

### The need for control structures

Almost any program encounters the need to make a decision. Consider the following simple case,



```
//the solution to the quadratic equation  $a*x*x + b*x + c = 0$ 
double a = 1, b=0.5, c = 1.0;
// two roots
double r1 = (b + sqrt(b*b-4*a*c))/(2*a);
double r2 = (b - sqrt(b*b-4*a*c))/(2*a); cout
<< "r1 " << r1 << " r2 " << r2 << endl;
```

This program will fail to produce a valid result because  $b^2 - 4ac$  is negative. If one is producing code with a large number of users it is necessary to check for conditions that indicate that the solution cannot be obtained. In this example, the discriminant should not be negative if complex numbers are not allowed. Thus, it is necessary to test the sign of the discriminant and take different actions accordingly.

C++ offers a large set of control constructs that allow a bool condition to affect the flow of control of program execution. In the simple example, the sign of the discriminant can be tested in order to take different actions.

```
c = 0.05;
double disc = b*b-4*a*c;
if(disc<0)
    cout << "The quadratic equation does not have" << endl
        << "a solution over the reals" << endl;
else
{
    r1 = (b + sqrt(b*b-4*a*c))/(2*a);
    r2 = (b - sqrt(b*b-4*a*c))/(2*a);
    cout << "r1 " << r1 << " r2 " << r2 << endl;
}
```

In this case the program outputs,

```
r1 0.361803 r2 0.138197
```

With the original value for  $c = 1.0$  the program would output,

```
The quadratic equation does not have
a solution over the reals
```

This behavior is much more pleasant than outputting

```
r1 -1.#IND r2 -1.#IND
```

which would occur without the “if” control statement.

### The **if** control structure

The most widely used control statement is **if**. The syntax is

```
if ( condition )  
    statement1  
[else  
    statement2]
```

The specification of the syntax uses [ ] brackets to indicate that the part in brackets is optional. That is, it is not necessary that the “else” part of the if statement be present.

A *statement* in C++ is either a single expression, terminated by a ; , or a set of expressions enclosed in a *scope*. A program scope is delineated by a matching pair of curly brackets { ... } . Any variables defined within the scope are not visible outside the scope, i.e. outside the curly brackets. So in our previous example the single statement,

```
cout << "The quadratic equation does not have" << endl  
    << "a solution over the reals" << endl;
```

is executed if the conditional expression is non-zero and the alternative scope,

```
{  
    r1 = (b + sqrt(b*b-4*a*c)) / (2*a) ;  
    r2 = (b - sqrt(b*b-4*a*c)) / (2*a) ;  
    cout << "r1 " << r1 << " r2 " << r2 << endl ;  
}
```

is executed if the expression is zero. The use of program scope enables code to be packaged into independent modules that define computation to be exercised under different conditions. This style is called *structured* programming. It is to be distinguished from early FORTRAN coding style which was a interlinked jumble executed through statement labels and GOTO expressions.

The data type of the **expression** must be capable of meaningful type casting to an integer type. Thus, any of the data types we have encountered so far can be used as the type of the **if** control expression. In the example, the control expression has the data type **double**.

Since the **if** structure is itself a statement, it is possible to create a chain of conditional expressions representing multiple outcomes,

```

double x=0, y=0, z=0;
int a =0, b=1, c=0;
if(a) {
    x = 5;
} else if(b) {
    if(c) {
        y = 1;
        z = 3;
    } else {
        z=1;
    }
} else if(c) {
    x = 4;
    y = 2;
    z = 7;
}

```

After execution the value of **x=0**, **y=0** and **z=1**. Verify this fact for yourself by tracing through the control paths. Obviously the nesting of control expressions can quickly get out of hand **an** be difficult to interpret. Some programmers try to make the structuring more explicit through decorative placement of the scope brackets, for example,

```

if(a){
    x = 5;
    y = 1;
} else {
    z = 1;
    x = 4;
}

```

These matters are left to the programmer's individual taste. Some find this style a bit too fancy and not contributing that much to readability.

Another possible form of the *condition* statement is an initialized declaration, as follows.

```

double x = 1;
double v = 2.0, y, z;
if(double x = sqrt(v)) {
    y = x*x*x;
    z = x*y;
}

```

In this case the condition is based on the value of **x**. If **x** is assigned a non-zero value, the compound statement following the **if** statement will be executed. The purpose of this

form is to restrict the assignment of the value of **x** to inside the **if** statement's scope. Inside the scope, **x= 1.41421356** and outside the value is **x= 1**.

