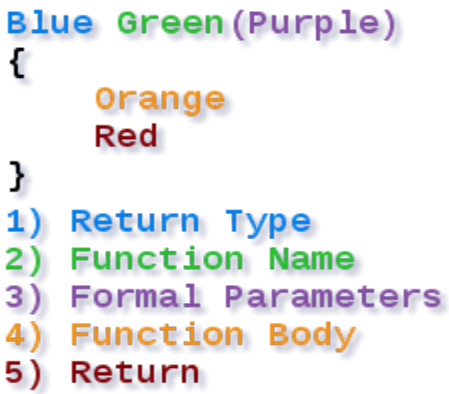
Functions



Functions break your program into separate tasks. This makes it so that your main function doesn't do all of the work. As the previous sentence denotes, main is a function and follows the same syntax as all other functions.

A function begins with its **return type**. This ranges from **int** to vector to string to your own types, such as a **class**. A **function name** should describe the task of the function. It is akin to naming a variable. The **formal parameters** are variables to hold the incoming data from elsewhere in your program. These may be any valid type, such as **int** or **char**. They are separated by a comma and they may be named differently than the incoming arguments as the variables here are local only to that function. Formal parameters may take no arguments. The **function body** is where the program statements are, just like a body of a loop or the body of main. The **return** statement is placed anywhere you would like to exit the function with the desired result. This means that there can be more than one **return** statement in a function. A function may be **void** (no value to be returned), and the keyword **return;** (optional unless you terminate early) is used versus **return variable;** Therefore, when you call a **void** function, you will not use it in an assignment statement. Math statements and boolean expressions may be done on the same line as the return statement. Here is a simple example of a void function and a function that adds two numbers together and returns an **int**:

```
void printDetails() {
    cout << "This program adds two numbers together\n";
    return; //Returns nothing  }


int addTwoNums(int num1, int num2)  {
    return num1 + num2; //Returns 400 }


int main() {  //Main usually has return 0; at end, but is not needed
    printDetails(); //Function invocation
    int number1 = 100, number2 = 300, result;
    result = addTwoNums(number1, number2); //Function invocation }
```

The function addTwoNums is used in the main function. This is called **function invocation** as it calls for the function to execute. The parameters in parenthesis (in main) are known as the **arguments** to the function. The order of the arguments will correspond to the formal parameters. That is, **number1** corresponds to the first parameter named **num1** and **number2** corresponds to the second parameter named **num2**. If

number1 and **number2** were switched, then the opposite would take place (**number2** corresponds to **num1** and **number1** corresponds to **num2**). The function executes and returns an **int**. The integer that is returned would be **400** as **num1 = 100** and **num2 = 300**. Here the values of **number1** and **number2** were copied to the function (see the **Call by Value** page), added, and then returned into **result**.

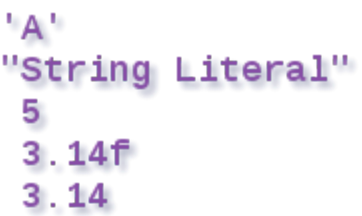
Function Declaration and Definition

1) Function Declaration;
2) Main Function
3) Function Definitions

To be more organized and formal, functions are split into two separate parts of your program. Before the main function, you have your **function declaration** (**prototype** / **signature**). This tells the compiler that a function has been created, so when the compiler comes to the function invocation it knows it exists and will find said function. For you, this is where you comment the **precondition** (what must be true before the function executes), **postcondition** (what must be true after the function executes), and what the function does. This is considered a statement, so there needs to be a semicolon after each **function declaration**. This is akin to declaring a variable. After the **main function** is where the **function definition** is. This is what the function does. Here is the previous example in the more organized and formal way:

```
//Precondition: None / Postcondition: Summary displayed to user
//Summary: Function to display what program does
void printDetails();


//Precondition: num1,num2 must hold an int value /
//Postcondition: num1,num2 are added and sum is returned /
//Summary: Function to add //two numbers
int addTwoNums(int num1, int num2);


int main() {
    printDetails(); //No arguments
    int number1 = 100, number2 = 300, result;
    result = addTwoNums(number1, number2); //Two arguments }


void printDetails() { //Restate function declaration sans semicolon
    cout << “This program adds two numbers together\n”;
    return; //Optional as function terminates at end regardless }


int addTwoNums(int num1, int num2) { //Restate sans semicolon
    return num1 + num2; }
```

Literals as Function Arguments



Variables are not the only thing that is allowed to be passed as an argument to a function. **Literals** are just as valid as variables when it comes to arguments for a function [except they cannot be called (passed) by reference]. A **literal** is a value, such as a **char**, **string**, **int**, **float**, or **double**, that has not been assigned to any variable. They are standalone values. Here is an example:

```
void literals(char a, string literal, int x, float y, double z)
{
    cout << a << ' ' << literal << endl;
    cout << x << ' ' << y << ' ' << z << endl;
}

int main()
{
    literals('A', "String Literal", 5, 3.14, 3.14);
}
```

The compiler does the job of assigning the literals that are a part of the function arguments to the variables of the function itself. **'A'** would be assigned to **a**, **"String Literal"** would be assigned to **literal**, and so forth. That way, if you are not planning on using the literals outside of the function itself, there is no reason to declare and define variables in main. This leads to shorter code and frees up variable names for something more important. What the **literals** function does is simply display to the console what has been stored in each variable from the formal parameters. This is the same action as declaring and initializing a variable and displaying it. The variables in the **literals** function are able to be modified.

Constant Reference Function Parameter

```
void function(const int &variable);
```

This technique is passing a variable by reference to a function but making it **const** so that it cannot be altered. This method is used for efficiency purposes by saving on memory as there is no copying of the data. In other words, a constant reference acts as an alias to the argument from the **function invocation**, and this alias can be used in the function, but not changed in any way. Examples include a class' copy constructor (see the **Class** section for more information) or a function that simply displays data.