ШАГ
компьютерная
АКАДЕМИЯ

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C++

Урок № 2

Условия

Содержание

Материалы урока прикреплены к данному PDF-файлу. Для доступа к материалам, урок необходимо открыть в программе Adobe Acrobat Reader.

1. Понятие оператора

В прошлом уроке вы познакомились с понятием переменная и тип данных. Кроме того, в примерах урока, а также домашнем задании мы с вами производили над переменными определенные действия, то есть оперировали данными. Вполне очевидно, что слова оператор и оперировать имеют одинаковое происхождение, следовательно, согласно простой логике:

Оператор — конструкция языка позволяющая производить различные действия над данными, приводящие к определенному результату.

Все операторы принято подразделять на группы по признаку их действия. Например, арифметические операции — операции, позволяющие производить арифметические действия над данными (сложение, вычитание и так далее). Обо всех подобных группах представленных в языке С, мы будем рассказывать в дальнейшем. На данный момент, следует обсудить более масштабную классификацию всех операторов, принятую вне зависимости от их влияния на содержимое переменных. Итак, все операторы делятся на:

1. **Унарные** — операторы, которым необходим, только один операнд (данные, над которыми производится действие). С примером унарного оператора вы уже знакомы из курса школьной математики — унарный минус, который позволяет превратить число в отрицательное (3 и –3), или

положительное (–(–3)). Т.е. общий синтаксис унарного оператора таков:

```
оператор операнд;     или    операнд оператор;
```

2. **Бинарные** — операторы, которым необходимо два операнда слева и справа от оператора. Таких операторов вы знаете множество — это +, –, * и т.д. И их общий синтаксис можно изобразить следующим образом:

```
операнд оператор операнд;
```

3. **Тернарные** — операторы, которым необходимо три операнда. В языке программирования C++ такой оператор всего один и с его синтаксисом мы познакомимся чуть позже.

Приоритет

Все операторы имеют приоритет. Ниже приведены операторы в соответствии с приоритетами. Более углубленно мы познакомимся с некоторыми в сегодняшнем уроке, другие узнаем в процессе дальнейшего обучения. Естественно, в данной таблице представлены не все операторы языка, а пока что наиболее актуальные для нас.

Таблица 1

Символьное обозначение операции	
Высший приоритет	Высший приоритет
() [] . ->	^
! *(ун) – (ун) ~ ++ ––	|
% * /	&&
+ –	||
<< >>	?:

Символьное обозначение операции	
Высший приоритет	Высший приоритет
< > <= >=	= += −= *= /= %= &= |= ^= >>= <<=
!= ==	
&	Низший приоритет

Теперь, когда фундамент знаний в области операторов заложен, вы можете переходить к более детальному изучению последних, а именно к следующему разделу урока.

2. Арифметические операции с числами

Хорошо забытое старое…

Итак, приступим. Как уже было отмечено ранее — арифметические операции — это операции, позволяющие производить арифметические действия над данными. Большинство из них вам знакомы с детства и, тем не менее, давайте, систематизируем наши знания с помощью таблицы представленной ниже.

Таблица 2.

Название операции	Символ, для обозначения в языке С	Краткое описание. Пример.
Сложение	+	Складывает два значения вместе, результатом является сумма операндов: 5+18 результат 23.
Вычитание	–	Вычитает значение, находящееся справа из значения, находящегося слева от оператора. Результат — разность операндов: 20–15 результат 5.
Умножение	*	Перемножает два значения, результатом является произведение операндов: 5*10 результат 50.
Деление	/	Делит значение, находящееся слева на значение, находящееся справа от оператора. Например: 20/4 результат 5.
Деление по модулю	%	Результатом этой операции является остаток от целочисленного деления, например, если мы делим 11 на 3, то целых частей у нас получается 3, (так как 3*3=9), в остатке будет 2, это число и будет результатом деления по модулю: 11/3 3 целых 2 в остатке 11%3 = 2 (остаток).

Примечание:

1. Операцию деления по модулю, можно применять только к целочисленным данным. Попытки нарушить данное правило приведут к ошибке на этапе компиляции.

2. Если меньшее число делится на большее с помощью %, то результатом будет само меньшее число. 3%10 = 3.

3. Делить по модулю на нуль нельзя, это приведет к некорректной работе программы на этапе выполнения.

Инкремент и декремент

Все вышеописанные операции, являлись бинарными, однако существуют еще и унарные арифметические операции, таких операций в школьном курсе нет, хотя на самом деле они очень просты:

1. **Инкремент** — обозначается конструкцией ++. Данный оператор увеличивает содержимое любой переменной на единицу и перезаписывает значение переменной. Например:

```
int a=8;
cout<<a; // на экране число 8
a++;
cout<<a; // на экране число 9
```

2. **Декремент** — обозначается конструкцией --. Данный оператор уменьшает содержимое любой переменной на единицу и перезаписывает значение переменной.

Например:

```
int a=8;
cout<<a; // на экране число 8
a--;
cout<<a; // на экране число 7
```

Достаточно просто, не правда ли?! Такие выражения могут быть представлены и так: **a=a+1** или **a=a–1**. Следует отметить, что для литералов ни инкремент, ни декремент не используются, т.к. совершенно не логично поступать следующим образом **5=5+1**. Это явная ошибка. Однако на этом мы не закончим знакомство с инкрементом и декрементом. В прошлом разделе урока мы выяснили, что синтаксис унарного оператора, может быть не только таким,

```
операнд оператор;
```

но и таким

```
оператор операнд;
```

Такие формы записи носят название постфиксной, (оператор располагается после значения) и префиксной (оператор располагается до значения). И инкремент, и декремент обладают обеими формами. Давайте разберемся, какие есть различия между формами, и в каких случаях эти различия имеют значение.

Пример 1

```
int a=8;
cout<<a; // на экране число 8
a++;
```

```
cout<<a; // на экране число 9
++a;
cout<<a; // на экране число 10
```

В данном примере нет никакой разницы, между префиксной и постфиксной формой. И в первом и во втором случае значение переменной a просто увеличивается на единицу.

Смысл использования различных форм оператора появляется только тогда, когда в строке кроме самого оператора, есть еще какая-нибудь команда.

Пример 2

```
int a=8;
cout<<++a; // на экране число 9
cout<<a++; // на экране число 9
cout<<a;   // на экране число 10
```

Прежде чем разбирать пример, давайте установим три правила:

1. Принцип выполнения команд в языке C++ неоднозначен. Поэтому ниже приводится таблица направления действия некоторых операторов.
2. Если кроме постфиксной формы инкремента или декремента, в строке есть еще какая-либо команда, то сначала выполняется эта команда, и только потом инкремент или декремент независимо от расположения команд в строке.
3. Если кроме префиксной формы инкремента или декремента, в строке есть еще какая-либо команда, то все

команды в строке выполняются справа налево согласно приоритету операторов.

Таблица 3

ОПЕРАТОР	НАПРАВЛЕНИЕ
() [] . ->	СЛЕВА НАПРАВО
* / % + –	
<< >> & ^ |	
<< = >> = ==! =	
&& ||	
Унарный – унарный + ! ++ ––	СПРАВА НАЛЕВО
?:	
+ = – = / = * = % = & = ^ = | =	

Теперь более подробно о примере:

- Изначально значение переменной равно числу 8.
- Команда **cout<<++a**; содержит префиксную форму оператора инкремент, следовательно, используя третье правило, описанное выше, мы сначала увеличиваем значение переменной a на единицу, а затем показываем его на экран с помощью команды **cout<<**.
- Команда **cout<<a++**; содержит постфиксную форму оператора инкремент, следовательно, используя второе правило, описанное выше, мы сначала показываем значение переменной (всё еще 9) на экран с помощью команды **cout<<**, а затем увеличиваем значение переменной **a** на единицу.
- При выполнении команды **cout<<a**; будет показано уже измененное (увеличенное) значение, то есть число **10**.

Исходя из предыдущих тем данного раздела урока, мы с вами теперь знаем, как упростить неудобную

и «некрасивую» запись типа **x=x+1** или **x=x–1**, превратив её в **x++**, или **x––**. Но таким образом, мы можем увеличивать и уменьшать значение переменной лишь на единицу, а как быть с любым другим числом? Например, как упростить запись:

X=X+12.

В данном случае, тоже есть простое решение — использовать так называемые комбинированные операторы или сокращенные арифметические формы. Выглядят они следующим образом:

Таблица 4

Название формы	Комбинация	Стандартная запись	Сокращенная запись
Присваивание с умножением	*=	A=A*N	A*=N
Присваивание с делением	/=	A=A/N	A/=N
Присваивание с делением по модулю	%=	A=A%N	A%=N
Присваивание с вычитанием	– =	A=A–N	A– =N
Присваивание со сложением	+=	A=A+N	A+=N

Мы рекомендуем вам в дальнейшем пользоваться сокращенными формами, так как это не только является хорошим тоном в программировании, но и значительно повышает читабельность программного кода. Кроме того, в некоторых источниках упоминается о том, что сокращенные формы обрабатываются компьютером

быстрее, повышая скорость выполнения программы. Теперь самое время убедиться во всем вышесказанном на практике, потому что, как говорится, лучше один раз увидеть, чем сто раз услышать. Вы уже умеете создавать проекты и добавлять в них файлы, собственно именно это от вас сейчас и требуется. Далее представлено несколько программ, которые вам необходимо набрать, что бы увидеть применение арифметических операции на практике. Начнем с проекта под названием *Game*.

