• ХИМИКОТЕХНОЛОГИЧЕН И МЕТАЛУРГИЧЕН УНИВЕРСИТЕТ – СОФИЯ

ИНФОРМАТИКА

част втора

Основи на програмирането на C++

лектор: гл. ас. д-р Стефан М. Панов

Катедра “Информатика”

Лекция 7 (15)

Функции в C++ - втора част

Създаване на ограничен масив

Типът псевдоним като тип на стойност, връщана от функция може с успех да се прилага за създаването на „**ограничен**“ масив. Както знаем, при изпълнение на C**++** код не е предвидена проверка за нарушение на границите при индексиране на масиви. Това означава, че може да бъде зададен индекс, превишаващ размера на масива и по тоя начин да се опитаме да четем или да пишем извън него. Но при създаването на ограничен, или още наречен безопасен, масив излизането извън неговите граници може да се предотврати. При работа с такъв масив всеки излизащ извън границите индекс не се допуска за **действително** индексиране.

Един от начините за създаване на ограничен масив е демонстриран в програма **Prog76.cpp**.

При използване на двете функции индексът на интересуващия ни елемент се задава като аргумент. Обърнете внимание на това, **че функцията put()**

връща указател към зададения елемент и затова се използва в лявата част на инструкциите за присвояване.

Да споменем, че можем да създадем и собствен безопасен масив, при работа с който се използват стандартните обозначения.

Презареждане на функции

Презареждането на функции е механизъм, позволяващ на няколко родствени функции да имат еднакви имена.

В **C++** няколко функции могат да имат еднакви имена, **но при условие, че техните параметри са различни.** Такава ситуация се нарича **презареждане на функции** (**function overloading**), а функциите, които са задействани — **презаредени** (**overloaded**). Презареждането на функции е един от начините за реализация на полиморфизма в **C++.**

Да разгледаме прост пример за презареждане на функции - **Prog77.cpp**.

Както е видно, функцията **f()** се презарежда три пъти. След като списъците от параметри за трите версии са различни, **компилаторът притежава достатъчно информация за да може да извика правилната версия на всяка функция**.

За определяне на това, каква версия на презаредената функция да се вика, компилаторът използва типа/типовете и/или броя на аргументите. Оттук, презаредените функции трябва да се отличават по типа/типовете и/или броя на аргументите. Независимо от това, че презаредените методи могат да се различават и по типовете на връщаните стойности, такъв вид информация не е достатъчен за C++ за да може във всички случаи компилаторът да реши, каква именно функция е нужно да се извика.

За да се разбере по-добре ползата от презареждането на функции, да разгледаме три функции от стандартната библиотека: **abs()**, **labs()** и **fabs().** Те са били определени навремето в езика **C**, **а после заради съвместимост**

включени в C++. Функция **abs()** връща абсолютната стойност (модула) на цяло число, функцията **labs()** връща модула на дълга целочислена стойност (тип **long**), а **fabs()** — модула на стойност с плаваща точка (типа **double**). **И понеже езикът C не поддържа презареждане на функции, всяка функция трябва да има собствено име, независимо от това, че и трите функции изпълняват (по същество) едно и също действие.** Това прави ситуацията да изглежда по-сложна, отколкото е в действителност. **С други думи, при едни и същи действия програмистът трябва да помни имената на три (в дадения случай) функции вместо на една**. Но в **C++**, както е показано в следващия пример **Prog78.cpp,** може да се ползва само едно име – примерно **myabs** - за всичките три функции.

Всяка от тях връща абсолютната стойност на своя аргумент. Във всяка ситуация при извикване компилаторът “знае” каква именно функция му е нужна. **За вземане на решение на него му е достатъчно “да погледне” типа**

на аргумента, предаван на функцията. Принципната значимост на **презареждането** се състои в това, че то **позволява да се обръщаме към свързани функции посредством едно, общо за всички име**. Следователно, името **myabs** представлява общото действие, което се изпълнява във всички случаи.

Заб. В зората на създаването на **C++** презаредените функции е трябвало явно да се обявяват като такива с помощта на ключовата дума **overload**. **Тази ключова дума повече не е нужна в C++**.

Съвет: Има смисъл да се презареждат само тясно свързани операции.

Аргументи, предавани на функция по подразбиране

В C++ можем да укажем на параметър някаква стойност, която автоматично ще се използва, ако при извикване на функцията не се задава

аргумент съответстващ на тоя параметър. Аргументите, предавани на функции по подразбиране, може да се използват за да се опрости обръщението към сложни функции, а също и като “съкратена форма” на презареждане на функции.

