Наръчник за програмиране

с C++

автор: Ангел Дойчев

фн: 2401332001

използван LLM: Chat GPT

Contents

[Въведение 6](#_Toc177915618)

[1. Типове данни в C++ 9](#_Toc177915619)

[1.1. Примитивни типове данни в C++ 9](#_Toc177915620)

[1.1.1. Целочислени типове данни 9](#_Toc177915621)

[1.1.3. Символен тип (char) 11](#_Toc177915622)

[1.1.4. Логически тип (bool) 12](#_Toc177915623)

[1.2. Композитни типове данни в C++ 13](#_Toc177915624)

[1.2.1. Масиви и указатели 13](#_Toc177915625)

[1.2.2. Структури и обединения 15](#_Toc177915626)

[1.2.3. Използване на enum за изброени стойности 17](#_Toc177915627)

[1.3. Константи и модификатори в C++ 19](#_Toc177915628)

[1.3.1. Константи 19](#_Toc177915629)

[1.3.2. Модификатори 20](#_Toc177915630)

[1.4. Примерни програми 22](#_Toc177915631)

[1.4.1. Създаване на променливи от различни типове данни 22](#_Toc177915632)

[1.4.2. Работа с различни модификатори и константи 24](#_Toc177915633)

[1.4.3. Допълнителен пример за constexpr и изчисления по време на компилация 25](#_Toc177915634)

[2. Оператори и изрази 25](#_Toc177915635)

[2.1. Аритметични оператори в C++ 25](#_Toc177915636)

[2.1.1. Приоритет на операторите и скоби 27](#_Toc177915637)

[2.2. Логически оператори и оператори за сравнение в C++ 27](#_Toc177915638)

[2.2.1. Логически оператори 28](#_Toc177915639)

[2.2.2. Оператори за сравнение 29](#_Toc177915640)

[2.2.3. Комбиниране на оператори за сравнение с логически оператори 30](#_Toc177915641)

[3. Управление на потока на изпълнение 30](#_Toc177915642)

[3.3. Оператор switch 30](#_Toc177915643)

[3.1. Условни оператори в C++ 31](#_Toc177915644)

[3.1.1. Оператор if: основна синтактична структура 31](#_Toc177915645)

[3.1.2. Условни изрази и вложени if конструкции 31](#_Toc177915646)

[3.1.3. Използване на else if и else 32](#_Toc177915647)

[3.2. Цикли в C++ 33](#_Toc177915648)

[3.2.1. Цикъл for: синтаксис и примери 33](#_Toc177915649)

[3.2.2. Цикъл while и do-while 34](#_Toc177915650)

[3.2.3. Оператори за прекъсване на цикли: break и continue 36](#_Toc177915651)

[4. Масиви и низове 36](#_Toc177915652)

[4.1. Масиви 37](#_Toc177915653)

[4.1.1. Деклариране на масиви и достъп до елементи 37](#_Toc177915654)

[4.1.2. Инициализация на масиви 37](#_Toc177915655)

[4.1.3. Обхождане на масиви чрез цикли 38](#_Toc177915656)

[4.2. Многомерни масиви 38](#_Toc177915657)

[4.2.1. Работа с двумерни масиви (матрици) 38](#_Toc177915658)

[4.2.2. Инициализация и обработка на многомерни масиви 39](#_Toc177915659)

[4.3. Низове в C++ 39](#_Toc177915660)

[4.3.1. Символни масиви (C-style низове) 39](#_Toc177915661)

[4.3.2. Работа с клас std::string за манипулация на низове 40](#_Toc177915662)

[4.4. Примерни задачи 41](#_Toc177915663)

[4.4.1. Операции с масиви (търсене, сортиране, сума на елементи) 41](#_Toc177915664)

[4.4.2. Обработка на низове (конкатенация, дължина на низ) 41](#_Toc177915665)

[5. Функции в C++ 42](#_Toc177915666)

[5.1. Основи на функциите 42](#_Toc177915667)

[5.1.1. Деклариране и дефиниране на функции 42](#_Toc177915668)

[5.2. Предаване на параметри 43](#_Toc177915669)

[5.2.1. Предаване по стойност 43](#_Toc177915670)

[5.2.2. Предаване по референция 44](#_Toc177915671)

[5.3. Рекурсивни функции 44](#_Toc177915672)

[5.3.1. Основни примери за рекурсия 45](#_Toc177915673)

[5.3.2. Проблеми с рекурсията 46](#_Toc177915675)

[5.4. Функции с променлив брой параметри 49](#_Toc177915682)

[5.4.1. Пример: Използване на функции с променлив брой аргументи 50](#_Toc177915683)

[6. Алгоритми и структури от данни 50](#_Toc177915684)

[6.1. Основни алгоритми 51](#_Toc177915685)

[6.1.1. Сортиране 51](#_Toc177915686)

[6.1.2. Търсене 53](#_Toc177915687)

[6.2. Структури от данни 55](#_Toc177915688)

[6.2.1. Стекове и опашки 55](#_Toc177915689)

[6.2.2. Линейни свързани списъци 57](#_Toc177915690)

### Въведение

В този раздел ще бъде направено детайлно въведение в света на езика C++, ще се разгледа неговата история, еволюция и значимост в съвременното програмиране. Ще се акцентира върху фундаменталните принципи, които го правят един от най-популярните и мощни езици за програмиране, както и върху разликите му с други езици.

#### **История на C++**

Езикът C++ е създаден през 1979 г. от Бярне Строуструп, който работи в Bell Labs. В началото езикът е наричан „C с класове“, тъй като основната му цел е била да разшири възможностите на езика C с концепции за обектно-ориентирано програмиране (ООП). C++ въвежда концепцията за класове и обекти, което го прави много по-гъвкав при създаването на сложни софтуерни системи. Първата официална версия на C++ се появява през 1985 г. и оттогава езикът продължава да се развива и подобрява с нови функционалности и стандарти.

#### **Важност на C++ в съвременното програмиране**

C++ е един от най-разпространените езици за програмиране, използван в множество различни области, включително разработка на софтуер, игри, операционни системи, вградени системи и много други. Важността на C++ произлиза от неговата мощ и гъвкавост. Той съчетава както ниско ниво на контрол върху хардуера, така и високо ниво на абстракция, което го прави подходящ за разнообразни приложения.

Езикът C++ е популярен в разработването на производителни системи, където ефективността и скоростта са от критично значение. Програмите на C++ се компилират директно до машинен код, което означава, че са изключително бързи и могат да използват хардуерните ресурси по оптимален начин. Това е причината C++ да бъде предпочитан за софтуер с висока производителност, като операционни системи, игрални двигатели, сървърни приложения и финансов софтуер.

#### **Основни принципи на C++**

C++ е многофункционален език, който поддържа няколко различни парадигми на програмиране, включително процедурно, обектно-ориентирано и дори функционално програмиране. Този многостранен подход прави C++ един от най-гъвкавите езици, позволявайки на програмистите да изберат най-подходящия метод за решаване на даден проблем.

##### **Обектно-ориентирано програмиране (ООП)**

Една от най-съществените характеристики на C++ е поддръжката му на обектно-ориентирано програмиране. ООП въвежда концепцията за "обекти" — структури от данни, които комбинират както стойности, така и функции, които манипулират тези стойности. Това позволява програмирането да бъде по-интуитивно и организирано, като насърчава повторното използване на код, капсулирането на логика и по-добрата поддръжка на големи проекти.

Основните концепции на ООП в C++ включват:

* **Класове и обекти**: Класовете дефинират шаблони за създаване на обекти (екземпляри на класове), като същевременно инкапсулират данни и функционалност.
* **Наследяване**: Наследяването позволява на един клас да наследява характеристики от друг, което улеснява повторното използване на код.
* **Полиморфизъм**: Полиморфизмът позволява на обектите от различни класове да се използват взаимозаменяемо, ако споделят общ интерфейс.
* **Капсулация**: Капсулацията ограничава достъпа до данните на обекта, като ги прави достъпни само чрез предварително дефинирани методи.

##### **Управление на паметта**

Една от отличителните черти на C++ е прякото управление на паметта. В сравнение с по-високо ниво езици като Python или Java, които използват автоматично управление на паметта (чрез „garbage collector“), C++ дава на програмистите пълен контрол върху динамичното заделяне и освобождаване на памет. Това позволява по-добра оптимизация на ресурси, но изисква по-внимателно програмиране, за да се избегнат грешки като "memory leaks" (изтичане на памет) или неправилно освобождаване на памет.

##### **Генеративно програмиране**

C++ поддържа генеративно програмиране чрез шаблони (templates). Това позволява създаването на функции и класове, които работят с произволни типове данни. Шаблоните значително увеличават гъвкавостта на кода и позволяват създаване на обобщени (generic) структури, които могат да бъдат повторно използвани за различни типове данни.

##### **Мултипарадигмен език**

C++ е **мултипарадигмен** език, което означава, че може да се използва за различни видове програмиране. Наред с обектно-ориентираното програмиране, езикът поддържа и процедурен стил (базиран на функции), както и елементи на функционално програмиране. Това прави C++ изключително гъвкав и подходящ за решаване на различни типове задачи.

#### **Сравнение с други езици за програмиране**

Въпреки че C++ споделя много общи черти с други езици, като C и Java, той има някои специфични предимства и недостатъци:

* **C**: C++ е надстройка над C и включва всички функционалности на C, но същевременно добавя обектно-ориентирани концепции и други удобства. Докато C е по-подходящ за ниско ниво програмиране и вградени системи, C++ е предпочитан за по-комплексни проекти, където е необходима по-добра абстракция.
* **Java**: Java е подобен на C++ по много начини, но е насочен към по-високо ниво програмиране и има автоматично управление на паметта, което улеснява програмистите, но ограничава контрола върху изпълнението. C++ предлага повече контрол и възможности за оптимизация, но изисква повече усилия за управление на ресурсите.
* **Python**: Python е език с динамично типизиране и е популярен заради своята простота и удобство за бързо прототипиране. Въпреки това, Python не може да се сравнява с C++ по отношение на производителност, особено за задачи, където ефективността на изпълнение е критична.

### 1. Типове данни в C++

### ****1.1. Примитивни типове данни в C++****

Примитивните типове данни са основните типове, които се използват за представяне на базови стойности в C++ програмирането. Те включват цели числа, реални числа, символи и логически стойности. Всеки тип заема различен размер в паметта и се използва в зависимост от нуждите на програмата.

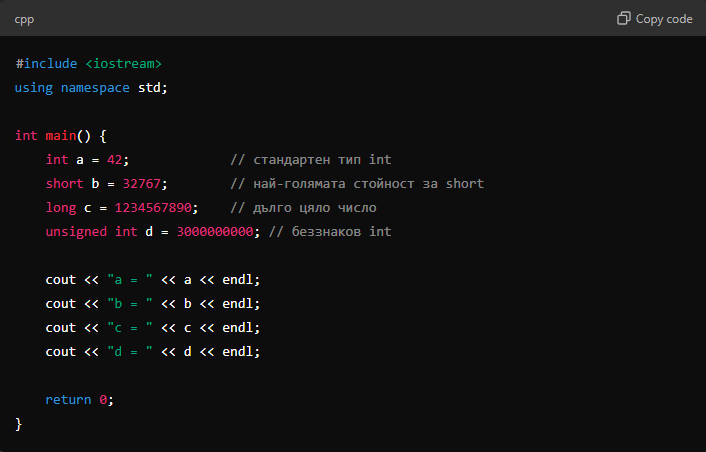
### **1.1.1. Целочислени типове данни**

Целочислените типове данни в C++ се използват за представяне на цели числа, като могат да бъдат **знакови** или **беззнакови**, т.е. те могат да приемат както положителни, така и отрицателни стойности или само положителни.

| **Тип данни** | **Размер (в байтове)** | **Диапазон на стойности** |
| --- | --- | --- |
| int | 4 | От -2,147,483,648 до 2,147,483,647 |
| short | 2 | От -32,768 до 32,767 |
| long | 4 или 8 | Зависи от платформата |
| long long | 8 | От -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807 |
| unsigned | 4 | От 0 до 4,294,967,295 |

* **int**: Основният тип за цели числа. Често използван, когато не се изисква специален размер.
* **short**: Използва се за числа с по-малък диапазон от стойности.
* **long** и **long long**: Поддържат по-големи цели числа.
* **unsigned**: Беззнаков тип, който съхранява само положителни стойности.

