дефинирани в функции, се заделят статично.
- Глобални променливи: Глобалните променливи, дефинирани извън функции, се заделят статично.
- Статични променливи: Статичните променливи, дефинирани с ключовата дума static, се заделят статично.

Пример:

```
int num = 10; // Паметта за num се заделя статично
```

5.4.2 Динамично разпределяне на паметта

При динамичното разпределяне на паметта, програмата заделя памет по време на изпълнение, когато е необходимо. Разпределя се върху хийпа на паметта. Тази памет е гъвкава и може да се разширява или намалява по време на изпълнение.

Характеристики:

- Заделяне по време на изпълнение: Паметта се заделя динамично от програмата по време на изпълнение, когато е необходимо.
- Променлив размер: Размерът на паметта, заделена динамично, може да се променя по време на изпълнение.
- Достъпност до освобождаване: Паметта, заделена динамично, е достъпна докато не бъде освободена от програмата.
- По-гъвкаво: Динамичното разпределяне е по-гъвкаво, тъй като размерът на паметта може да се променя по време на изпълнение.
- По-сложно управление: Управлението на динамично заделена памет е по-сложно, тъй като програмистът трябва да се грижи за освобождаването на паметта, когато вече не е необходима.

Примери:

– Динамични масиви: Динамичните масиви, дефинирани с ключовата дума new, се заделят динамично.

– Структури и класове: Паметта за структури и класове може да се заделя динамично, когато е необходимо.

Пример:

```
int *ptr = new int; // Паметта за ptr се заделя динамично
...
delete ptr; // Освобождаване на паметта
```

Използване на статично и динамично разпределяне

Статичното разпределяне е подходящо за променливи, чийто размер е известен преди компилация, или за променливи, чийто размер е фиксиран и не се променя по време на изпълнение.

Динамичното разпределяне е подходящо за променливи, чийто размер е неизвестен преди компилация, или за променливи, чийто размер се променя по време на изпълнение.

Важно е да се отбележи:

Освобождаване на паметта: Когато се използва динамично разпределяне, е важно да се освободи паметта, когато вече не е необходима. В противен случай, програмата може да изчерпи паметта.

Прекаленото използване на динамично разпределяне: Прекаленото използване на динамично разпределяне може да доведе до проблеми с производителността, тъй като програмата трябва да се справя с операции за заделяне и освобождаване на памет.

Глава 6

Алгоритми

Алгоритмите са основата на програмирането. Те са последователност от стъпки, които описват как да се реши даден проблем. Без алгоритми, програмите биха били хаотични и непредсказуеми.

Значението на алгоритмите в програмирането се крие в следните аспекти:

- Ефективност: Алгоритмите определят ефективността на програмата. Ефективни алгоритми използват по-малко ресурси (време, памет) за решаване на проблем.
- Яснота: Алгоритмите описват логиката на програмата по ясен и структуриран начин.
- Повторна употреба: Алгоритмите могат да се използват повторно в различни програми.
- Разбиране: Алгоритмите помагат на програмистите да разберат как работи програмата.
- Управление на сложността: Алгоритмите помагат да се управява сложността на програмите, особено при решаване на големи проблеми.

6.1 Анализа на сложността

Анализът на сложността на алгоритмите е ключов аспект в програмирането, който ни позволява да оценим ефективността на алгоритмите и да сравним различни алгоритми за решаване на един и същ проблем. Сложността на алгоритъма се измерва с времевата и паметната сложност.

