ХИМИКОТЕХНОЛОГИЧЕН И МЕТАЛУРГИЧЕН УНИВЕРСИТЕТ – СОФИЯ

ИНФОРМАТИКА

част първа

Основи на програмирането на C++

лектор: гл. ас. д-р Стефан М. Панов

Катедра “Информатика”

Лекция 4

Още основни елементи на езика C++

Константи (литерали)

Константите, наричани понякога литерали, са фиксирани стойности, които не могат да бъдат променяни от програмата. Всъщност, ние вече ги използвахме в предишните програми.

Константите могат да имат който и да е от основните типове данни. Начинът на представяне на една константа зависи от нейния тип. Символните константи се поставят в единични кавички (наричани и **апострофи**). Например, 'd' и '%' са символни литерали. Ако е необходимо да присвоим символ на променлива (в примера с име **ch**) от типа **char** се използва инструкция, подобно на: **ch = 'Z'**;

Типът **wchar_t** няма да го разглеждане.

Целочислените константи се задават като числа без дробна част. Например: **18** и **-356**. **Реалните константи трябва да съдържат десетична**

точка, последвана от дробната част на числото, например **77.123**. За реални константи може да използваме и експоненциалното представяне на числата (т.е. представяне с плаваща запетая (точка), наричано и представяне с мантиса и порядък). Например: **6.273E+8**

Когато имаме константа, на която не сме указали явно типа компилаторът се стреми да определи типа по определени правила. (**Но ако не сме сигурни какво ще бъде решението му можем да зададем типа явно.**) Например, ако сме задали литерал цяло число и то влиза в обхвата на типа **int**, тогава компилаторът ще присвои тип **int**. Ако е извън обхвата, но влиза в обхвата на типа **long int** ще избере **long int**. Ако обаче числото е огромно и пак е извън обхвата и компилатора поддържа типа **long long int**, ще избере него. Принципът при целите числа става ясен: числото трябва да влиза в обхвата на избрания тип и се избира най-малкия по заемана памет тип данни.

При константите реални числа обаче правилото е друго: те се считат за тип **double, т.е** . т.е. посоченото вече число е **77.123** е **double.**

Ние можем да зададем точния тип на една числова константа със съответната наставка (суфикс). Правилата са:

- Ако **реалното число** завършва с буква **F** ще бъде от типа **float**, но ако буквата е **L** се указва тип **long double**.
- За **цели числа** суфиксът **U** означава използване на модификатора на тип **unsigned**, а суфиксът **L** на модификатора **long**. (Ако се налага да използваме и двата модификатора, т.е. **unsigned long** е необходимо да укажем и двата суфикса **U** и **L**.) Следват няколко примера в таблица 4.1.

Тип на данни	Примерни константи
int	1, 567, 55000, -789
long int	75000L, -14L
unsigned int	20000U, 987U
unsigned long	72323UL, 430000UL
float	623.12F, 5.34e-4F
double	77.23, 823123.345, -12.678
long double	10008.212345L

Таблица 4.1 – примери за константи с явно зададен тип

Шестнадесетични и осмични константи

Понякога е удобно вместо десетична бройна система (БС) да използваме осмична или шестнадесетична. Вече знаем, че в осмичната бройна система основата е числото 8, а за запис на числата се употребяват цифрите от 0 до 7.

Примерно, в осмичната БС числото 10 има същата стойност, както числото 8 в десетичната.