#### Пример за работа с целочислени типове:



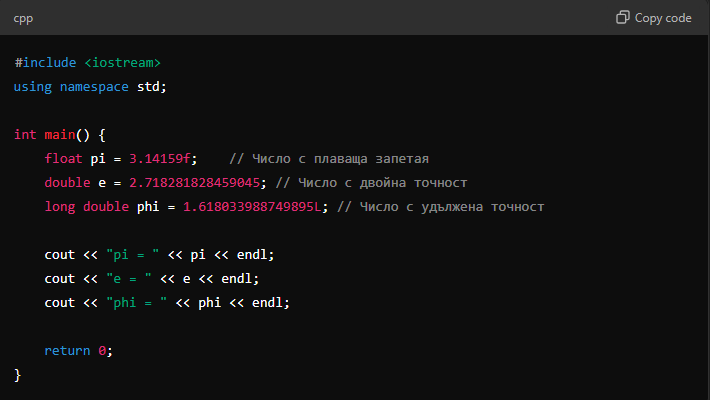
#### **1.1.2. Реални числа (с плаваща запетая)**

Реалните числа в C++ се използват за съхраняване на дробни стойности (например 3.14, 2.718). Те включват типове като float, double и long double, които се различават по точност и размер в паметта.

| **Тип данни** | **Размер (в байтове)** | **Диапазон на стойности** | **Прецизност** |
| --- | --- | --- | --- |
| float | 4 | ~3.4E-38 до 3.4E+38 | 6-7 цифри |
| double | 8 | ~1.7E-308 до 1.7E+308 | 15-16 цифри |
| long double | 10 или 16 | Зависи от платформата | Повече от 16 цифри |

* **float**: Поддържа реални числа с единична точност (по-малка точност, но по-малък размер в паметта).
* **double**: Поддържа числа с двойна точност, като често се използва поради по-високата точност.
* **long double**: Използва се, когато се изисква по-голяма прецизност от double.

#### Пример за работа с реални числа:



Забележка: Суфиксът f указва, че стойността е от тип float, докато суфиксът L указва, че стойността е long double.

### **1.1.3. Символен тип (char)**

Типът **char** се използва за представяне на единични символи. Символите се представят като цели числа (т.е. ASCII кодове) и заемат 1 байт памет. Диапазонът на стойности на char е от -128 до 127 за знакови типове или от 0 до 255 за беззнакови.

| **Тип данни** | **Размер (в байтове)** | **Диапазон на стойности** |
| --- | --- | --- |
| char | 1 | От -128 до 127 (знаков) |
| unsigned char | 1 | От 0 до 255 (беззнаков) |

#### Пример за работа със символен тип:



Тук символът 'A' има ASCII код 65, а '1' има ASCII код 49. Използването на **беззнакови символи** (например unsigned char) разширява диапазона от положителни стойности.

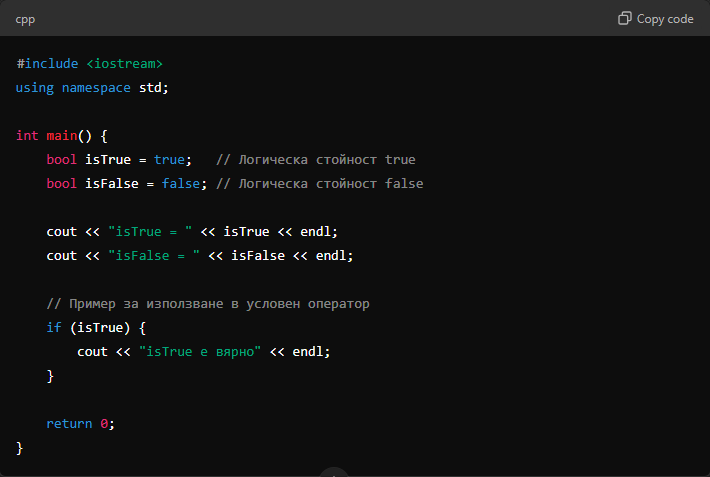
### ****1.1.4. Логически тип (bool)****

Типът **bool** в C++ представлява булева стойност, която може да бъде **true** (истина) или **false** (лъжа). Този тип се използва за логически операции и вземане на решения.

| **Тип данни** | **Размер (в байтове)** | **Стойности** |
| --- | --- | --- |
| bool | 1 | true, false |

Булевите стойности често се използват в условни оператори като if, цикли, и за проверка на резултати от логически операции.

#### Пример за работа с логически тип:



В този пример булевите стойности се използват в условен оператор if, като true се оценява като истина, а false – като лъжа.

### ****1.2. Композитни типове данни в C++****

Композитните типове данни в C++ позволяват съхранението на множество стойности от един и същи или различни типове. Те включват **масиви**, **указатели**, **структури**, **обединения** и **изброени стойности (enum)**.

### ****1.2.1. Масиви и указатели****

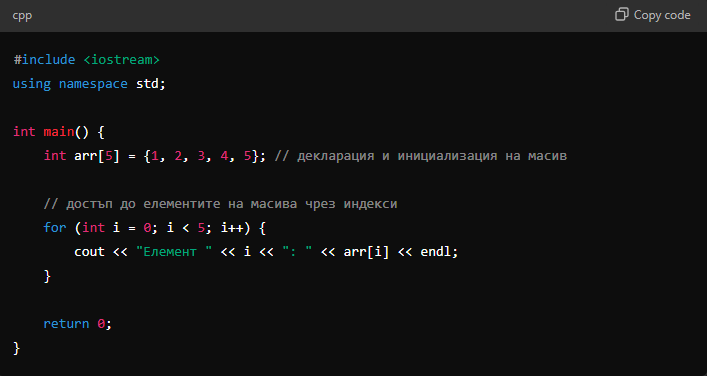
#### **Масиви**

Масивите са колекция от елементи със същия тип, които се съхраняват в последователни позиции в паметта. Индексите в масивите започват от нула, което означава, че първият елемент на масива има индекс 0.

**Синтаксис за декларация на масив:**

****

#### Пример за работа с масиви:



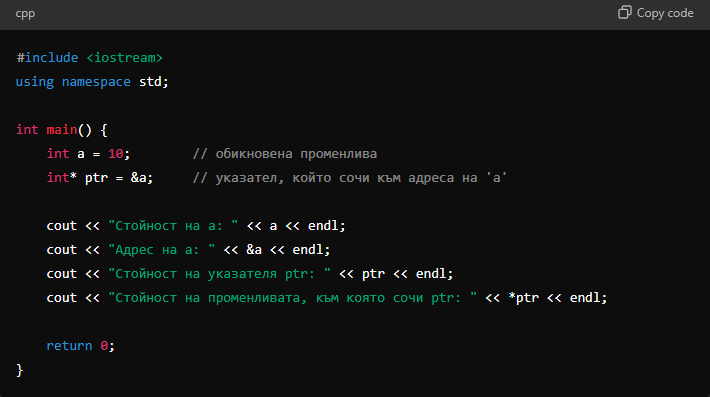
Тук имаме масив от пет цели числа, към който достъпваме чрез индекси в цикъл.

#### **Указатели**

Указателите са променливи, които съхраняват адреса на друга променлива. Указателите се използват за динамично управление на паметта, за предаване на променливи по референция в функции и други операции с памет.

**Синтаксис за декларация на указател:**

Пример за работа с указатели:

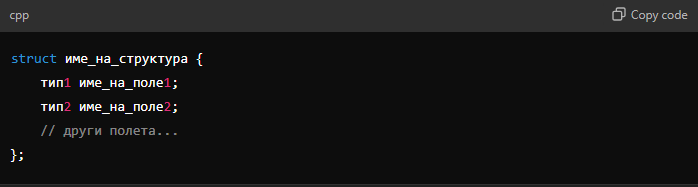


Тук указателят ptr съхранява адреса на променливата a, а чрез \*ptr достъпваме стойността, съхранена на този адрес.

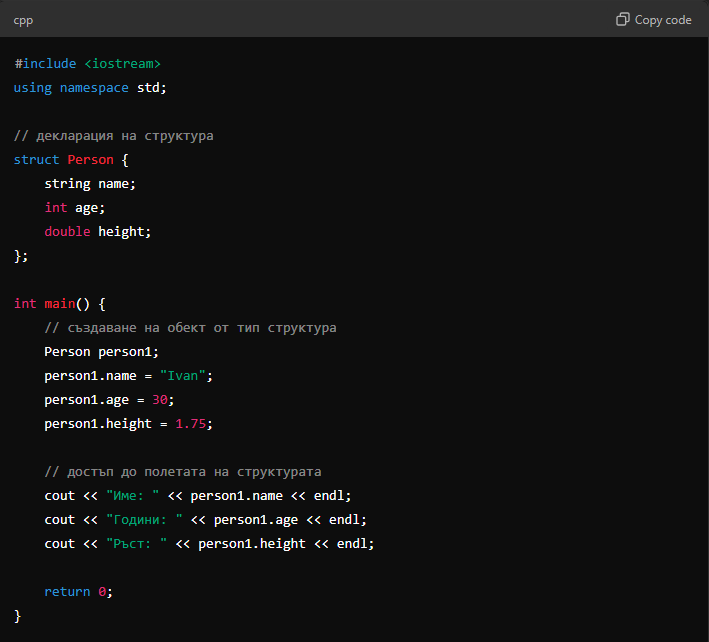
### ****1.2.2. Структури и обединения****

#### **Структури**

Структурата е композитен тип данни, който позволява съхранението на множество променливи с различен тип в една единствена единица. Всеки елемент на структурата се нарича **член** или **поле**.

**Синтаксис за декларация на структура:**

#### Пример за структура:



Тук структурата Person съхранява данни за човек, като име, възраст и ръст. Можем да създаваме обекти от този тип и да достъпваме техните членове.

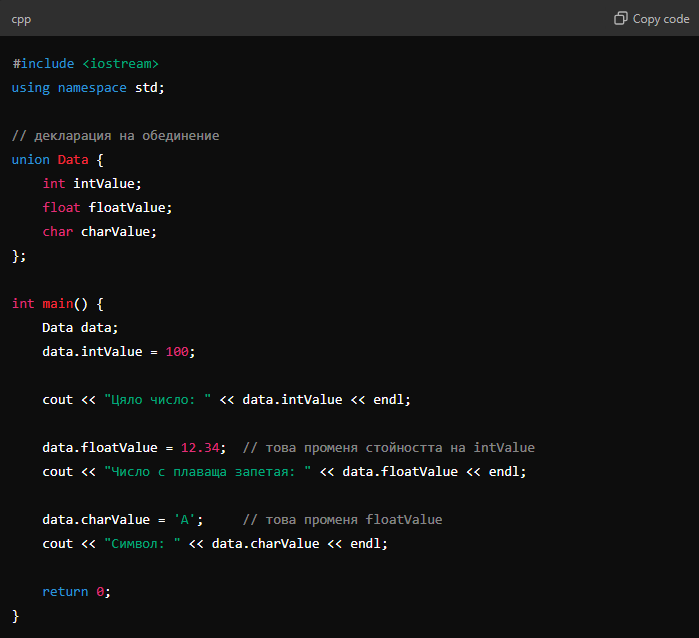
#### **Обединения**

Обединението е подобно на структурата, но при обединението всички членове споделят едно и също място в паметта, така че във всеки един момент може да бъде достъпен само един член.

