Functions

Functions Summary	Examples of functions we have used:
Functions are a collection of statements which perform a specific task. Typically, functions are used in making code more modular, reusable, and maintainable.	`sqrt(iX), fabs(dX), fileInput.eof(), szName.size(), szName.at(2)`

Defining a Function

In order to use a function, you first must define it. All functions are comprised of the following parts:

Return Data Type: The data type of the value that is sent from the function.
Name: What the function is called. This name should be a reflection of what the function accomplishes.
Parameter List: A list of variables that are being passed to the function. Each parameter must also be defined with what data type it is. In some cases, the list can be empty.
Body: A set of statements to execute when the function is called.

```
/*****************************************************************
int absoluteNumber(int iNumber)
Purpose:
  Find the absolute value of a number
Parameters:
  I int iNumber - Number to absolute
Return Value:
  The absolute value of iNumber
Notes:
  -
*****************************************************************/
int absoluteNumber(int iNumber)
{
   if(iNumber < 0)
       iNumber *= -1;

   return iNumber;
}
```

Documenting a Function
 	
When defining a function, it is important to document the intended behavior of the function. Doing so enhances the overall quality and usability of the codebase. This is especially important when documenting the behavior of parameters and return values.
 	
`Parameters:`
`I - Notes that the parameter is passed in with initial data and does not get modified.`
`O - Notes that this parameter is used to pass data out and does not come in with data initially.`
`I/O - Notes that this parameter is passed in with initial data and its data gets modified by the function execution.`
 	
For this class, you will document every function you create with this comment header block. It is typically a good idea to start with documentation first as it can help you with the function writing process.
	`/****************************************************************`
	`function prototype`
	`Purpose:`
	`   Description of the purpose of the function.`
	`Parameters:`
	`   List each of each parameter including data type and description.`
	`   Each item should begin with whether the parameter is passed in, out or both.`
	`   I Passed in.`
	`   O Passed out.`
	`   I/O Modified.`
	`Return Value:`
	`   List of values returned by the function, excluding parameters.`
	`Notes:`
	`   Description of any special information regarding this function.`
	`   This is a good place to state any assumptions the function makes.`
	`****************************************************************/`

Defining a Void Function
 	
Sometimes, it is not necessary to have your function return a value. However, as previously mentioned, your function definition must always return a data type. In these cases, we use the keyword `void` to note that a function does not return any data. Void functions **DO NOT** use return statements.
	`void displayHelloWorld()`
	`{`
	`   cout << "Hello world" << endl;`
	`}`
	
	`What happens if you return in a void function?`
	`You get an error. "return-statement with a value, in function returning ‘void’"`

Calling a Void Function

Once you have defined the function, you must call the function in order to have its code executed.

Note: `main` is a special function that is automatically called when a program starts. All other functions must be explicitly called in order to execute their code.

Calling a Function from Main

```
int main()
{
   // Call displayHelloWorld function
   displayHelloWorld();
}
```

The Call Stack

In C++, the call stack is a runtime data structure that stores information about active function calls in a program. When a function is called, a new stack frame is created and pushed onto the top of the call stack. The stack frame contains information such as the function's parameters, local variables, return address, and sometimes additional metadata required for function execution.

Note: This is just for your information. Call stacks are covered in detail in CS 2308.

Call Stack Visualization

```
        ------------------------------
        |     displayHelloWorld()     |  <-- Top of the Stack
        ------------------------------
        |           main()            |  <-- Bottom of the Stack
        ------------------------------
```

Calling a Function Before Defining It

You can think of calling functions in C++ similar to making a phone call. If you try to call a phone number before it's been assigned to anyone, you'll just hear a dial tone. Similarly, if you try to call a function before it's been defined in your code, the program will encounter an error.

```
int main()
{
   // Call displayHelloWorld function
   displayHelloWorld();
}

void displayHelloWorld()
{
   cout << "Hello world" << endl;
}

error: ‘displayHelloWorld’ was not declared in this scope
```

Function Prototypes

Function prototypes, also known as function declarations, serve to inform the compiler about a function before it's fully defined. By providing this information upfront, the compiler can recognize the function's usage throughout the codebase, enabling seamless integration even when the function definition appears later in the program.

Reminder: If you fully define the function before calling it, you do not need a function prototype.

```cpp
// Function prototype
void displayHelloWorld();

int main()
{
   // Call displayHelloWorld function
   displayHelloWorld();
}

// Define function
void displayHelloWorld()
{
   cout << "Hello world" << endl;
}
```

Function Parameters and Arguments

Sometimes functions need to take in data in order to perform a task, as we have seen with functions like sqrt() and fabs(). This data can be broken down into two categories:

Arguments: A value provided to a function's parameter during a function call. These can be literals, variables, or even expressions.