3. Применение арифметических операций

Пример №1. Игра

```
// примитивная игра для малышей
#include <iostream>
using namespace std;
int main()
{
    int buddies; // количество пиратов до битвы
    int afterBattle; // количество пиратов после битвы

    // Вы пират. Сколько человек в вашей команде,
    // если не считать вас?
    cout<<"You the pirate. How many the person in "
          "your command, without you?\n\n";
    cin>>buddies;

    // Внезапно на вас нападает 10 мушкетеров
    cout<<"Suddenly you are attacked by 10 "
          "musketeers \n\n";

    // 10 мушкетеров и 10 пиратов погибают в схватке.
    cout<<"10 musketeers and 10 pirates perish in "
          "fight.\n\n";

    // подсчет оставшихся в живых
    afterBattle=1+buddies-10;

    // Осталось лишь ... пиратов
    cout<<"Remains only "<<afterBattle<<" pirates\n\n";

    // Состояние убитых насчитывает 107 золотых монет
```

```
    cout<<"The condition killed totals 107 gold "
          "coins \n\n";

    // Это по ...  монет на каждого
    cout<<"It on "<<(107/afterBattle)<<"coins on "
          "everyone";

    // Пираты устраивают большую драку из-за оставшихся
    cout<<"Pirates arrange greater fight because "
          "of remained\n\n";

    // ... монет
    cout<<(107%afterBattle)<<"coins \n\n";

    return 0;
}
```

В данном примере используется правило деления целого на целое. При таком делении дробная часть, даже если должна быть, обрезается. Более подробно об этом будет рассказано в разделе урока — «Преобразование типов». В выражении (**107/afterBattle**) — мы узнаем сколько монет получит каждый пират, если разделить их поровну. Кроме того, оператор деления по модулю, помогает нам выяснить, сколько останется монет, которые невозможно разделить, то есть мы получим остаток от деления 107 на количество выживших пиратов. Вот и все особенности примера.

Пример №2. Окружность

В данном примере будет продемонстрировано использование арифметических операторов в программах, производящих математические вычисления.

Мы убедимся, что знание арифметических операторов дает возможность решать простые задачи. Однако, мало уметь использовать операторы, необходимо понимать каков будет результат их использования. Об этом и пойдет речь в следующем разделе.

```
// программа для выяснения параметров окружности
#include <iostream>
using namespace std;

int main()
{
    const float PI=3.141592;// обозначение константы -
                            // числа пи

    // объявление переменных для хранения параметров
    float radius, circumference, area;

    // приглашение ввести радиус
    cout<<"Welcome to program of work with rounds\n\n";
    cout<<"Put the radius from rounds\n\n";
    cin>>radius;
    cout<<"\n\n";

    area=PI*radius*radius; // подсчет площади круга
    circumference=PI*(radius*2); // подсчет длины
                                 // окружности

    // вывод результатов
    cout<<"Square of round: "<<area<<"\n\n";
    cout<<"length of round: "<<circumference<<"\n\n";
    cout<<"THANKS!!!  BYE!!!\n\n";

    return 0;
}
```

4. Преобразование типов

Когда, мы что-либо делаем, нам, несомненно, важно знать каков будет результат. Вполне очевидно, что из тех ингредиентов из которых, скажем, варится суп харчо, вряд ли можно приготовить торт со взбитыми сливками. Следовательно, результат напрямую зависит, от составных частей. То же самое происходит с переменными. Если, скажем, складывается два числа типа **int**, вполне понятно, что результат так же будет иметь тип **int**. А вот как быть, если данные имеют разные типы? Именно об этом мы и поговорим в текущем разделе данного урока.

Итак, прежде всего, давайте разберемся с тем, как типы данных взаимодействуют друг с другом. Существует так называемая иерархия типов, где все типы размещены по старшинству. Для того, что бы разбираться в преобразовании типов, необходимо всегда помнить порядок типов этой иерархии.

```
bool, char, short-int-unsigned int-long-unsigned
long-float-double-long double
```

Несмотря на то, что некоторые типы имеют одинаковый размер, в них помещается разный диапазон значений, например, **unsigned int** в отличие от **int** может поместить себя в два раза больше положительных значений, и потому является старше по иерархии, при этом оба типа имеют размер 4 байта. Кроме того, следует отметить, очень важную особенность, отраженную здесь, если в преобразовании

типов участвуют такие типы, как **bool**, **char**, **short**, они автоматически преобразовываются к типу **int**.

Теперь, давайте рассмотрим, различные классификации преобразований.

Классификация по диапазону содержащихся значений

Все преобразования можно разделить на две группы относительно местоположения в иерархии типов участвующих в преобразовании.

1. Сужающее преобразование — при таком преобразовании — больший тип данных в иерархии преобразуется к меньшему типу, безусловно, в этом случае может произойти потеря данных, поэтому с сужающим преобразованием, следует быть осторожными. Например:

```
int A=23.5;
cout<<A; // на экране 23
```

2. Расширяющее преобразование. Данный вид преобразования, ведет к так называемому расширению типа данных от меньшего диапазона значений к большему диапазону. В качестве примера предлагается такая ситуация.

```
unsigned int a=3000000000;
cout<<a; // на экране 3000000000
```

В данном случае **3000000000** — это литерал типа **int**, который благополучно расширяется до **unsigned int**, что и позволяет нам увидеть на экране именно **3000000000**,

а не что-то другое. Тогда, как в обычный **int** такое число не поместиться.

Классификация по способу осуществления преобразования

Вне зависимости от направления преобразования, оно может быть осуществлено одним из двух способов.

1. Неявное преобразование. Все вышеописанные примеры относились к этому типу преобразования. Такой вид преобразования также называют автоматическим, т.к. оно происходит автоматически без вмешательства программиста, другими словами, мы ничего не делаем для того, что бы оно произошло.

```
float A=23.5; - double стал  float без
                каких-либо дополнительных действий
```

2. Явное преобразование. (второе название — приведение типов). В данном случае, преобразование производится программистом, тогда, когда это необходимо. Давайте рассмотрим простой пример такого действия:

```
double z=37.4;
float y=(int) z;
cout<<z<<"*** "<<y; // на экране 37.4 ***37
```

(int)z — есть явное сужающее преобразование от типа **double** к типу **int**. В этой же строке происходит расширяющее неявное преобразование от полученного типа **int** к типу **float**. Следует запомнить, что любое преобразование носит временный характер и действует только

в пределах текущей строки. То есть переменная **z** как была **double**, так и останется на протяжении всей программы, а ее преобразование в **int** носило временный характер.

Преобразование типов в выражении

И вот мы наконец-то подошли к тому, о чем говорили в самом начале данного раздела урока, как выяснить какого типа будет результат какого-то выражения. Давайте попробуем это вычислить, пользуясь полученными знаниями. Предположим у нас есть следующие переменные:

```
int I=27;
short S=2;
float F=22.3;
bool B=false;
```

Пользуясь этими переменными, мы собираемся составить такое выражение:

```
I-F+S*B
```

В переменную какого типа данных нам следует записать результат? Решить это просто, если представить выражение в виде типов данных:

```
int-float+short*bool
```

Напоминаем, что **short** и **bool** сразу же примут тип **int**, так что выражение будет выглядеть так:

```
int-float+int*int,  при этом false станет 0
```

Умножение **int** на **int** даст, несомненно, результат типа **int**. А вот сложение **float** с **int**, даст на выходе **float**, так как, здесь вступает в игру новое правило:

Если в каком-либо выражении используются разные типы данных, то результат, приводится к большему из этих типов.

Ну и, наконец — вычитание из **int** типа **float**, согласно только что упомянутому правилу снова даст **float**.

Таким образом, результат выражения будет иметь тип **float**.

```
float res= I-F+S*B; // 27-22.3+2*0
cout<<res; // на экране число 4.7
```

Теперь, когда вы знакомы с правилом, вам нет необходимости, детально разбирать выражение, достаточно просто найти наибольший тип, именно он и будет результирующим.

Примечание: *Будьте очень внимательны также и при сочетании переменных с одинаковыми типами данных. Например, помните, если целое делится на целое, то и получиться целое. То есть, int A=3; int B=2; cout<<A/B; // на экране 1, так как результат — int и дробная часть утеряна. cout<<(float) A/B; // на экране 1.5, так как результат — float.*

Пример, использующий преобразование типов

Теперь давайте закрепим знания на практике. Создадим проект и напишем нижеследующий код.

```
#include <iostream>
using namespace std;
int main(){
```

```
    // объявление переменных и запрос на ввод данных
    float digit;
    cout<<"Enter digit:";
    cin>>digit;
    /* Даже, если пользователь ввел число
       с вещественной частью, результат выражения
       запишется в int и вещественная часть будет
       утеряна, разделив число на 100 мы получим
       количество сотен в нем.*/
    int res=digit/100;
    cout<<res<<"hundred in you digit!!!\n\n";
    return 0;
}
```

Теперь, когда вы рассмотрели пример, вы, конечно же, убедились, что с помощью преобразования типов можно не просто организовать временный переход из одного типа в другой, но и решить простую логическую задачу. Следовательно, вы должны отнестись к этой теме с должным вниманием. Понимание преобразования в дальнейшем поможет вам не только решать интересные задачи, но избежать ненужных ошибок.