## The **switch** statement and the **enum** data type

In order to keep complex decision options manageable and readable, the **switch** control statement is a good alternative. The **enum** data type (See Lecture 2, Figure 1) is often used in conjunction with the switch statement and so is introduced here as well.


A programmable form of data type is provided by the **enum** statement. The **enum** type is represented by a finite set of elements with names or labels. The meaning of these names is significant to the programmer and by using the enum labels throughout the program, its intent is easier to interpret. The labels are associated with integer values by the enumeration and thus the enum names can be used in computation as well as in conditional expressions. It is this latter application that is of particular interest in this section.

A simple example of an **enum** declaration follows,

```
enum event_types { null_event, mouse_event, key_event };
```

The enum set has a name, **event\_types**, and the set has three members, **null\_event**, **mouse\_event** and **key\_event**. The meaning of these labels is clearly associated with computer interrupt events that are triggered by the mouse or keyboard. The integer value assigned to each label is, by default: 0 to the first label; and for each additional label the value is incremented by 1. Thus in the example, **null\_event==0** and **key\_event == 2**; The sequence of integers can be altered by setting one or more of the labels to a specific integer,

```
enum event_types { null_event=5, mouse_event, key_event };
```

In this case, **mouse\_event == 6**, and the integer assignment sequence increments as before. 

The enum syntax allows the programmer to manipulate each label assignment as they see fit, as long there are no duplicate assignments. For example, event\_types could be defined as,

```
enum event_types{ null_event='n', mouse_event='m', key_event='k' };
```

In this case, the values could be printed with some meaningful association to the label.

The output of the program,

```
char my_type = null_event;  
cout << "What is my type? Answer " << my_type << endl;
```

produces,

What is my type? Answer n

With the old enum assignment, the output would have been,

What is my type? Answer 0

a much less informative statement. It is emphasized that **enum** elements are each a single integer value, thus a string of characters cannot be a valid **enum** element.

An example of the switch control statement using the **enum** just defined is as follows:

```
event_type e;
...

// e is set by an interrupt
...
switch (e)
{
    case null_event:
    {
        //do something appropriate for null
        break;
    }
    case mouse_event:
    {
        //do something appropriate for mouse event
        ...
        break;
    }
    case key_event:
    {
        // do something appropriate for a key event
        ...
        break;
    }
    default:
    {
        // a unrecognized event has occurred
        ...
    }
}
...// the break statement transfers control to this point
```

There are several parts to the switch statement. The first element is **switch(e)**.

This part of the statement indicates the variable that will control the setting of the switch. In this case, the switch is based on the type of event being received during an event interrupt signal on the computer I/O interface. The **case** part of the statement indicates the beginning of code that is executed for a particular value of **e**. The **case** keyword is followed by the particular value of **e** that causes the program section of this case to execute. The case statement is terminated by **:**. If there is just one line of

computation for a case there is no reason to have explicit scope brackets. However usually there are a series of steps necessary to handle the case, so scope brackets are used.

After the case has been handled, it is necessary to finish with a **break**, which transfers control to the end of the switch statement scope. Without the **break** statement, control would pass to the next case statement. This behavior is sometimes convenient if there are several cases that have the same computation. For example,

```
unsigned x;  
//some value assigned to x  
switch (x)  
{  
    case 0:  
    case 1:  
    case 2:  
        {  
            // computation for cases 0, 1 and 2  
            break;  
        }  
    case 3:  
        ...  
}
```

The switch statement in general has the following syntax.

```
switch ( condition )  
    case constant-expression : statement  
    [default : statement]
```

The **condition** must be capable of being cast to a integer type. The **constant-expression** is any integer value. The code to implement the **case** follows the colon, **:**. It is optional to have a final **default** catch-all case for switch values that are not handled by the specified cases.

## Compound Arithmetic Operators

Several C++ compound arithmetic operators were overlooked in previous Lectures. A compound operator carries out an operation on a variable and then assigns the resulting value to the variable.

The compound operators are defined by example below.

```
double a = 1.0, b = 2.1;  
a += b; // The result is a == 3.1  
a -= b; // The result is a == -1.1  
a *= b; // The result is a == 2.1  
a /= b; // The result is a == 0.47619
```

## The while control loop

A mainstay of computing is the *loop* where the flow of control is repeated until a condition is reached. The simplest form of loop control is the **while** statement. An example of the **while** statement follows.

```
double sum = 0;
unsigned i = 100;
while(--i>0)
    sum += sqrt(static_cast<double>(i));
```

The first aspect of this program to notice is the new operator form, **--i**. This is a unary operation on any type that can be cast to an integer, which subtracts 1 from the operand. This form has the **--** as a prefix. An alternative form is **i--**, the postfix form. The difference between these forms is that with the prefix form the 1 is subtracted before the statement is completed. In the postfix form, the 1 is subtracted after the next operation is performed. For example consider the program,

```
unsigned i = 2;
unsigned j = i--;
unsigned k = --i;
```

After execution, the value of **j** is 2 and **k** is 0. A similar unary operator pair is available for adding 1 to the operand, i.e., **++i** and **i++**.