Шестнадесетичната БС използва цифрите от 0 до 9 плюс буквите от A до F, означаващи шестнадесетичните "цифри" 10, 11, 12, 13, 14 и 15. Например, 10_{16} е равно на 16_{10}. Доколкото тези две бройни системи се използват в програмите доста често, в езика C++ е разрешено да се задават **целочислени** литерали не в десетична, а в осмична или шестнадесетична система. **Шестнадесетичният литерал трябва да започва с префикса 0x (нула и латинска буква x, малка или главна) а осмичният само с нула.** Примери:

```
int hex = 0xFF; // 255 в десетична БС

int oct = 013; // 11 в десетична БС
```

Низови константи

Езикът C++ поддържа още един вграден тип константи, наречен **низ** (на английски **string**). **Низът е набор от символи, затворен в двойни кавички**, например **"tova e tekst"**. Ние вече използвахме низове в нашите програми в **cout**-инструкциите, с помощта на които извеждахме текст на екрана. Това, на което се налага да обърнем внимание е, че C++ позволява да дефинираме низове, но въпреки това няма вграден основен тип данни за низ. Низовете в C++, както ще се убедим по-късно, се поддържат във вид на символни масиви. (Освен това, C++ поддържа низов тип и с помощта на библиотечния клас **string**, но ние в нашия курс няма да стигнем до класове.)

Заб. Още веднъж се напомня да се прави разлика между символ и низ. **Зависи какви кавички използваме!** Например **'d'** е символ, но **"d"** е низ.

Спецификаторът const и именовани константи

Спецификаторът **const** (наричан и **квалификатор**) винаги предшества типа при обявяване на променлива. Той се използва основно за създаване на **именовани константи**. Това е много удобно когато една и съща стойност трябва да се използва на много места в програмата. Ако бихме използвали числова константа, да кажем числото 20 и тя се среща примерно 50 пъти в една голяма програма, евентуалната промяна на тоя константа изисква да променим кода на 50 места. При именованата константа промяната е само една, там където е обявена константата.

Пример: **const int Temp_v_gradusi = 20**;

Спецификаторът **const** има и по-специална употреба. Примерът, който дадохме може да се тълкува и така – променливата **Temp_v_gradusi** се инициализира с цялото число 20 и чрез добавянето на **const** тая стойност не

може да се модифицира оттам насетне при изпълнение на нашата програма. Но паметта, която е заделена за **Temp_v_gradusi** (да си припомним какво е дефиниция на променлива!) би могла да се промени от външни устройства, примерно датчик. Обявявайки променливата **Temp_v_gradusi** с помощта на спецификатора **const**, **можем да докажем**, че произволни регистрирани изменения на променливата са били предизвикани изключително от външни събития.

Управляващи символни последователности

Наричат ги и escape-последователности. **Въпреки, че се състоят от два символа, те се възприемат като един символ**. **Първият символ е винаги обратната наклонена черта \.**

Вече знаем, че с помощта на символните константи (винаги затворени в единични кавички) може да изведем нужния печатен символ. Но в ASCII

Управляваща последователност	Описание
\a	Издава звуков сигнал
\b	Връща курсора една позиция назад
\f	Премества курсора на нова страница
\n	Премества курсора на нов ред
\r	Връща курсора в началото на реда
\t	Хоризонтална табулация
\v	Вертикална табулация
\\	Обратно наклонена черта
\?	Въпросителен знак
\'	Апостроф
\"	Кавички
\xHH	Символ, зададен чрез 16-ично число
\OOO	Символ, зададен чрез осмично

Таблица 4.2 - управляващите символни последователности

таблицата има и управляващи символи, които нямат печатен образ, примерно символът за отиване на нов ред. Освен това в езика C++ някои символи (като единичните и двойни кавички) имат специално предназначение и не могат да се използват непосредствено. Това са двете основни причини, поради които в C++ са въведени управляващите символни последователности, показани на табл. 4.2. От практическа гледна точна за нас са по-важни последователностите в лилаво. Таблица 4.2 се нуждае от пояснения:

- **Всички показани последователности при използване трябва да се поставят вътре в някой символен низ**;

- Последните два реда в червено показват как може да вмъкнем в един низ **произволен** символ от ASCII таблицата.

