Functions

Functions

A **function** is a group of statements that perform a particular task.
You may define your own functions in C++.

Using functions can have many advantages, including the following:
- You can reuse the code within a function.
- You can easily test individual functions.
- If it's necessary to make any code modifications, you can make modifications within a single function, without altering the program structure.
- You can use the same function for different inputs.

> Every valid C++ program has at least one function - the **main()** function.

The Return Type

The **main** function takes the following general form:

```
int main()
{
  // some code
  return 0;
}
```

A function's **return type** is declared before its name. In the example above, the return type is **int**, which indicates that the function returns an integer value.
Occasionally, a function will perform the desired operations without returning a value. Such functions are defined with the keyword **void**.

> **void** is a basic data type that defines a valueless state.

Defining a Function

Define a C++ function using the following syntax:

```
return_type function_name( parameter list )
{
   body of the function
}
```

return-type: Data type of the value returned by the function.
function name: Name of the function.
parameters: When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function.
body of the function: A collection of statements defining what the function does.

> Parameters are **optional**; that is, you can have a function with no parameters.

Defining a Function

As an example, let's define a function that does not return a value, and just prints a line of text to the screen.

```
void printSomething()
{
 cout << "Hi there!";
}
```

Our function, entitled **printSomething**, returns **void**, and has no parameters.
Now, we can use our function in **main()**.

```
int main()
{
  printSomething();

  return 0;
}
```

Try It Yourself

To **call** a function, you simply need to pass the required parameters along with the function name.

Functions

You must declare a function prior to calling it.
For example:

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

void printSomething() {
 cout << "Hi there!";
}

int main() {
 printSomething();

 return 0;
}
```

Try It Yourself

Putting the declaration after the **main()** function results in an error.

Functions

A function **declaration**, or **function prototype**, tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.
For example:

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

//Function declaration
void printSomething();

int main() {
  printSomething();

  return 0;
}

//Function definition
void printSomething() {
  cout << "Hi there!";
}
```

Try It Yourself

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Function Parameters

For a function to use **arguments**, it must declare formal **parameters**, which are variables that accept the argument's values.

For example:

```
void printSomething(int x)
{
   cout << x;
}
```

This defines a function that takes one **integer** parameter and prints its value.

Formal parameters behave within the function similarly to other local variables. They are created upon entering the function, and are destroyed upon exiting the function.

Function Parameters

Once parameters have been defined, you can pass the corresponding arguments when the function is called.
For example:

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

void printSomething(int x) {
  cout << x;
}

int main() {
  printSomething(42);
}

// Outputs 42
```

Try It Yourself

The value 42 is passed to the function as an argument, and is assigned to the formal **parameter** of the function: **x**.

Making changes to the parameter within the function does not alter the argument.

Function Parameters

You can pass different arguments to the same function.
For example:

```
int timesTwo(int x) {
   return x*2;
}
```

The function defined above takes one integer parameter and returns its value, multiplied by 2.

We can now use that function with different arguments.

```
int main() {
  cout << timesTwo(8);
  // Outputs 16

  cout <<timesTwo(5);
  // Outputs 10

  cout <<timesTwo(42);
  // Outputs 84
}
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Multiple Parameters

You can define as many parameters as you want for your functions, by separating them with **commas**.

Let's create a simple function that returns the sum of two parameters.

```
int addNumbers(int x, int y) {
 // code goes here
}
```

As defined, the **addNumbers** function takes two parameters of type int, and returns int.

Data type and **name** should be defined for each parameter.

Multiple Parameters

Now let's calculate the sum of the two parameters and return the result:

```
int addNumbers(int x, int y) {
  int result = x + y;
  return result;
}
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Multiple Parameters

Now we can call the function.

```
int addNumbers(int x, int y) {
  int result = x + y;
  return result;
}

int main() {
  cout << addNumbers(50, 25);
  // Outputs 75
}
```

Try It Yourself

You can also assign the returned value to a variable.

```
int main() {
  int x = addNumbers(35, 7);
  cout << x;
  // Outputs 42
}
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Multiple Parameters

You can add as many parameters to a single function as you want.
For example:

```
int addNumbers(int x, int y, int z, int a) {
  int result = x + y + z + a;
  return result;
}
```

Try It Yourself

If you have multiple parameters, remember to separate them with **commas**, both when declaring them and when passing the arguments.

Random Numbers

Being able to generate **random** numbers is helpful in a number of situations, including when creating games, statistical modeling programs, and similar end products.

