3 Conditional evaluation of code

3.1 Flow Control

In many tasks, the next instruction will depend on the results of a previous calculation and this means that the actual instructions that are performed can vary when the program is executed at different times. In the previous example, the program will print an error and stop if the file cannot be opened, which could occur because the disk is full.

3.1.1 if

C++ supports two main conditional constructs: if and switch. The general form of the if command is

\[
\text{if(expression) statement;}
\]

\[
\text{else statement;}
\]

where a statement may be a single command, a block of commands, enclosed in braces \{\} or nothing at all; the else command is optional. If expression evaluates to true, anything other than 0, the statement following if is executed; otherwise the statement following else is executed, if it exists. Let’s consider a concrete example:

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

int main()
{
    float a;

    cout << "Please enter a number ";
    cin >> a;

    if(a > 0) cout << a << " is positive\n";
    else cout << a << " is not positive\n";
}
```
The above program takes a user-entered number and determines whether or not it is positive. If-else commands may be nested and this language feature can be used to add a zero test to the above program:

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

int main()
{
  float a;

  cout << "Please enter a number ";
  cin >> a;

  if(a > 0) cout << a << " is positive\n";
  else
    if(a == 0) cout << a << " is zero\n";
    else cout << a << " is not positive\n";
}
```

**Important note** The test for equality is the `==` relational operator. One of the most common programming mistakes is to use the assignment operator `=` instead of the relational operator `==`. If `a=0` cout << a << " is zero\n";

The above code will compile, but it doesn’t do what you think! The expression `a=0` assigns the value 0 to the variable `a` and returns true if successful. Thus, instead of testing for zero, the statement automatically sets `a` to zero.

### 3.1.2 The `?` command

The `?` operator is a shorthand for simple if-else statements. The syntax is

```
expression 1 ? expression 2 : expression 3
```
and if expression 1 is true, then expression 2 is evaluated; if expression 1 is false expression 3 is evaluated.

\[
\begin{align*}
x &= 10; \\
y &= x > 9 \ ? \ 100 : 200;
\end{align*}
\]

In the above example, \(y\) is assigned the value 100; if \(x\) had been less than 9, \(y\) would have the value 200. In if-else terms the same code would be

\[
\begin{align*}
x &= 10; \\
\text{if}(x > 9) \ y &= 100; \\
\text{else} \ y &= 200;
\end{align*}
\]

For clarity, it is generally better to avoid the \(?\) operator and write out the full if-else construction. Nevertheless, you might encounter it in programs written by others.

### 3.1.3 switch

The `switch` command is used to construct multiple-branch selections and tests the value of an expression against a list of integer or character constants. Unlike `if`, `switch` can only test for equality, but it can be useful in certain situations, such as menu operations.

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

main()
{
    char ch;
    double x=5.0, y=10.0;

    cout << " 1. Print value of x\n";
    cout << " 2. Print value of y\n";
    cout << " 3. Print value of xy\n";
```
3 CONDITIONAL EVALUATION OF CODE

```cpp
    cin >> ch;

    switch(ch)
    {
    case '1':
        cout << x << "\n";
        break;
    case '2':
        cout << y << "\n";
        break;
    case '3':
        cout << x*y << "\n";
        break;
    default:
        cout << "Unrecognised option\n";
        break;
    } \End of switch statement
    \End of main function
```

The `case` keyword is used to demark individual tests and the `break` keyword breaks out of the switch command at the end of each test segment. The optional `default` block corresponds to the final else, it executes if none of the tests are true. If the `break` command is omitted then the next command(s) in the `switch` statement will execute. This can be used to produce the same behaviour for many tests. *i.e.*

```cpp
    switch(ch)
    {
    case '1':
    case '2':
    case '3':
        cout << "1, 2 or 3 pressed\n";
```
3.2 Iterative execution of commands — Loops

Almost all programming tasks require the repeated execution of the same, or very similar, commands. Instead of writing the same, or almost, the same instruction several times, it would be much better to write the instruction once and tell the computer to perform it several times, such a construct is known as a loop.

3.2.1 for loops

The most versatile loop construct in C++ is the for loop, which repeats a block of code a number of times until a predefined condition is satisfied. The generic form of this construct is

\[
\text{for} (\text{initialisation}; \text{condition}; \text{increment}) \text{ statement};
\]

where the initialisation command is executed at the start of the first loop. The condition is tested at the top of each loop and if it is true then the loop commands are executed. The increment command is executed at the end of each loop.

