2.2.1: Assignment of Values to Variables

This simplest thing you can do to a variable is assign a value to it. You have already seen above that you can assign a value to a variable when you declare it. You can also do this at any point in a program. Following are some examples:

```
int i;
int j,k;
i=0;
i=j;
k=1;
i=j=k;       // also legal
```

2.2.2: Converting Variables

As we will see below when we discuss arithmetic, we sometimes need to convert variables from one type to another. For instance consider the following:

```
float x=2.345;
int i;
i=x;         // assigning a floating point value to an integer
```


The C++ compiler knows how to do simple conversions of this type, but it prefers it if you explicitly tell it that you want to make a conversion. The syntax for doing this is:

```
float x=2.345;
int i;
i=int(x);    // explicitly convert x to type int
x=float(i); // convert i to a floating point number
```

This is generally called a **cast** and the above is an example of **casting** an int to a float.

2.2.3: Constants

In an example above, we declared a variable pi to store the value of π and we assigned it a reasonable value. It is pretty clear to us that we don't want the value of to be allowed to change anywhere in the program, but the compiler doesn't know that. We can however tell it that we would like to keep the value of pi a constant, and not let it be changed. The way to do this is to use the **const** qualifer:

```
// An if-else statement
const int max(50);
int j;
cin >> j;                           // the user types in a number
if (j<max) {
      cout << “OK” << endl;         // print OK if j less than 50
} else {
      cout << “Sorry” << endl;      // print sorry if j is not
}
```

3.1.3: Conditions

What are the conditions we can test on in a program? The following table lists some of the conditions which C++ supports:

Condition	Operator	Explanation
(a < b)	<	True if a is less than b
(a > b)	>	True if a is greater than b
(a <= b)	<=	True if a is less than or equal to b
(a >= b)	>=	True if a is greater than or equal to b
(a == b)	==	True if a is equal to b
(a != b)	!=	True if a is not equal to b
(a && b)	&&	True if a and b are true
(a || b)	||	True if a or b is true

You can make arbitrarily complicated expressions such as the following which will execute if **a** is greater than **b** or **a** is identical to **c**:

```
if ((a>b) || (a==c)) {
      i=i*2;      // Whatever you want to execute if condition is true
}
```

One extremely common programming mistake is the following:

```
if (a=b)  cout << “I didn’t mean to do this” << endl;       // wrong
if (a==b) cout << “What I really meant to do” << endl;      // right
```

The first statement is allowed, but what it does is first set the value of **a** equal to the value of **b**. It then checks the value of **a** to see whether it is equal to the value that C++ uses by convention to indicate true before deciding what to do. There are cases where this is what you want to do, but it is unlikely that you will need to do this. Watch out for this bug!

3.3: How Long Are Variables Valid?

As was discussed above, variables can be defined at any point in your program and so are automatically made during the program execution only when they are needed. However, these variables are also removed from memory after they have been used. This means variables only have a limited range of validity within your code and this is called their **scope**. Consider the following example:

```
const int max(50);
int j;
cin >> j;                       // the user types in a number
if (j<max) {
      int k=j*j;                // k equals the square of j
      cout << k << endl;        // prints square of j, if j less than 50
}
```

Here, the variable k is only created if j is less than 50, otherwise it never exists. It therefore makes sense that trying to use k **after** the end of the *if* statement could give problems, if the value of j the user typed in happened to be 50 or more. To avoid such problems, then if k was created, it is removed from memory when the program execution reaches the end brace of the *if* statement. It is said to go **out of scope** at this point. Any attempt to use it later will give a compilation error; hence the following is not allowed:


```
const int max(50);
int j;
cin >> j;                       // the user types in a number
if (j<max) {
      int k=j*j;                // k equals the square of j
}
cout << k << endl;              // does not compile
```

Does this mean we cannot use any variable called k in the rest of the program for any other purpose? No! Although it is not recommended as it can be very confusing, we are allowed to do the following:

```
const int max(50);
int j;
cin >> j;                       // the user types in a number
int k=j*j*j;                    // k equals the cube of j;
if (j<max) {
      int k=j*j;                // k equals the square of j
}
cout << k << endl;              // always writes out the cube of j
```

we would find that the **read_in** function would not change any of its arguments as it is only able to change copies of the arguments.