**Синтаксис за декларация на обединение:**

#### 

#### Пример за обединение:

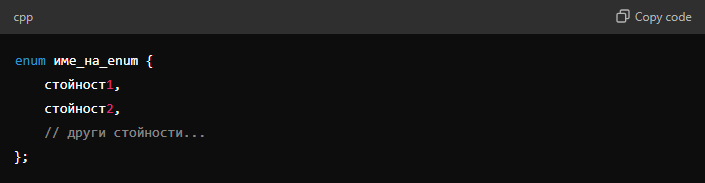


Тъй като всички членове на обединението споделят едно и също място в паметта, присвояването на стойност към един член променя другите.

### ****1.2.3. Използване на**** enum ****за изброени стойности****

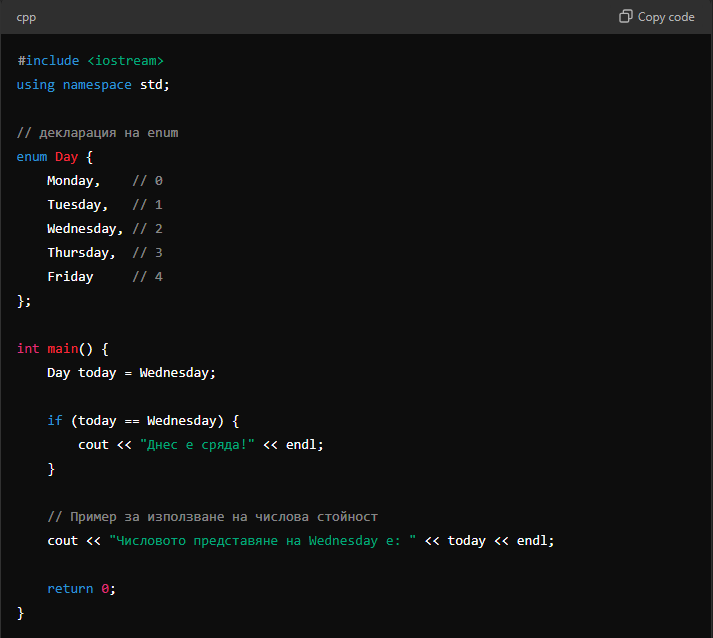
enum е тип данни, който позволява дефинирането на множество константи с уникални имена, свързани с целочислени стойности. Това е полезно, когато имаме набор от предварително дефинирани стойности.

**Синтаксис за декларация на enum:**



По подразбиране, първата стойност в изброението е 0, втората е 1 и т.н., но можем да присвояваме собствени стойности на елементите.

#### Пример за enum:



В този пример дефинираме изброен тип Day, който съдържа дните от седмицата. Променливата today приема стойността Wednesday, която по подразбиране има числово представяне 2.

### ****1.3. Константи и модификатори в C++****

Константите и модификаторите в C++ са важни концепции, които помагат за по-добро управление на данните и ефективност при програмирането. **Константите** осигуряват неизменност на стойностите, докато **модификаторите** определят начина, по който се интерпретират и съхраняват тези стойности в паметта.

### ****1.3.1. Константи****

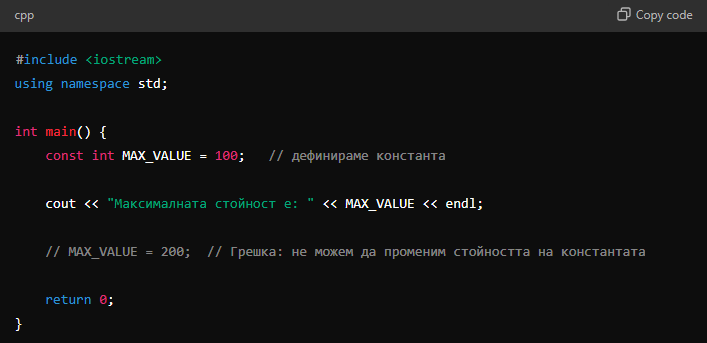
#### const **(константни променливи)**

Ключовата дума **const** указва, че една променлива не може да бъде променяна след като й бъде присвоена стойност. Тя често се използва за защита на данни, които трябва да останат неизменни през целия цикъл на програмата.

**Синтаксис:**

#### 

#### Пример за const:



В този пример константата MAX\_VALUE има стойност 100 и не може да бъде променена след инициализацията.

#### constexpr **(константи на ниво компилация)**

Ключовата дума **constexpr** указва, че дадена стойност може да бъде изчислена по време на компилация, вместо по време на изпълнение. Това може да доведе до оптимизация на програмата, когато стойността е известна предварително.

**Синтаксис:**

#### 

#### Пример за constexpr:



Тук функцията getSquare изчислява квадрата на числото по време на компилация, тъй като както аргументът, така и резултатът са означени с constexpr.

### ****1.3.2. Модификатори****

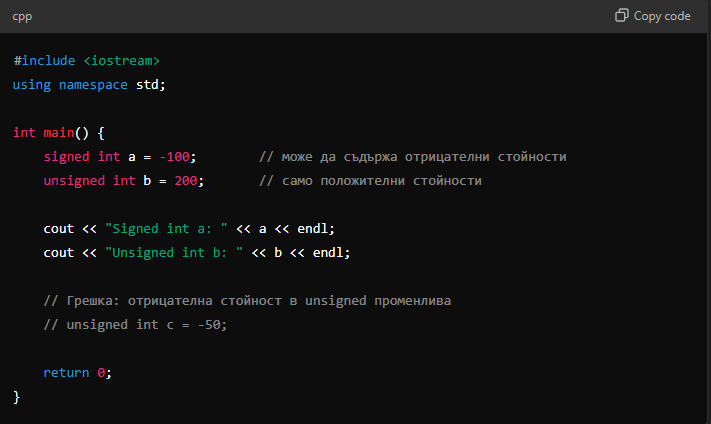
Модификаторите се използват за промяна на свойствата на основните типове данни, като позволяват управлението на размера на променливата и нейния диапазон от стойности. Най-често срещаните модификатори са **signed** и **unsigned**.

#### signed **и** unsigned

Модификаторите **signed** и **unsigned** указват дали дадена променлива може да съхранява както положителни, така и отрицателни стойности (**signed**), или само положителни стойности (**unsigned**). Тези модификатори обикновено се използват с целочислени типове като int и char.

* **signed**: Може да съдържа както положителни, така и отрицателни стойности.
* **unsigned**: Може да съдържа само положителни стойности, но с удвоен максимален диапазон спрямо знаковите числа.

#### Пример за signed и unsigned:



В този пример signed int може да приема отрицателни стойности, докато unsigned int може да съхранява само положителни числа, но в по-голям диапазон.

#### signed **и** unsigned **с** char

Типът **char** по подразбиране е знаков или беззнаков, в зависимост от компилатора. Модификаторите **signed** и **unsigned** могат да бъдат приложени и към char, за да определят дали символът трябва да се интерпретира като число със или без знак.

* **signed char**: Стойностите варират от -128 до 127.
* **unsigned char**: Стойностите варират от 0 до 255.

#### Пример за signed и unsigned с char:



Тъй като типът char обикновено се интерпретира като символ, използваме static\_cast<int> за да изведем числовото представяне на знаковия и беззнаковия символ.

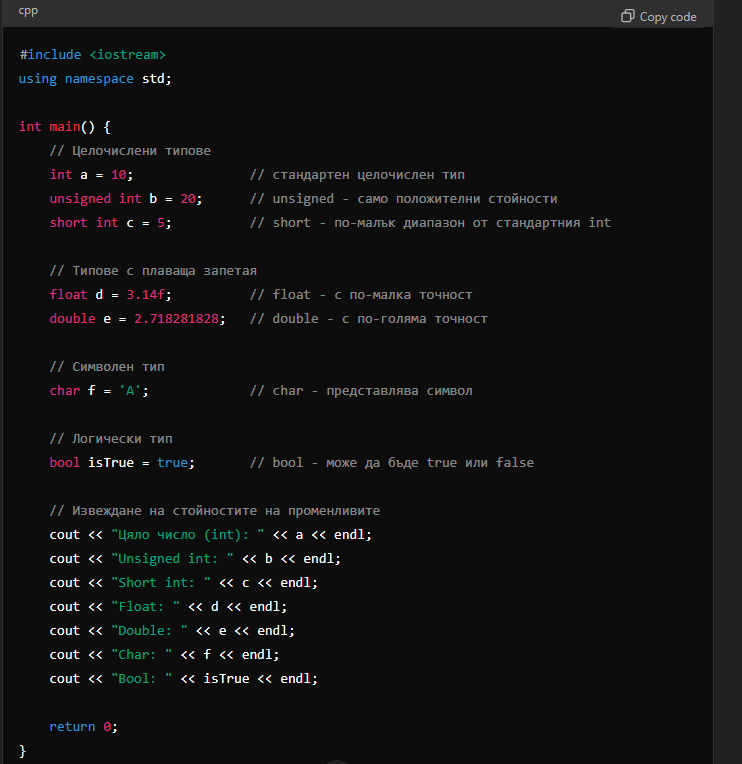
### ****1.4. Примерни програми****

В тази подточка ще разгледаме практически примери за създаване и използване на променливи от различни типове данни в C++, както и за работа с модификатори и константи. Тези примери ще демонстрират как да създавате променливи, да използвате константи и да прилагате модификатори за контрол върху стойностите и типа данни.

### ****1.4.1. Създаване на променливи от различни типове данни****

В C++ можем да създаваме променливи от различни примитивни и композитни типове данни, като използваме различните ключови думи, които сме обсъждали досега. Ето някои примери за създаване и работа с различни типове променливи.

#### Примерна програма:

****

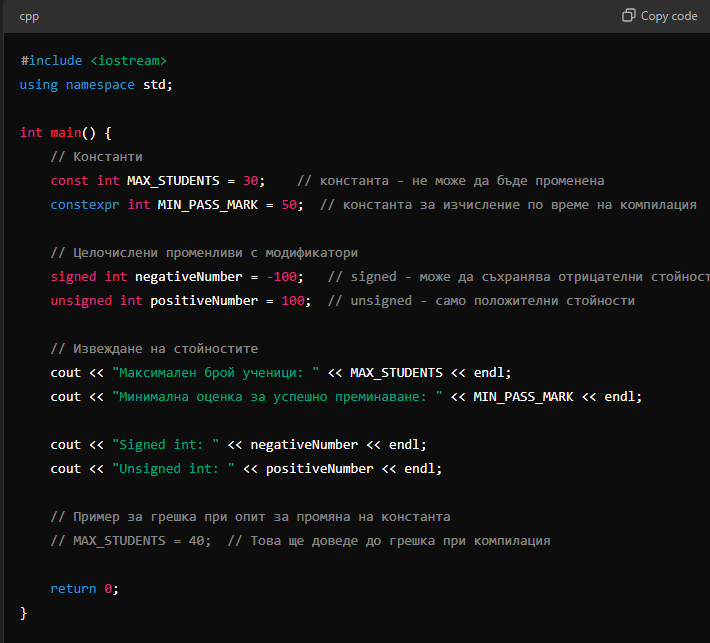
**Обяснение на кода:**

* Променливата a е стандартен целочислен тип int.
* Променливата b е от тип unsigned int, което означава, че може да приема само положителни стойности.
* Променливата c е от тип short int, който заема по-малко място в паметта и има по-малък диапазон от стандартния int.
* Променливите d и e са съответно от тип float и double, които се използват за числа с плаваща запетая с различна точност.
* Променливата f е от тип char, който съхранява символи.
* Променливата isTrue е от логически тип bool, който може да съхранява стойности true или false.

### ****1.4.2. Работа с различни модификатори и константи****

В този пример ще създадем програма, която използва **константи**, модификатори като **signed** и **unsigned**, както и **constexpr** за работа с изчисления по време на компилация.