In the C++ standard library, you can access a pseudo random number generator function that's called **rand()**. When used, we are required to include the header **<cstdlib>**.

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

int main() {
  cout << rand();
}
```

Try It Yourself

This will output a random number.

Random Numbers

A **for** loop can be used to generate multiple random numbers.

```
int main() {
  for (int x = 1; x <= 10; x++) {
    cout << rand() << endl;
  }
}
```

```
/* Output:
41
18467
6334
26500
19169
15724
11478
29358
26962
24464
*/
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Random Numbers

Use the **modulo** (%) operator to generate random numbers within a specific range.
The example below generates whole numbers within a range of 1 to 6.

```
int main () {
 for (int x = 1; x <= 10; x++) {
 cout << 1 + (rand() % 6) << endl;
 }
}

/* Output:
6
6
5
5
6
5
1
1
5
3
*/
```

Try It Yourself

However, the **rand()** function will only return a pseudo random number. This means that each time the code is run, it generates the same numbers.

The srand() Function

The **srand()** function is used to generate truly random numbers.
This function allows to specify a **seed** value as its parameter, which is used for the **rand()** function's algorithm.

```
int main () {
 srand(98);

 for (int x = 1; x <= 10; x++) {
  cout << 1 + (rand() % 6) << endl;
 }
}
```

Try It Yourself

Changing the seed value changes the return of rand(). However, the same argument will result in the same output.

Truly Random Numbers

A solution to generate truly random numbers, is to use the current time as a seed value for the srand() function.
This example makes use of the **time()** function to get the number of seconds on your system time, and randomly seed the rand() function (we need to include the <ctime> header for it):

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

int main () {
 srand(time(0));

 for (int x = 1; x <= 10; x++) {
  cout << 1 + (rand() % 6) << endl;
 }
}
```

Try It Yourself

time(0) will return the current second count, prompting the **srand()** function to set a different seed for the **rand()** function each time the program runs.

Using this seed value will create a different output each time we run the program.

Default Values for Parameters

When defining a function, you can specify a **default** value for each of the last parameters. If the corresponding argument is missing when you call a function, it uses the **default** value.

To do this, use the assignment operator to assign values to the arguments in the function definition, as shown in this example.

```
int sum(int a, int b=42) {
 int result = a + b;
 return (result);
}
```

This assigns a default value of 42 to the **b** parameter. If we call the function without passing the value for the b parameter, the default value will be used.

```
int main() {
  int x = 24;
  int y = 36;

  //calling the function with both parameters
  int result = sum(x, y);
  cout << result << endl;
  //Outputs 60

  //calling the function without b
  result = sum(x);
  cout <<  result << endl;
   //Outputs 66

  return 0;
}
```

Try It Yourself

The second call to the function does not pass a value for the second parameter, and the default value of 42 is used, instead.

Using Default Arguments

Another example:

```
int volume(int l=1, int w=1, int h=1) {
  return l*w*h;
}

int main() {
  cout << volume() << endl;
  cout << volume(5) << endl;
  cout << volume(2, 3) << endl;
  cout << volume(3, 7, 6) << endl;
}

/* Output
1
5
6
126
*/
```

Try It Yourself

As you can see, default parameter values can be used for calling the same function in different situations, when one or more of its parameters are not used.

Overloading

Function **overloading** allows to create multiple functions with the **same name**, so long as they have different parameters.

For example, you might need a **printNumber()** function that prints the value of its parameter.

```
void printNumber(int a) {
 cout << a;
}
```

This is effective with integer arguments only. Overloading it will make it available for other types, such as **floats**.

```
void printNumber(float a) {
 cout << a;
}
```

Now, the same **printNumber()** function name will work for both integers and floats.

Function Overloading

When overloading functions, the definition of the function must differ from each other by the types and/or the number of arguments in the argument list.
For example:

```
void printNumber(int x) {
   cout << "Prints an integer: " << x << endl;
}
void printNumber(float x) {
   cout << "Prints a float: " << x << endl;
}
int main() {
 int a = 16;
 float b = 54.541;
 printNumber(a);
 printNumber(b);
}

/* Output:
Prints an integer: 16
Prints a float: 54.541
*/
```

Try It Yourself

As you can see, the function call is based on the argument provided. An **integer** argument will call the function implementation that takes an **integer** parameter. A **float** argument will call the implementation taking a **float** parameter.

Function Overloading

You **can not** overload function declarations that differ only by **return** type.
The following declaration results in an error.