За тази цел се използва асимптотичната нотация:

- $O(\dots)$ ("Голямо-О" Нотация) - обозначава най-лошият случай за протичане на алгоритъма ("Протича не по-бавно отколкото ... ")
- $\Theta(\dots)$ ("Тета" Нотация) - обозначава средната сложност на алгоритъма ("Протича толкова бързо колкото ... ")
- $\Omega(\dots)$ ("Омега" Нотация) - обозначава най-добрият случай за протичане на алгоритъма ("Протича не по-бързо отколкото ... ")

O, Ω и Θ са специални оператори, обозначаващи множества от функции от даден порядък на величина:

$$f(n) \in \begin{cases} O(g(n)) \Leftrightarrow \ f \preceq g \Leftrightarrow \ \limsup_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| < \infty \\ \Theta(g(n)) \Leftrightarrow \ f \asymp g \Leftrightarrow \ \lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| \in (0, \infty) \\ \Omega(g(n)) \Leftrightarrow \ f \succeq g \Leftrightarrow \ \liminf_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| > 0 \end{cases}$$

Свойства:

$$f(n) + g(n) = O(\max(f, g)(n))$$
$$f(n) \cdot g(n) = O((f \cdot g)(n))$$

Времева сложност

Времевата сложност описва колко време е необходимо на алгоритъма да се изпълни в зависимост от размера на входните данни. Измерва се с броя на операциите, които алгоритъмът извършва.

Означения:

$O(1)$: Константна сложност - броят на операциите е независим от размера на входните данни.

$O(n)$: Линейна сложност - броят на операциите е пропорционален на размера на входните данни.

$O(n^2)$: Квадратна сложност - броят на операциите е пропорционален на квадрата на размера на входните данни.

$O(\log n)$: Логаритмична сложност - броят на операциите е пропорционален на логаритъма на размера на входните данни.

Пример:

Линейно търсене: Алгоритъмът за линейно търсене има времева сложност $O(n)$, тъй като преглежда всички елементи на масива, за да намери търсеното число.

Двоично търсене: Алгоритъмът за двоично търсене има времева сложност $O(\log n)$, тъй като разделя масива на половина при всяка стъпка.

Как се изчислява времевата сложност на стъпките:

Линеен код: събира се.

Вложени цикли: умножават се.
(Примерно два вложени `for` цикъла протичат в $O(n^2)$)

Рекурсии: в зависимост от бройката и размерите на подпроблемите се използва Мастър теорема.

Паметна сложност

Паметната сложност описва колко памет е необходима на алгоритъма да се изпълни в зависимост от размера на входните данни. Измерва се с броя на променливите, които алгоритъмът използва.
Означения:

$O(1)$: Константна сложност - алгоритъмът използва фиксирано количество памет.

$O(n)$: Линейна сложност - алгоритъмът използва пропорционално на размера на входните данни количество памет.

$O(n^2)$: Квадратна сложност - алгоритъмът използва пропорционално на квадрата на размера на входните данни количество памет.

Пример:

Сортиране с мехурчета: Алгоритъмът за сортиране с мехурчета има паметна сложност $O(1)$, тъй като използва само фиксирано количество памет за променливи.

Сортиране със сливане: Алгоритъмът за сортиране с сливане има паметна сложност $O(n)$, тъй като използва допълнителен масив с размер n за съхранение на междинните резултати.

Мастър теорема

Мастър теоремата е важен инструмент в анализа на алгоритмите, особено за алгоритми от тип "разделяй и владей". Тя ни позволява да намерим асимптотичната сложност на рекурсивни алгоритми, които разделят проблема на k по-малки подпроблеми, решават ги рекурсивно и след това комбинират решенията.

Мастър теоремата се прилага за рекурентни уравнения от вида:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) =$$

$$= aT\left(\frac{n}{b}\right) + \Theta(n^c \log^k(n)) = \begin{cases} \Theta(1) & , n = 1 \\ \Theta(n^c) & , a < b^c,\ k = 0 \\ \Theta(n^c \log^{k+1}(n)) & , a = b^c,\ k > -1 \\ \Theta(n^c \log\log(n)) & , a = b^c,\ k = -1 \\ \Theta(n^c) & , a = b^c,\ k < -1 \\ \Theta(n^{\log_b(a)}) & , a > b^c \end{cases}$$

където:

$T(n)$ е времевата сложност на алгоритъма за вход с размер n.
a е броят на подпроблемите.
b е размерът на всеки подпроблем.
$f(n)$ е времевата сложност на комбинирането на решенията на подпроблемите.