#### Примерна програма:

****

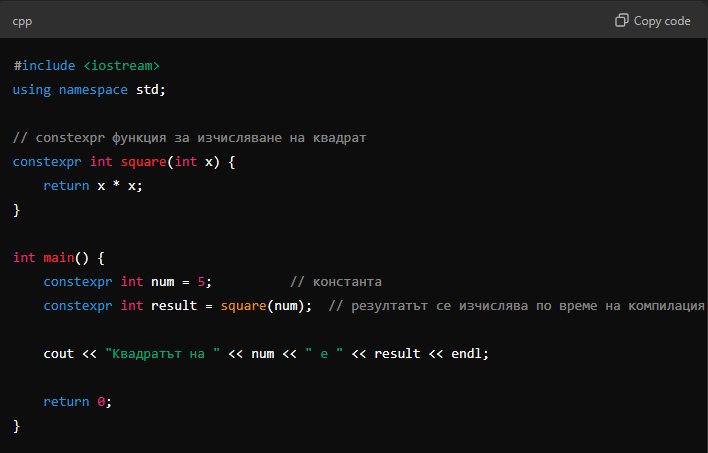
**Обяснение на кода:**

* Променливата MAX\_STUDENTS е декларирана като **const**, което означава, че не може да бъде променяна след инициализацията.
* Променливата MIN\_PASS\_MARK е декларирана като **constexpr**, което позволява нейното използване в изчисления по време на компилация.
* negativeNumber е знакова променлива (**signed**), която може да приема както положителни, така и отрицателни стойности.
* positiveNumber е беззнакова променлива (**unsigned**), която може да приема само положителни стойности.
* Опит за промяна на константата MAX\_STUDENTS би довел до грешка при компилация.

### ****1.4.3. Допълнителен пример за**** constexpr ****и изчисления по време на компилация****

Ключовата дума **constexpr** позволява да се дефинират константи и функции, които могат да бъдат изчислени по време на компилация. Това може да доведе до оптимизация на програмата.

#### Пример с constexpr функция:

****

**Обяснение на кода:**

* Функцията square е декларирана като **constexpr**, което означава, че резултатът може да бъде изчислен по време на компилация.
* Променливата result ще бъде изчислена по време на компилация, което прави програмата по-ефективна.

### 2. Оператори и изрази

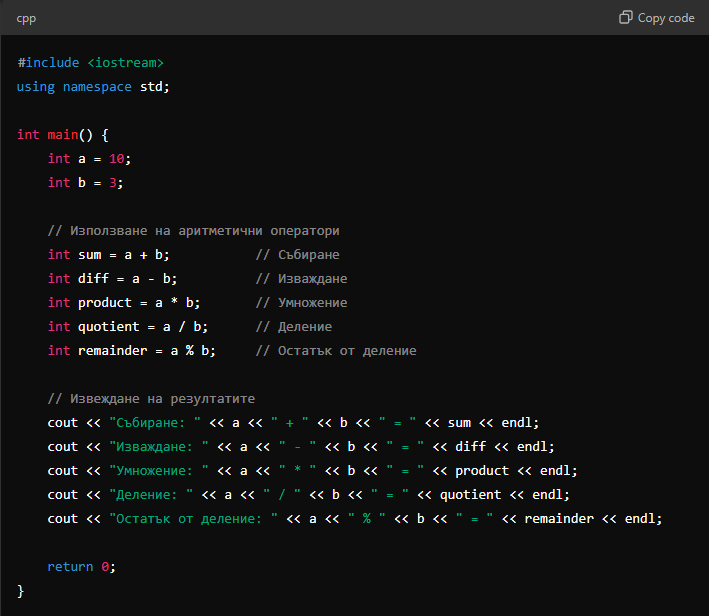
### ****2.1. Аритметични оператори в C++****

Аритметичните оператори в C++ се използват за извършване на основни математически операции като събиране, изваждане, умножение, деление и намиране на остатък. Тези оператори са част от стандартния език и могат да се използват с цели числа и числа с плаваща запетая.

#### **Основни аритметични оператори:**

1. **Събиране (+)** – използва се за събиране на две числа.
2. **Изваждане (-)** – използва се за изваждане на едно число от друго.
3. **Умножение (\*)** – използва се за умножение на две числа.
4. **Деление (/)** – използва се за деление на две числа.
5. **Остатък (%)** – използва се за намиране на остатъка от целочислено деление. Работи само с цели числа.

#### **Пример с основните аритметични оператори:**

****

**Обяснение на кода:**

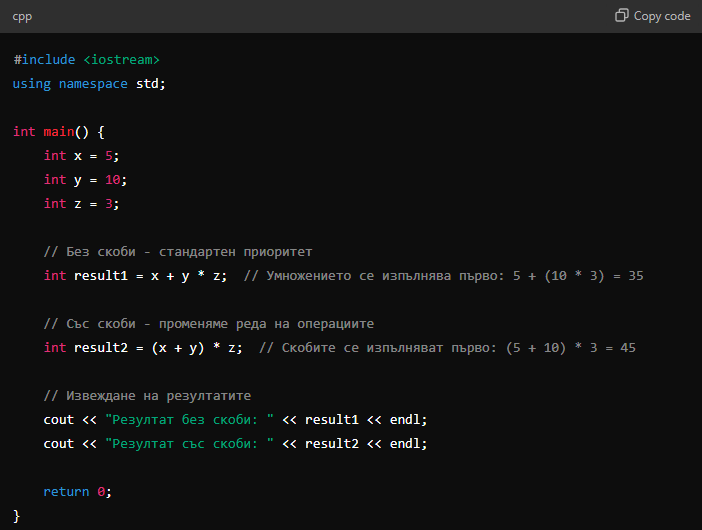
* Създадохме две цели числа, a и b, и използвахме основните аритметични оператори за извършване на различни операции.
* Операторът % намира остатъка от делението на a на b. Този оператор се използва само с цели числа.
* Резултатите от операциите се извеждат в конзолата.

### ****2.1.1. Приоритет на операторите и скоби****

Приоритетът на операторите определя реда, в който те се изпълняват в изразите. В C++ аритметичните оператори имат различен приоритет:

* **Умножение (\*)**, **деление (/)** и **остатък (%)** имат по-висок приоритет от **събиране (+)** и **изваждане (-)**.
* Скобите () променят реда на изпълнение, като операторите вътре в скобите се изчисляват първи, независимо от приоритета на операторите извън тях.

#### **Пример за приоритет на операторите:**

****

**Обяснение на кода:**

* В първия случай, умножението се изпълнява преди събирането поради по-високия си приоритет, което дава резултат 35.
* Във втория случай, скобите променят реда на операциите, така че първо се събират x и y, а след това се умножават със z, което дава резултат 45.

### ****2.2. Логически оператори и оператори за сравнение в C++****

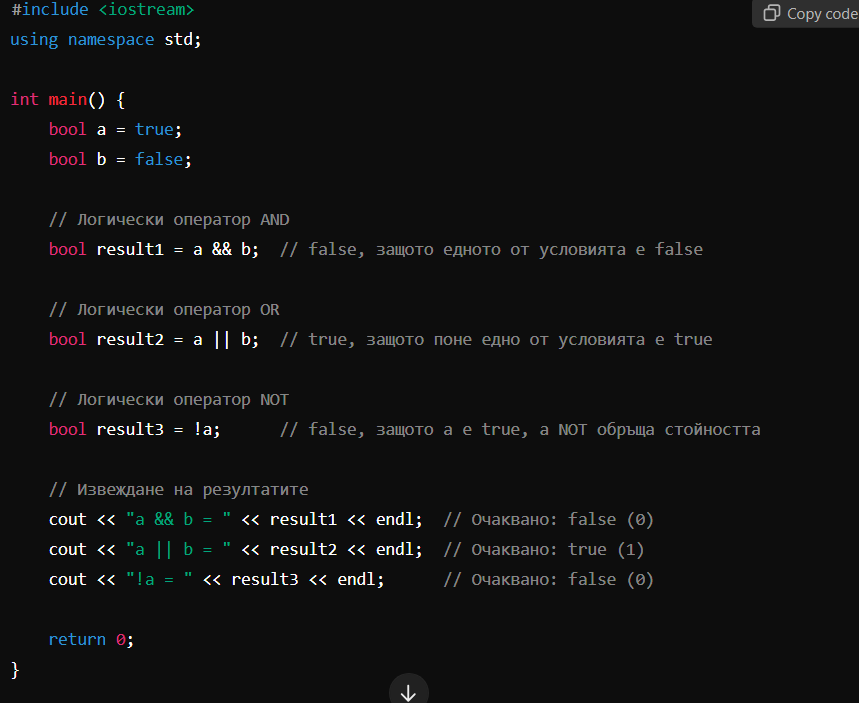
В C++ логическите оператори и операторите за сравнение се използват за създаване на условия и проверка на връзки между стойности. Те са от съществено значение при контролирането на потока на изпълнение в програмите, особено в условни конструкции и цикли.

### ****2.2.1. Логически оператори****

Логическите оператори се използват за комбиниране на логически изрази или за проверка на тяхната истинност:

1. **AND (&&)** – връща **true** само ако и двете условия са истинни.
2. **OR (||)** – връща **true**, ако поне едно от условията е истинно.
3. **NOT (!)** – обръща стойността на логическия израз (ако е **true**, става **false**, и обратно).

#### **Пример с логически оператори:**



**Обяснение на кода:**

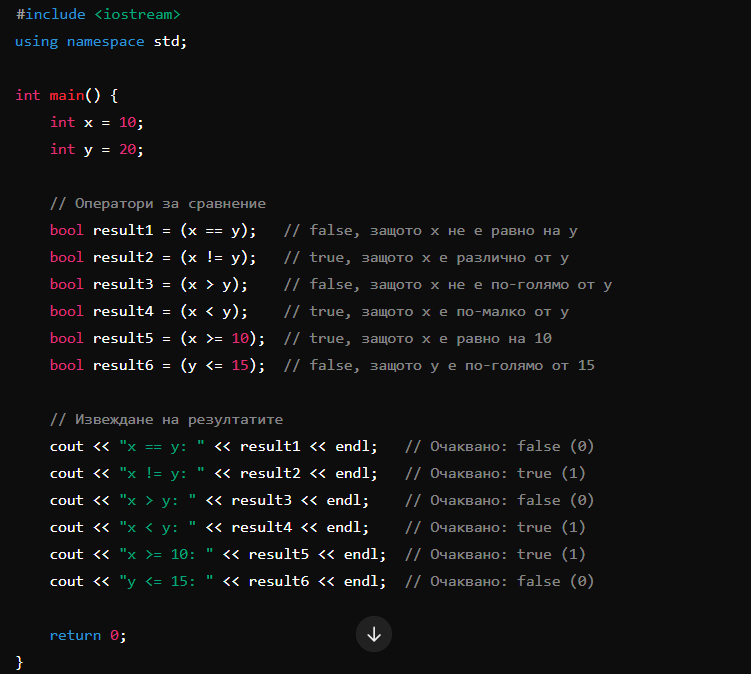
* Логическият оператор && връща **false**, защото само едно от условията е **true**.
* Операторът || връща **true**, защото поне едно от условията е **true**.
* Операторът ! обръща стойността на a, която е **true**, и връща **false**.

### ****2.2.2. Оператори за сравнение****

Операторите за сравнение се използват за сравняване на две стойности и връщат булев резултат (**true** или **false**):

1. **Равно (==)** – проверява дали две стойности са равни.
2. **Неравно (!=)** – проверява дали две стойности не са равни.
3. **По-голямо (>)** – проверява дали едната стойност е по-голяма от другата.
4. **По-малко (<)** – проверява дали едната стойност е по-малка от другата.
5. **По-голямо или равно (>=)** – проверява дали едната стойност е по-голяма или равна на другата.
6. **По-малко или равно (<=)** – проверява дали едната стойност е по-малка или равна на другата.

