ХИМИКОТЕХНОЛОГИЧЕН И МЕТАЛУРГИЧЕН УНИВЕРСИТЕТ – СОФИЯ

ИНФОРМАТИКА

част първа

Основи на програмирането на C++

лектор: гл. ас. д-р Стефан М. Панов

Катедра “Информатика”

Лекция 2

Възникване на C++
Първа програма на C++

Цел на курса — да се научим да пишем програми на C++ — един от най-мощните програмни езици и в наши дни.

Езикът C++ — това е ключ към съвременното обектно-ориентирано програмиране (ООП). Той е създаден за разработка на високопроизводително програмно осигуряване и е изключително популярен сред програмистите. C++ дава концептуалния фундамент, на който се опират другите езици за програмиране и много от съвременните средства за обработка на данни. Неслучайно потомци на C++ са такива утвърдени и популярни езици като C# и Java. Последните два използват същия базов синтаксис и същите принципи за разработка.

Историята за създаването на C++ започва с езика C. Можем да твърдим, че C++ представлява супермножество на езика C. (Всички компилатори на C++ могат да се използват и за компилация на C-програми.) C++ е разширена и подобрена версия на езика C, в която са реализирани принципите на ООП. Но C++ включва също и редица други усъвършенствания на C, например разширения набор от библиотечни функции. За да разберем напълно и оценим по достойнство C++ е необходимо да осмислим много "как" и "защо" по отношение на езика C.

Създаване на езика C

Появата на езика **C** разтърси компютърния свят защото той коренно промени подходът към програмирането и неговото възприятие. **C** се счита първият съвременен "програмистки език", понеже до неговото изобретяване компютърните езици са се разработвали главно като учебни

упражнения или са били резултат от действията на бюрократични структури. **C** е бил замислен и разработен от практикуващи програмисти и отразява техния подход към програмирането. Средствата му са били многократно обмисляни, шлифовани и изтествани от хора, които действително са работели с тоя език. В резултат **C** бързо завоювал умовете и сърцата на многочислени привърженици.

Езикът C е изобретен в началото на 70-те от Денис Ричи (Dennis Ritchie) за компютри PDP-11 (разработка на компанията DEC — Digital Equipment Corporation), работещи под управлението на ОС UNIX.

В продължение на години стандарт за езика **C** де-факто е служил езикът, поддържан от ОС UNIX и описан в книгата на Брайан Кърниган (Brian Kemighan) и Д. Ричи „The C Programming Language“ (Prentice-Hall, 1978). Обаче формалното отсъствие на стандарт станало причина за натрупващи се разминавания между различните реализации на езика. За да се променят нещата, в началото на 1983 г. е бил учреден комитет по създаване на ANSI-стандарта, призван — веднъж и завинаги — да определи езика **C**. Крайната версия на тоя стандарт (с название C89, но има и по-нови) е била приета

през декември 1989. Тя именно е фундаментът, на който бил построен езикът **C++**.

C често е наричан компютърен език от “средно ниво”. Това определение няма негативен оттенък, защото изобщо не означава, че **C** е по-маломощен и развит в сравнение с езиците от “високо ниво” или е сложен за използване, подобно на асемблер. (За езикът асемблер е характерно символното представяне на реален машинен код, който може да изпълнява даден компютър/ЦП.) В случая “средно ниво” означава, че **C** съчетава елементи на езиците от високо ниво (например Pascal, Modula-2 или Visual Basic) с функционалността на езика асемблер.

От гледна точка на теорията в един език от високо ниво е заложен стремежът да се даде на програмиста всичко, което той може да поиска във вид на вградени средства. Език от ниско ниво от своя страна не дава на програмиста нищо, освен достъп до реални машинни команди. Език от средно ниво предоставя на програмистите някакъв (неголям) набор от инструменти, **позволявайки им сами да разработват конструкции от по-**

високо ниво. С други думи езикът от средно ниво предлага вградена мощ в съчетание с гъвкавост.