Унифицированная инициализация

В C++ 11 был добавлен механизм унифицированной инициализации, который позволяет задать значение различным программными конструкциям (переменным, массивам, объектам) единообразным способом. Рассмотрим на примере инициализации переменных.

```
int a = {11};  // В a записывается значение 11
int b{33};     // В b записывается значение 33
```

Для того, чтобы задать значения переменным мы используем {}. Как видно из примера это можно сделать двумя способами. Такая форма инициализации также называется **списковой инициализацией**.

Сужение и списковая инициализация

Что произойдет при выполнении кода?

```
int x = 2.88;
cout<<x; // на экране отобразится 2
```

Как вы уже знаете из изученного материала, в данном примере происходит неявное сужающее преобразование, так как мы присваиваем переменной **x** целого типа значение типа **double**. Однако, если использовать **списковую инициализацию** компилятор генерирует ошибку на этапе компиляции, так как эта форма инициализации защищает от сужения. Она не дает записать значение большего размера в тип, который не поддерживает такой диапазон значений.

```
int x = { 2.88 }; // ошибка на этапе компиляции.
                  // 2.88 — double, а x переменная
                  // целого типа

char ch = { 777 };// ошибка на этапе компиляции.
                  // 777 — int, а ch переменная
                  // символьного типа 777 не попадает
                  // в диапазон значений char
```

С другой стороны:

```
char ch2 = { 23 };  // всё правильно. 23 попадает
                    // в диапазон char
```

```
double x = { 333 }; // всё правильно 333 - int
                    // и попадает в диапазон double
```

Если вы хотите выявлять потенциальные проблемы с потерей данных на этапе компиляции вы можете использовать списковую инициализацию.

5. Логические операции

В программировании, зачастую, необходимо не только производить какие-то вычисления, но и сравнивать величины между собой. Для этого используются, так называемые логические операции. Результатом логических операций всегда является либо значение **true**, либо значение **false**, то есть истина или ложь. Логические операции делятся на три подгруппы:

1. Операторы сравнения;
2. Операторы равенства;
3. Логические операторы объединения и отрицательная инверсия.

Теперь давайте более детально разберем каждую группу операторов.

Операторы сравнения

Используются тогда, когда необходимо выяснить каким образом две величины относятся друг к другу.

Таблица 5

Символ, обозначающий оператор	Утверждение
<	Левый операнд менше чем правый операнд
>	Левый операнд больше чем правый операнд
<=	Левый операнд меньше или равен правому операнду
>=	Левый операнд больше или равен правому операнду

Смысл операций сравнения (второе название — операции отношений) состоит в том, что если утверждение, заданное с помощью оператора верно, выражение, в котором он участвует, заменится на значение **true**, если не верно — на значение **false**. Например:

```
cout<<(5>3); // на экране будет единица, так как
             // утверждение (5>3) истина.
cout<<(3<2); // на экране будет 0, так как (3<2) ложь.
```

> **Примечание:** *Вместо значений false и true на экран выводится 0 и 1, так как они эквивалентны значениям ложь и истина. В языке C++ в роли истины также может выступать любое другое число отличное от 1 и 0, как положительное, так и отрицательное.*

Операторы равенства

Используются для проверки на полное соответствие или несоответствие двух величин.

Таблица 6

Символ, обозначающий оператор	Утверждение
==	Левый операнд равен правому
!=	Левый операнд не равен правому

Применение этих операторов совпадает с принципом применения предыдущей группы, то есть, на выходе выражение заменяется либо на истину, либо на ложь, в зависимости от утверждения.

```
cout<<(5!=3); // на экране будет единица,
              // так как утверждение (5!=3) истина.
cout<<(3==2); // на экране будет 0, так как (3==2) ложь.
```

Логические операции объединения и отрицательная инверсия

В большинстве случаев невозможно обойтись только одним утверждением. Чаще всего необходимо комбинировать утверждения тем или иным образом. Например, чтобы проверить находится ли число в диапазоне от 1 до 10, необходимо проверить два утверждения: число должно одновременно >=1 и <=10. Для того чтобы реализовать такую комбинацию необходимо ввести дополнительные операторы.

Таблица 7

Операция	Название
&&	И
||	ИЛИ
!	НЕ

Логическое И (&&)

Логическое И объединяет вместе два утверждения и возвращает истину только в том случае, если и левое и правое утверждения истинны. Если хотя бы одно из утверждений или оба ложны, объединенное выражение заменяется на ложь. Логическое И работает по сокращенной схеме, то есть, если первое утверждение ложь, второе уже не проверяется.

Таблица 8

Утверждение 1	Утверждение 2	Утверждение1 && Утверждение 2
true	true	true
true	false	false
false	true	false
false	false	false

Теперь рассмотрим пример, в котором программа получает число и определяет, попадает ли это число в диапазон от 1 до 10.

```
#include <iostream>
using namespace std;

int  main()
{
    int N;
    cout<<"Enter digit:\n";
    cin>>N;
    cout<<((N>=1)&&(N<=10));
    cout<<"\n\nIf you see 1 your digit is in "
          "diapazone\n\n";
    cout<<"\n\nIf you see 0 your digit is not in "
          "diapazone\n\n";

    return 0;
}
```

В данном примере, если оба утверждения будут верными, на место выражения подставится 1, в противном случае — 0. Соответственно пользователь сможет проанализировать сложившуюся ситуацию, используя инструкции программы.

Логическое ИЛИ (||)

Логическое ИЛИ объединяет вместе два утверждения и возвращает истину только в том случае, если хотя бы одно из утверждений верно, и ложь в том случае, если оба утверждения не верны. Логическое ИЛИ работает по сокращенной схеме, то есть, если первое утверждение истина, второе уже не проверяется.

Таблица 9

Утверждение 1	Утверждение 2	Утверждение1 && Утверждение 2
true	true	true
true	false	true
false	true	true
false	false	false

Еще раз рассмотрим пример, в котором программа получает число и определяет, попадает ли это число в диапазон от 1 до 10. Только, теперь используем ИЛИ.

```
#include <iostream>
using namespace std;

int main()
{
    int N;
    cout<<"Enter digit:\n";
    cin>>N;
    cout<<((N<1)||(N>10));
    cout<<"\n\nIf you see 0 your digit is in "
          "diapazone\n\n";
    cout<<"\n\nIf you see 1 your digit is not in "
          "diapazone\n\n";
    return 0;
}
```

В данном примере если оба утверждения будут ложными, (то есть число будет не меньше 1 и не больше 10) на место выражения подставится 0, в противном случае — 1.

Соответственно пользователь, также как и в предыдущем примере, сможет проанализировать сложившуюся ситуацию и сделать вывод.

Логическое НЕ (!)

Логическое НЕ является унарным оператором и в связи с этим не может называться оператором объединения. Оно используется в том случае, если нужно изменить результат проверки утверждения на противоположный.

Таблица 10

Утверждение	! Утверждение
true	false
false	true

```
// на экране будет 1, так как (5==3) ложь и её
// инверсия это - истина.
cout<<!(5==3);
// на экране будет 0, так как (3!=2) истина и её
// инверсия это - ложь.
cout<<!(3!=2);
```

Логическое отрицание возвращает на место утверждения ложь, если последнее истинно, и наоборот, истину, если утверждение ложно. Данный оператор можно применить для сокращения постановки условия. Например, выражение

```
b==0
```

можно сокращенно записать с помощью инверсии:

```
!b
```

обе записи дают на выходе истину, в случае если **b** будет равно нулю.

В данном разделе мы рассмотрели всевозможные логические операции, которые позволяют определить истинность любого утверждения. Однако описанные здесь примеры являются неудобными для рядового пользователя, так как анализ результатов должен производить не он, а программа. Кроме того, если в зависимости от утверждения необходимо не просто выдавать на экран результат его проверки, а производить какое-либо действие, тут уже пользователь точно бессилен. В связи с этим, обладая знаниями логических операций необходимо получить дополнительную информацию для возможности реализации того или иного действия в зависимости от условия. Именно об этом и пойдет речь в следующем разделе нашего урока.

6. Конструкция логического выбора if

Сейчас мы с вами познакомимся с оператором, который позволяет превратить обычную линейную программу в программу «мыслящую». Данный оператор позволяет проверить какое-то утверждение (выражение) на истинность и в зависимости от полученного результата произвести то или иное действие.

Для начала рассмотрим общий синтаксис данного оператора:

```
if (утверждение или выражение)
{
    действие 1;
}

else
{
    действие 2;
}
```

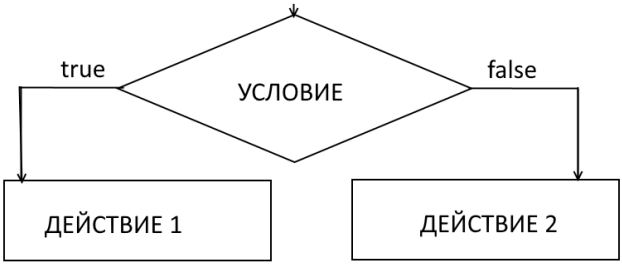

Рисунок 1

Основные принципы работы оператора if

В качестве утверждения или выражения может выступать какая-либо конструкция, содержащая логические операторы или же арифметическое выражение.