```
int printName(int a) { }
float printName(int b) { }
double printName(int c) { }
```

Try It Yourself

Although each function uses the same name, the only difference from one to the other is the **return** type, which is not allowed.

Recursion

A **recursive function** in C++ is a function that calls itself.

To avoid having the recursion run indefinitely, you must include a termination condition.

Recursion

To demonstrate recursion, let's create a program to calculate a number's **factorial**.
In mathematics, the term factorial refers to the product of all positive integers that are less than or equal to a specific non-negative integer (n). The factorial of **n** is denoted as **n!**
For example:

```
4! = 4 * 3 * 2 * 1 = 24
```

Recursion is a method of solving a problem where the solution depends on the solutions to smaller instances of the same problem.

Recursion

Let's define our function:

```
int factorial(int n) {
  if (n==1) {
    return 1;
  }
  else {
    return n * factorial(n-1);
  }
}
```

The if statement defines the exit condition. In this case, it's when n equals one, return 1 (the factorial of one is one).
We placed the recursive function call in the else statement, which returns n multiplied by the factorial of n-1.
For example, if you call the factorial function with the argument 4, it will execute as follows: return 4 * factorial(3), which is 4*3*factorial(2), which is 4*3*2*factorial(1), which is 4*3*2*1.

> The **factorial** function calls itself, and then continues to do so, until the argument equals 1.

Recursion

We're now at the point where we can call our **factorial** function.

```
int factorial(int n) {
  if (n==1) {
    return 1;
  }
  else {
    return n * factorial(n-1);
  }
}
int main() {
  cout << factorial(5);
}

//Outputs 120
```

Try It Yourself

Another name for the exit condition is **the base case**.

> Keep in mind that a **base case** is necessary for real recursion. Without it, the recursion will keep running forever.

Arrays and Functions

An **array** can also be passed to a function as an argument.
The parameter should be defined as an array using square brackets, when declaring the function.
For example:

```
void printArray(int arr[], int size) {
  for(int x=0; x<size; x++) {
    cout <<arr[x];
  }
}
```

Try It Yourself

> Tap **Try It Yourself** to play around with the code!

Arrays and Functions

We can use our function in main(), and call it for our sample array:

```
void printArray(int arr[], int size) {
  for(int x=0; x<size; x++) {
    cout <<arr[x]<< endl;
  }
}
int main() {
  int myArr[3]= {42, 33, 88};
  printArray(myArr, 3);
}
```

Try It Yourself

The **printArray** function takes an array as its parameter (int arr[]), and iterates over the array using a **for** loop.
We call the function in main(), which is where we pass the **myArr** array to the function, which prints its elements.

Remember to specify the array's name **without** square brackets when passing it as an argument to a function.

Function Arguments

There are two ways to pass arguments to a function as the function is being called.

By value: This method copies the argument's actual value into the function's formal parameter. Here, we can make changes to the parameter within the function without having any effect on the argument.

By reference: This method copies the argument's reference into the formal parameter. Within the function, the reference is used to access the actual argument used in the call. This means that any change made to the parameter affects the argument.

By default, C++ uses call **by value** to pass arguments.

Passing by Value

By default, arguments in C++ are passed **by value**.
When passed by value, a copy of the argument is passed to the function.

Example:

```
void myFunc(int x) {
  x = 100;
}

int main() {
  int var = 20;
  myFunc(var);
  cout << var;
}
// Outputs 20
```

Try It Yourself

Because a copy of the argument is passed to the function, the original argument is not modified by the function.

Passing by Reference

Pass-by-reference copies an argument's address into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
To pass the value by reference, argument **pointers** are passed to the functions just like any other value.

```
void myFunc(int *x) {
  *x = 100;
}

int main() {
  int var = 20;
  myFunc(&var);
  cout << var;
}
// Outputs 100
```

Try It Yourself

As you can see, we passed the variable directly to the function using the **address-of operator &**.
The function declaration says that the function takes a **pointer** as its parameter (defined using the *** operator**).

As a result, the function has actually changed the argument's value, as accessed it via the pointer.

Summary

Passing by value: This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

Passing by reference: This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. So, changes made to the parameter also affect the argument.

In general, passing by value is faster and more effective. Pass by reference when your function needs to modify the argument, or when you need to pass a data type, that uses a lot of memory and is expensive to copy.

End.