Бидейки език от средно ниво, **С** позволява манипулиране на битове, байтове и адреси, т.е. основните елементи, с които работи компютърът. По такъв начин в **С** не е предвидено отделяне на апаратните средства на компютъра от програмните. Например, размерът на целочислените стойности в **С** е пряко свързан с размера на думата на централния процесор. В повечето езици от високо ниво съществуват вградени инструкции, предназначени за четене и запис на дискови файлове. В езика **С** всички такива процедури се изпълняват посредством извикване на библиотечни функции, а не с помощта на ключови думи, определени в самия език. Такъв подход повишава гъвкавостта на **С**.

С позволява на програмиста (по-точно, стимулира го) да дефинира подпрограми за изпълнение на операции от високо ниво. **Тия подпрограми се наричат функции**. Функциите имат много голямо значение за езика **С**. Можем да ги наречем строителни блокове на **С**. Програмистът може без особени усилия да създаде библиотека от функции, предназначена за

изпълнение на различни задачи, които да се ползват в неговите програми. В тоя смисъл програмистът може да персонализира **C** в съответствие със своите потребности.

Необходимо е да споменем още един аспект на езика **C**, много важен за **C++**: С е структуриран език. Най-характерната особеност на един структуриран език се явява **използването на блокове**. **Блок — това е набор от инструкции, които са логически свързани между една с друга**. Например, представете си инструкция **IF**, която при успешна проверка на своя логически израз е длъжна да изпълни 5 отделни инструкции. **Ако тия инструкции са групирани и се обръщаме към тях като към единно цяло, то такава група образува блок**.

Структурираният език поддържа **концепцията за подпрограми и локални променливи**. Локална променлива е обикновена променлива, която е известна само на подпрограмата, в която тя е дефинирана. Структурираният език също поддържа редица циклични конструкции **като while, do-while и for**.

Накрая (особено важно), **C** не носи отговорност за действията на програмиста. На програмиста се позволява да прави всичко, което поиска, но за последствията, т.е. за всичко, което прави програмата, отговаря не езикът, а той. **C** предоставя на програмиста практически пълна власт над компютъра, но това е и тежко бреме на голяма отговорност.

Предпоставки за възникване на езика C++

Справедливо може да зададем въпроса: След като **C** е толкова добър и полезен защо е изобретен **C++**? **Отговорът е: Заради нарастващата сложност на създаваните програми**. **C++** е един от начините да се справим с тоя проблем. Нека разгледаме етапите накратко и обобщено:
Първоначално, на първите компютри е писано направо на машинен език (изпълним от процесора код). Това е особено трудно, но докато програмите не са превишавали няколкостотин команди методът е имал право на съществуване. Следващата стъпка е възникване на *езика асемблер*, където има символно задаване на машинните команди. Всяка процесорна

архитектура има свой собствен специфичен асемблерен (и машинен) език! Използват се етикети, символи и изрази като операнди, за да представят адреси и други константи, освобождавайки програмиста от досадните изчисления на ръка. Асемблерният език се преобразува в изпълним от процесора код чрез помощна програма позната също като асемблер. Процесът на преобразуване е познат като асемблиране на кода.

Но програмите продължавали да имат все по-големи размери и стремежът на програмистите да се справят с все по-голямото ниво на сложност довело до появата на езици от по-високо (в сравнение с асемблер) ниво, даващи повече инструменти за работа.

Първият такъв широко разпространен език за програмиране бил FORTRAN. Независимо от това, че той се явява значителна крачка по пътя на прогреса в областта на програмирането, FORTRAN все пак трудно може да се нарече език способстващ за написването на ясни и прости за разбиране програми. Края на 60-те години на миналия век се счита за периода на появата на структурното програмиране – метод, реализиран много успешно в езика **C**. С помощта на структурното програмиране е могло да се пишат програми със

средна сложност, при това **без особено героични усилия от страна на програмиста**. Но ако програмният проект достигал определен размер, то даже с използването на структурните методи неговата сложност рязко нараствала и се оказвала непреодолима за възможностите на програмиста. Когато (в края на 70-те) към "критичната" точка достигнали много проекти, започнали да се раждат **нови технологии** за програмиране. Една от тях получила название **обектно-ориентирано програмиране (ООП).** Въоръжени с методите на ООП програмистите могли да се справят с програми с далеч по-голям размер, отколкото преди. Но езикът **C**, иначе с много достойнства, не поддържал методите на ООП. Стремежът да се получи обектно-ориентирана версия на езика **C** в края на краищата довел и да появата на **C++**.