- **if(X>Y)** — обычное утверждение, будет истинным, если **X** действительно больше **Y**

```
int X=10,Y=5;
if(X>Y){ // истина
    cout<<"Test!!!"; // на экране Test
}
```

- **if(A>B&&A<C)** — комбинированное утверждение, состоящее из двух частей, будет истинно, если обе части будут верными.

```
int A=10,B=5,C=12;
if(A>B&&A<C){ // истина
    cout<<"A between B and C"; // на экране A between
                               // B and C
}
```

- **if(A–B)** — арифметическое выражение, будет истинным, если **A** не равно **B**, т.к. в противном случае (если они равны) их разность даст нуль, а нуль это ложь.

```
int A=10,B=15;
if(A-B){
    // -5 это истина
    cout<<"A != B"; // на экране A != B
}
```

- **if(++A)** — арифметическое выражение, будет истинным, если **A** не равно **–1**, т.к. если **A** равно **–1** увеличение на **1** даст нуль, а нуль это ложь.

```
int A=0;
if(++A){ // 1 это истина
    cout<<"Best test!!";// на экране Best test!!
}
```

- **if(A++)** — арифметическое выражение, будет истинным, если **A** не равно **0**, т.к. в данном случае используется постфиксная форма инкремента, сначала произойдет проверка условия и будет обнаружен нуль, а потом увеличение на единицу.

```
int A=0;
if(A++){ // 0 это ложь
    cout<<"Best test!!"; // эту фразу мы не увидим,
                         // т.к. if не выполнится
}
```

- **if(A==Z)** — обычное утверждение, будет истинным, если **A** равно **Z**.
- **if(A=Z)** — операция присваивания, выражение будет истинным, если **Z** не равно нулю.

Примечание: *Типичная ошибка. Очень часто вместо операции проверки на равенство ==, по не- внима- тельности указывается операция присва- ивания =, и смысл выражения может радикально измениться. Такая банальная опечатка может привести к не- корректной работе всей программы. Рассмотрим два казалось бы идентичных примера.*

Правильный пример

```
#include <iostream>
using namespace std;
int main(){
    int A,B; // объявляем две переменные
    // просим пользователя ввести в них данные
    cout<<"Enter first digit:\n";
    cin>>A;
    cout<<"Enter second digit:\n";
    cin>>B;
    if(B==0){ // если B содержит ноль сообщаем об ошибке
        cout<<"You can't divide by zero!!!";
    }
    else{ // в противном случае
        cout<<"Result A/B="<<A<<"/"<<B<<"="<<A/B;
        // выдаем результат деления A на B
    }
    cout<<"\n The end. \n"; // конец
    return 0;
}
```

Пример с ошибкой

```
#include <iostream>
using namespace std;
int main(){
    int A,B; // объявляем две переменные
             // просим пользователя ввести в них данные
    cout<<"Enter first digit:\n";
    cin>>A;
    cout<<"Enter second digit:\n";
    cin>>B;
    // Приравниваем B к нулю и проверяем условие,
    // оно автоматически ложно
    if(B=0){// эта часть не выполнится никогда,
            // т.к. условие всегда ложно
       cout<<"You can't divide by zero!!!";
```

```
        // сообщаем об ошибке
    }
    else{// всегда выполняется эта часть,
         // в которой A делится на новоиспеченный нуль
         /* В этой строке произойдет ошибка на этапе
            выполнения, т. к. компьютер попытается
            разделить число на нуль */
       cout<<"Result A/B="<<A<<"/"<<B<<"="<<A/B;
    }
    cout<<"\n The end. \n"; // Эту фразу мы не увидим
                            // никогда.
    return 0;
}
```

1. Как вы уже успели заметить, если содержимое круглых скобок будет являться истиной, то выполнится действие 1, заключенное в фигурные скобки конструкции **if**, при этом действие 2 блока **else** будет проигнорировано.
2. Если же содержимое круглых скобок ложно, выполнится действие 2, заключенное в фигурные скобки конструкции **else**, при этом действие 1 будет проигнорировано.
3. Конструкция **else** является необязательной. Это означает, что если нет необходимости делать что-либо при ложности утверждения, данную конструкцию можно не указывать. Например, программу, использующую защиту против деления на нуль, можно записать таким образом:

```
#include <iostream>
using namespace std;
int main(){
```

```
    int A,B; // объявляем две переменные
    // просим пользователя ввести в них данные
    cout<<"Enter first digit:\n";
    cin>>A;
    cout<<"Enter second digit:\n";
    cin>>B;
    if(B!=0){ // если B не равно нулю
        cout<<"Result A/B="<<A<<"/"<<B<<"="<<A/B;
        // производим вычисления
    }
        // в противном случае не делаем ничего
        cout<<"\nThe end.\n";
    return 0;
}
```

4. Если к блоку **if** или **else** относится только одна команда, то фигурные скобки можно не указывать. С помощью этого правила сделаем программу еще короче:

```
#include <iostream>
using namespace std;
int main(){
    int A,B; // объявляем две переменные
    // просим пользователя ввести в них данные
    cout<<"Enter first digit:\n";
    cin>>A;
    cout<<"Enter second digit:\n";
    cin>>B;
    if(B!=0) // если B не равно нулю
    // производим вычисления
    cout<<"Result A/B="<<A<<"/"<<B<<"="<<A/B;
    // в противном случае не делаем ничего
    cout<<"\nThe end.\n";
    return 0;
}
```

Мы только что познакомились с условным оператором **if** и обсудили основные принципы его действия. Прежде чем переходить к рассмотрению специфических особенностей **if** и практическим примерам, сделаем небольшое отступление и посмотрим на еще один оператор, с помощью которого можно поставить простое условие.

> **Примечание.** *Будьте внимательны: оператор if и оператор else неразрывны! Попытка вписать между ними строку кода, приведет к ошибке на этапе компиляции.*

Фрагмент кода с ошибкой

```
....
    if(B==0){ // если B содержит ноль
        // сообщаем об ошибке
        cout<<"You can't divide by zero!!!";
    }
    cout<<"Hello";// Ошибка!!!! Разрыв конструкции
                  // if - else!!!
    else{ // в противном случае
        cout<<"Result = "<<A/B;// выдаем результат
                               // деления A на B
    }
....
```

Тернарный оператор

Некоторые условия являются очень примитивными. Например, возьмем нашу программу деления двух чисел. Она проста и с точки зрения действий и с точки зрения кода. На операторы **if** и **else** приходится по одной строке

кода — действия. Такую программу, можно упростить еще больше, используя тернарный оператор.

Для начала рассмотрим его синтаксис:

```
УТВЕРЖДЕНИЕ ИЛИ ВЫРАЖЕНИЕ?ДЕЙСТВИЕ1:ДЕЙСТВИЕ2;
```

Принцип действия прост — если УТВЕРЖДЕНИЕ ИЛИ ВЫРАЖЕНИЕ — истина, выполняется ДЕЙСТВИЕ 1, если — ложь, выполняется ДЕЙСТВИЕ 2.

Давайте рассмотрим действие данного оператора:

```
#include <iostream>
using namespace std;
int main(){
    int A,B; // объявляем две переменные

    // просим пользователя ввести в них данные
    cout<<"Enter first digit:\n";
    cin>>A;
    cout<<"Enter second digit:\n"
    cin>>B;

    /* В данном случае, если B не будет равно нулю,
       выполниться та команда, которая стоит после
       знака вопроса и на экране покажется результат
       деления. В противном случае выполниться
       команда стоящая после знака двоеточие и на
       экране будет сообщение об ошибке деления на нуль.
    */
    (B!=0)?cout<<"Result A/B="<<A<<"/!"<<B<<"="<<
         A/B:cout<<"You can't divide by zero!!!";

    // конец программы
    cout<<"\n The end. \n";
    return 0;
}
```

Не правда ли, код стал еще оптимальнее!? Для закрепления полученной информации приведем еще один, более сложный, пример. Программа, будет определять, какое из двух чисел, введенных пользователем является большим, а какое меньшим.

```
#include <iostream>
using namespace std;

int main(){
    int a,b; // объявляем две переменные

    // просим пользователя ввести в них данные
    cout<<"Enter first digit:\n";
    cin>>a;
    cout<<"Enter second digit:\n";
    cin>>b;
 */ Если, (b>a), то на место оператора ?: подставится b,
    в противном случае на место оператора подставится a,
    таким образом, то число, которое больше запишется
    в переменную max.
 */
    int max=(b>a)?b:a;

 /* Если, (b<a), то на место оператора ?: подставится b,
    в противном случае на место оператора подставится a,
    таким образом, то число, которое больше запишется
    в переменную min.
 */
    int min=(b<a)?b:a;

    // Вывод результата на экран.
    cout<<"\n Maximum is \n"<<max;
    cout<<"\n Minimum is \n"<<min<<"\n";
    return 0;
}
```

Итак, давайте твердо уясним следующее: если условие и действия от него зависящие, достаточно просты, будем использовать тернарный оператор. Если же нам необходима сложная конструкция, то, безусловно, используем оператор **if**.

7. Лесенка if – else if

Из прошлого раздела урока вы узнали о существовании условных операторов. Теперь неплохо было бы получить информацию об особенностях их работы.

Предположим нам необходимо написать программу для учета денежной скидки, в зависимости от суммы. Например, если покупатель приобрел товара на сумму больше 100 грн., он получает скидку 5%. Больше 500 грн. — 10%, и, наконец больше 1000 грн. — 25%. приложение должно выдать сумму, которую должен уплатить покупатель, если последний получил скидку. Теперь необходимо найти оптимальный вариант решения задачи.