Referring back to the while loop program, the argument of the while statement must be an expression that can be cast to an integer type. If the integer takes on the value zero then the loop is not executed and control passes to the statement after the end of the scope of the loop. In this case the **bool** expression, **--i>0**, is non-zero until **i=0**. Thus, the summation will add 99 values of **sqrt(static\_cast<double>(i))**, where **i** is in the range **99<i<0**. The resulting sum is **661.462947103**. Note that if the **while** condition is already zero when the statement is first encountered, no loop will occur.

### The continue statement

It can be the case that conditions can arise where processing within the loop scope should be discontinued and the loop started again at the next iteration. Suppose the program from above is modified to only compute the sum of the square roots for the case where the fractional part is less than 0.5. This result can be achieved as follows.

```
double sum = 0;
unsigned i = 100;
while(--i>0)
{
    double v = sqrt(static_cast<double>(i));
    unsigned k = static_cast<unsigned>(v); //next smaller integer
    double f = v-k;//get the fraction
    if(f>=0.5)
        continue;
    sum += v;
```

```
}
```

The **continue** statement has the effect of skipping to the end of the loop to trigger the start of a new iteration. In this case, the result is **342.996580654**.

### The use of **break** in a loop

The **break** statement may be used inside a loop construct to terminate the loop process. As an illustration of its use, suppose it is desired to stop the summation of the previous program when the sum reaches or exceeds a particular value as follows.

```
double sum = 0;
unsigned i = 100;
while(--i>0 && sum < 100.0)
{
    double v = sqrt(static_cast<double>(i));
    unsigned k = static_cast<unsigned>(v); //next smaller integer
    double f = v-k;//get the fraction
    if(f>=0.5)
        continue;
    sum += v;
    if(sum>100)
        break;
}
```

After the loop is terminated by the **break** statement the value of **i=72** and **sum=100.938439119**.

A typical use of the **while** loop is to process an indefinite number of occurrences, where the termination condition is detected within the loop scope and the a **break** is used to go on to the next computation. For example,

```
while(1)
{
    //do some computations
    ...
    bool c = ... //some condition
    if( c )
        break;
}
```

If the condition **c** never occurs (**c** always 0) then an *infinite loop* occurs. Such cases can easily arise and so this form of programming should be avoided if possible, but is sometimes necessary when the number of loop iterations to achieve a task is unknown.

### do while

Another form of the **while** statement is sometimes useful. The keyword **do** is used at the start of a body of computation and terminated with a **while** conditional statement. The following example illustrates this form.



```

unsigned int counter = 5;
unsigned long factorial = 1;
do {
    factorial *= counter--; //Multiply, then decrement.
} while (counter > 0);
cout << "Factorial " << factorial << endl;

```

The output of this program is,

```
Factorial 120
```

The **do while** control form insures that the code inside the **do** scope is executed at least once, regardless of the condition. That is, the code is executed before testing the **while** condition.

### The **for** loop

The **for** loop construct is perhaps the most frequently used statement in C++. The **for** statement is divided into three compartments, separated by a semicolon.

```

for( initial conditions ; condition for termination ; loop updates ) {
    // loop code
}

```

The *initial conditions* compartment is executed before any other part of the **for** statement. The purpose of this compartment is to initialize loop variables prior to the start of the iterations. After the initialization compartment is executed the *condition for termination* compartment is evaluated. If this condition evaluates to **0** then the loop is terminated without proceeding to the final compartment or to the body of the loop. If the condition is non-zero then the loop code is executed. Finally at the end of the loop, the indicated variables in *loop updates* compartment are incremented according to the arithmetic operations. Note that the variables are updated prior to the evaluation of the condition for termination. Note that both the initial conditions and loop updates compartments allow the specification of multiple variables, separated by commas.