Значение на анализа на сложността

- Сравняване на алгоритми: Анализът на сложността ни позволява да сравним различни алгоритми за решаване на един и същ проблем и да изберем най-ефективния.
- Оптимизиране на алгоритми: Анализът на сложността ни помага да оптимизираме алгоритмите, като намалим времевата и паметната им сложност.
- Разбиране на ограниченията: Анализът на сложността ни показва ограниченията на алгоритмите, например, кога алгоритъмът става непрактичен за големи размери на входните данни.

6.2 Търсене

Търсенето на елементи е често срещана задача в програмирането, при която се цели да се намери конкретен елемент в множество от данни. Различните алгоритми за търсене се различават по ефективност, сложност и приложимост за различни видове данни.

Линейно търсене

Линейното търсене е най-простият алгоритъм за търсене на елемент в множество от данни. Този алгоритъм преглежда последователно всеки елемент в множеството, докато не намери търсения елемент или не стигне до края на множеството.

Как работи:

1. Започва от първия елемент в множеството.
2. Сравнява текущия елемент с търсения елемент.
3. Ако елементите са равни, алгоритъмът е успешен и връща индекса на елемента.
4. Ако елементите не са равни, алгоритъмът преминава към следващия елемент в множеството.
5. Процесът се повтаря, докато не се намери търсения елемент или не се стигне до края на множеството.

Предимства:

- Прост за имплементиране: Лесен за разбиране и кодиране.
- Работи с всякакви данни: Не се изисква сортиране на множеството от данни.

Недостатъци:

- Бавен за големи множества от данни: Трябва да се прегледат всички елементи, което може да е бавно за големи множества.
- Времева сложност: $O(n)$: В най-лошия случай, алгоритъмът ще трябва да прегледа всички n елемента.

Реализация в C++ :

```cpp
// Функция за линейно търсене
int linearSearch(int array[], int length, int target) {
    for (int i = 0; i < length; i++) {
        // Върни индекса на елемента, ако е равен на target
        if (array[i] == target) return i;
    }

    return -1; // Върни -1, ако не е намерен елемент
}
```

Двоично търсене

Двоичното търсене е много по-ефективен алгоритъм за търсене в сравнение с линейното търсене, но работи само със **сортирани** множества от данни. Този алгоритъм разделя множеството от данни на половина при всяка стъпка, като отхвърля половината, която не съдържа търсения елемент.

Как работи:

1. Започва от средата на множеството.
2. Сравнява елемента в средата с търсения елемент.
3. Ако елементите са равни, алгоритъмът е успешен и връща индекса на елемента.
4. Ако елементите не са равни, алгоритъмът отхвърля половината от множеството, която не съдържа търсения елемент.
5. Процесът се повтаря, като се разделя останалата част от множеството на половина, докато не се намери търсения елемент или не останат елементи за проверка.

Предимства:

- Бърз за големи множества от данни: Времевата сложност е $O(logn)$, което е много по-бързо от $O(n)$ за линейното търсене.
- Ефективен за големи множества: Позволява бързо намиране на елементи в големи множества от данни.

Недостатъци:

- Работи само със сортирани данни: Не може да се използва за несортирани множества от данни.
- По-сложен за имплементиране: Изисква повече код и логика за имплементиране.

Реализация в C++ :

```cpp
// Функция за двоично търсене
int binarySearch(int array[], int length, int target) {
    int left = 0; // Лявата граница
    int right = length - 1; // Дясната граница

    while (left <= right) {
        int mid = left + (right - left) / 2; // Средата

        if (array[mid] == target) {
            return mid; // Върни индекса
        } else if (array[mid] < target) {
            left = mid + 1; // Отхвърли дясната
        } else {
            right = mid - 1; // Отхвърли дясната
        }
    }

    return -1; // Върни -1, ако не е намерен
}
```

6.2.1 Обхождане на графика

Графите са математически структури, които представляват взаимоотношения между обекти. Те се състоят от върхове (или нодове) и ръбове, които свързват върховете.