#### **Пример с оператори за сравнение:**



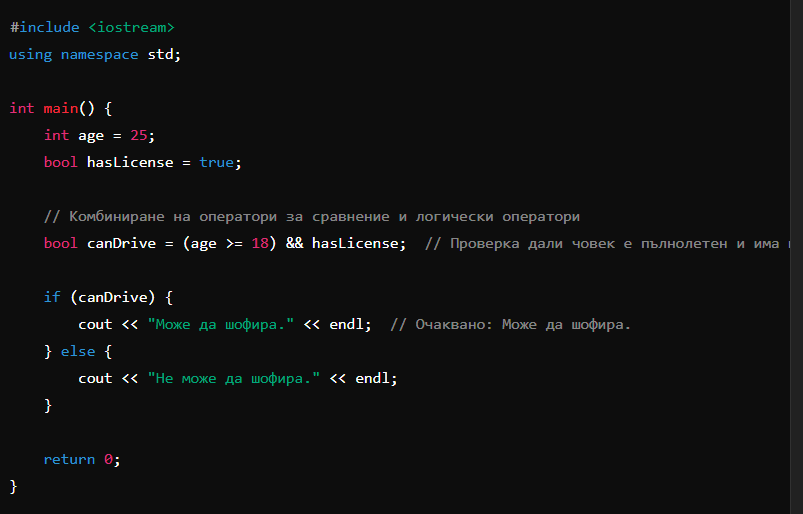
**Обяснение на кода:**

* Сравняваме две цели числа x и y с различни оператори за сравнение.
* Резултатите са булеви стойности (**true** или **false**) и се извеждат в конзолата.
* Например, (x == y) е **false**, защото x не е равно на y, докато (x < y) е **true**, защото x е по-малко от y.

### ****2.2.3. Комбиниране на оператори за сравнение с логически оператори****

Можем да комбинираме операторите за сравнение и логическите оператори за по-сложни условия.

#### **Пример за комбиниране:**



**Обяснение на кода:**

* Проверяваме дали човек може да шофира, като използваме комбинация от оператори за сравнение (>=) и логически оператор (&&).
* Ако възрастта е 18 или повече и има шофьорска книжка, резултатът е **true** и съобщението "Може да шофира." се извежда.

### 3. Управление на потока на изпълнение

### 3.3. Оператор switch

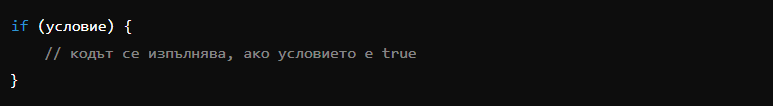
* Условно разклонение с switch и case.
* Използване на default в switch.

### ****3.1. Условни оператори в C++****

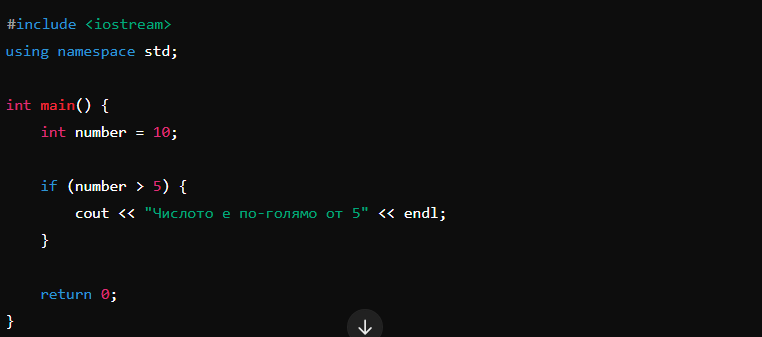
### **3.1.1. Оператор** if**: основна синтактична структура**

Условният оператор if се използва за изпълнение на код само ако дадено условие е изпълнено (**true**). Това е най-основният начин за контрол на потока на програмата въз основа на логически условия.

**Синтаксис:**



#### **Пример с if:**



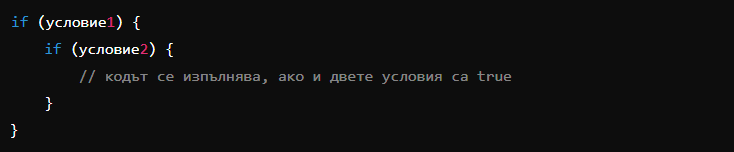
**Обяснение:**

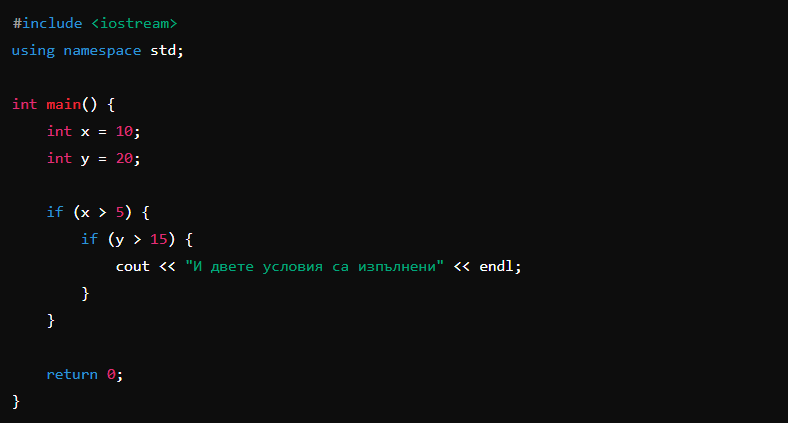
* В този пример условието number > 5 е **true**, затова съобщението ще бъде отпечатано.

### ****3.1.2. Условни изрази и вложени if конструкции****

Вложените if конструкции позволяват да имаме условие вътре в друго условие. Това дава възможност за по-сложни проверки.

**Синтаксис:**

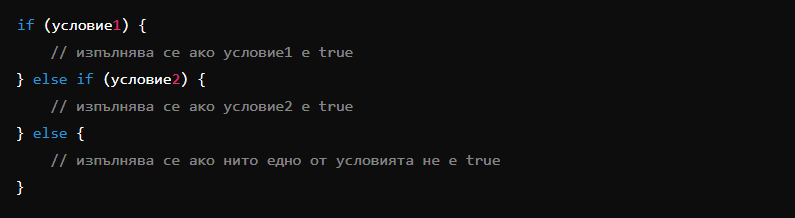
****

**Пример с вложен if:**

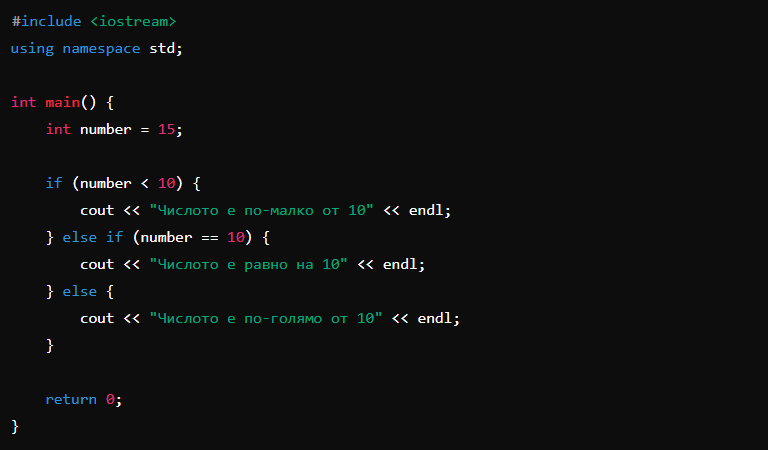
### ****3.1.3. Използване на** else** if ****и**** else

Когато имаме повече от едно условие за проверка, можем да използваме else if за проверка на допълнителни условия и else за действие, ако никое от предишните условия не е изпълнено.

**Синтаксис:**



#### **Пример с if, else if и else:**

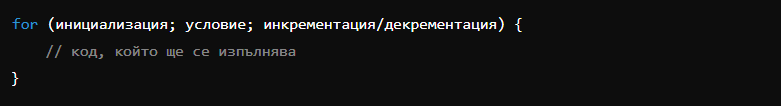


### ****3.2. Цикли в C++****

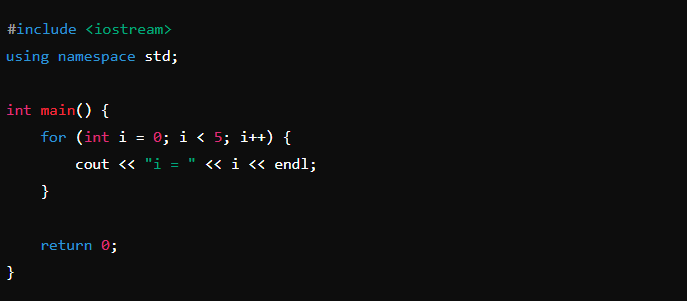
### ****3.2.1. Цикъл**** for****: синтаксис и примери****

Цикълът for се използва за повторение на блок код фиксиран брой пъти. В синтаксиса на цикъла се задават начална стойност, условие за продължаване и стъпка.

**Синтаксис:**



#### **Пример с цикъл for:**



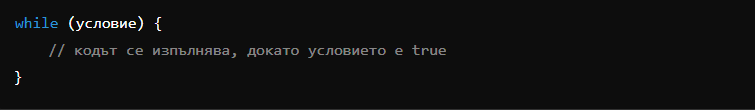
**Обяснение:**

* В този пример цикълът ще изпълни тялото си 5 пъти (докато i < 5).

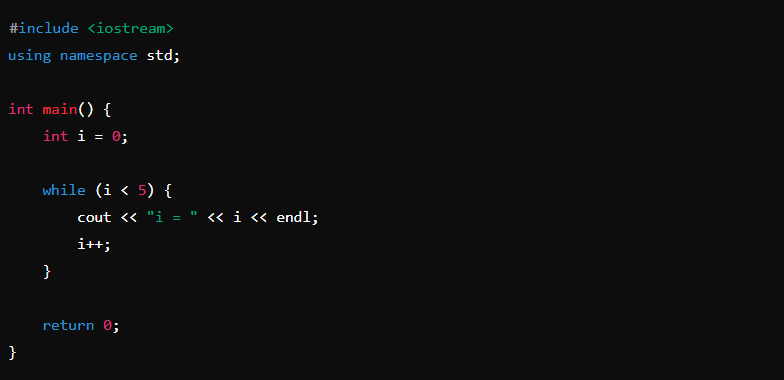
### **3.2.2. Цикъл** while **и** do-while

**Цикъл while** изпълнява тялото си, докато дадено условие е изпълнено (**true**). Ако условието не е вярно в началото, цикълът няма да се изпълни нито веднъж.

**Синтаксис:**

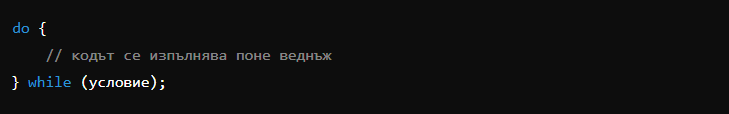


#### **Пример с while:**

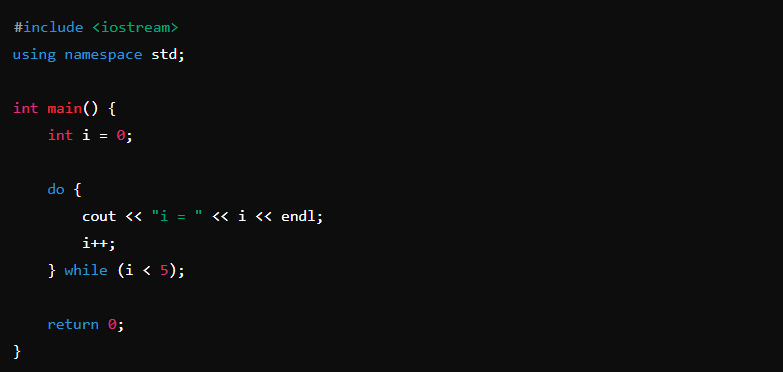


**Цикъл do-while** изпълнява тялото на цикъла поне веднъж, независимо дали условието е изпълнено, и след това проверява условието.