Задаването на аргументи, предавани функции по подразбиране като синтаксис е аналогично на инициализация на променливи. Да разгледаме следния пример, в който се обявява функция **myfunc().** Тя приема един аргумент от тип **double** със стойност по подразбиране **0.0** и един символен аргумент със стойност по подразбиране '**R**'.

```
void myfunc(double num = 0.0, char ch = 'R')  { ...}
```

След такова обявление **myfunc()** може да се вика по един от следните начина:

```
myfunc(198.234, 'A'); // Предаваме явно зададените стойности.
```

myfunc(10.1); // За параметъра **ch** се прилага аргументът,

// зададен по подразбиране (' **R** ').

myfunc(); // И за двата параметъра се прилагат

// аргументите зададени по подразбиране.

За да се предугади максимално възможното количество ситуации и се осигури тяхната коректна обработка, **функциите често се обявяват с по-голям брой параметри, отколкото е необходимо в най-разпространените случаи**. Ето защо, благодарение на прилагането на аргументи по подразбиране не е нужно да се указват всички аргументи (използвани в общия случай), **а само тия, които имат смисъл за дадената ситуация**.

Колко е полезно предаването на аргументи по подразбиране е показано за функцията **clrscr(),** представена в програма **Prog79.cpp**. Функцията **clrscr()** изчиства екрана чрез извеждане на последователност от символи за нов ред

(Това не е най-ефективният начин, но е подходящ за дадения пример). Понеже в най-често използвания видеорежим за изображения се извеждат 25 реда текст на екрана, то като аргумент по подразбиране се използва стойността 25. Но тъй-като в други режими на екрана може да се извеждат повече или по-малко от 25 реда аргументът, действащ по подразбиране може да се преопредели явно, указвайки нужната стойност.

При създаване на функции, имащи стойности на аргументите, предавани по подразбиране е необходимо да се помнят три неща:

Стойностите по подразбиране трябва да бъдат зададени веднъж, при това при първото обявяване на функцията във файла. (най-добре в прототипа)

Разните версии на презаредена функция могат да имат различни по стойност аргументи, действащи по подразбиране.

Всички параметри, които приемат стойност по подразбиране трябва да са разположени вдясно от останалите.

Както вече се спомена, едно от приложенията на предаване на аргументи по подразбиране е “съкратената форма” на презареждане на функции.

За да разберем това да си представим, че е нужно да създадем две “адаптирани” версии на стандартната функция **strcat().** Едната версия трябва да присъединява **цялото** съдържания на единия низ в края на другия. Втората приема трети **аргумент, който задава броя на** присъединяемите символи.

Макар че за постигане на поставената цел може да се реализират две версии на функцията **mystrcat()** съществува по-прост начин на решение на тая задача. Използвайки предаване на аргументи по подразбиране може да създадем само една функция **mystrcat(),** както е постъпено в следващата програма **Prog80.cpp**. Както се вижда, ако стойността на **len** е равно на -1

функцията **mystrcat()** добавя към низа **s1** целия низ, адресиране от параметър **s2**. Осигурявайки за параметъра **len** аргумент предаван по подразбиране, двете операции може да се обединят в една функция.

Тоя пример позволи да се демонстрира как аргументите, предавани на функция по подразбиране, осигуряват основа за съкратена форма на обявление на презаредени функции.

Относно използването на аргументи, предавани по подразбиране

Независимо от това, че аргументите предавани на функции по подразбиране са много мощно средство в програмирането (при тяхното коректно използване), с тях понякога могат да възникнат проблеми. Тяхното предназначение е да позволят на функциите ефективно да изпълняват своята работа, осигурявайки при цялата простота на тоя механизъм значителна гъвкавост. **Ако не съществува някаква единна стойност, която обикновено се присвоява на даден параметър, то няма смисъл да се**

обявява съответен аргумент по подразбиране. **И последно, случайното използване на аргумента по подразбиране не трябва да води до необратими отрицателни последствия.** **Та нали такъв аргумент може просто да се забрави да се укаже при извикване на някоя функция**. И ако това се случи, подобна „издънка“ не би трябвало да води , например, до загуба на важни данни!

Презареждане на функции и неоднозначност

Неоднозначност възниква тогава, когато компилаторът не може да определи различието между две презаредени функции.

