Functions

Functions

- Named blocks of code
  - May take parameters, and/or return values

This function accepts an int as an input argument, and returns a double back to the calling code

return is used to specify the value passed back to the caller. It also terminates the function

This function does not return a value, so the return type is given as void

```
double factorial( int num )
{
        double result = 1;
        for ( int i = 2; i <= num; i++ )
                result *= i;
        return result;
}

void multiply( double num )
{
        cout << "Result is " << num * 2;
}
```

Calling Functions

- Call functions using their name

Here the function is used in place of a simple value. It could also have been used in an assignment, such as

```
double res;
res = factorial( n );
```

```
int main()
{
        int n;
        cout << "Number to factorialize: ";
        cin >> n;
        cout << "Factorial is " << factorial( n ) << "\n";
        double val;
        cout << "Number to multiply by 2: ";
        cin >> val;
        multiply( val );
        return 0;
}
```

Default Function Arguments

- C++ allows a default value to be used when the argument is not supplied on call

```
void multiply( double num, double multiplier )
{
        cout << "Result is " << num * multiplier;
}
```

Function implementation, .cpp file.

```
void multiply( double num, double multiplier = 2 );
```

Function prototype, .h file.

If the 2nd argument is not specified when the function is called, the value 2 given here is used.

Overloading

- Functions may be overloaded, based on number and/or types of arguments

```
void divide( double num, double divisor )
{
    cout << "Result is " << num / divisor;
}

void divide( int num, int divisor )
{
    cout << "Result is " << num / divisor;
}
```

divide(3.0, 2.0); ➡ 1.5

divide(3, 2); ➡ 1

The types of all the arguments are taken into account in deciding which overload to call, but the return type is ignored.

Argument Passing

- Default behaviour is pass by value
  - Function receives local copy of argument on call
  - Any changes are not visible to caller

```
void divide( double num, double divisor )
{
        num = num / divisor;
        cout << "Result is " << num << "\n";
}
```

Calling it and displaying the argument:

```
double val;
cout << "Number to divide by 2: ";
cin >> val;
divide( val, 2 );
cout << "val is " << val << "\n";
```

```
Number to divide by 2: 4
Result is 2
val is 4
```

val, the argument passed, has not been changed by the function.

Argument Passing

- C++ supports pass by reference
  - Function receives reference to actual parameter
  - Changes are reflected in original

```
void divide( double & num, double divisor )
{
        num = num / divisor;
        cout << "Result is " << num << "\n";
}

double val;
cout << "Number to divide by 2: ";
cin >> val;
divide( val, 2 );
cout << "val is " << val << "\n";
```

```
Number to divide by 2: 4
Result is 2
val is 2
```

The value of the variable val has been changed by the function.

const

- const keyword indicates value of a variable cannot change
  - Requires initialisation
- Can also be applied to function parameters
  - Indicates that function will not change the value of the argument
- Use of const is recommended wherever possible

```
void divide( const double & num, double divisor )
{
        num = num / divisor;
        cout << "Result is " << num << "\n";
}
```

This line generates a compiler error of "l-value specifies const object" as an attempt is made to change a const argument.

Local Variables

- C++ supports block scoping of variables

```
void fred()
{
        int n = 4;
        int m = 7;
}

void joe()
{
        int n = 2;
        n = m;
}
```

This variable n has no relationship with the variable also named n in fred.

Gives a compiler error:
'm' : undeclared identifier

Lifetime

- Scoping defines rules for visibility of variables
- Lifetime defines how long values are preserved
  - Not always the same as scope

```
void countup()
{
        int counter = 0;
        counter++;
        cout << "Count is: " << counter << endl;
}
```

```
Count is: 1
Count is: 1
Count is: 1
Count is: 1
Count is: 1
Count is: 1
```

This time count is given a value, but is reinitialized to this value each time the function is called.

Local variables defined in this way are sometimes called 'automatic' variables as they are created automatically on the stack as the function is called, and then removed as the function exits. They are efficient in memory use as they only exist while they are being used, but because they are recreated each time the function is called they get reinitialized.

static

- Static alters default lifetime of a variable
  - Preserves values between calls
  - Even if variable has local (block) scope

```
void countup()
{
        static int counter = 0;
        counter++;
        cout << "Count is: " << counter << endl;
}
```

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
```

static variables defined in this way become part of the permanent data of the application. The initial value is set once, at some point before the variable is first used.

Global Symbols

- Variables may be defined outside a function
  - Global variables
- Scope and lifetime are equivalent

```
int counter = 0;
void countup()
{
        counter++;
        cout << "Count is: " << counter << endl;
}
void resetCounter()
{
        counter = 0;
}
```

'Global' variable declared outside of function.

- Use of global variables is not encouraged

External Symbols

- Global variables may be shared with code in other source files
  - Use extern keyword to specify use of an externally defined variable
  - Important for linking of different files into a single executable

File1.cpp

```
int counter = 0;

void countup()
{
        ...
```

File2.cpp

```
extern int counter;

void resetCounter()
{
        counter = 0;
}
```

Using static with Global Symbols

- Keyword static applied to a "global" symbol means the symbol has file scope
  - Can be seen throughout source file
  - Can not be seen from an other source file
  - "Hides" names

```
static int counter = 0;
void countup()
{ ...
```

counter is now only available to functions within the same file. Other source files could have their own variables called counter, but this causes confusion and should be avoided.