Поява и еволюция на езика C++

Създаден е от датчанинът Бярне Строуструп (Bjame Stroustrup) в 1979 год. в компанията Bell Laboratories, САЩ като (както вече се подчерта) **C** е фундамента, на който се изгражда новия **C++**. **Основната разлика между C и C++ е, че C++ съдържа вградена в езика поддръжка на обектно-ориентирано програмиране. В C++ са добавени класове и обекти, множествено наследяване, виртуални функции, overloading, шаблони, обработка на изключения и вградени оператори за работа с динамична памет**.

Струва си да се подчертае все пак, че в основата на **C++** не е само **C**. Строуструп утвърждава, че някои обектно-ориентирани средства са били вдъхновени от друг обектно-ориентиран език, а именно Simula67. Така **C++** представлява симбиоза на две мощни методологии за програмиране. Важно е също да се помни и следното: Тъй като **C++** е супермножество на езика **C**, то научавайки се да програмирате на **C++**, вие ще можете да програмирате и на **C**! (Но при ограничен обем от часове нещата са по-различни ;)

Новият език е разработван от Bjame Stroustrup постепенно. В 1983 год. създаващият се език вече престанал да бъде просто допълнена версия на класическия **C** и бил преименуван от «С с класове» в «С++». Първото му официално представяне става октомври, 1985 год. До началото на официалната стандартизация езикът се е развивал основно от Stroustrup в отговор на заявките на програмисткото общество. Вместо стандарт се ползвали написаните от него трудове по **C++** (описание на езика, ръководство по **C++** и т.н.). Едва в 1998 г. е бил ратифициран първият международен стандарт на езика **C++**: ISO/IEC 14882:1998. (Виж таблица 1.1) Стандартът **C++** от 2003 год. се състои от две основни части: описание на ядрото на езика и описание на стандартната библиотека. В него са отстранени откритите грешки и недомислици на предишната версия на стандарта. Версията от 2007 г. описва основно разширенията на стандартната библиотека. Новото в стандарта от 2011 г. са допълнения в ядрото на езика, засягащи тънкости в дълбочина и не са от интерес за нашия курс. Версията от 2014 г. е незначително разширение на предишната – главно отстраняване на бъгове и леки подобрения. Последната е от 2017 г.

Година	C++ Стандарт	Неформално име
1998	*ISO/IEC 14882:1998*	C++98
2003	*ISO/IEC 14882:2003*	C++03
2007	*ISO/IEC TR 19768:2007*	C++TR1
2011	*ISO/IEC 14882:2011*	C++11
2014	*ISO/IEC 14882:2014(E)*	C++14
2017	*ISO/IEC 14882:2017(E)*	C++17

Таблица. 1.1 Стандарти на езика C++

Какво е това ООП ?

Обектно-ориентираният подход към програмирането позволява една задача да се разложи на съставни части така, че всяка част да представлява самостоятелен **обект, който съдържа собствени инструкции и данни**. При такъв подход съществено се понижава общата сложност на програмите, т.е. улеснява се програмиста. **Всички обектно-ориентирани езици се характеризират с три общи признака: капсулация, полиморфизъм и наследяване**.

Капсулация

Като правило, програмите се състоят от два основни елемента: инструкции (код) и данни. Кодът изпълнява действията, а данните представляват информацията, върху която са насочени тези действия. **Капсулацията (понякога „капсуловане“) е механизъм, свързващ в едно кодът и данните, които той обработва**, с цел да ги обезопаси както от външно вмешателство, така и от неправилно използване. **При такова свързване (обединение) се**

създава обект! Вътре в обекта кода, данните или и двете могат да бъдат или скрити в “рамките” на тоя обект или открити. Скритият код (или данни) е известен и достъпен само от другите части на същия обект. Откритият код (или данни) е достъпен от всяка част от програмата. Това носи съществени предимства, но последното може да се оцени само с практика!