**Синтаксис:**



#### **Пример с do-while:**



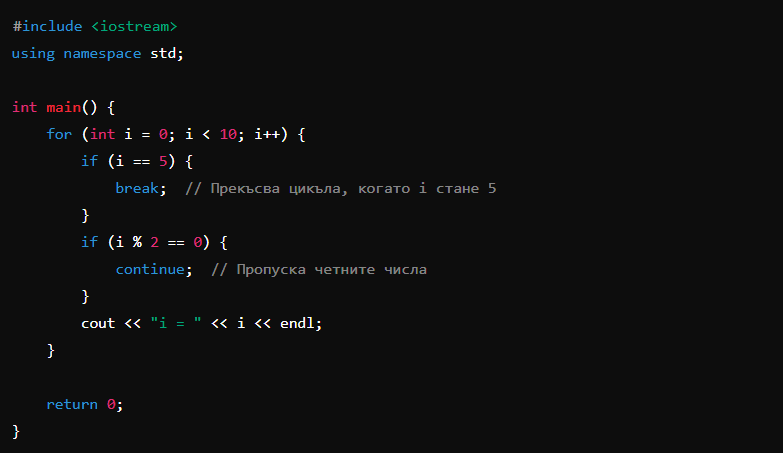
**Обяснение:**

* Тялото на цикъла ще се изпълни поне веднъж, дори ако условието не е изпълнено първоначално.

### ****3.2.3. Оператори за прекъсване на цикли:**** break ****и**** continue

* **Оператор break** – прекъсва изпълнението на цикъла и извежда програмата извън цикъла.
* **Оператор continue** – прекъсва текущата итерация на цикъла и преминава към следващата итерация.

#### **Пример с break и continue:**



**Обяснение:**

* **break** прекъсва цикъла, когато i достигне 5, така че циклите спират.
* **continue** пропуска четните стойности на i и преминава директно към следващата итерация.

### ****4. Масиви и низове****

Масивите и низовете са основни структури от данни в C++, които се използват за съхранение и манипулиране на колекции от елементи. Масивите представляват последователни блокове от памет, които могат да съдържат множество елементи от един и същ тип, докато низовете са специални видове масиви, които съхраняват символи.

### ****4.1. Масиви****

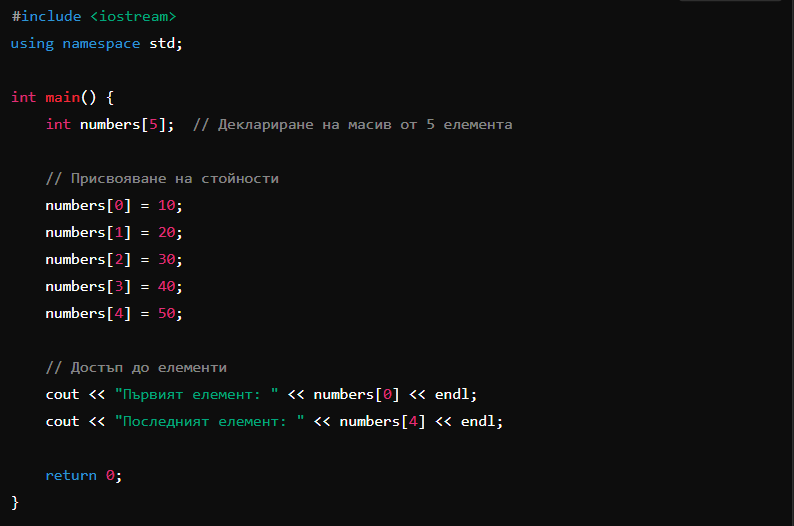
### ****4.1.1. Деклариране на масиви и достъп до елементи****

Масивите се декларират чрез посочване на типа данни, последван от името на масива и размера в квадратни скоби.

**Синтаксис:**

#### 

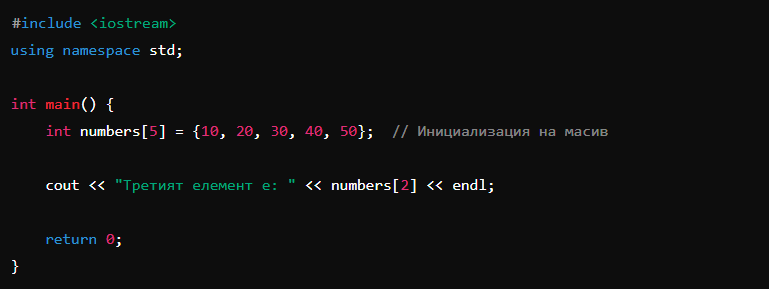
#### **Пример: Деклариране и достъп до елементи**



### ****4.1.2. Инициализация на масиви****

Можем да инициализираме масив директно по време на деклариране.

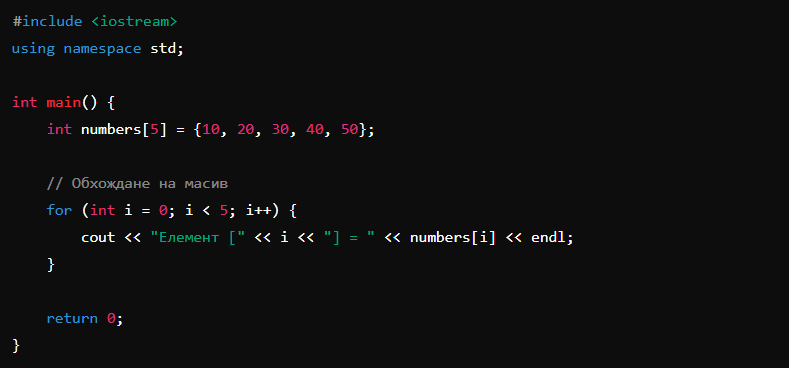
**Пример: Инициализация при деклариране**



### ****4.1.3. Обхождане на масиви чрез цикли****

Масивите могат лесно да бъдат обхождани с помощта на цикли, за да се обработват техните елементи.

**Пример: Обхождане на масив с for цикъл**



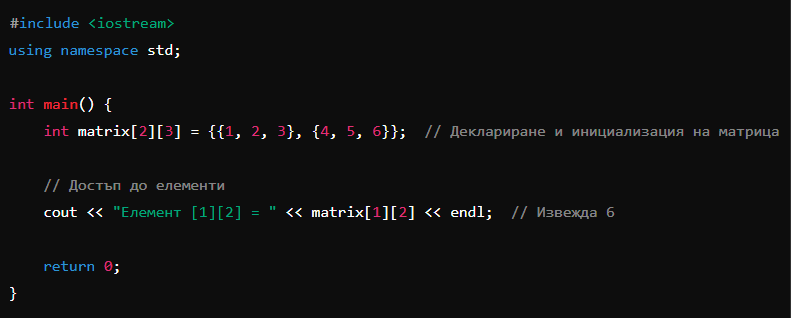
### ****4.2. Многомерни масиви****

### ****4.2.1. Работа с двумерни масиви (матрици)****

Двумерните масиви са колекции от елементи, организирани в редове и колони. Те могат да се представят като матрици.

**Синтаксис:**

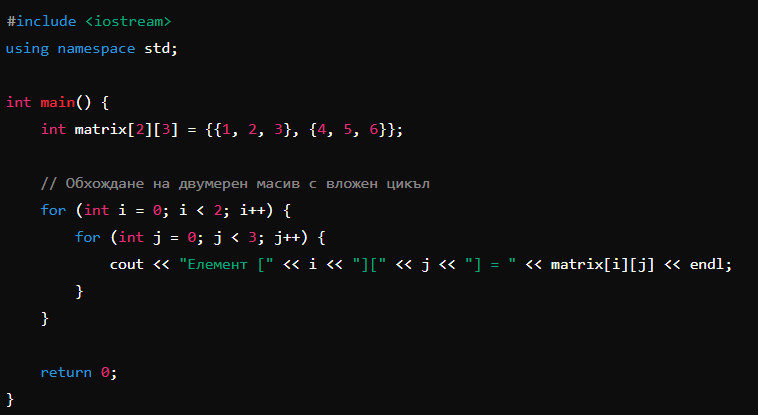
#### **Пример: Деклариране на двумерен масив**



### ****4.2.2. Инициализация и обработка на многомерни масиви****

Можем да обхождаме и обработваме елементите на двумерни масиви с вложени цикли.

**Пример: Обхождане на двумерен масив**

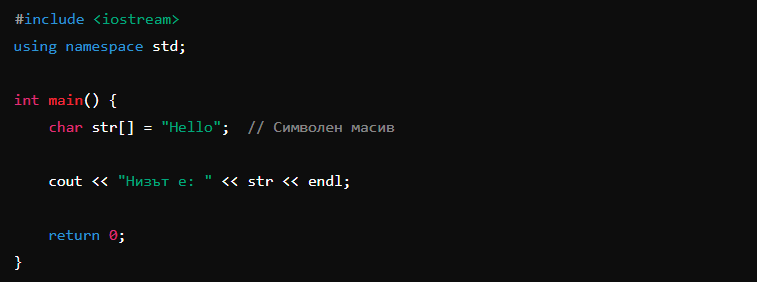


### ****4.3. Низове в C++****

### ****4.3.1. Символни масиви (C-style низове)****

C-style низовете са масиви от символи, завършващи с нулев терминатор (\0), който указва края на низа.

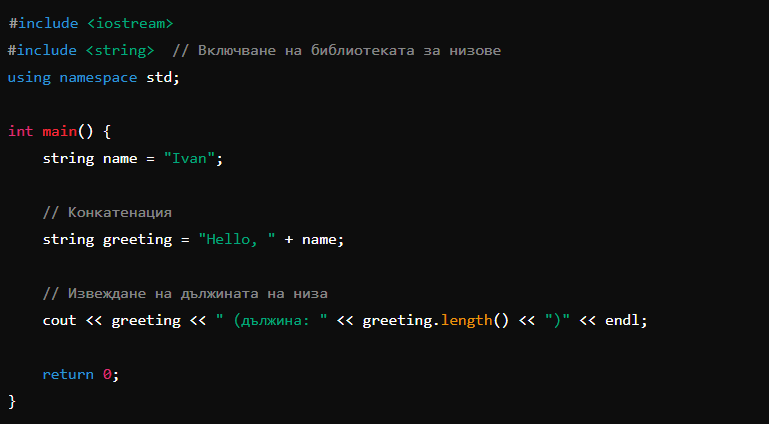
**Пример: Деклариране на C-style низ**



### ****4.3.2. Работа с клас**** std::string ****за манипулация на низове****

Класът std::string е част от стандартната библиотека на C++ и предлага удобни функции за работа с низове.

**Пример: Работа с std::string**

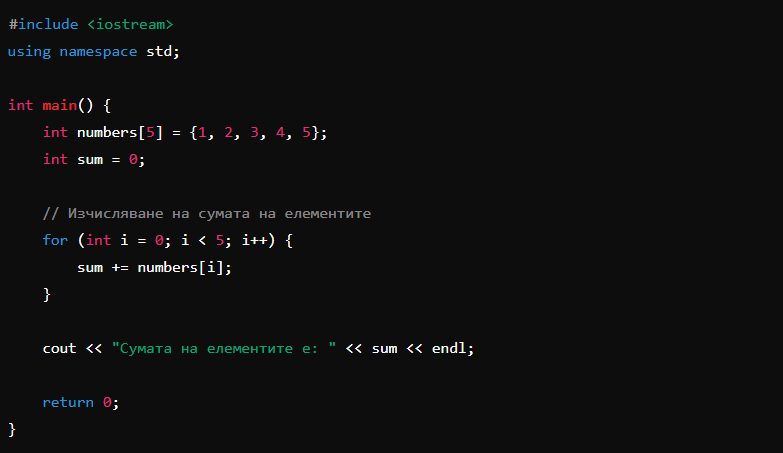
**Обяснение:**

* Класът std::string предоставя методи като .length() за извличане на дължината на низа и оператори за конкатенация.