Вариант решения № 1

```
#include <iostream>
using namespace std;
int main(){
    // объявляется переменная,для хранения
    // первоначальной суммы
    int summa;
    // запрос на ввод суммы с клавиатуры
    cout<<"Enter item of summa:\n";
    cin>>summa;
    if(summa>100){// если сумма больше 100 грн.,
                  // скидка 5%
      cout<<"You have 5% discount!!!\n";
      cout<<"You must pay - "<<summa-summa/100*5<<"\n";
    }
    if(summa>500){// если сумма больше 500 грн., скидка 10%
       cout<<"You have 10% discount!!!\n";
```

```
        cout<<"You must pay - "<<summa-summa/100*10<<"\n";
    }

    if(summa>1000){// если сумма больше 1000 грн.,
                   // скидка 25%
      cout<<"You have 25% discount!!!\n";
      cout<<"You must pay - "<<summa-summa/100*25<<"\n";
    }
    else{ // в противном случае, скидки нет
       cout<<"You have not discount!!!\n";
       cout<<"You must pay - "<<summa<<"\n";
    }
    return 0;
}
```

Данный пример, на первый взгляд у начинающего программиста не вызывает нареканий, однако, давайте рассмотрим ситуацию, в которой программа отработает весьма некорректно. Сумма, введенная с клавиатуры, равна **5000**. Эта цифра превышает **1000**, следовательно, мы должны получить **25%** скидку. Однако, произойдет совсем другое.

1. Каждый оператор **if** является самостоятельным и не зависит от других **if**, следовательно, вне зависимости от того, какое из **if** выполнится, проверка условия все равно будет осуществляться для всех операторов.
2. Сначала, осуществится проверка условия **if(summa> 100)**. **5000**, естественно больше **100**, условие истинно и выполняется тело **if**. На экране мы получаем:

```
You have 5% discount!!!
You must pay - 4750
```

3. Однако, на этом программа не остановится — далее будет проанализировано условие **if(summa>500)**. **5000** больше **500**, условие снова истинно и выполняется тело **if**. На экране мы получаем:

```
You have 10% discount!!!
You must pay - 4500
```

4. Ну и, наконец программа проверит условие **if(summa>1000)**, которое тоже окажется истинным, так как **5000** больше **1000**. И, действие связанное с **if**, выполняется и теперь. На экран выводится:

```
You have 25% discount!!!
You must pay - 3750
```

Таким образом, вместо одной информационной надписи, мы получаем три. Такое решение задачи является нерентабельным. Попробуем оптимизировать его.

Вариант решения № 2

```
#include <iostream>
using namespace std;

int main(){
    // объявляется переменная,для хранения
    // первоначальной суммы
    int summa;

    // запрос на ввод суммы с клавиатуры
    cout<<"Enter item of summa:\n";
    cin>>summa;
```

```
    // если сумма в диапазоне от 100 грн. до 500 грн.,
    // скидка 5%
    if(summa>100&&summa<=500){
        cout<<"You have 5% discount!!!\n";
        cout<<"You must pay - "<<
              summa-summa/100*5<<"\n";
    }

    // если сумма в диапазоне от 500 грн.
    // до 1000 грн., скидка 5%
    if(summa>500&&summa<=1000){
        cout<<"You have 10% discount!!!\n";
        cout<<"You must pay - "<<
              summa-summa/100*10<<"\n";
    }

    if(summa>1000){ // если сумма больше 1000 грн.,
                    // скидка 25%
        cout<<"You have 25% discount!!!\n";
        cout<<"You must pay - "<<
              summa-summa/100*25<<"\n";
    }
    else{ // в противном случае, скидки нет
        cout<<"You have not discount!!!\n";
        cout<<"You must pay - "<<summa<<"\n";
    }
    return 0;

}
```

Для начала, снова представим, что пользователь ввел сумму размером 5000 грн.

1. Сначала, осуществится проверка условия **if(summa>100&&summa<=500)**. 5000 не входит в заданный диапазон, условие ложно и тело **if** выполнятся не будет.

2. Далее будет проанализировано условие **if(summa> 500&&summa<=1000)**. **5000** не входит и в этот диапазон, условие снова ложно и тело **if** выполнятся не будет.
3. И, наконец программа проверит условие **if(summa> 1000)**, которое окажется истинным, так как **5000** больше **1000**. И, действие связанное с **if**, выполнится. На экран выводится:

```
You have 25% discount!!!
You must pay - 3750
```

Казалось бы на этом можно остановиться, но давайтека проверим еще один вариант. Например, пользователь вводит значение **600**. И, на экране появляются следующие данные:

```
Enter item of summa:
600
You have 10% discount!!!
You must pay - 540
You have not discount!!!
You must pay - 600
Press any key to continue
```

Такой поворот событий объясняется легко:

1. Сначала, осуществится проверка условия **if(summa> 100&&summa<=500)**. **5000** не входит в заданный диапазон, условие ложно и тело **if** выполнятся не будет.
2. Далее будет проанализировано условие **if(summa> 500&&summa<=1000)**. **5000** входит в этот диапазон,

условие истинно и тело **if** выполнится, на экран выведется сообщение о **10%** скидке.

3. И, наконец программа проверит условие **if(summa>1000)**, которое окажется ложным. Действие связанное с **if** выполняться не будет, но у данного самостоятельного оператора **if**, есть собственный **else**, который отработает в нашем случае. На экран выводится сообщение об отсутствии скидки.

Вывод: *во-первых, мы выяснили, что оператор* **else** *относится только к последнему* **if**. *Во-вторых, пришли к тому, что и данная реализация программы нас не устраивает.*

Рассмотрим еще один пример решения. Название проекта *Discount3*.

Вариант решения № 3

```
#include<iostream>
using namespace std;
int main(){
    // объявляется переменная,для хранения
    // первоначальной суммы
    int summa;
    // запрос на ввод суммы с клавиатуры
    cout<<"Enter item of summa:\n";
    cin>>summa;
    if(summa>1000){ // если сумма больше 1000 грн.,
                    // скидка 25%
        cout<<"You have 25% discount!!!\n";
        cout<<"You must pay - "<<
        summa-summa/100*25<<"\n";
    }
```

```
    else{ // если сумма не больше 1000 грн.
          // продолжаем анализ
        if(summa>500){ // если сумма больше 500 грн.,
                       // скидка 10%
            cout<<"You have 10% discount!!!\n";
            cout<<"You must pay - "<<
                  summa-summa/100*10<<"\n";
        }
        else{ // если сумма не больше 500 грн.
              // продолжаем анализ
            if(summa>100){ // если сумма больше 100 грн.,
                           // скидка 5%
                cout<<"You have 5% discount!!!\n";
                cout<<"You must pay - "<<
                summa-summa/100*5<<"\n";
            }
            else{ // если сумма не больше 100 грн.
                  // скидки нет
                cout<<"You have not discount!!!\n";
                cout<<"You must pay - "<<summa<<"\n";
            }
        }
    }
    return 0;
}
```

Внимательно проанализировав данный пример, вы заметите, что каждый следующий **if**, может выполниться только, в том случае, если не выполнился его «предшественник», так как находится внутри конструкции **else** последнего. Таким образом, мы наконец-то нашли оптимальный код реализации. Структура, которую мы только что создали называется «лесенка» **if else if**, так как условия в ней располагаются в виде лестницы. Теперь, мы с вами знаем, какая это полезная конструкция. Остался последний штрих.

Оптимизация кода

В предыдущем разделе урока прозвучало правило: Если к блоку **if** или **else** относится только одна команда, то фигурные скобки можно не указывать. Дело в том, конструкция **if else** считается одной цельной командной структурой. Следовательно, если внутри некоторых **else** нет ничего кроме вложенной конструкции, фигурные скобки таких **else** можно опустить:

```
#include <iostream>
using namespace std;

int main(){
    // объявляется переменная,для хранения
    // первоначальной суммы
    int summa;
    // запрос на ввод суммы с клавиатуры
    cout<<"Enter item of summa:\n";
    cin>>summa;
    if(summa>1000){ // если сумма больше 1000 грн.,
                    // скидка 25%
        cout<<"You have 25% discount!!!\n";
        cout<<"You must pay - "<<
              summa-summa/100*25<<"\n";
    }
    // если сумма не больше 1000 грн.
    // продолжаем анализ
    else if(summa>500){ // если сумма больше 500 грн.,
                        // скидка 10%
        cout<<"You have 10% discount!!!\n";
        cout<<"You must pay - "<<
              summa-summa/100*10<<"\n";
    }
    // если сумма не больше 500 грн.
    // продолжаем анализ
```

```
    else if(summa>100){ // если сумма больше
                        // 100 грн., скидка 5%
        cout<<"You have 5% discount!!!\n";
        cout<<"You must pay - "<<
              summa-summa/100*5<<"\n";
    }
    return 0;
}
```

Вот и всё! Задача решена. Мы получили цельную конструкцию множественного выбора, состоящую из отдельных, взаимозависимых условий. Теперь можно переходить к следующим разделам урока, где мы с вами подробно рассмотрим еще несколько примеров использования **if else**.

8. Практический пример создание текстового квеста

Постановка задачи

Вы конечно знакомы с таким жанром игр, как квест. Герой такой игры должен выполнять различные задания, отвечать на вопросы, принимать решения, от которых зависит результат игры. Мы с вами попробуем сейчас создать так называемый текстовый квест (квест без графики). Наша задача предлагать герою варианты действий, и в зависимости от его выбора строить ситуацию.