Видове графи:

Ненасочени графи: Ръбовете не са ориентирани, т.е. връзката между два върха е двупосочна.

Насочени графи: Ръбовете са ориентирани, т.е. връзката между два върха е еднопосочна.

Тегловни графи: Ръбовете имат тегла, които представляват разстояние, цена или друга величина.

С цел нагледност, ще дефинираме графите по следният начин:

```
// Използваме множество за да избегнем повторни връзки между ръбовете
#include <set>
#include <queue>

// Структура за връх
struct Vertex {
    int value = -1; // Стойност на върха
    bool visited = false; // Обходен ли е върхът
    // Множество от съседните върхове
    std::set<Vertex*> neighborhood;
};

// Клас за граф
class Graph {
private:
    int vNum; // Брой върхове
    Vertex* vertexList; // Списък от върховете

public:
    // Създава граф с n върха
    Graph(int n) : vNum(n) {
        vertexList = new Vertex[n];

        for (int i = 0; i < n; i++) {
            vertexList[i].value = i;
        }
    }

    // Добавя (двупосочна) връзка между върховете a и b
    void addEdge(int a, int b) {
        vertexList[a].neighborhood.insert(&vertexList[b]);
        vertexList[b].neighborhood.insert(&vertexList[a]);
    }

    // DFS и BFS
    ...

}
```

За обхождането на графи и търсенето на елементи във върховете се използват алгоритми като:

Depth-First Search (DFS)

DFS, или Обхождане в дълбочина, е алгоритъм за обхождане на структури от данни, като дървета и графи. Той започва от даден връх, който се обозначава като корен, и обхожда структурата, като се движи по-дълбоко в нея, преди да се върне назад. DFS е рекурсивен алгоритъм, който се реализира чрез рекурсивна функция. Протича в $\Theta(\#\text{върхове}) + \Theta(\#\text{ръбове}) = \Theta(n)$ време.

В реални приложения, DFS се използва за:

– Намиране на всички върхове в граф.

– Проверка за цикли в граф.

– Намиране на най-краткия път в граф.

– Търсене на топологично сортиране на граф.

Реализация в `C++` :

```cpp
// Функция за DFS
Vertex* dfs(int start, int target) {
    Vertex* s = &vertexList[start];

    // Целта е намерена
    if (target == start) return s;

    // Отбелязваме текущия връх като посетен
    s->visited = true;

    for (Vertex *v : s->neighborhood) {
        // Пропуска вече посетени върхове
        if (v->visited) continue;

        // Продължава да търси в следващия връх рекурсивно
        return dfs(v->value, target);
    }

    // Не сочи към нищо когато целта не е намерена
    return nullptr;
}
```

След търсенето е удачно да се върнат `visited` стойностите на върховете на `false`, за да се избегнат нежелани грешки при повторни търсения.

Breadth-First Search (BFS)

BFS, или Обхождане в ширина, е алгоритъм за обхождане на структури от данни, като дървета и графи. Той започва от даден връх, който се обозначава като корен, и обхожда структурата, като се движи по-широко в нея, преди да се движи по-дълбоко. BFS е нерекурсивен алгоритъм, който се реализира чрез опашка. Протича в $\Theta(\#\text{върхове}) + \Theta(\#\text{ръбове}) = \Theta(n)$ време.

В реални приложения, BFS се използва за:

– Намиране на всички върхове в граф.

– Проверка за цикли в граф.

– Намиране на най-краткия път в граф.