```cpp
#include <iostream>

using namespace std;

int main()
{
    for(int i=1; i <= 10; i++) cout << i << i*i << "\n";
}
```

The above example prints the numbers 1 to 10 and their squares. In this case the increment operator ++ has been used as the increment command. The comma operator allows multiple variables to control for loops:

```cpp
#include <iostream>

using namespace std;
```
int main()
{
    int i,j;

    for(i=1, j=20; i < j; i++, j-=5)
    {
        cout << i << " " << j << "\n";
    }
}

produces the output

1 20
2 15
3 10
4 5

The components of a for loop may be any valid C++ commands, or they may be absent, allowing powerful generalisations. For example

for(x=0; x!=100; ) cin >> x;

will run until the user enters 100.

for(;;) cout << "An infinite loop\n";

A for loop with no components is an infinite loop, it can only be terminated by using a break command somewhere in the loop. Time delay loops may be constructed by omitting the body of a for loop e.g.

for(t=0; t < 100; t++); // wait for a while

3.2.2 while and do-while loops

Another type of loop is the while loop, which has the functional form
and continues to iterate until the condition becomes true. The while loop tests the condition at the top of the loop and so the statement will not execute if the condition is initially false. A common use of while loops is for convergence tests

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

main()
{
    double num=50;

    while(num > 0.0001)
    {
        num /= 2.0;
    }

    cout << num << "\n";
}
```

This loop repeatedly halves the variable num until it is less than a specified tolerance.

A do-while loop is similar to a while loop, but performs the test at the bottom of the loop rather than the top. It has the form

```cpp
do{statement;} while(condition);
```

The do loop above could equally well have been a do-while loop:

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

main()
{
```
double num=50;

do
{
    num /= 2.0;
}
while(num > 0.0001);

cout << num << "\n";
}

Remember, a do-while loop always forces at least one execution of the commands in the body of the loop, whereas a while loop does not.

3.3 Jump commands

In certain situations, you might want to jump to a new point in the program skipping over the intermediate instructions. The most common jump command is return which is used to return from functions to the main program, see §4. The break command has already been mentioned when describing the switch construct. It has a more general scope, however, and can be used to force the immediate termination of any loop. For example

for(int i=0; i <= 10; i++)
{
    cout << i;
    if(i==5) break;
}

will only display the numbers 1 to 5 on the screen because break overrides the conditional test

i <= 10 in the for loop.

The “opposite” of the break command is continue, which forces another iteration of the loop, skipping any intermediate code. In a for loop, the increment command is performed and then the conditional test. In while and do-while loops control passes directly to the conditional tests. This can be used to force the earlier evaluation of the condition.
#include <iostream>
using namespace std;

int main()
{
    bool flag=false;
    char ch;

    while(!flag)
    {
        cin >> ch; //read in a character

        if(ch=='x')
        {
            flag = true;
            continue;
        }

        cout << "You have entered the character " << ch << "\n";
    }
}

The preceding program takes input from the keyboard and prints the individual characters entered. If the user enters an x the loop terminates without printing that character. This program also illustrates another common C++ programming practice, using boolean variables as test variables. If flag is false (0), then not flag (!flag) is 1 which is true and the while loop executes. If flag is set to true (1), not flag will be zero, which is false and the loop terminates. If you understand this program, then you should have no problem with relational expressions in C++. 
Functions are the building blocks of C++ programs and, in their simplest form, are merely groups of instructions. In general, however, a function can take a number of variables (arguments), operate upon them and return a value to the part of the program from which the function was called. You should already be familiar with the most important function `main()`. The most general form of a function definition is shown below:

```
return data type function_name( arguments )
{
    body of function
}
```

The `arguments` to a function take the form of a comma-separated list of type definitions i.e. `(double a, int b, ...)`. If the function does not return a value (a subroutine), the return data should be of the type `void`.

Consider now a function `square()` that returns the square of its argument:

```cpp
int square(int a)
{
    return(a*a);
}
```

The function “expects” an integer variable as its argument and returns a integer variable. The `return` keyword causes the function to return to the point in the program from which it was called and sets the return value, if there is one.

In a well-written program tasks that are performed many times should be “packaged away” in functions and it is the functions that are then called repeatedly. The advantage of this approach is that once the function has been written and tested it can be trusted not to be the source of any errors in the program, which facilitates debugging. In addition, if a more efficient method of achieving the same result is found, the function can be changed in one place and the rest of the program will not have to be altered.

The following example shows how to use functions in a C++ program:

```cpp
#include <iostream>
using namespace std;
```
//Declare a function that multiplies two integers
int product(int a, int b)
{
    return(a*b);
}

//main body of program
int main()
{
    int i,j; //Declare two integers

    //Loop i varies from 1 to 10, and j varies from 10 to 1
    for(i=1, j=10; i<=10; i++, j--)
    {
        cout << i << " " << j << " " << product(i,j) << "\n";
    }
}

Note that the choice of variable names used within the product() function is totally independent of any other variables in the program. It is a common misconception that if the arguments of the function are a and b then the function must be called using the variable names a and b.

4.1 Where to define functions

In C++, nested functions are not allowed, which means that functions cannot be defined within other functions. In particular, all functions must be defined outside the main() function. For example, the following is illegal code.

int main()
{
    //Nested function --- illegal
    double square(double x) {return x*x;}
}
square(2.0);
}

A legal version of the above code would be

//Function definition outside main -- legal
double square(double x) {return x*x;}

int main()
{
    square(2.0);
}

In addition, functions must be defined before they can be used. The following is again illegal code:

int main()
{
    //Function called before definition
    square(2.0);
}

double square(double x) {return x*x;}

Actually, we do not have to define the entire function before it is called, we can specify the function prototype — its name, arguments and return type — and delay specification of the function body. The function prototype is defined in the same way as a function but with a semi-colon placed after the closing parenthesis, see below.

//Function prototype (establish the interface)
double square(double x); //Note the semi-colon here
int main()
{
    // Call function (interface defined so OK)
    square(2.0);
}

// Now define body of square function, as usual
double square(double x) { return x*x; }

4.2 Function libraries

The core C++ language is very compact, but much of the power of the language rests in the vast
number of existing libraries, which contain many thousands of specialised functions. We have
already used the I/O libraries, but there are many, many others. Another commonly used library
is the math library, cmath. The standard trigonometric, hyperbolic and exponential functions are
all present: e.g. sin(), exp(), acos(), log(), tanh(). Also of use are the function pow(x,y)
which raises the number x to the power y and sqrt(), the square-root function.

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

// Define our own cube root function
double cbrt(double arg)
{
    double result;
    result = pow(arg,(1.0/3.0));
    return(result);
}

int main()
{

int i;
double x;

// Loop over the numbers 1 to 10
for(i=1; i<=10; i++)
{
    x = i*i*i; // This will transform the integer result to a double
    cout << x << " has the cube root " << cbrt(x) << "\n";
}

This example also illustrates the use of local variables in functions. The variable result is only defined within the function cbrt(), it cannot be accessed from main() and is a form of data encapsulation.

4.3 Modifying the data in function arguments

The functions defined thus far cannot change the value of the arguments passed to them. Consider the following example

#include<iostream>
using namespace std;

void square(int x) { x = x*x; }

int main()
{
    int a=5;
    square(a);
    cout << a << endl;
}
The result of running this program is that the number 5 will be printed on the screen. The variable \texttt{a}, defined in \texttt{main}, has not been changed by the function \texttt{square}. This is because the local variable \texttt{x} is a \texttt{copy} of the value stored in the variable \texttt{a}. The value of the copy is changed within \texttt{square}, but the original remains unchanged. The behaviour of the function is known as \textbf{call by copy} and is the “default” for C++ functions.

If we want the function to be able to modify the value of its arguments we must change this behaviour as shown below

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

void new_square(int &x) { x = x*x; }

int main()
{
    int a=5;
    new_square(a);
    cout << a << endl;
}
```