Код реализации

```
#include <iostream>
using namespace std;

int main()
{
    // Добро пожаловать. Три испытания чести.
    // Злой маг похитил принцессу и ее судьба в твоих
    // руках. Он предлагает тебе пройти 3 испытания
    // чести в его лабиринте.
    cout<<"Welcome. Three tests of honour.
           The malicious magician has stolen\n\n";
    cout<<"\nprincess and its destiny in your hands.
           It suggests you\n";
    cout<<"\nto pass 3 tests of honour in its
           labyrinth.\n";
```

```
bool goldTaken, diamondsTaken, killByDragon;

// Ты входишь в первую комнату, здесь очень много
// золота.
cout<<"You enter into the first room, here it is "
      "a lot of gold.\n\n";
// Возьмешь ли ты его?
cout<<"Whether you will take it?(1-yes, 0-no)\n\n";
cin>>goldTaken;
if(goldTaken) // если возьмешь
{
    // Золото остается тебе, но ты провалил
    // испытание. ИГРА ОКОНЧЕНА!!!
    cout<<"Gold remains to you, but you have "
          "ruined test. GAME is over!!!\n\n";
}
else // если нет
{
    // Поздравляю, ты прошел первое испытание
    // чести!
    cout<<"I congratulate, you have passed "
          "the first test abuse!\n\n";
    // Ты переходишь в следующую комнату.
    // Она полна бриллиантов
    cout<<"You pass in a following room. "
          "It is full of brilliants \n\n";
    // Возьмешь ли ты бриллианты?
    cout<<"Whether you will take brilliants? "
          "(1-yes,0-no)\n\n";
    cin>>diamondsTaken;
    if(diamondsTaken)// если возьмешь
    {
        // Бриллианты остаются тебе,
        // но ты провалил второе испытание
        cout<<"Brilliants remain to you, but you "
              "have ruined the second test\n\n";
        // ИГРА ОКОНЧЕНА!!!
```

```
        cout<<"GAME is over!!!\n\n";
    }
    else // если нет
    {
    // Поздравляю, ты прошел второе испытание
    // чести!!!
    cout<<"I congratulate, you have passed "
          "the second test abuse!!!\n\n";
    // Ты входишь в третью комнату.
    cout<<"You enter into the third room. \n\n";
    // На крестьянина напал дракон!
    // Двигаться дальше
    cout<<"The person was attacked by "
          "a dragon! To move further \n\n";
    // не обращая на них внимания
    cout<<"Not paying to them of attention "
          "(1-yes,0-no)?\n\n";
    cin>>killByDragon;

        if(killByDragon)// если возьмешь
        {
            // Ты пытаешься проскользнуть мимо,
            // но дракон
            cout<<"You try to pass past, "
                  "but a dragon \n\n";
            // замечает твое присутствие.
            cout<<"notices your presence\n\n";
            // Он превращает тебя в пепел.
           // Ты мертв!!!
            cout<<"It transforms you into ashes. "
                  "You are dead!!!\n\n";
            // ИГРА ОКОНЧЕНА!!!
            cout<<"GAME is over!!!\n\n";
        }
        else// если нет
        {
```

```
                // Поздравляю, ты с честью прошел все
                // испытания!!!
                cout<<"I congratulate, you with "
                      "honour have was tested all!!! "
                      "\n\n";
                // Принцесса достается тебе!!!
                cout<<"Princess gets to you!!!\n\n";
            }
        }
    }
    return 0;
}
```

Несмотря на примитивность примера, вы можете убедиться в том, что уже сейчас, имея минимальные знания мы можем написать программу, способную развлечь среднестатистического малыша. Это происходит потому, что у нас в руках есть мощное средство — условные операторы.

9. Практический пример на принадлежность точки кольцу

Постановка задачи

На плоскости нарисовано кольцо с центром в точке **(x0, y0)**. И радиусами границ **r1<r2**. На этой же плоскости дана точка с координатами **(x, y)**. Необходимо определить, принадлежит ли эта точка кольцу. Название проекта **Circle2**.

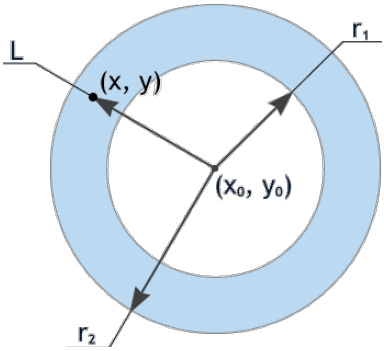

Рисунок 2

Решение задачи

Для решения задачи необходимо вычислить расстояние от центра кольца до точки, и сравнить его с радиусами:

1. Если длина отрезка от центра до точки меньше чем радиус внешней окружности и при этом больше, чем радиус внутренней окружности, то точка принадлежит кольцу.

if(L>=r1&&L<=r2)

В противном случае точка не принадлежит кольцу.

2. Для выяснения расстояния, от центра до точки мы воспользуемся теоремой Пифагора — *Квадрат гипотенузы равен сумме квадратов катетов*. Следовательно — *длина гипотенузы равна корню квадратному из суммы квадратов катетов*.

Примечание: *Нам понадобятся дополнительные знания для получения степени и корня квадратного.*

- В программу необходимо подключить библиотеку для использования математических функций под названием **math.h**.
- Для возведения в степень используется функция **pow(double num,double exp)**, где **num** это число для возведения в степень, а **exp** — сама степень.
- Для извлечения корня квадратного используется функция **sqrt(double num)**, где **num** это число из которого извлекают корень. **c=sqrt(pow(a,2)+ pow(b,2))**;

L=c;

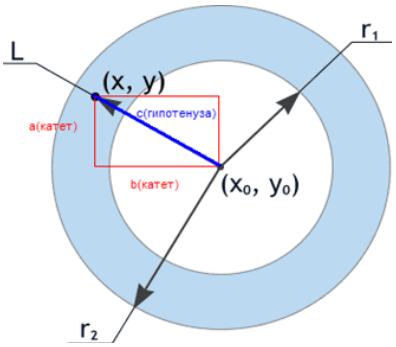

Рисунок 3

3. Осталось выяснить длины катетов, на рисунке видно, как это сделать.

a=y–y0;
b=x–x0;

Теперь осталось собрать все части решения в единое целое.

```
#include <iostream>
#include <math.h>

using namespace std;
int main() {
    // Объявление переменных
    int x0, y0, r1, r2, x, y;
    float L;
    // Запрос на ввод необходимых данных
    cout<<"Input coordinates of circle's "
          "center (X0, Y0):";
    cin>>x0>>y0;
    cout<<"Input circle radiuses R1 and R2:";
    cin>>r1>>r2;
    cout<<"Input point coordinates (X, Y): ";
    cin>>x>>y;
    // Выведение формулы
    L = sqrt(pow(x - x0, 2) + pow(y - y0, 2));
    // Анализ результатов
    if ((r1 < L)&& (L < r2 )) {
        cout<<"This point is situated inside "
              "the circle.\n";
    }
    else {
        cout<<"This point is not situated "
              "inside the circle.\n";
    }
    return 0;
}
```

10. Структура множественного выбора switch

Мы уже знакомы с конструкцией, анализирующей условия — конструкцией **if**, а также с тернарным оператором. Еще один оператор выбора — оператор **switch**. Представьте, что необходимо написать программу, в которой используется меню, состоящее из пяти пунктов. Например, маленькое приложение для малышей, умеющее складывать, вычитать и т.п. Можно реализовать обработку выбора с помощью лесенки **if else if**, вот так:

```
#include <iostream>
using namespace std;
int main(){
   // объявление переменных и ввод значения
   // с клавиатуры
   float A,B,RES;
   cout<<"Enter first digit:\n\";
   cin>>A;
   cout<<"Enter second digit:\n";
   cin>>B;

   // реализация программного меню
   char key;
   cout<<"\nSelect operator:\n";
   cout<<"\n + - if you want to see SUM.\n";
   cout<<"\n - - if you want to see DIFFERENCE.\n";
   cout<<"\n * - if you want to see PRODUCT.\n0";
   cout<<"\n / - if you want to see QUOTIENT.\n";
```

```
    // ожидание выбора пользователя
    cin>>key;
    if(key=='+') { // если пользователь выбрал
                   // сложение
       RES=A+B;
       cout<<"\nAnswer: "<<RES<<"\n";
    }
    else if(key=='-'){ // если пользователь выбрал
                       // вычитание
          RES=A-B;
          cout<<"\nAnswer: "<<RES<<"\n";
       }
       else if(key=='*'){ // если пользователь выбрал
                          // умножение
             RES=A*B;
             cout<<"\nAnswer: "<<RES<<"\n";
          }
          else if(key=='/'){ // если пользователь
                             // выбрал деление
             if(B){ // если делитель не равен нулю
                RES=A/B;
                cout<<"\nAnswer: "<<RES<<"\n";
             }
             else{ // если делитель равен нулю
                 cout<<"\nError!!! Divide by "
                       "null!!!!\n";
              }
           }
           else{ // если введенный символ некорректен
               cout<<"\nError!!! This "
                     "operator isn't correct\n";
           }
       }
    }
    return 0;
}
```

Вышеописанный пример вполне корректен, но несколько громоздко выглядит. Данный код можно значительно упростить, именно для этого используется **switch**. Он позволяет сравнить значение переменной с целом рядом значений и, встретив совпадение, выполнить определенное действие.

Общий синтаксис и принцип действия

Для начала рассмотрим общий синтаксис оператора:

```
switch(выражение){
case значение1:
    действие1;
    break;

case значение2:
    действие2;
    break;

case значение3:
    действие3;
    break;

......

default:
    действие_по_умолчанию;
    break;
}
```

Давайте проанализируем данную форму записи:

1. **Выражение** — те данные, которые необходимо проверить на соответствие. Здесь может указываться переменная (но только типа **char** или целочисленная),

либо выражение, результатом которого являются целочисленные данные.