Реализация в C++ :

```cpp
// Функция за BFS
Vertex* bfs(int start, int target) {
    std::queue<Vertex*> q; // Празна опашка за върхове
    Vertex *s = &vertexList[start];

    // Целта е намерена
    if (target == start) return s;

    // Отбелязваме текущия връх като посетен
    s->visited = true;
    q.push(s);

    while (!q.empty()) {
        // Вземаме следващия елемент в опашката
        Vertex *u = q.front();
        q.pop(); // Махаме го от опашката

        // Вкарваме всеки съседен връх в опашката
        for (Vertex *v : u->neighborhood) {
            // Пропуска вече посетени върхове
            if (v->visited) continue;

            // Целта е намерена
            if (target == v->value) return v;

            v->visited = true;
            q.push(v);
        }
    }
```

```
86      // Отбелязва обратно всички върхове като непосетени
87      flushVisited();
88
89      // Не сочи към нищо когато целта не е намерена
90      return nullptr;
91  }
```

6.3 Сортиране

Сортирането в C++ е процесът на подреждане на елементите на масив по определен признак. Обикновено се подреждат (сортират) числа по възходящ или низходящ ред.

В C++ има различни алгоритми за сортиране, като Bubble Sort, Insertion Sort, Quick Sort, Merge Sort, и др.

Може да се използва вградената функция `std::sort()`, която е много ефективна и лесна за използване. За да използвате `std::sort()`, трябва да включите заглавния файл <algorithm>. Функцията `std::sort()` приема два итератора, които определят началото и края на масива, който искате да сортирате.

Пример:

```
1  #include <algorithm>
2
3  int main() {
4      int a[] = {5, 2, 8, 1, 9};
5      int n = sizeof(a) / sizeof(a[0]);
6
7      std::sort(a, a + n);
8  }
```

Въпреки лесното използване, не е най-бързият алгоритъм за сортиране. Поради тази причина ще разгледаме някои други алгоритми за сортиране. В алгоритмите, които ще разглеждаме, ще имаме предвид естественото сортиране по големина (от най-малко към най-голямо).

Swap

Размяната на две променливи лежи в основата на повечето алгоритми за сортиране. Стандартно се използва временна променлива, която да съхранява едната от променливите, докато ги разменя.

Реализация в `C++` :

```
// Стандартна функция за размяна на две променливи
void swap(<type> &a, <type> &b) {
    <type> temp = a;
    a = b;
    b = temp;
}
```

За напреднали: Работейки с битовете на променливите, могат да се разменят две променливи, без да се създава временна променлива, чрез битовия оператор XOR (`^`) по следния начин:

```
// Функция за размяна на две променливи чрез XOR
void swap(<type> &a, <type> &b) {
    a ^= b;
    b ^= a;
    a ^= b;
}
```

По този начин могат да се разменят променливи от примитивен тип, или променливи с примитивен характер (напр. `std::string` могат да се разменят по този начин, като се използват `char` адреси в параметрите на функцията).

Bubble sort

Bubble Sort е алгоритъм за сортиране, който е сравнително прост за разбиране и реализиране. Той работи, като сравнява съседни елементи в масив и ги разменя, ако са в неправилен ред. Процесът се повтаря, докато масивът не е сортиран. Bubble Sort е "in-place" алгоритъм, което означава, че не е нужно да се създава нов масив за сортиране. Той е нестабилен алгоритъм, което означава, че елементите с еднаква стойност не запазват относителното си положение в масива след сортиране. Bubble Sort е сравнително бавен алгоритъм, особено за големи масиви.

Реализация в `C++` :

```cpp
// Функция за Bubble Sort
void bubbleSort(int arr[], int size) {
    // Първоначално се предполага, че масивът е сортиран
    bool sorted = true;

    for (int i = 0; i < size - 1; i++) {
        // Разменя съседни елементи, ако са в неправилен ред
        if (arr[i] > arr[i + 1]) {
            // Отбелязва масива за повторна проверка
            sorted = false;
            swap(arr[i], arr[i + 1]);
        }
    }

    // Повтаря сортирането, докато масивът не е сортиран
    if (!sorted) bubbleSort(arr, size);
}
```

Времева сложност: $O(n^2)$

Insertion sort

Insertion Sort е друг алгоритъм за сортиране, който е лесен за разбиране и реализиране. Той работи, като изгражда сортиран подмасив от масива, елемент по елемент. Insertion Sort е "in-place" алгоритъм, което означава, че не е нужно да се създава нов масив за сортиране. Той е стабилен алгоритъм, което означава, че елементите с еднаква стойност запазват относителното си положение в масива след сортиране. Insertion Sort е ефективен за малки масиви, но е бавен за големи масиви.