Now, the number 25 will be printed on the screen. The only difference between the two functions \texttt{square} and \texttt{new\_square} is the introduction of an ampersand, \&, in the definition of the arguments to \texttt{new\_square}. This is an instruction to the C++ to change the default behaviour and to use \textbf{call by reference} rather than \textbf{call by copy}. Now, the original variable, rather than a copy of its value, is directly accessed by the function.

4.4 Function overloading

The function \texttt{square} was written assuming that the data was \texttt{int}, but what happens when we call it with \texttt{double} data.

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

int square(int x) {return x*x;}

int main()
{
    double a=1.5;
    cout << square(a) << endl;
}

The answer is that the double passed to the function will be converted into an integer and the program displays the number 1. Most compilers will issue a warning when compiling the above code.

We must write a new function that calculates the square of a double datum.

double square(double arg) {return (arg * arg);}

In older languages, such as C and FORTRAN, it’s impossible to have two functions with the same name. In C++, you can write many functions with the same name, a property known as function overloading. To avoid confusion, functions with the same name should serve the same purpose, usually acting on different types of variables or classes. The only limitation is that C++ uses the arguments to the function to decide which version of the function to call. Thus, it is impossible to have two functions with the same name and arguments, but different return types.

    int solve_pde(double a);
    float solve_pde(double a); // Illegal
    int solve_pde(double a, int b) // OK

The compiler will determine which version of the function to use when it is called. The following code is illegal in C, but will compile and run in C++.

    #include <iostream>
using namespace std;

int square(int arg)
{
    cout << "Integer square() called " << endl;
    return (arg * arg);
}

double square(double arg)
{
    cout << "Double square() called " << endl;
    return (arg * arg);
}

main()
{
    int i=10;
    double a=20;

    cout << i << " squared is " << square(i) << "\n";
    cout << a << " squared is " << square(a) << "\n";
}
5 Pointers

Pointers are guaranteed to send the novice C/C++ programmer into wide-eyed shock and trembling fear. They will seem strange at first, but represent a very powerful mechanism for handling data. In short, a pointer is nothing more than a variable that holds a memory address, rather than a value. A pointer points to a data value, rather than being a value itself. Think of memory as a sequence of labelled boxes each containing a piece of paper with a number written on it. You can either be told the number directly (a variable) or find it by knowing which box it's in (a pointer).

Many people unfamiliar with pointers often ask why. “Why do I need to worry about pointers in C and not in other languages?” The answer lies in the philosophical differences between languages. In fact, all languages use pointers, but often hide them from the programmer. C gives the programmer the choice of when to use pointers, or not, with the attendant flexibility (and complexity). In practical terms, using pointers generally increases efficiency and permits the easier management of dynamic data structures. Furthermore, pointers must be used if one wants to alter the arguments passed to a function (call by reference).

In C/C++ a pointer is defined by placing a * in front of the variable name e.g. double *p defines a pointer, p, to a double type variable. The following example defines a variable, a, and two pointers, p1, p2. The memory address of a is assigned to the pointer p1 and copied to p2.

```
main()
{
    double a, *p1, *p2;

    a = 10.0;
    p1 = &a; // The &a means 'the address of a'
    p2 = p1; // Copy address of a to p2

    cout << a << " has the address " << p1 << "\n";
    cout << "We can use p1 " << *p1 << " to recover the value of a " << a << "\n";
```
5 POINTERS

a++; // Increment a

cout << "We can also use p2 " << *p2 << " to recover the value of a " << a << "\n";
}

There are two operators associated with pointers: & and *, which are, loosely speaking, inverses. The address of the variable a is obtained by using the command &a, whereas *p gives the value of the variable at the address p, e.g. *p1, *p2, as shown in the program above.