2. **Case Значение1, case значение2, case значение3** — Целочисленные или символьные постоянные значения с которыми сверяется выражение.
3. **Действие1, действие2, действие3** — действия, которые должны выполниться, если значение выражения совпало со значением **case**.
4. Если произошло совпадение и благополучно выполнилось действие связанное с совпавшим **case**, **switch** заканчивает свою работу и программа переходит на следующую строку за закрывающейся фигурной скобкой оператора **switch**. За данную функцию отвечает оператор **break** именно он останавливает выполнение **switch**.
5. Если в ходе анализа совпадений не произошло срабатывает секция **default** и выполняется **действие_по_умолчанию**. Оператор **default** является аналогом оператора **else**.

Теперь давайте посмотрим, каким образом можно упростить приведенный в начале темы пример.

Оптимизация примера

```
#include <iostream>
using namespace std;
int main(){
    // объявление переменных и ввод значения
    // с клавиатуры
    float A,B,RES;
    cout<<"Enter first digit:\n";
```

```
cin>>A;
cout<<"Enter second digit:\n";
cin>>B;

// реализация программного меню
char key;
cout<<"\nSelect operator:\n";
cout<<"\n + - if you want to see SUM.\n";
cout<<"\n - - if you want to see DIFFERENCE.\n";
cout<<"\n * - if you want to see PRODUCT.\n";
cout<<"\n / - if you want to see QUOTIENT.\n";

// ожидание выбора пользователя
cin>>key;

// проверяется значение переменной key
switch(key){
case '+': // если пользователь выбрал сложение
    RES=A+B;
    cout<<"\nAnswer: "<<RES<<"\n";
    break; // parada do switch

case '-': // если пользователь выбрал вычитание
    RES=A-B;
    cout<<"\nAnswer: "<<RES<<"\n";
    break; // остановка switch

case '*': // если пользователь выбрал умножение
    RES=A*B;
    cout<<"\nAnswer:  "<<RES<<"\n";
    break; // остановка switch case '/':
           // если пользователь выбрал деление

case '/': // если пользователь выбрал деление
    if(B){ // если делитель не равен нулю
        RES=A/B;
        cout<<"\nAnswer: "<<RES<<"\n";
    }
```

```
        else{ // если делитель равен нулю
            cout<<"\nError!!! Divide by null!!!!\n";
        }
        break; // остановка switch
    default: // если введенный символ некорректен
        cout<<"\nError!!! This operator isn't "
              "correct\n";
        break; // остановка switch
    }
    return 0;
}
```

Как видите, код теперь выглядит гораздо проще и его удобнее читать.

Оператор **switch** достаточно прост в обращении, однако необходимо знать некоторые особенности его работы:

1. Если в **case** используются символьные значения, они должны указываться в одинарных кавычках, если целочисленные, то без кавычек.
2. Оператор **default** может располагаться в любом месте системы **switch**, выполняться он все равно будет в том случае, если нет ни одного совпадения. Однако правилом «хорошего тона» является указывать **default** в конце всей конструкции.

```
switch(выражение){
case значение1:
    действие1;
    break;

case значение2:
    действие2;
    break;
```

```
default: действие_по_умолчанию;
    break;

case значение3:
    действие3;
    break;
}
```

3. После самого последнего оператора в списке (будь то **case** или **default**) оператор **break** можно не указывать.

```
switch(выражение){

case значение1:
    действие1;
    break;

case значение2:
    действие2;
    break;
default:
    действие_по_умолчанию;
    break;

case значение3:
    действие3;
}
```

```
switch(выражение){

case значение1:
    действие1;
    break;

case значение2:
    действие2;
    break;

case значение3:
    действие3;
    break;
default:
    действие_по_умолчанию;
}
```

4. Оператор **default** можно вообще не указывать, в случае, если не найдется совпадений, просто ничего не произойдет.

```
switch(выражение){
case значение1:
    действие1;
    break;
```

```
case значение2:
    действие2;
    break;

case значение3:
    действие3;
    break;
}
```

5. В случае, если необходимо выполнять один и тот же набор действий для разных значений проверяемого выражения, можно записывать несколько меток подряд. Рассмотрим пример программы, которая переводит систему буквенных оценок в цифровые.

```
#include <iostream>
using namespace std;

int main(){
    // объявление переменной, для хранения буквенной
    // оценки
    char cRate;

    // просьба ввести буквенную оценку
    cout<<"Input your char-rate\n";
    cin>>cRate;

    // анализ введенного значения
    switch (cRate) {
        case 'A':
        case 'a':
            // оценка A или a равноценна 5
            cout<<"Your rate is 5\n";
            break;
```

```
        case 'B':
        case 'b':

        // оценка B или b равноценна 4
        // а также оценка C или c равноценна 4-
        case 'C':
        case 'c':
            cout<<"Your rate is 4\n";
            break;

            // оценка C или c равноценна 3
            cout<<"Your rate is 3\n";
            break;

        case 'D':
        case 'd':
            // оценка D или d равноценна 2
            cout<<"Your rate is 3\n";
            break;

        case 'F':
        case 'f':
            // оценка F или f равноценна 2
            cout<<"Your rate is 2\n";
            break;

        default:
            // остальные символы некорректны
            cout<<"This rate isn't correct\n";
    }
    return 0;
}
```

Пример примечателен тем, что с помощью идущих подряд **case** достигается регистронезависимость. То есть, неважно, какую именно букву введет пользователь — заглавную или строчную.

Распространенная ошибка

Всё самое главное об операторе **switch** сказано, осталось лишь получить информацию о том, с какой проблемой может столкнуться программист, используя данный оператор.

Если случайно пропустить **break** в любом блоке **case**, кроме последнего, и этот блок в последствии отработает, то выполнение **switch** не остановиться. Тот блок оператора **case**, который будет идти вслед за уже выполнившимся, так же выполниться без проверки.

Пример ошибки

```
#include <iostream>
using namespace std;
int main(){
    // реализация программного меню
    int action;
    cout<<"\nSelect action:\n";
    cout<<"\n 1 - if you want to see course "
          "of dollar.\n";
    cout<<"\n 2 - if you want to see course "
          "of euro.\n";
    cout<<"\n 3 - if you want to see course "
          "of rub.\n";

    // ожидание выбора пользователя
    cin>>action;
    // проверяется значение переменной action
    switch(action){
    case 1: // если пользователь выбрал доллар
        cout<<"\nCource: 28 gr.\n";
        break; // остановка switch
    case 2: // если пользователь выбрал евро
        cout<<"\nCource: 30 gr.\n";
```

```
        // break; закомментированна остановка switch
    case 3: // если пользователь выбрал рубли
        cout<<"\nCource: 0.44 gr.\n";
        break; // остановка switch
    default: // если выбор некорректен
        cout<<"\nError!!! This operator isn't "
              "correct\n";
        break; // остановка switch
    }
    return 0;
}
```

Ошибка произойдет в том случае, если будет выбран **2** пункт меню. В **case** со значением **2** закомментирован оператор остановки **break**. На экране результат такой ошибки выглядит следующим образом:

```
Select action:
1 - if you want to see course of dollar.
2 - if you want to see course of euro.
3 - if you want to see course of rub.
2
Cource: 30 gr.
Cource: 0.44 gr.
Press any key to continue
```

Кроме необходимой информации на экране показалось то, что находилось в блоке **case**, располагающемся после ошибочной конструкции. Следует избегать таких опечаток, так как они приводят к ошибкам на этапе выполнения.

11. Понятие enum, как перечислимого типа

Сейчас пришло время изучить еще один тип данных — **enum**. **Перечисление** (*enum*) — это набор именованных целочисленных констант.

Допустим, что в вашей программе необходимо создать несколько констант с кодом соответствующей страны. Это бы выглядело вот так:

```
const int USA = 1;
const int France= 33;
const int Ukraine = 380;
const int Italy = 39;
const int Australia = 61;
```

Однако такой способ только нагромождает программный код, он занимает слишком большое количество строк. И, как видим, это еще не все страны. Естественно, что хочется как-то систематизировать, сгруппировать эту информацию. Именно перечисление позволяет решить такую проблему. Синтаксис:

```
enum имя { константа1 [= значение1],
           константа2 [= значение2], …};
```

Тогда список констант будет иметь вот такой вид:

```
enum countries { USA =1, France=33, Ukraine=380,
                 Italy=39, Australia =61};
```

После создания такого перечисления стран имена USA, France, Ukraine, Italy, Australia становятся все теми же целочисленными константами, что и в примере выше.