Примерно следната инструкция ще отпечата 3 пъти символа въпросителен знак, т.е на екрана ще видим **? ? ?**

cout <<"? \x3F \077\n";

Това е така защото ASCII кодът на въпросителния знак е 63, а 63 в 16-ична БС е 3F. В осмична БС 63 е 77. Показаното в таблицата **\xHH** означава, че трябва да въведем две 16-ични цифри след \x т.е. **H** идва от **hexadecimal (16-ично).** Когато ASCII кодът на символа е до 15, т.е. може да се зададе и само с една 16-ична цифра, т.е. верни са и двата варианта - \x0A и \xA.

Същият принцип важи когато искаме да зададем символ с осмично число, т.е. **O** в **\OOO** от таблицата означава **octal**, т.е. осмична цифра. Пишем толкова цифри, колкото са необходими за осмичното представяне на съответния символ, но можем и да добавяме незначещи нули, както в горния пример за числото 63_{10} за извеждане на въпросителния знак.

Тъй като задаването на символ чрез шестнадесетично или осмично число е по-общо, тоя начин на задаване може да се прилага вместо всяка от показаните 11 последователности. Примерно вместо **"\n"** за преминаване на нов ред може да използваме **"\x0A"** или **"\012"**. Аналогично вместо **"\""** може да използваме **"\x22"** или **"\042"** (и трите варианта печатят една двойна кавичка в низ) и т.н. (**ако има време да се покаже програма_7a**)

Оператори

В C++ е дефиниран широк набор от вградени оператори, даващи възможност на програмиста лесно да създаде и пресметне най-разнообразни изрази. **Оператор (operator) — това е символ, който указва на компилатора да изпълни конкретни математически действия или логически манипулации**. В C++ има 4 общи класа оператори: аритметични, побитови, логически и оператори за отношения. Освен тях са определени и

други оператори със специално предназначение. Ще разгледаме 3 класа оператори, без побитови.

Аритметични оператори

В табл. 4.3 са изброени аритметичните оператори в C++. Действието на

Оператор	Действие
+	събиране
-	изваждане или унарен минус
*	умножение
/	деление
%	деление по модул
--	намаляване с 1 (декремент)
++	увеличаване с 1 (инкремент)

Таблица 4.3 – аритметични оператори

операторите +, -, * и / е същото, както в алгебрата, те могат да се прилагат за данни от **всеки** основен **числов тип**. Например, след прилагане на оператора деление (/) към цяло число остатъкът се изоставя. Конкретен пример: резултатът от целочисленото деление 13/5 ще е равен на 2. Остатъкът от деление се получава с помощта на оператора деление по модул (%). Например, 13 % 5 е равно на 3. Но тоя оператор не се прилага към реални числа (т.е. числа от тип float или double). **Операторите се прилагат върху операнди**. В дадения пример операндите са числата 13 и 3. Има оператори които се прилагат винаги върху 2 операнда (такива са *,/,% и +), други винаги са върху един операнд (-- и ++). Операторът **"-"** е по-различен. Говорим за „**изваждане**“, когато той се прилага върху два операнда и за „**унарен минус**“, когато се прилага върху един операнд. Унарният минус по същество представлява умножение на стойността на единствения операнд по -1. **Т.е, едно число, пред което има знак "-" мени своя знак на противоположния.**

Инкремент и декремент

Операторът инкремент изпълнява събиране на операнда с числото 1, а операторът декремент изважда 1 от своя операнд. Това значи, че инструкцията **x = x + 1 ;** е аналогична на **++x ;** Също така инструкцията **x = x - 1 ;** е аналогична на **--x ;** И двата оператора се срещат в по две разновидности: говорим за **префиксна форма**, когато стоят пред своя операнд (**++x** и **--x**), и **постфиксна форма** когато са след него него (**x++** и **x--**). Коя форма ще се избере е от значение, когато инкрементът или декрементът се среща като част от по-сложен израз, но това ще се разгледа по-късно при указателите. Сега само ще се подчертае, че повечето C++ компилатори създават по-ефективен код за операциите инкремент и декремент в сравнение с кода, генериран при използването на обикновения оператор събиране или изваждане на единица. Затова се предпочита да се използват операторите инкремент и декремент, където това е възможно.

