2. New features in C++

C++ is the object-orientated version of C (taking its name from the increment operator in C). It is defined as a superset of C so that all the C language is included in C++, although much of it is surpassed by more effective methods in C++. As well as introducing OO features, C++ also contains some general improvements in the languages (for example in C++ // denotes a comment to the end of the line).

Classes

Classes in C++ are implemented in a manner very similar to structures (many C++ compilers work by translating the code into C and then compiling it - classes are translated into structures). They consist of a series of variable and function declarations. These are then local to the class and are accessed in the same way as members of a structure when an instance of the class is created. For example:

```
class blackBox { ------------------------------------------------  Class defined
private:          // Don't worry what this does just yet
      int a;
      int b;
public:           // Don't worry what this does just yet
      int set(char which, int value) {
            if (which == 'a') {
                  a = value;
                  return 1;
                  } else if (which == 'b') {
                  b = value;
                  return 1;
                  } else {
                  return 0;
                  }
            }
      int get(char which) {
            if (which == 'a') {
                  return a;
            } else if (which == 'b') {
            return b;
            } else {
            return 0;
            }
      }
};

/* Note that like structures, classes require a semi-colon after the closing
curly bracket */

/* It is legal to have another function called get as the other get function is
within the class */
int get(char which) {
      return 0;
}

int main() {
      blackBox myBox; ------------------------------------------  Instance of class created
      int current = 1;

      myBox.set('a',1);
      current = myBox.get('a');     // Current is now equal to 1
      current = get('a');           // Current is now equal to 0
}
```

New and Delete operators

C++ contains in-built dynamic memory operations - rather than using *malloc*, *calloc* and *free* in C++ you use the `new` and `delete` operators. Using `new` and `delete` is a lot simpler than *malloc* and *calloc*, for example:

```
int main() {
int *size;
int *array;

size = new int;         // Returns a pointer to a dynamic allocated new integer
*size = 5;
array = new int[*size]; // Returns a pointer to an array of 5 integers

//... Rest of program
delete size;            // Frees the memory pointed to by p;
delete []array;         // The [] are required to tell the compiler that it is
an
                        // array that is being deleted, not just a single
                  variable
}
```

`New` and `delete` allow for much more intuitive and 'cleaner' use of dynamic memory, especially when used with class constructors and destructors and operator overloading (all covered later on).

Streams and File I/O

Rather than using C style I/O routines, C++ defines some classes to deal with I/O and some standard variables corresponding to *stdout*, *stdin* and *stderr*. In shift operators are used to input and output variables, for example:

```
#include <iostream.h>
// The header file for C++ I/O

int main() {
int i;
char c;
char *string;

string = new [200];
cin >> i;   // Equivalent to fscanf(stdin,"%d",&i);
cin >> c;   // Equivalent to fscanf(stdin,"%c",&c);
cin.getline(string,200,'\n');
/* Inputs a line to the given buffer (string) of up to the given size of the
buffer (200 - 1 for the '\0') or the given character ('\n') */

if (string[0] == '\0') {
      cerr << "Blank line entered.\n";
      // Equivalent to fprintf(stderr,"Blank line entered.\n");
      }
      cout << string << ' ' << c << ' ' << i << '\n';
      // Equivalent to fprintf(stdout,"%s %c %d\n",string,c,i);
}
```

Not only is this new notation clearer but it is also much more easily extended to cover new types than C style I/O. Files work in a very similar manner, the types `ifstream` and `ofstream` representing input and output files respectively. For example:

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

```
int main(int argc, char **argv) {
ifstream in;
ofstream out;
char temp;

// Open the files
in.open(argv[1]);
out.open(argv[2]);

// Check they are working
if (in.bad()) {
      cerr << "Can't open file: " << argv[1] << '\n';
      return 0;
      }
      if (out.bad()) {
      cerr << "Can't open file: " << argv[2] << '\n';
      return 0;
      }

      // Copy across the data
      while (!in.eof()) {     // While not at EOF
      in >> temp;
      out << temp;
      }

      // Close the files
      in.close();
      out.close();
      return 1;
}
```

Reference variables

One non-OO feature introduced in C++ is reference variables. These are declared as pointers, but use '&' instead of '*', they behave as 'half pointers', they contain the address of a variable, but they don't need a special operator to access them and can only ever point at one object. They are best thought of as 'another name for', for example:

```
int main() {
int a = 1;
int b = 2;
int &ref = a;

if (ref == a) {
      // This code is always executed
      // ...
      }
      ref = b;
      // Sets the value of a to the value of b (2)
      // not ref to refer to b;
      a = 1;      // Reset a
      if (ref == b) {
      // This code will never be executed as ref still refers to a
      // ...
      }
      return 1;
}
```

The main use of reference variables is in functions, where they make call by reference much easier than having to continually use pointers:

```
void normalSquare(int a) {    // Clearly useless
a = a * a;
}

void referenceSquare(int &a) {
a = a * a;
}

/* Far more useful as when the function is called, the reference is intialised
to refer to the parameter, thus making the function equivalent to: */
void pointerSquare(int *a) {
*a = *a * *a;
}

referenceSquare(v);
pointerSquare(&v);
```

Reference variables don't really increase the power of the language; anything you can do with reference variable you could do with pointers in C, but they do make it a lot simpler and the code more legible. Also when passing classes, passing a constant reference to them is far faster than passing the actual class (see copy constructors), regardless of whether you want to pass by value or by reference.

It is possible to mix pointers and reference variables (i.e. `int &*p` are reference to an integer pointer), but as with other features - "just because you can doesn't mean you should"; it is one way of creating very confusing programs very quickly and should be avoided.

Function Overloading

In C++ the compiler differentiates between functions not only on the basis of name but also on the arguments that they take, so it is legal to have two functions with the same name, as long as they will always take different arguments, for example:

```
// Legal
int quadratic (int terms, int *coefficients, int value) {
int total = 0;
int x = 1;
int i;

for (i = terms - 1, i <= 0; ++i) {
      total += coefficients[i] * x;
      x *= value;
      }
      return total;
}
```

```
float quadratic (int terms, float *coefficients, float value) {
float total = 0;
float x = 1;
int i;

for (i = terms - 1, i <= 0; ++i) {
      total += coefficients[i] * x;
      x *= value;
      }
      return total;
}

// Illegal
int input(FILE *target);
float input(FILE *target);
// Functions have the same arguments, return type is irrelevant
char convert(char value = '\0');
int convert(int value = '\0');
// A call to convert() is ambiguous
```

Boolean Values

C++ defines a new primitive type, `bool` and the corresponding keywords `true` and `false` to go with it. On early systems this was defined as follows:

```
typdef int bool
#define true 1
#define false 0
```

although more modern systems will actual implement them as a separate type. It is worth using booleans because they make the code more conceptually clear, for example:

```
int full(class bucket); // Does it return 1/0 or a percentage?
bool full(class bucket);// Much more explicit
```

If you then have problems with old compilers you can add the typedef and preprocessor directives.

Task

1. Try re coding some old C programs using the new features in C++, all hybrids are legal C++ as the standard defines it to include all the C language.
2. Try recoding some C structured data types in C++ using classes to gather all the info in one place.
3. Is there any difference in the following sections of code?

   ```
   cout << 1 << 2;
   (cout << 1) << 2;
   cout << (1 << 2);
   ```