Полиморфизъм (от гр. „много форми“)

За да се разгледа вторият основен признак са необходими поне начални програмни познания. Да разгледаме конструкцията „стек“ – в него добавянето и премахване на елементи става на принципа „последният дошъл е първият обслужен“. Примерно трябва да напишем програма, в която са нужни 3 различни типа стек в зависимост от типа на данните с които се работи – за цели числа, за реални числа и за символни низове (група от символи). Алгоритъмът за реализация на стека е един и същи, независимо от типа на данните. Въпреки това, в класическото програмиране програмистът трябва да напише 3 отделни подпрограми (функции), с отделни имена и всяка със собствен интерфейс. **Благодарение на**

полиморфизма може да се създаде един общ интерфейс, който подхожда и за трите ситуации (за нашия пример) . Изборът на конкретната функция в зависимост от контекста (ситуацията) е отговорност на компилатора и не се налага да бъде правен ръчно от програмиста. Т.е. програмистът използва общия интерфейс, а останалото се върши от компилатора.

Наследяване

Наследяването е механизъм, благодарение на който един обект може да продобие свойствата на друг. С помощта на наследяването се реализира концепцията на йерархическата класификация. Много от областите на знанието се организират във вид на управляема йерархическа (низходяща) класификация. Например, ябълката „Златна превъзходна“ се явява част от класификацията „ябълка“, която на свой ред се явява част от класа „плод“, а той — част от още по-големия клас „храна“. Класът „храна“ притежава определени качества (ядливост, хранителна стойност и др.), които са приложими и към подкласа „плод“. Но освен тия качества, класът „плод“ има и специфични характеристики (сочност, сладост и др.), които го

отличават от другите хранителни продукт. В класа „ябълка“ се определят качества, специфични за ябълките (растат на дървета, в умерен климат и др.). Класът „Златна превъзходна“ наследява качествата **на всички предходни класове** и освен това определя нови качества, които са уникални само за тоя сорт ябълки.

Ако не се използва йерархическото представяне на признаците, то за всеки обект би се наложило в явен вид да се определят всички присъщи му характеристики. Но благодарение на наследяването за един обект е нужно са се доопределят само тия качества, които го правят уникален вътре в неговия клас, доколкото той (обектът) наследява общите атрибути на своя родител. Следователно, именно механизмът на наследяването позволява на един обект да представлява конкретен екземпляр на по-общия клас.

Фактът, че **C++** поддържа ООП, не означава, че един **C++**-програмист може да пише единствено обектно-ориентирани програми. Едно от най-важните достойнства на езика **C++** (както и на неговия предшественик, езика **C**) е гъвкавостта.

Връзка на C++ с езиците Java и C#

Много от вас знаят за съществуването на програмните езици, Java и C#. Java е разработен в компанията Sun Microsystems, а C# в Microsoft. Какво е отношението на тези два езика към C++ ?

C++ е родителски език и за Java, и C#. И макар разработчиците на Java и C# да са добавяли към първоизточника, премахвали от него или модифицирали някои средства, **като цяло** синтаксисът на трите езика е практически идентичен. Нещо повече, обектният модел, използван от **C++**, е подобен на обектните модели на Java и C#. Което значи, че знаейки **C++** вие можете сравнително лесно да изучите Java или C#. Обратното е също вярно: ако знаете Java или C#, изучаването на **C++** не представлява особен проблем.

Основното различие между C++, Java и C# е в типа изчислителна среда, за която се е разработвал всеки от тия езици. **C++** се е създавал с цел създаване на високоефективни програми, предназначени за изпълнение под управлението на определена ОС и ЦП от конкретен тип.

