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

3.4 Изключения (Exceptions) и работа с тях

Глава 4

Съставни типове данни

4.1 Масиви

4.2 Структури

4.3 Класове

4.4 Обединения

4.5 Изброяване

4.6 Колекции

Глава 5

Манипулиране на паметта

5.1 Указатели

5.2 Референции

5.3 Адресна аритметика

5.4 Динамично и статично разпределяне на паметта

Глава 6

Алгоритми

Реализация на софтуерно приложение "..."