Parameters: A function input specified in a function definition. Upon a call, the parameter's memory location is allocated, and the parameter is assigned with the argument's value. Upon returning to the original call location, the parameter is deleted from memory.

When calling a function, the number of arguments **MUST** match the number of non-default function parameters.

```cpp
void displayMovie(string szMovie, double dRating)
{
   cout << szMovie << " has a rating of " << dRating << endl;
}

int main()
{
   string szMovie = "Jaws";
   double dRating = 4.5;
   displayMovie(szMovie, dRating);
}

What happens if we call displayMovie(dRating, szMovie)?
could not convert ‘dRating’ from ‘double’ to ‘std::string’ {aka
‘std::__cxx11::basic_string’}
```

Pass by Value

By default, C++ passes parameters using pass by value. This means that a copy of the argument's value is made and passed to the function. Any modifications made to the parameter inside the function do not affect the original argument outside the function.

```
What is the output of the provided program?
main: 10 100
functionA: 20 110
main: 10 100
```

```
void functionA(int iX, int iY)
{
   iX = iX + 10;
   iY = iY + 10;
   cout << "functionA: " << iX << " " << iY << endl;

}

int main()
{
   int iX = 10, iY = 100;
   cout << "main: " << iX << " " << iY << endl;

   // Call function A
   functionA(iX, iY);

   cout << "main: " << iX << " " << iY << endl;
}
```

Setting Default Parameters

Similar to structures, you can specify default values for one or more parameters in a function definition. These default values are used when the function is called without providing arguments for those parameters. When defining default function parameters, there are a couple of rules that you must follow:

1. Default parameters must be specified from right to left. Non-default parameters cannot follow default parameters.
2. Once a default parameter is provided, all subsequent parameters in the list must also have default values.
3. Default values can be literals, constants, or expressions of compatible types.

```
void addNumbers(int iX, int iY = 10)
{
    cout << "iSum: " << iX + iY << endl;
}

addNumbers(10, 20);
iSum: 30

addNumbers(10);
iSum: 20

addNumbers();
Invalid
```

Parameter Error Checking

When dealing with a function that has parameters, typically it is a good idea to check that values are within an expected range. If a parameter is not in the expected range, the function should take one or more of various actions, like outputting an error message, assigning a valid value, returning a value indicating failure, exiting the program, etc.

Typically, error messages are printed to `cerr` as it is an output stream in C++ that is used specifically for writing error messages to the standard error stream. Unlike `cout`, `cerr` is unbuffered, meaning that output written to `cerr` is immediately displayed on the console without waiting for the buffer to be flushed.

```
double calculateRectangleArea(double dLength, double dWidth)
{
   if (dLength <= 0 || dWidth <= 0)
   {
      cerr << "Error: Invalid dimensions. Length and width must
be positive numbers." << endl;

      // Return a negative value to indicate error
      return -1.0;
   }

       // Calculate and return the area
       return dLength * dWidth;
}
```

Returning a Value from a Function

Functions can return data using a `return` statement. When a function returns data, it's called a value-returning function. Such functions can only return **a single** piece of data through a return statement*. The function exits either after returning data or when the last statement of the function body has been executed, whichever occurs first.

*You can return values through other methods like references, which are discussed later on.

```
int sumNumbers(int iX, int iY)
{
   int iResult;
   iResult = iX + iY;
   return iResult;
}

int sumNumbers(int iX, int iY)
{
   return iX + iY;
}
```

Forgetting the Return in a Value-Returning Function

A common error when dealing with value-returning functions is to forget to return a value. When a value-returning function is missing a return statement, the returned value is undefined. This means that sometimes the function will unintentionally return the correct answer. Make sure that your value-returning functions are always returning a value.

```
int sumNumbers(int iX, int iY)
{
   int iResult;
   iResult = iX + iY;
}

What warning can be used to help identify this issue?
warning: no return statement in function returning non-void
[-Wreturn-type]
```

Calling a Value-Returning Function

Although calling a value-returning function is similar to calling a void function, there is a key difference. Mainly, when you call a value-returning function, you will want to use the value it returns. This could be done either by assigning the return to a variable or even directly printing out the return via an output statement.