Езиците Java и C# са разработени в отговор на уникалните потребности на силно разпределена мрежова среда, която често е и типичен пример за съвременна изчислителна среда. Java позволява създаването на междуплатформен (съвместим с няколко ОС) **преносим** програмен код за Internet. Образно казано, Java-програмата може свободно "да се носи из Internet просторите". C# е разработен за средата .NET Framework (Microsoft), която поддържа многоезично програмиране (mixed-language programming) и компонентно-ориентиран код, изпълняем в мрежова среда.

Java и C# позволяват създаването на преносим код, работещ в разпределена среда, но това си има цена. Java и C# програмите се изпълняват по-бавно от **C++**-програмите.

Извод: ако ще се създават високоефективни приложения, използвайте C++. Ако са нужни преносими програми, ползвайте Java или C#.

Затова и въпросът "Кой език е по-добър?" е поставен некоректно. Уместно е въпросът да се зададе другояче: "Кой език най-добре подхожда за решението на дадената задача?".

Написване и компилация на първата ни програма

Най-трудно при изучаване на един език за програмиране е това, че нито един негов елемент не съществува изолирано от другите. Компонентите на езика работят заедно, образно казано в „задружен колектив”. Такава тясна взаимовръзка усложнява разглеждането на един аспект от C++ без разглеждането на другите. Често обсъждането на едно средство предвижда предварително запознанство с други. Ще започнем с кратко описание на една проста, но реална програма. **Стъпките са: въвеждаме текста, а след това компилация и изпълнение**.

Ние ще работим основно в **Dev-C++** среда за разработка. Избрана е защото това е **безплатна среда, сравнително лесна за работа**. Текстът на една програма може да се въведе като **сорс файл** (наричан често изходен код) в самата **Dev-C++**, което е доста удобно. Обаче може и отделно в някой

текстов редактор (примерно Notepad или WordPad), но не и като docx файл в MS Word. Разликата е, че с Notepad/ WordPad се създава текстов файл, докато във втория случай ще имаме текстов файл плюс форматиране, което ще попречи

```
/*
  Programa # 1 - Nashata pyrva C++ programa.
  Vyvedete taia programa, posle ia kompiliraite i izpylnete.
*/
#include <iostream>
using namespace std;
// main() - nachalo na izpylnenieto na programata.
int main()
{
   cout << "Nashata pyrva C++ programa.";
   return 0;
}
```

Фиг. 2.1 Текстът на нашата първа програма

на работата на **C++** компилатора. Ако работим в **Dev-C++** среда най-лесният начин да започем е първо да изпълним **File/New/Source File**.

След като въведем текста (алтернативата е да го копираме отнякъде, но без форматиране) избираме **File/Save As** за да го запомним като файл в предварително направена папка за нашите учебни програми. Името на сорс файла формално може да бъде произволно, но **C++ програмите** обикновено се съхраняват с разширение **.cpp**

Как и защо се създава проект в **Dev-C++** ще бъде показано на упражнение:

File/New/Project и т.н. вместо **File/New/Source File**. Сега само ще посочим, че ако имаме отнякъде готова програма (.cpp файл) можем веднага с мишката да я „хвърлим“ от File Explorer в основния прозорец на **Dev-C++** средата вместо да създаваме нов сорс файл.

Следващата стъпка е да извикаме компилатора с **Execute/Compile** за нашият .cpp файл. Ако има **синтактични грешки, т.е. написаното не отговаря на изискванията на синтаксиса на езика C++**, трябва да отстраним грешките и да изпълним действието отново. Резултатът от работата на компилатора е създаването на **обектен файл** със същото име и с **разширение .obj или .o**. В зависимост от настройките на **Dev-C++** средата обектният файл може да се запише и в друга папка, различна от мястото на сорс файла. При успешно създаден обектен файл в **Dev-C++** среда автоматично, без наша намеса, се извиква свързващият редактор (линкера) за създаването на **изпълним файл** със същото име и **разширение .exe** (виж и лекция 7 от първи семестър). Изпълнимият файл може да се пусне за изпълнение с **Execute/Run** в **Dev-C++** среда или в прозорецът на **„Command Prompt“** от самата операционна система като изпишем името на нашия файл с или без разширение .exe. Резултатът от изпълнението (от ЦП) на нашата програма в **Dev-C++** среда се вижда в нов прозорец (Output Window).