Възможни са ситуации, в които компилаторът е неспособен да направи избор между две (или повече) коректно презаредени функции. Съответно кодът няма да се компилира. **Основна причина за неоднозначност в C++ е автоматичното преобразуване на типове**. Както е известно, **в C++** се прави опит автоматично да се преобразува типът на аргумента, използван при

извикване на функция, в типа на параметъра, определен във функцията. Да разгледаме пример **Prog81.cpp**.

При викане на функция **myfunc()** с аргумент цяло число 10, **в програмата се внася неоднозначност, защото за компилатора е неизвестно, в какъв тип се очаква да се преобразува тоя аргумент**: **float** или **double**. **И двете преобразования са допустими.** В такава неоднозначна ситуация ще се издаде съобщение за грешка и програмата няма да се компилира.

Да подчертаем, грешката не е в презареждането на функцията myfunc(), а в конкретното нейно извикване.

Още един пример за неоднозначност, предизвикан от автоматичното преобразуване на типове в **C++** е даден в **Prog82.cpp**. Типовете **unsigned char** и **char** са различни. При викане на функцията **myfunc()** с целочислен аргумент **88** компилаторът “не знае” каква функция да изпълни, т.е. в

стойност от какъв тип да се преобразува числото **88**: от типа **char** или от **unsigned char**? **И двете преобразования тук са напълно допустими**.

Неоднозначност може да се предизвика и при използването в презаредни функции на аргументи, предавани по подразбиране.

Да разгледаме например следващата програма **Prog83.cpp**. Компилаторът "не знае" дали да извика версията на функцията **myfunc(),** която приема един аргумент или да използва възможността да предаде аргумент по подразбиране към версията, която приема два аргумента.

Рекурсивни функции

Рекурсивната функция е функция, която вика сама себе си.

Когато една функция вика сама себе си в стека се заделя памет за новите локални променливи и параметри и кодът на функцията от самото ѝ начало се изпълнява с тия нови променливи. Рекурсивното викане не създава ново

копие на функцията. Нови са само аргументите. При връщане от рекурсивно извикване от стека се извличат старите локални променливи и параметри и изпълнението на функцията се възобновява от “вътрешната” точка на нейното извикване.

При написване на рекурсивна функция е необходимо да включим в нея инструкция за проверка на условието (най-често if-инструкция), която би осигурила изход от функцията без изпълнение на рекурсивно извикване. Ако не направим това, извиквайки такава функция, няма как да се завърнем от нея. При работа с рекурсия това е най-разпространеният тип грешки. Затова при разработка на програми с рекурсивни функции не си струва да се скъпим на инструкции **cout**, за да сме наясно какво се случва в конкретната функция.

Основното достойнство на рекурсията се състои в това, че някои типове алгоритми се реализират рекурсивно по-просто, отколкото техните

итеративни еквиваленти. Например, алгоритъмът за сортировка **Quicksort** много трудно може да бъде написан без рекурсия. Освен това, някои задачи (най-вече такива, свързани с изкуствен интелект) просто са създадени за рекурсивни решения. И на последно място, при някои програмисти процесът на мислене е организиран така, че им е по-лесно да мислят рекурсивно, отколкото итеративно.

Във следващата програма_84 функцията **reverse()** използва рекурсия за показване на екрана на своя низов аргумент в обратен ред.

```
using namespace std; // nashata 84-ta programa
#include <iostream> // e primer za rekursivna funkcia
void reverse(char *s);
int main() {
	char str[] = "Tova e primer";
	reverse(str);
	return 0;
```

```
}
void reverse(char *s) {
//     cout << *s << endl; // pri debugvane
       if(*s) {
              reverse(s+1);
       }
       else {
              return;
       }
       cout << *s ; // cout << *s << "\n";
}
```

Функцията **reverse()** проверява, дали като аргумент ѝ е предаден указател към нула (която нула е нулевият символ за край на низ). Ако не е, то функцията вика себе си с указател към следващия символ в низа. Тоя "завиващ се" процес се повтаря докато към функцията не се предаде указател към нула. Когато накрая се открие символът за край на низа започва процесът на "развиване". Т.е. след като завърши текущото извикване на функцията се преминава към завършване на предпоследното

викане на функцията, т.е. до изпълнение на инструкцията **cout << *s ;** където като аргумент участва последният символ. След това се преминава към завършване на предпредпоследното извикване и т.н. докато завърши първото извикване на функцията с аргумент първия символ в низа. В резултат началният низ посимволно се извежда на екран в обратен ред.

Създаването на рекурсивни функции често предизвиква трудности при начинаещи програмисти, но с времето за много от тях използването на рекурсията става обичайна практика.

Край на лекция 7.