5.1 Pointers and arrays

There is a very close relationship between pointers and arrays. In fact, the array name on its own is a pointer to the first element of the array (the address at which the array starts). Arrays are arranged sequentially in memory and so you can use pointer arithmetic to access array elements: p[1] is the same as *p++. Consider the following example code:

char str[80], *p;

p = str; // Assign the address of the array to p1

// Now we can use p or str to access the array
cout << " We can also express this by " << *(p+4) << "\n";

5.1.1 Dynamic memory allocation and pointers

In many cases, the size of the array will not be known when writing the program. For example, if we are reading in data from a file then the total number of data will depend on the size of the file. In these situations there are two options:

- Static Allocation: Allocate a fixed, but large, amount of storage, double A[1000]; (possibly limited by stack size).
5 POINTERS

- Dynamic Allocation: Determine the exact amount of storage required as part of the program.

In C++ dynamic memory allocation is handled by pointers, which are declared using the * operator.

```cpp
double *A_dynamic;
```

We have declared the variable `A_dynamic` as a pointer to a double, or an address in memory. Declaring a pointer does not actually allocate any storage for a double variable, it merely "points" to an area of memory that can be used to store double variables.\(^1\) The use of pointers for dynamic memory allocation may seem rather convoluted, but pointers have many other uses.

Having declared a pointer, we can use it to allocate storage for as many double data as required. The instruction

```cpp
A_dynamic = new double[100];
```

allocates an array of 100 double variables and the pointer refers to the first of these variables. We use the standard array syntax to access the variables, `A_dynamic[0]` is the first entry in the array, etc. Of course, if we knew that we needed storage for 100 data we could have defined the array statically.

The following program assigns storage for a number of int variables read in from a file, for details on file I/O see §2.4.2.

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

int main()
{
    //Declare and null a pointer to integer data

    int i;
    for(i = 0; i < 100; i++)
    {
        //... allocate storage and initialize
    }
}
```

\(^1\)The introduction of a pointer does introduce an additional memory overhead because we have to store the pointer itself. A bad choice of data structure with multiple pointers can be very wasteful.
int *A_dynamic=0;

cout << "Reading in data from file" << endl;
ifstream in_file("input.dat");

//The number of data must be the first entry in the file
int num_data=0;
in_file >> num_data;

//Allocate storage
A_dynamic = new int[num_data];

//Loop over the number of data and read from file
for(unsigned i=0;i<num_data;i++)
{
    in_file >> A_dynamic[i];
}

//Close the file
in_file.close();

//Calculate the sum of all the data
double sum=0.0;
for(unsigned i=0;i<num_data;i++) {sum += A_dynamic[i];}

//Print out the average value
cout << "Average value " << sum/num_data << endl;

//Free the memory allocated
delete[] A_dynamic; A_dynamic=0;
If the file `input.dat` contains the single line

```
10 1 2 3 4 5 6 7 8 9 10
```
the output from the program is

**Average value 5.5**

Once we have allocated memory for variables using the `new` keyword, then we are in charge of cleaning up when we have finished with it. The keyword `delete` is used to deallocate memory. If an array has been allocated, square brackets must be added after the delete command, e.g. `delete[] array_data`.

If memory has been deleted then it can be reused by any other programs running on the computer. If it is accidentally used by your program again, without reallocation, the result is a nasty run-time error known as a **segmentation fault**. These are particularly hard to track down because if the memory has not yet been grabbed by another program then everything will appear to be fine. For this reason, it is very good practice to “null out” unused pointers. When a pointer is declared, initialise it to null, 0, and when the allocated memory has been deleted reset the value of the pointer to zero.

### 5.2 Pointers and functions

Pointers come into their own when used in conjunction with functions. A function cannot permanently alter the value of an argument unless it is passed as a pointer (call by reference). If the argument is not passed as a pointer, a local copy of the variable is made and destroyed on return from the function (call by value).

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

// Function without pointer
void square(double a)
{
}
```
double result;
result = a*a;
a = result;
cout << " In the function square " << a << "\n";
}