Reminder: A value-returning function will always return a value of the specified data type.

```
int absoluteNumber(int iNumber)
{
   if(iNumber < 0)
       iNumber *= -1;

   return iNumber;
}

int main()
{
   int iX, iAbsoluteValue;

   cout << "Enter number: ";
   cin >> iX;

   iAbsoluteValue = absoluteNumber(iX);

   cout << "|" << iX << "| = " << iAbsoluteValue;
   cout << endl;
}
```

Pass by Reference

In addition to passing by value, C++ also supports the ability to access and modify the memory location of a variable passed to it as an argument. This is referred to as pass by reference. This means that any changes made to the parameter inside the function directly affect the original variable outside the function.

To pass arguments by reference, you use the reference operator & in the function parameter list.

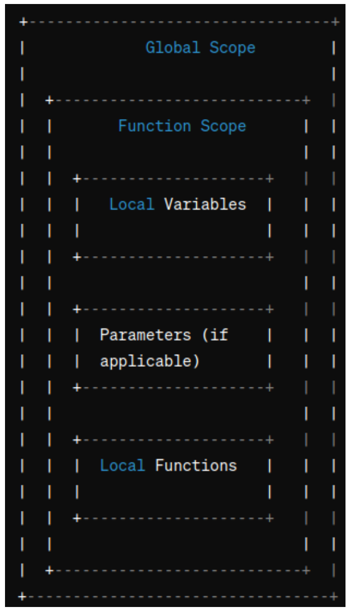
```
What is the output of the provided program?
main: 10 100
functionA: 20 110
main: 20 110
```

```
void functionA(int& iX, int& iY)
{
   iX = iX + 10;
   iY = iY + 10;
   cout << "functionA: " << iX << " " << iY << endl;

}

int main()
{
   int iX = 10, iY = 100;
   cout << "main: " << iX << " " << iY << endl;

   // Call function A
   functionA(iX, iY);

   cout << "main: " << iX << " " << iY << endl;
}
```

Understanding Scope

Understanding scope in C++ is crucial for managing the visibility and lifetime of variables and functions within a program. Scope often refers to the region of a program where a particular identifier, such as a variable or a function, is valid and accessible. Below are some brief categories of scope.

Global Scope	Function Scope	Namespace Scope
Variables and functions declared outside of any function. Accessible from anywhere in the program.	Variables declared within a function or passed as parameters. Limited to the function in which they are declared.	Variables, functions, and classes declared within a namespace. Provides a way to organize and encapsulate code.



Tracing a Function Call

Tracing is a powerful technique for debugging, understanding, and optimizing code. Tracing can be done either manually (by hand or via output statements) or through the use of a debugger.

```
      Function Call    |   Return Value
  ---------------------+--------------------
    Passed Parameters and Body Execution
```

```cpp
int functionA(int iA, int iB)
{
        int iC = iA * iB;
        return iC;
}
```

Trace the function call functionA(1,2) using a function t-chart.

```
  functionA(1,2)   |  2
  -----------------+-----------
iA = 1
iB = 2
iC = iA * iB = 1 * 2 = 2
```

Unit Tests

Another common way to test if programs are behaving properly is to use unit testing. Unit testing is the process of individually testing a small part or unit of a program, typically a function. In C++, we use the assert() to create unit tests. Assert() is a macro (similar to a function) from the cassert library that prints an error message and exits the program if assert()'s input expression is false.

Example of failed assertion:

```
int main(): Assertion `convertToInches(0 ,1) == 1'
failed.
```

What's wrong with the provided program?
It is subtracting iInches instead of adding iInches.

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

int convertToInches(int iFeet, int iInches)
{
   int iToInches = 0;

   iToInches = iFeet * 12 - iInches;

   return iToInches;
}

int main()
{
   // This assert would pass
   assert(convertToInches(1 ,0) == 12);
   // This assert would fail
   assert(convertToInches(0 ,1) == 1);
}
```

Overloading Function Definitions

C++ allows you to create multiple functions with the same name as long as they have different parameter lists. Typically, these functions perform similar tasks but may operate on different types of data or accept different numbers of parameters.

However, overloading functions has some drawbacks as it can cause ambiguity, complexity, potential bugs, and compile-time overhead.

```
void displayMovie(string szMovie)
{
   cout << szMovie << " has not released";
}

void displayMovie(string szMovie, double dRating)
{
   cout << szMovie << " has a rating of " << dRating << endl;
}

szMovie = "Jaws: The Revenge";
dRating = 2.4;
displayMovie(szMovie, dRating);

szMovie = "Jawsnado";
displayMovie(szMovie);
```