### ****4.4. Примерни задачи****

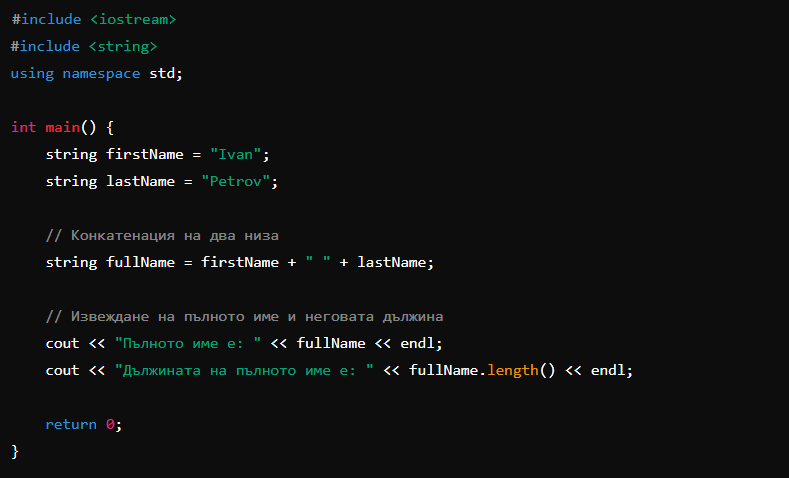
### ****4.4.1. Операции с масиви (търсене, сортиране, сума на елементи)****

**Пример: Сума на елементи в масив**



### ****4.4.2. Обработка на низове (конкатенация, дължина на низ)****

**Пример: Конкатенация и дължина на низове**



### ****5. Функции в C++****

Функциите в C++ са основен инструмент за организиране на кода в по-малки, управляеми части. Те позволяват многократна употреба на код и улесняват разделението на задачите.

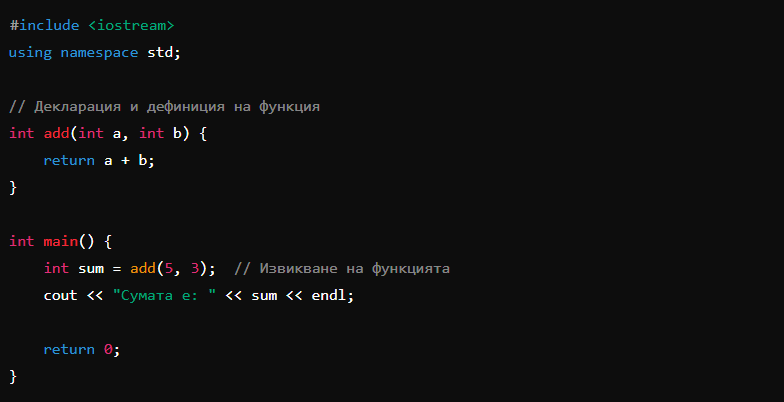
### ****5.1. Основи на функциите****

### ****5.1.1. Деклариране и дефиниране на функции****

Функциите се състоят от заглавие (декларация), в което се указват типът на връщаната стойност, името на функцията и списъкът с параметри. След това следва дефиницията, която съдържа тялото на функцията.

**Синтаксис:**

#### **Пример: Функция, която връща стойност и приема аргументи**

**Обяснение:**

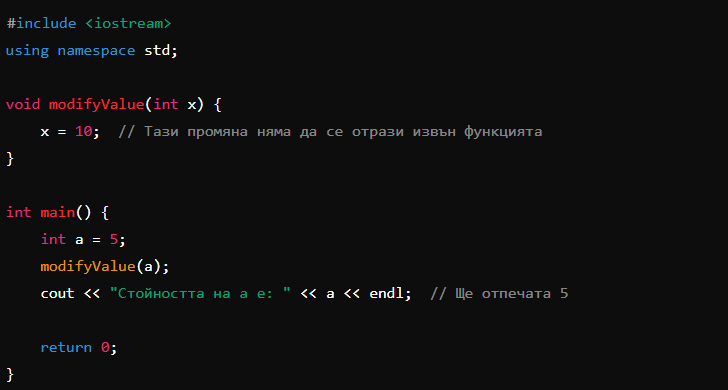
* Функцията add приема два целочислени аргумента и връща тяхната сума.

### ****5.2. Предаване на параметри****

### ****5.2.1. Предаване по стойност****

При предаването по стойност, параметрите се копират и промените в тях не се отразяват върху оригиналните променливи извън функцията.

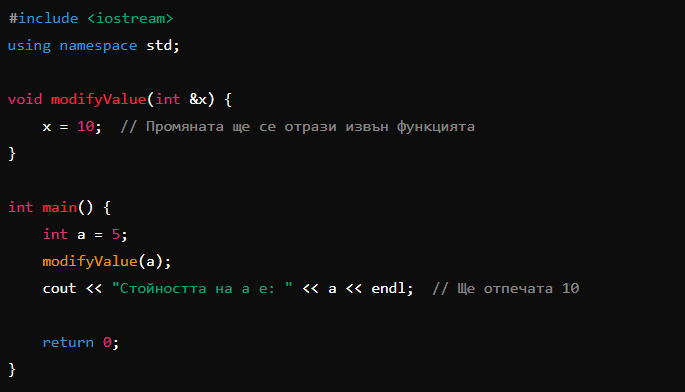
**Пример: Предаване по стойност**



### ****5.2.2. Предаване по референция****

При предаване по референция функцията получава пряк достъп до променливата и промените се отразяват върху оригинала.

**Пример: Предаване по референция**



### ****5.3. Рекурсивни функции****

Рекурсивната функция е такава, която извиква самата себе си. Това е полезно за решаване на проблеми, които могат да бъдат разделени на по-малки подобни подпроблеми.

### ****5.3.1. Основни примери за рекурсия****

**Пример 1: Факториел**

Факториелът на едно число nnn (обозначен като n!n!n!) е произведението на всички числа от 1 до nnn.

**Пример 2: Фибоначи**

Числата на Фибоначи са поредица, при която всяко число е сума от предходните две. Формулата е F(n)=F(n−1)+F(n−2)F(n) = F(n-1) + F(n-2)F(n)=F(n−1)+F(n−2), като F(0)=0F(0) = 0F(0)=0 и F(1)=1F(1) = 1F(1)=1.

### 

### ****5.3.2. Проблеми с рекурсията****

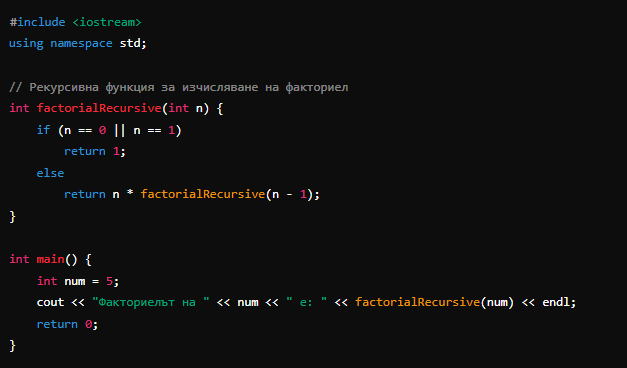
Основните проблеми, свързани с рекурсията, са:

* **Дълбочина на рекурсията** – прекалено дълбоката рекурсия може да доведе до препълване на стека (stack overflow).
* **Използване на памет** – всяко рекурсивно извикване изисква нова рамка в стека на паметта, което може да доведе до голяма употреба на памет при сложни изчисления.

**Факториел с рекурсия**

Рекурсивното решение на факториела се основава на математическата дефиниция:

* n!=n×(n−1)!n! = n \times (n-1)!n!=n×(n−1)!, като 0!=10! = 10!=1.

**Факториел с итерация**

Итеративното решение използва цикъл, за да изчисли факториела. В този случай се избягва използването на рекурсия.

**Обяснение:**

* **Рекурсивно решение:** Функцията извиква сама себе си, докато достигне базовия случай n==1n == 1n==1. Всеки път, когато функцията се извиква, текущата стойност на nnn се умножава по резултата от следващата рекурсивна стъпка.
* **Итеративно решение:** Използва цикъл for, за да умножи всички числа от 1 до nnn, което също води до изчисляване на факториела, но без да използва допълнителни стъпки на рекурсия.

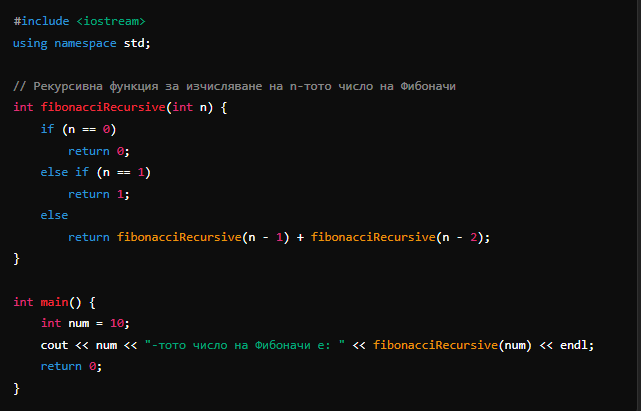
### Пример за изчисляване на числата на Фибоначи: Рекурсия срещу Итерация

Числата на **Фибоначи** се дефинират по следния начин:

* F(0)=0F(0) = 0F(0)=0
* F(1)=1F(1) = 1F(1)=1
* F(n)=F(n−1)+F(n−2)F(n) = F(n-1) + F(n-2)F(n)=F(n−1)+F(n−2) за n≥2n \geq 2n≥2

Числата на Фибоначи чрез рекурсия

Рекурсивното решение директно следва математическата дефиниция:



Числата на Фибоначи чрез итерация

Итеративното решение използва цикъл, за да изчисли числата на Фибоначи без нужда от рекурсия:

### Обяснение:

* **Рекурсивно решение:** Функцията се извиква рекурсивно за стойностите n−1n-1n−1 и n−2n-2n−2, докато достигне базовите случаи F(0)F(0)F(0) и F(1)F(1)F(1). Всяко извикване изчислява част от предишната стъпка.
* **Итеративно решение:** Използва два променливи a и b, които представляват предишните две числа на Фибоначи. На всяка стъпка от цикъла се изчислява следващото число, като стойностите на предишните две числа се обновяват.

### Разлики:

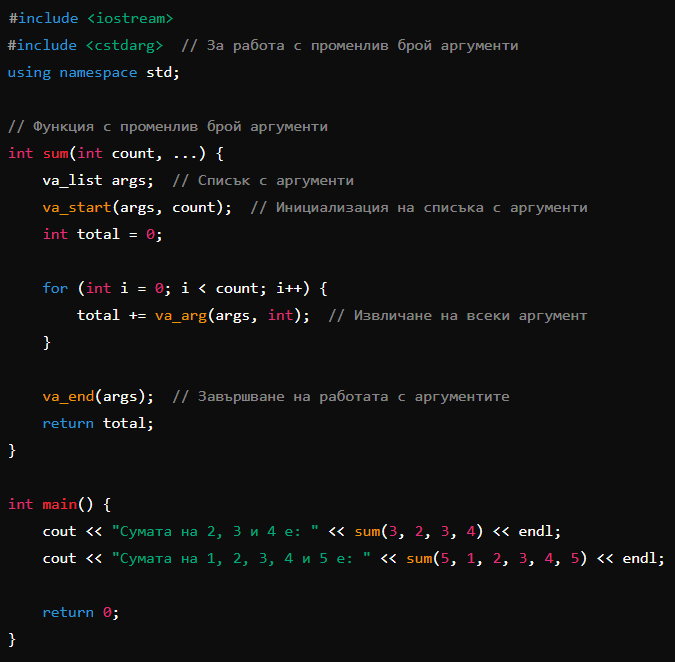
* **Рекурсията** е по-интуитивна и лесна за разбиране, но е по-неефективна, особено за по-големи стойности на nnn, защото води до множество излишни изчисления на едни и същи стойности (експоненциално време за изпълнение).
* **Итерацията** е по-ефективна, тъй като се изпълнява в линейно време и не изисква допълнителна памет за рекурсивните извиквания.

