

Functions

Functions allow us to split a program up into different sub-tasks.

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

int hello();

int
main()
{
    hello();
    return (0);
}
int
hello()
{
    cout << "hello" << endl;
    return (0);
}
```

- this program illustrates a very simple function. In this case the function is called `hello()` and it prints out the word `hello` on the screen
- the function is declared at the top of the program before `int main()`
- when you declare a function you have to give its type, just like a variable. In this case `hello()` is of type `int` but functions can also be of type `float` or any other variable type
- the name of the function is always followed by a pair of round brackets e.g. `hello()`
- the function itself is written after the main program - after the last curly bracket. It starts with the type of the function (in this case `int`) followed by the name of the function (in this case `hello()`).
- the body of the function is written between curly brackets and finishes with `return (0);` with what it returns matching its type

Functions with Arguments

Example of 1 argument:

- we pass information from main to a function by using arguments:

```
#include <iostream>
using namespace std;
int square(int);
```

```
int
main()
{
    int x = 4, y = 5;
    square(x);
    square(y);
    return (0);
}
// Definition of function square:
int
square(int z)
{
    cout << "the square is " << z * z << endl;
    return (0);
}
```

- here we have created a function called `square()` which takes an integer argument. The argument appears in the round brackets following the name of the function.
- in this program the variables `x` and `y` are given as arguments to `square()`
- the function `square()` contains within itself a variable called `z`. This variable only exists within `square()`. It cannot be used by any other part of the program. Likewise `x` and `y` exist only in the main program. They cannot be used by `square()`
- when the line `square(x);` appears in the main program the computer copies the value of `x` in the main program to the value of `z` in `square()`. The function then carries out operations on `z` - in this program it prints the value `z*z` to the screen.
- we can call a function as many times as we like. Each time we can give it a different argument. For example, in the previous program we called `square()` twice. The first time we gave the variable `x` as the argument (*to get 16*). The second time we gave the variable `y` (*to get 25*).
- each time the computer copied the value of the argument to `z` and carried out the same operations on `z`

Example of 2 arguments:

- a function can have more than one argument. Here is a function called `add()` which has two:

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

int add(int, int);

int
main()
{
    int x = 2, y = 3;
    add(x, y);
    return (0);
}

// Definition of the add function
int
add(int p,int q)
{
    int sum;
    sum = p + q;
    cout << "the sum is " << sum << endl;
    return (0);
}
```

- this function takes two integer arguments and prints out their sum
- when we declare the function above the main program we have to give the type of the arguments in the round brackets after the name of the function

```
int add(int, int);
```

How to return data from the function to the main program

- we have seen how to pass data from the main program to the function via arguments. We can pass data back from the function to the main program via the `return` statement

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

int square(int);
int
main()
{
    int x = 4, y = 5, xsqu, ysqu;
    xsqu = square(x);
    ysqu = square(y);
    cout << "the square is " << xsqu << endl;
    cout << "the square is " << ysqu << endl;
    return (0);
}
int
square(int z)
{
    return (z * z);
}
```

- here we have modified the function `square()` so that it returns the square of its argument to the main program. We can use statements like

```
xsqu = square(x);
```

to pass the data from the function to variables in the main program

- if the function returns an integer value to the main program then the function is declared as type integer. That's why in the above program `square()` is declared by the statement

```
int square(int);
```

A function of type `float`

```
#include <iostream>

using namespace std;

float square(float);

int
main()
{
    float x = 3.5, xsqu;

    xsqu = square(x);

    return (0);
}

float
square(float z)
{
    return (z * z);
}
```

- here we have modified `square()` so that it accepts a `float` argument and returns a `float` value to the main program. Notice how the declaration statement

```
float square(float);
```

now uses `float` instead of `int`

- it is also possible to have functions of other types

Using Arrays as Arguments

The following program creates a function `sum_array()` which uses a one-dimensional array as an argument

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

const int ARR_SIZE = 5;

int sum_array(int [ARR_SIZE]);

int
main()
{
    int arr[ARR_SIZE] = {1,2,3,4,5}, sum;
    sum = sum_array(arr);
    cout << "the sum is " << sum << endl;
    return (0);
}

int
sum_array(int bar[ARR_SIZE])
{
    int sum, i;

    sum = 0;
    for(i = 0; i < ARR_SIZE; i++) {
        sum += bar[i];
    }
    return (sum);
}
```

- in the function declaration at the top of the program we include the size of the array in square brackets

```
int sum_array(int [ARR_SIZE]);
```

- strictly speaking it's not necessary to actually write the size: we could just write

```
int sum_array(int []);
```

leaving the square brackets empty but for the time being it's better if you include the size

Using a 2D Array as an Argument

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

```

const int NO_ROWS = 5;
const int NO_COLS = 5;

int print_array(int [NO_ROWS] [NO_COLS]);

int
main()
{
    int arr[NO_ROWS] [NO_COLS] = { {1,2,3,4,5},
                                   {1,2,3,4,5},
                                   {1,2,3,4,5},
                                   {1,2,3,4,5},
                                   {1,2,3,4,5}};
    print_array(arr);
    return (0);
}

int
print_array(int bar [NO_ROWS] [NO_COLS])
{
    int i, j;

    for (i = 0; i < NO_ROWS; i++) {
        for (j = 0; j < NO_COLS; j++) {
            cout << bar[i][j] << " ";
        }
        cout << endl;
    }
    return (0);
}

```

- here we have to give the number of rows and columns in the function declaration

```
int print_array(int [NO_ROWS] [NO_COLS]);
```

strictly speaking it's not necessary to supply the number of rows but for the moment it is better if you do

Does altering the value of the argument in the function affect any variables in the main program?

- The following program uses a function with a single integer argument. The value of the argument is changed within the function

```
#include <iostream>
```

```
using namespace std;

int add_one(int);

int main()
{
    int x = 3;

    add_one(x);
    cout << "in the main program x = " << x <<
endl;
    return (0);
}

int
add_one(int x)
{
    x++;
    cout << "in the function x = " << x <<endl;
    return (0);
}
```

- the variable `x` in the main program and the variable `x` in the function are two different variables. When the function `add_one()` is called the value of `x` in the main program is passed to `x` in the function but after that any alterations to `x` in the function do **not** affect `x` in the main program.

- the above program would print out

```
in the function x = 4
in the main program x = 3
```

Functions can alter the values of arrays in the main program

- There is a big difference between the way functions treat array arguments vs. ordinary variable arguments
- on the previous slide we saw that if a variable is passed as an argument to a function then the function cannot alter the value of the variable in the main program.
- this is not true of array arguments: functions can alter the values of arrays in the main program

```
#include <iostream>
using namespace std;
int add_one(int [5]);
int
main()
{
    int i, arr[5] = {1,2,3,4,5};
    add_one(arr);
    for (i = 0; i < 5; i++) {
        cout << arr[i] << " ";
    }
    return (0);
}
int
add_one(int bar[5])
{
    int i;
    for(i = 0; i < 5; i++) {
        bar[i]++;
    }
    for (i = 0; i < 5; i++) {
        cout << bar[i] << " ";
    }
    cout << endl;
    return (0);
}
```

- here an array is passed to a function and each member is incremented by one. This change will also affect the array back in the main program. The above program would print out:

```
2 3 4 5 6
2 3 4 5 6
```