5.1.2: Knowing Where The Data Are

If we were able to tell our **read_in** function where in memory our variables **my1**, **my2** and **my3** were instead of simply what the values of those variables were, the **read_in** function could use that knowledge to change the values of the variables in the main function. The way we will do this is to use a pointer.

5.1.3: What is a Pointer?

A pointer is type of variable which is able to store the location of another variable. For example a pointer to an int is a variable which can contain the location in the computer's memory where an integer variable is stored. Let's look at a couple of simple examples to see how to define and use a pointer variable:

```
int j;                  // j is defined to be an integer variable
j=1;                    // set the value of j to 1
int *pj;                // pj is defined to be a pointer to an integer
pj=&j;                  // pj is now pointing to j
cout << j << endl;      // print value of j, which is 1
cout << *pj << endl;    // print value pointed to by pj, which is 1
*pj=2;                  // sets j to 2
cout << j << endl;      // print the value of j, which is now 2
```


There is a lot of new syntax in this example, and it is a bit tricky. Lets go through it line by line, and with the help of a table showing a small section of the computer memory. The situation after the first five lines have executed might be as follows:

Memory Location	Variable Name	Variable Value
4004		
4008	j	1
4012		
4016	pj	4008
4020		

We declare a new variable j on the first line. When we do this, the computer sets aside some memory for the variable; in our example, four bytes at location 4008. The value of j at this point is whatever happened to be in this location beforehand and so is completely unpredictable. We then assign j a value of 1 on the second this case ip1) to the second pointer (ip2) so that they now both point at the same address, and hence the same variable.

5.2: References

C++ offers one other method of accessing memory locations. This is through **references**, which are very similar to pointers but have a different syntax. They hide much of the explicit use of pointers and so can be more convenient. However, they are more restricted in what they can be used for. If you would like to know more about this, then you should look up their use in a C++ book.

To define a reference, then we use a **&** (rather than a ***** as for pointers) but otherwise references are used very like normal variables. Below is an example, similar to the one above, of how to use this syntax:

```
int x=1,y=2;                  // two integers
int &ir1=x,&ir2=y;            // two references to int
                              // ir1 points to x, ir2 points to y
y=ir1;                        // y is now 1
ir1=0;                        // x is now 0
```


The main limitation here is that references must point to a variable immediately and which variable they point to cannot be changed later, hence several lines of the previous example have no equivalent here.

By using references, we don't have to explicitly use pointers and the code looks like we are using normal variables. The C++ compiler takes care of the references for us. Arguments which are references behave in the way which FORTRAN programmers expect arguments to behave.

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

void main(void) {
      ofstream myfile("poly.txt");
      float dx=0.10;
      float x,poly;
      for (int i=0;i<100;i++) {
            x=dx*i;
            poly=3.0*x-2.2*x*x+0.2*x*x*x;
            myfile << x << '\t' << poly << endl;
      }
}
```

To use an **fstream**, we first need to attach the stream to a particular file. We do this with the stream **constructor**. The **constructor** is a very important part of C++ which will be covered in detail in the 2nd year course, but the basic idea is that it is just a way of initialising a variable (often called an object when it is a user defined type). In this case the object we wish to initialise is the output file stream which is an object of the type **ofstream**. The name of our stream variable is **myfile** and we initialise it to connect to the file we wish to use, with the first statement in the main program above. This creates a new variable which is an output stream that flows to our file instead of to the terminal. We can then cause output to go to this file in a similar way to how we send output to the screen using the **<< operator**, as shown in the last line.

Similarly you can read in from files using streams of type **ifstream**.

8.4: Dynamic Allocation of Memory

Quite often you will want to write a program that allows the user to choose parameters which affect how much memory your program needs. For example you may want the user to be able to choose the size of an array. Dynamic memory allocation lets you allocate space for arrays as the program is running.

The **new** operator is used to dynamically allocate space for variables. The following code demonstrates its use:

```
float *fltptr=new float;        // allocate a single float variable
double *efield;                 // declare a pointer
efield=new double[100];         // dynamically allocate an array
int sz;
cin >> sz;                      // let user specify array size
int *myarray=new int[sz];       // dynamically allocate array
```

The **new** operator returns a pointer to the object you have asked it to allocate.