// Function with pointer
void square_pointer(double *a)
{
    double result;
    result = (*a)*(*a);
    *a = result;
    cout << " In the function square_pointer " << *a << "\n";
}

main()
{
    double a = 5;

    //Call function without pointer
    square(a);
cout << " After the function square " << a << "\n";

    //Call function with pointer
    square_pointer(&a); // Note that you need to pass the address of a
cout << " After the function square_pointer " << a << "\n";

}

This program produces the output

%./a.out
In the function square 25
After the function square 5
In the function square_pointer 25
After the function square_pointer 25

The variable a in main is not altered by the function square(), which does not work with pointers. In contrast, a is permanently altered by square_pointer().

In C/C++, arrays are always passed to functions as pointers. Remember that the pointer to an array is merely the name of the array with no square brackets. The pointers may then be used from within the function to access the array elements directly. An array argument to a function may be specified as usual int array[10], as an array of unspecified length int array[] or as a pointer int *array. Note: C/C++ does not do any bound checking so int array[10] is not as safe as it looks!

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

//Dot product of two vectors
double dot_product(int n, double *x1, double x2[]) {
  int i;
  double result=0.0;

  //Loop over the entries of the vectors
  for(i=0;i<=(n-1);i++) {
    result += x1[i]*x2[i];
  }

  return(result);
}
```
main()
{
    double x[3] = {1.0, 1.5, 3.0}, y[3] = {2.0, 5.0, 10.0};
    double product;

    // Call the function
    product = dot_product(3, x, y);
    cout << product;
}

**IMPORTANT NOTE:** When a multi-dimensional array is used as an argument to a function, only a pointer to the first element is passed. In order to find the elements in memory, the compiler needs to know the dimensions of all indices other than the first. The following is illegal

```c
void test_function(double x[][][[]]);
```

and will generate a compiler error. The legal version would be

```c
void test_function(double x[][5][6]);
```

and is rather limiting. Fortunately, using the dynamic memory allocation capabilities in C/C++ one is able to circumvent this restriction, although it’s not entirely trivial.
5.3 Dynamic memory allocation

Pointers are used in dynamic allocation of memory. A pointer to the first entry in an array must be defined before memory can be allocated. C++ makes dynamic allocation very easy with the `new` and `delete` operators.

```cpp
main()
{
  int i;
  double *array; // define a pointer to a double

  array = new double (100.0); // allocate a single double with // initial value of 100.0

  delete array; // free the memory allocated to array

  array = new double [10]; // allocate an array of 10 doubles // cannot specify initial values for arrays

  // Assign values to the array
  for(i=0;i<10;i++) {array[i] = 10.0*(i+1);}

  delete [] array; // Must use delete [] to free arrays
}
```

5.4 Warning!

Pointers are a blessing and a curse. They are absolutely necessary for many programs, but an error with pointers can be very hard to track down. The problem is that if you use a bad pointer, you are reading and writing to an unknown piece of memory. In the worst cases, you can overwrite other portions of your code, or even the operating system. In the best case, you won’t notice anything wrong. A common symptom of pointer problems is when your code runs fine on one
type of machine (e.g. Linux PC), but crashes with a segmentation fault on a different type of machine (e.g. Sun server). **Always** make sure that your pointer points to something before you try and use it. It is often a good idea to null-out pointers (initialise pointers to zero) i.e \texttt{double *p=NULL;} in order to minimise the instances of bad pointers.