Приоритет при аритметичните оператори

Аритметичните оператори се подчиняват на определен ред при изпълнение на действията, който се нарича **приоритет** и е показан на таблица 4.4.

Приоритет	Оператори
Най-висок	++ --
	- (унарен минус)
	* / %
Най-нисък	+ -

Таблица 4.4 – приоритет при аритметичните оператори

Операторите имащи еднакъв приоритет се изчисляват от компилатора отляво надясно. За промяна на реда на изчисление може да се използват кръгли скоби. **Една или повече операции, затворени в кръгли скоби, придобиват по-висок приоритет в сравнение с другите операции в израза**.

Оператори за отношения и логически оператори

Операторите за отношения и логическите (булевите) оператори, показани на таблица 4.5 често вървят “ръка за ръка” и се използват за получаване на резултати във вид на стойности **ИСТИНА/ЛЪЖА** (**true/false**). Накратко, операторите за отношения оценяват по “двубална система” отношенията между двете стойности, а логическите определят различни начини на съчетание на стойностите **ИСТИНА** и **ЛЪЖА**.

Обърнете внимание на това, че в езика **C++** в качеството на оператор за отношение “не равно” се употребява “!=”, а за оператора “равно” — двойния символ за равенство “==“. Единичния символ за равенство “=“ е запазен за инструкцията за присвояване.

Резултатът от изпълнението на операторите за отношения и логическите оператори има тип bool със стойности true или false.

Оператори за отношения	Значение
==	Равно
!=	Не равно
>	По-голямо
<	По-малко
>=	По-голямо или равно
<=	По-малко или равно
Логически оператори	Значение
&&	И
II	ИЛИ
!	НЕ

Таблица 4.5 – Оператори за отношения и логически оператори

Операндите, участващи в операциите за отношение, могат да бъдат практически от всеки тип, главното е да могат да се сравняват. **От своя страна операндите на логическите оператори трябва да са от тип bool и резултатът от една логическа операция също винаги ще има тип bool**.
Но доколкото в C++ което и да е ненулево число се оценява като истина (true), а нулата е еквивалентна на лъжа (false), то логическите оператори могат да се използват във всеки израз, който дава нулев или ненулев резултат.

Заб. Таблиците на истинност при извършване на операциите конюнкция, дизюнкция и отрицание с използването съответно на логическите оператори &&, || и ! са разгледани в нашата първа лекция.

Независимо от това, че C++ не притежава вграден логически оператор "изключващо ИЛИ" (XOR), ние може да напишем логически израз, който да

изпълнява липсващата операция. Убедете се, че изразът (a || b) && !(a && b) изпълнява операцията “изключващо ИЛИ”(a и b са логически променливи).

В следващата програма са показани резултатите от прилагането на операциите конюнкция, дизюнкция и изключващо ИЛИ (сума по модул 2).

```
#include <iostream> // nashata sedma=b programa
using namespace std;
int main()
{
	bool a,b;
	cout << "Vavedete a (0 ili 1): ";
	cin >>a;
	cout << "Vavedete b (0 ili 1): ";
	cin >>b;
	cout << "a AND b: " << (a && b) << "\n";
	cout << "a OR  b: " << (a || b) << "\n";
```

```
    cout << "a XOR b: " << ((a || b) && !(a && b))<< "\n";
  return 0;
}
```

Новото в тая програма е употребата на още един вграден идентификатор, **cin**. Той е съставен от частите на думите **console input** и в повечето случаи **означава въвеждане на данни от клавиатура**. В качеството на оператор за въвеждане се използва **“>>“**.

При изпълнение на тая инструкция стойността, въведена от потребителя (която в дадения пример трябва да бъде 0 или 1), се съхранява в променливата, посочена от дясната страна на оператора.