Заб. В някои случаи, вкл. и за **Dev-C++** средата когато работим без създаване на проект, след успешно създаване на изпълним файл от обектния последният (обектният) може да се изтрие автоматично. Това е причината да не можем да го видим в папката, където се намират сорс файлът и изпълнимият файл. В зависимост от настройките на текущия проект създаването на обектен и изпълним файл може да става и в други папки, различни от тая на сорс файла.

Поредов разбор (анализ) на нашата първа програма

```
/*
  Programa # 1 - Nashata pyrva C++ programa.
  Vyvedete taia programa, posle ia kompiliraite i izpylnete.
*/
```

Това е коментар. Подобно на повечето програмни езици C++ позволява вмъкването в изходния код на програмата на **коментари, съдържанието на които се пренебрегва от компилатора**. В нашия конкретен случай коментарът просто идентифицира програмата и ни напомня, какво трябва да правим с нея. **В общия случай с помощта на коментари се описват или разясняват действия, изпълними в програмата и тия разяснения са предназначени за тези, които ще четат (анализират) изходния код**.

Коментар — това е текст с пояснително съдържание, вмъкнат в програмата.

В C++ се поддържат два типа коментари. Първият, показан в началото на нашата програма се нарича **многоредов**. Той е длъжен да започва със символите **/ *** и да завършва със същите, но в обратен ред, т.е. ***/**. Всичко, което се намира между тях компилаторът го игнорира. Коментарите от тоя тип, както следва от названието, могат да заемат няколко реда, но може да са и на един ред.

```
// main() - nachalo na izpylnenieto na programata.
```

Така изглеждат вторият тип коментари, **едноредовите** в C++. **Такъв коментар започва с двойка символи //и завършва в края на реда**.

Характерът на използваните коментари е лично дело на програмиста, но добре написаните такива улесняват значително разбирането на една

програма. Оттам и препоръката да не се спестяват коментари даже когато сроковете за разработка на едно ПО са пределно кратки.

```
#include <iostream>
```

Включва библиотечния заглавен файл (хедър-файл) **iostream** в програмата. Един хедър-файл представлява външен изходен файл, вмъкван **винаги в началото** на C++ програмата с помощта на **директивата # include**. **iostream е необходим за правилната работа на входно-изходната система на C++.** Символите **<>** означават, че файлът ще търси в папките, съдържащи файловете на стандартната библиотека (пътят към тия папки зависи от реализацията на компилатора). По-късно ще се запознаем по-подробно със заглавните файлове и ще узнаем, защо са така важни за C и C++ програмите.

C/C++ препроцесор (англ. preprocessor) — програма, подготвяща кода на една C/C++ програма за компилация. Извиква се от компилатора като първа стъпка в процеса на транслация. **Командите на препроцесора (една от тях е**

include) се наричат директиви и са редове от изходния код на програмата. **Започват винаги с #**.

using namespace std;

Езикът C++ позволява писане на големи програми, при което могат да се ползват много на брой различни готови библиотеки. Имената на елементите на отделните библиотеки (примерно имена на готови функции) могат да съвпадат. Когато ползваме някоя готова функция как компилаторът да знае коя точно функция имаме предвид ние, от коя библиотека? Решението е използване на "Пространството на имената“ (namespace), като едно пространство съдържа имената на една библиотека. В нашия ред **ключовата дума „using“** информира компилатора за използването на заявеното пространства на имена **std**. **Именно в това пространство е обявена цялата стандартна библиотека на C++**. **Всеки програмен език, вкл. C++, притежава предварително дефинирано множество от ключови думи.**

int main()

Както се съобщава във вече разгледания по-нагоре едноредов коментар именно от тоя ред започва изпълнението на нашата програма.