An example follows.

```

for( unsigned i = 1, fact=1, j=0 ; i<=5; ++i, ++j) {
    fact *=i;
    cout << "Factorial[" << j <<" ]= " << fact << endl;
}

```

The output of the program is

```
Factorial[0 ]= 1
Factorial[1 ]= 2
```

```
Factorial[2 ]= 6
Factorial[3 ]= 24
Factorial[4 ]= 120
```

Note that **i**, **fact**, and **j** are declared and initialized in the first compartment. The condition to terminate the loop is **i<=5**, followed by a set of loop variable update expressions. If there is no condition specified then the loop will never terminate, as in the following statement, which does nothing forever.

```
for( ; ; );
```

Some programmers like to show off their knowledge of the for loop construct by carrying out all computation inside the **for** specification, e.g.,

```
for( unsigned i = 0, fact = 1; i<5; fact*=(i++ +1) )
    cout << "Factorial[" << i << "]= " << fact << endl;
```

In this case there is no computation at all being done in the body of the loop. All variables are local to the scope of the loop. Note that the loop counter **i** is being updated after its use in computing the factorial product through the use of the post-fix version of the increment operator. The output of this program is the same as before.

## Loop Invariants

*Invariants that can be removed from the loop*

In scientific computing it is important to gain as much efficiency from the code as possible. Therefore it is the responsibility of the programmer not to put computations inside the loop that are the same for each loop. The result of such computations are called *loop invariants*. Loop invariants are also conditions that must be true each time a loop executes. Uncovering such conditions can be an aid in writing programs that behave correctly.

An example of unnecessary computation inside a loop is shown in the following program.

```
unsigned n = 10000000;
double angle = 180, sum = 0;
for(double x =0; x<=angle; x+=angle/n)
{
    double x_radians = x*3.14159/180.0;
    double dx = angle*3.14159/(180*n);
    sum += x_radians*sin(x_radians)*dx;
}
```

This program computes an approximation to  $\int_0^{\text{angle}} x \sin(x) dx$

For **angle=180**, the result should be  $\pi$ . The value of sum after the program executes is **3.1415900566**. Unfortunately, there are many unnecessary computations in this implementation. The increment **angle/n** is computed each time, but is always the same. Likewise **3.14159/180.0**, and **dx** are all invariant within the loop.

The program above takes 5125 msec to execute. However if the program is altered as follows,

```
unsigned n = 10000000;  
double angle = 180.0;  
double angler = (angle*3.14159/180.0), sum = 0.0, dx= angler/n;  
for(double x_radians =0; x_radians<=angler; x_radians+=dx)  
    sum += x_radians*sin(x_radians);  
sum*=dx;
```

the execution time is only 1219 msec. In some cases the C++ compiler is smart enough to detect these useless computations, but it is good coding practice to keep invariants out of the loop.

#### *Invariants to insure correctness*

Another use of loop invariants is to help write correct programs and avoid costly debugging. Suppose the goal is to count how many digits there are in an integer. One problem is that 0 and 1 both have one digit, which complicates testing for termination of a counting program. Consider the following program<sup>1</sup>.

```
unsigned count = 0, n = m; //m is some unsigned integer;  
while (condition) {  
    ++count; n  
    /= 10;  
}
```

It is not immediately clear what expression to implement for the **condition** statement. Since **n** is unsigned, a condition of **n<0** is always false and **n>=0** is always true. The only alternative is **n!=0**, or equivalently **n>0**. Unfortunately, 0 is an integer so with **n=0** as an input, the program would skip any computation and the value of count would be 0, not the correct number of digits in 0.

It is clear that even a simple situation involving loops can be very error prone. The use of a loop invariant will make clear what the condition has to be. Recall that a loop invariant is a property that has some value before the loop starts and its value is not changed by any computation in the loop. Therefore it has the same value after the loop terminates.

One invariant property that immediately comes to mind is the number of digits in **m**, denoted by a function **D(m)**. At any given point in the execution of the full loop **D(n)** is not known, but an invariant relation involving **D** is known. That is,

---

<sup>1</sup> From an article in Dr. Dobb's journal, February 01, 2004, by Andrew Koenig and Barbara Moo

At the beginning of the loop, **count = 0**, and **D(n) = D(m)** as desired. During the loop, **count** is increased by one and **D(n)** is decreased by one, so the sum is a loop invariant as expected. The next observation is that when **n<10**, **D(n) = 1**. In this case,

$\text{count} + 1 = D(m)$ , or  $\text{countp} = D(m)$ , where  $\text{countp} = \text{count} + 1$ .

This result provides the solution to the problem. The correct program is as follows.

```
count + D(n) = D(m)
unsigned countp = 1, n = m; //some unsigned integer;
while (n >= 10) {
    ++countp;
    n /= 10; } //after termination, countp = D(m)
```



Note that  $\text{countp} + D(n)$  is also a loop invariant, where  $\text{countp} + D(n) = D(m) + 1$ .