Также в качестве примера перечисление можно использовать в виде списка монет США. Уникальность этих монет в том, что практически каждая из них кроме своего номинала имеет уже устоявшееся собственное название. К примеру, один цент называют пении, а 5 центов — никель. За значение константы возьмем номинал монеты, а в качестве ее имени ее общепринятое название.

```
enum coins { penny = 1, nickel = 5, dime = 10,
             quarter = 25, half = 50,
             dollar_coin = 100 };
```

Использование перечислений позволяет упростить понимание программ, написанных другим программистом, хоть это и более трудоемко. Посмотрим на перечисление американских монет в программе ниже:

```
#include <iostream>
using namespace std;

int main() {
    // Объявление пречисления монет США
    enum coins { penny = 1, nickel = 5, dime = 10,
                 quarter = 25, half = 50,
                 dollar_coin = 100 };
    // Объявление переменной для монеты
    int coin;
    cout << "Please enter a value of american coin"
         << endl;
    cin >> coin;
```

```
switch (coin)
{
    case penny:
        // Выводим на экран, что пенни
        // соответствует одному центу
        cout << "penny = 1 cent " << endl;
        // Дополняет вывод описанием монеты
        // На одной стороне присутствует Авраам
        // Линкольн, а Мемориал Линкольна -
        // на другой.
        cout << "It has Abraham Lincoln on one "
                "side and the Lincoln Memorial "
                "on the other." << endl;
        break;
    case nickel:
        // Выводим на экран, что никель
        // соответствует 5 центам
        cout << "nicel = 5 cents" << endl;
        // Дополняет вывод описанием монеты
        // На лицевой стороне монеты изображен
        // Томас Джефферсон и Монтичелло на
        // обратной стороне.
        cout << "It has Thomas Jefferson "
                "on the front and Monticello on "
                "the back." << endl;
        break;
    case dime:
        // Выводим на экран, что дайм
        // соответствует 10 центам
        cout << "dime = 10 cents" << endl;
        // Дополняет вывод описанием монеты
        // На лицевой стороне монеты изображен
        // Франклин Д. Рузвельт и факел
        // с обратной стороны
        cout << "It has Franklin D. Roosevelt "
                "on the front and a torch on "
                "the back." << endl;
        break;
```

```
        case quarter:
            // Выводим на экран, что 1/4 доллара
            // состовляет 25 центов
            cout << "quater = 25 cents" << endl;
            // Дополняет вывод описанием монеты
            // На лицевой стороне монеты изобажен
            // Джордж Вашингтон и на обратной стороне -
            // либо эмблема Соединенных Штатов,
            // либо дизайн одного из 50 штатов
            cout << "It has George Washington on "
                    "the front and either a United "
                    "States emblem or a design of "
                    "one of the 50 states on "
                    "the back." << endl;
            break;
        case half:
            // Выводим на экран, что 1/2 доллара
            // составляет 50 центов
            cout << "half = 50 cents " << endl;
            // Дополняет вывод описанием монеты
            // На лицевой стороне монеты изображен
            // Джон Ф. Кеннед и с обратной стороны -
            // Германский герб.
            cout << "It has John F.Kenned on "
                    "the front and the Presidential "
                    "Coat of Arms on the back." << endl;
            break;
        case dollar_coin:
            // Выводим на экран, что 1 монета доллара
            // составляет 100 центов
            cout << "dollar coin = 100 cents" << endl;
            // Дополняет вывод описанием монеты
            // На нем изображена родная американская
            // героиня Сакагавея на личевой стороне и
            // лысый орел на обратной.
            cout << "It features native American "
                    "heroine Sacagawea on the front "
                    "and a bald eagle on the back."
                     << endl;
```

```
            break;
        default:
            cout << "not found" << endl;
    }
    return 0;
}
```

Ниже предоставлен пример работы данной программы:

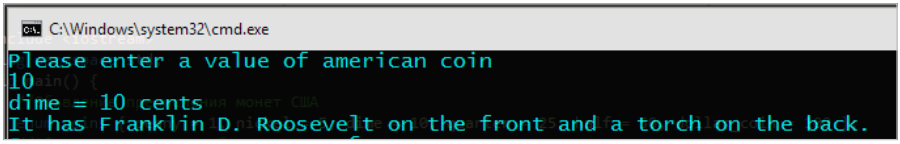

Рисунок 4

А теперь, рассмотрим другой пример — наша задача смоделировать поведение робота при его передвижении. Как современное устройство, наш робот знает о четырех сторонах света: 1 — север, 2 — запад, 3 — юг, 4 — восток. А также он может выполнять простые команды: 0 — продолжить движение, 1 — поворот направо, –1 — поворот налево.

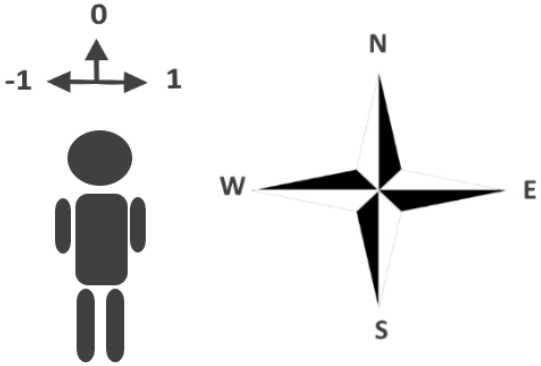

Рисунок 5

Так вот, наша цель определить направление робота после выполненной команды. Что для этого нужно знать? Первое, это куда изначально смотрит робот, а второе — выполняемая команда в текущий момент.

```
enum command {straight, right, left = -1};
enum direction {north = 1, west, south, east};
```

Можно заметить, что инициализация констант необязательна. Тогда какие значения будут иметь эти константы? Если ни одна из констант не была проинициализирована, тогда первая из них будет по умолчанию содержать значение ноль, а все остальные на единицу больше, то есть **straight = 0**, **right = 1**, но **left = –1**. Точно с такой же закономерностью следующие константы имеют значения: **north = 1**, **west = 2**, **south = 3**, **east = 4**.

Чаще всего перечисления лучше всего использовать совместно со **switch** конструкцией. Именно таким способом более понятно какие действия нужно выполнять по текущему значению **case**.

А теперь рассмотрим саму программу:

```
#include <iostream>
using namespace std;
int main() {

    // Объявление пречислений для команд и направления
    enum command { straight, right, left = -1 };
    enum direction { north = 1, west, south, east };
    // Объявление переменных для принятой команды
    // и изначального направления робота
    int Command, Direction;
```

```
cout << "Enter the start direction the robot" << endl;
cout << "\t 1- North\n"
     << "\t 2- West\n"
     << "\t 3- South\n"
     << "\t 4- East\n";
 cin >> Direction;
cout << "Select action :" << endl;
cout << "\t 0- straight on\n"
     << "\t 1- turn right\n"
     << "\t -1- turn left\n";
 cin >> Command;

switch (Direction)
// определяем изначальное направление робота
{
   case north: // если робот изначально смотрел
               // на север
      switch (Command)
      // определяем дальнейшее поведение робота
      {
         case straight:
              cout << "The robot is looking "
                      "at the notrh\n";
              break;
         case right:
              cout << "The robot is looking "
                      "at the east\n";
              break;
         case left:
              cout << "The robot is looking "
                      "at the west\n";
              break;
      }
      break;
   case west: // если робот изначально смотрел
              // на запад
      switch (Command)
```

```
        // определяем дальнейшее поведение робота
        {
            case straight:
                cout << "The robot is looking "
                        "at the west\n";
                break;
            case right:
                cout << "The robot is looking "
                        "at the north\n";
                break;
            case left:
                cout << "The robot is looking "
                        "at the south\n";
                break;
        }
        break;

    case south: // если робот изначально смотрел на юг
        switch (Command)
        // определяем дальнейшее поведение робота
        {
            case straight:
                cout << "The robot is looking "
                        "at the south\n";
                break;
            case right:
                cout << "The robot is looking "
                        "at the west\n";
                break;
            case left:
                cout << "The robot is looking "
                        "at the east\n";
                break;
        }
        break;

    case east: // если робот изначально смотрел
               // на восток
```

```
        switch (Command)
        // определяем дальнейшее поведение робота
        {
           case straight:
              cout << "The robot is looking at "
                      "the east\n";
              break;
           case right:
              cout << "The robot is looking "
                      "at the south\n";
              break;
           case left:
              cout << "The robot is looking "
                      "at the north\n";
              break;
        }
        break;
        default:
        cout<<"not found"<< endl;
    }
    return 0;
}
```

Ниже предоставлен пример работы программы:

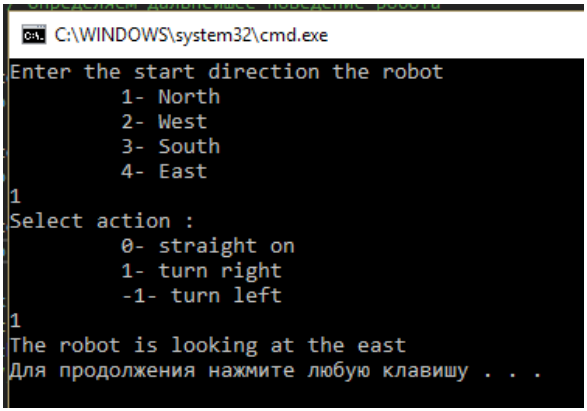

Рисунок 6

Можем убедиться, что без использования перечисления очень легко запутаться в поведении самой программы. И именно перечисления помогают упростить понимание такого кода.

В сегодняшнем уроке мы с вами познакомились с операторами, позволяющими производить анализ каких-либо данных. Теперь вы можете переходить к выполнению домашнего задания. Желаем удачи!

11. Домашнее задание

1. Напишите программу, проверяющую число, введенное с клавиатуры на четность.
2. Дано натуральное число **a (a<100)**. Напишите программу, выводящую на экран количество цифр в этом числе и сумму этих цифр
3. Известно, что 1 дюйм равен 2.54 см. Разработать приложение, переводящие дюймы в сантиметры и наоборот. Диалог с пользователем реализовать через систему меню.
4. Написать программу-калькулятор. Пользователь вводит два числа и выбирает арифметическое действие (+, -, *, /, максимум, минимум). Вывести на экран результат действия.

Урок № 2
Условия