### ****5.4. Функции с променлив брой параметри****

В C++ можем да създаваме функции, които приемат различен брой аргументи чрез използване на специални конструкции.

### ****5.4.1. Пример: Използване на функции с променлив брой аргументи****

Един от начините за създаване на функция с променлив брой параметри е чрез използването на ... (елипсиса), съвместно с библиотеката <cstdarg>.



### ****6. Алгоритми и структури от данни****

Алгоритмите и структурите от данни са основни концепции в програмирането, които позволяват ефективно управление на данните и оптимизация на времето за изпълнение на различни задачи.

### ****6.1. Основни алгоритми****

### ****6.1.1. Сортиране****

##### **Bubble Sort (Мехурчесто сортиране)**

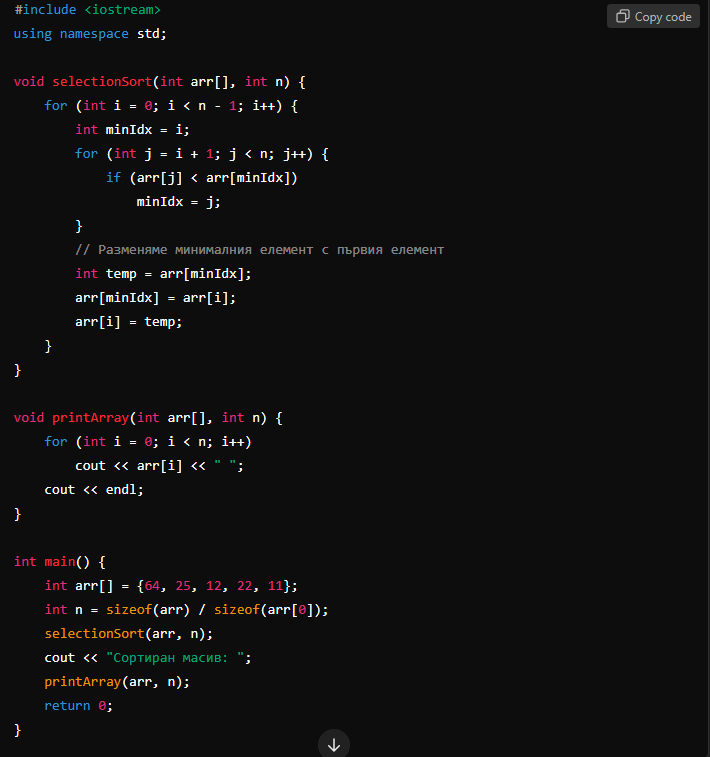
Bubble Sort е прост алгоритъм за сортиране, който повтаряно обхожда списъка и разменя съседни елементи, ако са в грешен ред. Процесът се повтаря, докато списъкът не бъде сортиран.

**Пример: Bubble Sort**

##### **Selection Sort (Сортиране чрез избор)**

Selection Sort търси най-малкия (или най-големия) елемент от несортираната част на масива и го поставя в правилната позиция. Това се повтаря за всяка позиция в масива.

**Пример: Selection Sort**



### ****6.1.2. Търсене****

##### **Линейно търсене (Linear Search)**

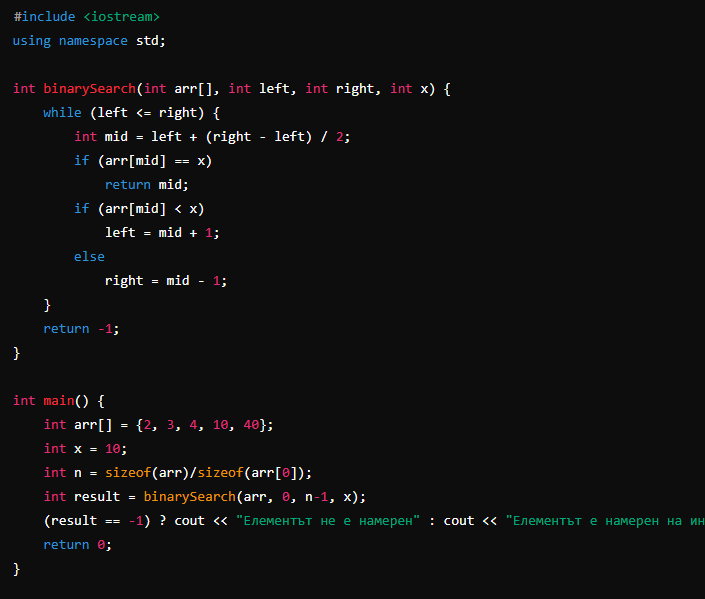
Линейното търсене обхожда всеки елемент от масива и проверява дали той е равен на търсения елемент.

**Пример: Линейно търсене**

##### **Бинарно търсене (Binary Search)**

Бинарното търсене е по-ефективен алгоритъм за търсене, който работи само с вече сортирани масиви. Алгоритъмът разделя масива на половини, като проверява средния елемент и елиминира едната половина на всеки етап.

**Пример: Бинарно търсене**



### ****6.2. Структури от данни****

### ****6.2.1. Стекове и опашки****

##### **Стек (Stack)**

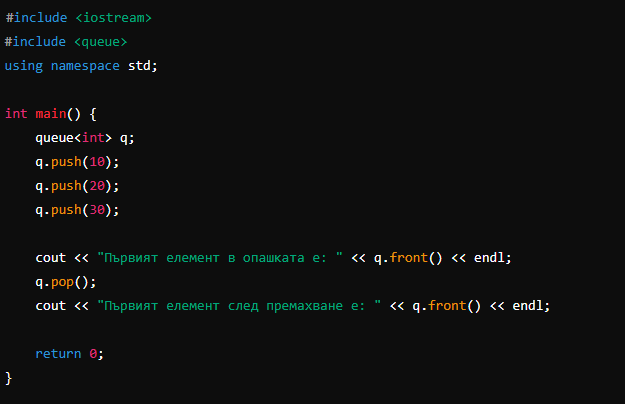
Стекът е структура от данни, която работи по принципа "Last In, First Out" (LIFO), тоест последният добавен елемент се премахва първи.

**Пример: Стек с вградени библиотеки**

##### **Опашка (Queue)**

Опашката е структура от данни, която работи по принципа "First In, First Out" (FIFO), тоест първият добавен елемент се премахва първи.

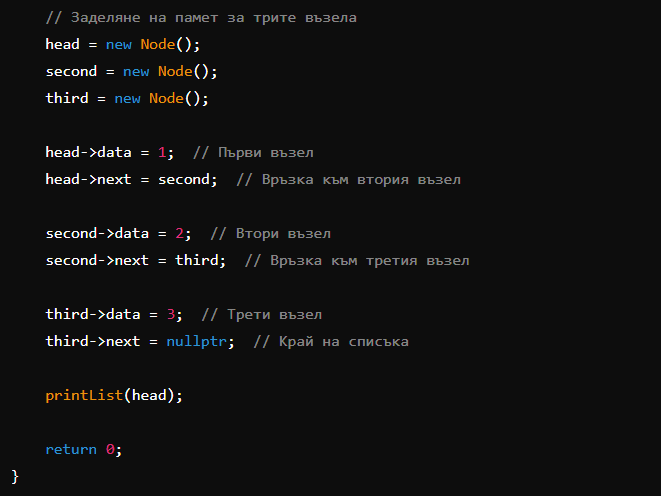
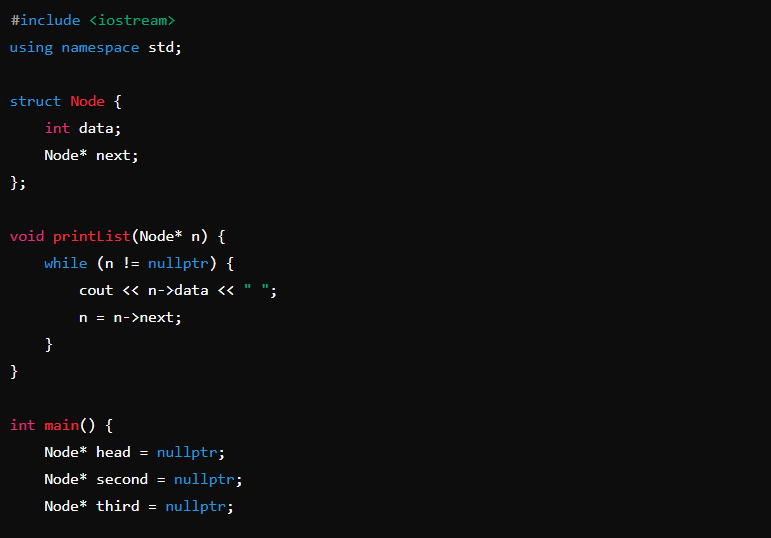
**Пример: Опашка с вградени библиотеки**



### ****6.2.2. Линейни свързани списъци****

Свързаният списък е динамична структура от данни, която се състои от възли (nodes), всеки от които съдържа данни и указател към следващия възел. Свързаните списъци могат да бъдат еднопосочни или двупосочни.

**Пример: Линеен свързан списък (еднопосочен)**



### Примерна Програма:

Тази програма на C++ симулира процеса на продажба на продукти от две различни компании за период от една седмица. Основната ѝ цел е да събира информация за продажбите на всеки продукт през всеки ден от седмицата и да извършва анализ на резултатите.

### Структура на програмата:

1. **Дефиниране на клас Product:**
   * Класът съдържа данни за всеки продукт като:
     + Име на марката (brand).
     + Количество в склада (qtyInStorage).
     + Общо продадени продукти (soldProducts).
     + Брой продадени продукти за всеки ден от седмицата (mostSoldPerDay[7]).
   * Чрез метода **конструктор** при създаването на обект от този клас се задава марката на продукта и началното количество на склад.
   * Включени са гетъри и сетъри, които осигуряват достъп до тези данни и позволяват промяната им, като се използва принципът на **енкапсулация**.
2. **Основната функция main:**
   * **Инициализация на продуктите:** Създават се два продукта - "Coca Cola" и "Pepsi", като и двата започват с 50 броя наличност в склада.
   * **Въвеждане на данни:** Програмата изисква от потребителя да въведе броя на продадените продукти за всеки ден от седмицата за двата продукта. Ако потребителят въведе невалидна стойност (нула или отрицателно число), програмата иска ново въвеждане за съответния ден.
   * **Актуализиране на данните:** Всеки път, когато се въведат валидни продажби, програмата:
     + Увеличава общия брой продадени продукти.
     + Намалява количеството продукти в склада.
     + Записва продажбите за конкретния ден в масива mostSoldPerDay.
3. **Общ брой продадени продукти:**
   * След като въведените данни са събрани за двата продукта, програмата изчислява и извежда общия брой на продадените продукти през седмицата.
4. **Сравнение на продажбите между двата продукта:**
   * Програмата проверява кой от двата продукта е продал повече бройки през седмицата и извежда съответното съобщение:
     + Ако първият продукт е продал повече, програмата изписва, че той е по-популярен.
     + Ако вторият продукт има повече продажби, се извежда, че той е предпочитан.
     + В случай на равни продажби за двата продукта, се съобщава, че и двата продават еднакво.
5. **Намиране на деня с най-много продажби:**
   * Програмата събира броя на продадените продукти от двата продукта за всеки ден от седмицата.
   * Накрая, тя идентифицира деня с най-много общи продажби и го извежда в текстова форма (като конкретен ден от седмицата – Понеделник, Вторник и т.н.).

### 