Заб. Конзола в програмирането е комбинация от монитор и клавиатура (или друго входно устройство).

Обърнете внимание: въпреки, че променливите **a** и **b** в показаната програма са от типа **bool**, ние въвеждаме целочислени стойности (0 или 1). В това

няма нищо странно, доколкото C++ автоматично преобразува числото 1 в **true**, а 0 — във **false** за булеви променливи. И обратно, при извеждане на екрана на bool-стойност, тя автоматично се преобразува в числото 1, ако е **true** и в 0, ако е **false**.

Както операторите за отношения, така и логическите оператори имат по-нисък приоритет в сравнение с аритметичните оператори. Това означава, че изразът 10 > 1+12 ще се изчисли така, както както все едно той е бил записан като: 10 >(1 + 12)

Резултатът, разбира се, ще бъде равен на ЛЪЖА. Освен това, да погледнем още един път следната инструкция от предишната програма:

```
cout << "a AND b: " << (a && b) << "\n";
```

Без кръгли скоби, в които е затворен изразът a && b тук не може да се мине, защото оператора && има по-нисък приоритет от оператора за извеждане

на данни “<<“. Същите разсъждения са в сила и за следващите 2 реда от програмата.

Приоритетът на операторите за отношения и логическите оператори е показан на следващата таблица 4.6.

Приоритет	Значение
Най-висок	!
	> >= < <=
	== !=
	&&
Най-нисък	||

Таблица 4.6 – приоритет при операторите за отношения и логическите оператори

С помощта на логически оператори могат да се обединят в един израз произволно количество операции за отношения.

Например, в този израз са обединени три операции за отношения:

var > 17 || !(10 < count) && 9 <= item

Изрази и неявни преобразувания на типове

Израз е комбинация от операнди и оператори. Операндите могат да бъдат променливи, константи и функции връщащи резултат. За промяна на приоритет в един израз могат да се използват и кръгли скоби. **В най-простия случай израз може да бъде само променлива или само константа. Всеки израз има тип. Това е типът на стойността, получена след изчисляване на израза. В C++ е допустимо в един израз да участват операнди от различни типове.** В тоя случай компилаторът преобразува **автоматично** типовете на

всички операнди до един общ тип по определени правила по време на изчисление на израза. Да разгледаме подробно това неявно преобразуване:

Първо, всички **char** и **short int** стойности автоматично се преобразуват към типа **int**. Тоя процес се нарича **целочислено разширение** (**integral promotion**). Второ, всички операнди се преобразуват към типа на най-големия операнд. Този пък процес се нарича **разширение на типа** (**type promotion**), при това той се изпълнява пооперационно. Например, ако единият операнд има тип **int**, а другият — **long int**, то типът **int** се разширява до тип **long int**. Или, ако поне един от операндите има тип **double**, всеки останал друг операнд се привежда до тип **double**. Това означава, че такива преобразувания, като от тип **char** в тип **double**, са напълно допустими. След преобразуването и двата операнда ще имат един и същи тип, такъв ще бъде и типът на резултата след операцията. Да разгледаме, например,

преобразуването на типовете, схематично представено на рис. 4.1. Отначало

char ch; int i; float f; double d;

result = (ch / i) + (f * d) - (i * f);