Реализация в C++ :

```cpp
// Функция за Insertion Sort
void insertionSort(int arr[], int size) {
    // Изследваме всяка стойност от масива
    for (int i = 1; i < size; i++) {
        int key = arr[i];

        int j = i - 1;
        // Докато изследвания елемент е по-малък от предходните,
        while (j >= 0 && arr[j] > key) {
            // разменяме надолу (Bubble-down)
            swap(arr[j + 1], arr[j]);
            j--;
        }
    }
}
```

Времева сложност: $\Theta(n \log n), O(n^2)$

Quick Sort

Quick Sort е един от най-ефективните алгоритми за сортиране, който използва стратегия "разделяй и владей". Работи, като избира "пивот" (pivot) елемент и разделя масива на две подмасиви - елементи по-малки от пивота и елементи по-големи от него. Алгоритъмът рекурсивно прилага същата логика на подмасивите, докато не се получи сортиран масив. Не изисква допълнителна памет за нови масиви, тъй като сортира на място. Не запазва относителното положение на елементите с еднаква стойност. За практическо приложение, той е предпочитан за сортиране на големи набори от данни.

Реализация в C++ :

```cpp
// Функция за разделение на подмасиви
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Избиране на пивот

    int i = low - 1; // Индекс на по-малкия елемент
    for (int j = low; j < high; j++) {
        // Ако текущият елемент е по-малък или равен на пивота
        if (arr[j] < pivot) {
            i++; // Увеличава индекса на по-малкия елемент
            swap(arr[i], arr[j]);
        }
    }

    // Поставяне на пивота на мястото му
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

// Функция за Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Индекс, на който пивотът е поставен
        int pivot = partition(arr, low, high);

        // Рекурсивно сортиране на елементите преди и след пивота
        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
    }
}
```

Времева сложност: $\Theta(n \log n), O(n^2)$

Merge Sort

Merge Sort е друг ефективен алгоритъм за сортиране, който използва стратегия "разделяй и владей". Работи, като разделя масива на две равни подмасиви, рекурсивно сортира подмасивите и след това слива (merge) сортираните подмасиви в един сортиран масив. Алгоритъмът рекурсивно прилага същата логика на подмасивите, докато не се получи сортиран масив. Използва допълнителна памет за сливане на сортираните подмасиви. Merge sort е стабилен, тъй като запазва относителното положение на елементите с еднаква стойност. Той е един от най-ефективните алгоритми за сортиране, който е стабилен и ефективен за големи масиви, бърз и предсказуем по отношение на своята сложност.

Реализация в C++ :

```cpp
// Функция за сливане на два сортирани масива
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1; // Брой елементи вляво
    int n2 = right - mid; // Брой елементи вдясно

    int *lArr = new int[n1]; // Временен ляв масив
    int *rArr = new int[n2]; // Временен ляв масив

    // Копиране на елементите от ляво и дясно в заделените масиви
    for (int i = 0; i < n1; i++) lArr[i] = arr[left + i];
    for (int j = 0; j < n2; j++) rArr[j] = arr[mid + 1 + j];

    // Сливане на сортираните подмасиви
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (lArr[i] <= rArr[j]) {
            arr[k] = lArr[i++];
        } else {
            arr[k] = rArr[j++];
        }
        k++;
    }

    // Копиране на останалите елементи
    while (i < n1) arr[k++] = lArr[i++];
    while (j < n2) arr[k++] = rArr[j++];

    // Освобождаване на временната памет
    delete[] lArr;
    delete[] rArr;
}

// Функция за Merge sort
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid); // Сортира лявата част
        mergeSort(arr, mid + 1, right); // Сортира дясната част
        merge(arr, left, mid, right); // Слива сортираните части
    }
}
```

Времева сложност: $O(n \log n)$

Реализация на софтуерно приложение "Калкулатор за матрици"