С функцията main () започва изпълнението на всяка C++ програма.

Всички **C++** програми се състоят от една или няколко функции. Под функция ние разбираме подпрограма. **Всяка C++ функция има име, но само една от тях се нарича main() и винаги трябва да се включва във всяка C++ програма.** Изпълнението на една C++ програма започва с извикване на main() и обикновено завършва с връщане от main (). Отварящата къдрава скоба **{** на следващия ред указва началото на кода на main (). **(Добре е още тука да се каже, че в C++ { винаги означава начало на нещо, а } означава край на това, което е било започнато със съответстващата ѝ отваряща скоба.)** Ключовата дума **int** (съкращение от integer), стояща пред името main(), означава типа данни за стойността, връщана от функцията main ().

Както ще видим по-късно **C++** поддържа няколко вградени типа данни, един от които е **int** за целите числа.

```
cout << "Nashata pyrva C++ programa.";
```

Това е инструкция за извеждане на данни. При нейното изпълнение на екрана на компютъра, в отделен прозорец, ще се появи съобщение „Nashata pyrva C++ programa.“ В тая инструкция се използва **операторът за извеждане „<<”**. **Той извежда изразът, стоящ отдясно, на устройството, указано отляво**. Думата **cout** е вграден идентификатор (съставен от двете думи **console output**), който в повечето случаи означава екрана на компютъра. Обърнете внимание, че разглежданата инструкция завършва с ; (точка и запетая). **В действителност всички изпълними C++ инструкции завършват с точка и запетая**. Съобщението "Nashata pyrva C++ programa." представлява низ. **В C++ под низ се подразбира последователност от символи, затворени в двойни кавички**.

А със следващия ред завършва функцията main ().

```
return 0;
```

При неговото изпълнение функцията main () връща на извикващия процес (в ролята на който обикновено е ОС) стойността 0. За повечето операционни системи нулевата стойност, която се връща, свидетелства за нормалното завършване на програмата. Другите стойности могат да означават завършване свързано с някаква грешка. **Думата return също се отнася към ключовите думи в C++ и се използва за завършване на дадена функция.** След нея обикновено има върната стойност, но може да съществуват и функции, които не връщат нищо.

Затварящата къдрава скоба } в последния ред указва края на кода (тялото) на main(). Ако в нашата програма return отсъстваше, тя (програмата) би завършила автоматично при достигане на тая затваряща скоба. **Кодът на всяка фукция завършва със затварящата къдрава скоба }.**

Обработка на синтактичните грешки

На всеки програмист е известно колко лесно е да се допуснат случайни грешки при въвеждане на текста на една програма. За наше добро, при опит да се компилира такава програма компилаторът ще сигнализира със съобщение за наличието на синтактични грешки. Мнозинството от C++ компилаторите се опитват "да видят" смисъл в изходния код на програмата, независимо от това, какво сме въвели. **Затова съобщението за грешка не винаги отразява истинската причина за проблема.** Например, ако в нашата програма случайно или нарочно пропуснем откриващата къдрава скоба след функцията **main()**, компилаторът ще посочи като източник на грешка инструкцията **cout**. Ето защо при получаване на съобщение за грешка прегледайте и два-три реда от кода непосредствено предшестващ реда с "откритата" грешка. Просто понякога компилаторът започва "да надушва лошото" едва няколко реда след реалното местоположение на грешката.

Предупреждения

C++ компилаторите дават като резултат от своята работа не само съобщения за грешки, но и предупреждения (warnings). В езикът C++ “по рождение” е заложено великодушно отношение към програмиста, т.е. той позволява на програмиста практически всичко, което е коректно от гледна точка на синтаксиса. Но даже и на “всепрощаващия” C++ компилатор някои синтактично правилни неща могат да се сторят подозрителни. Именно в такива ситуации се издава предупреждение. **Тогава програмистът сам е длъжен да прецени, доколко справедливи са подозренията на компилатора.** Според някои експерти, доста компилатори са прекалено бдителни и предупреждават по повод на напълно коректни инструкции.