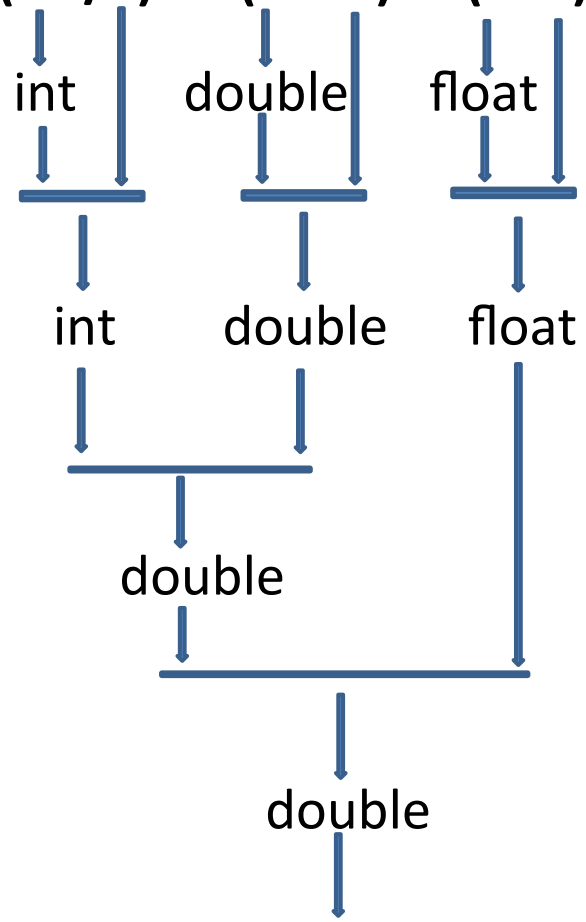

Рис. 4.1. Пример за преобразуване на типове в C+ +

символът **ch** се подлага на процес на “разширение” на типа и се преобразува в стойност от тип **int**. После резултатът от операцията **ch/i** се привежда към тип **double**, понеже резултатът от произведението **f*d** има тип double. Резултатът от целия израз ще получи също тип **double**, защото към момента на последното изчисление и двата операнда имат тип **double**.

За типът **bool**: Както вече се каза по-рано, **стойности от тип bool** автоматично се преобразуват **в цели числа 0 или 1** при използване в израз от целочислен тип. При обратното преобразуване на целочислен резултат в тип bool **нулата се преобразува във false, а ненулевите стойности - в true**.

Явно преобразуване на типове

Езикът С поддържа операция за явно (изрично) преобразуване на един тип в друг. Тя има следния синтаксис:

(тип) операнд където

тип указва към какъв тип трябва да се преобразува операнд.

операнд

може да бъде променлива или константа от всеки един от базовите типове.

Явното преобразуването на типове се разглежда като унарен оператор и поради това той има същия приоритет, както и другите унарни оператори.

Понякога операцията **явно преобразуване на тип** се оказва много полезна. Например, в следващата програма с реализация на цикъл се използва една

целочислена променлива, участваща в израз, резултатът от изчисляването на който трябва да се получи с дробна част.

```
#include <iostream> // nashata osma programa
using namespace std;
int main() {   // Izvejdame k i  stoinostta k/2  s drobna chast.
      int k;
      for (k=1; k<=50; k++ )
            cout << k << "/ 2 e ravno na: " << (float) k / 2 << '\n';
      return 0;
}
```

Изпреварващо показваме инструкция за цикъл for, в нея **k** се мени от 1 до 50.

Без операцията **явно преобразуване на тип** в програмата би имало само целочислено деление. Преобразуването в случая гарантира, че на екрана ще се изобрази и дробната част на резултата.

Използване на интервал и кръгли скоби

Всеки израз в C++ може да включва интервали (или символи за табулация) за по-голяма прегледност на кода. Например, следващите два израза са напълно еднакви, но вторият се чете по-леко:

x=10/y*(127/x);

x = 10 / y * (127/x) ;

Кръглите скоби (също както в алгебрата) повишават приоритета на операциите, съдържащи се вътре в тях. Използването на допълнителни кръгли скоби не води до грешки или забавяне на изчислението на израза. С други думи, от тях няма никаква вреда, но често полза! Те помагат да изясните (в началото на вас, а после и на другите, които ще разучават вашия код) точния ред на изчисленията. Кой от двата следващи израза е по-лесен за разбиране? **x = y/3-34*temp+127;** **x = (y/3) - (34*temp) + 127;**