Chapter 5

Functions in C++

Function Basics

Definition, prototypes, function calls

Functions in C++

Functions are a named list of statements that perform a specific task, and do that specific task well

A ***function definition*** consists of the functions return type, name and body

- Function names or identifiers are typically verbs, whereas variable identifiers are typically nouns.

A ***function call*** is an invocation of a function's name, which triggers the execution of code within that function/

```
void makePizza() { /* Code... */ }
```

examples: basic_function.cpp

Function Arguments / Parameters

A ***parameter*** is a variable that represents the input of a function and is specified in the function definition / implementation.

```
void findMax(int var1, int var2) { ... }
```

- A parameter of a function is only allocated in memory as long as the function is being executed
- Once the function completes, the values are cleared from memory (unless using pass by reference)

An ***argument*** is a value that is passed to a function and becomes the function parameter during a function call.

```
int num1 = 10;
int num2 = 20;
findMax(num1, num2);
```

Function Return Types

- A function may ***return*** a single value using a return statement
- The ***return type*** of a function is a data type representing the type of the value returned by the function.
- Return types can be any data type, just like variable data types:
  - ***int, double, float, char, string, vector<string>,*** etc.
  - ***Void -*** method that does not return a values

```
void makePizza() { /* Code... */ }
```

Return type

example: findMax.cpp

Why Functions?

Functions are one of the most useful concepts in any programming language

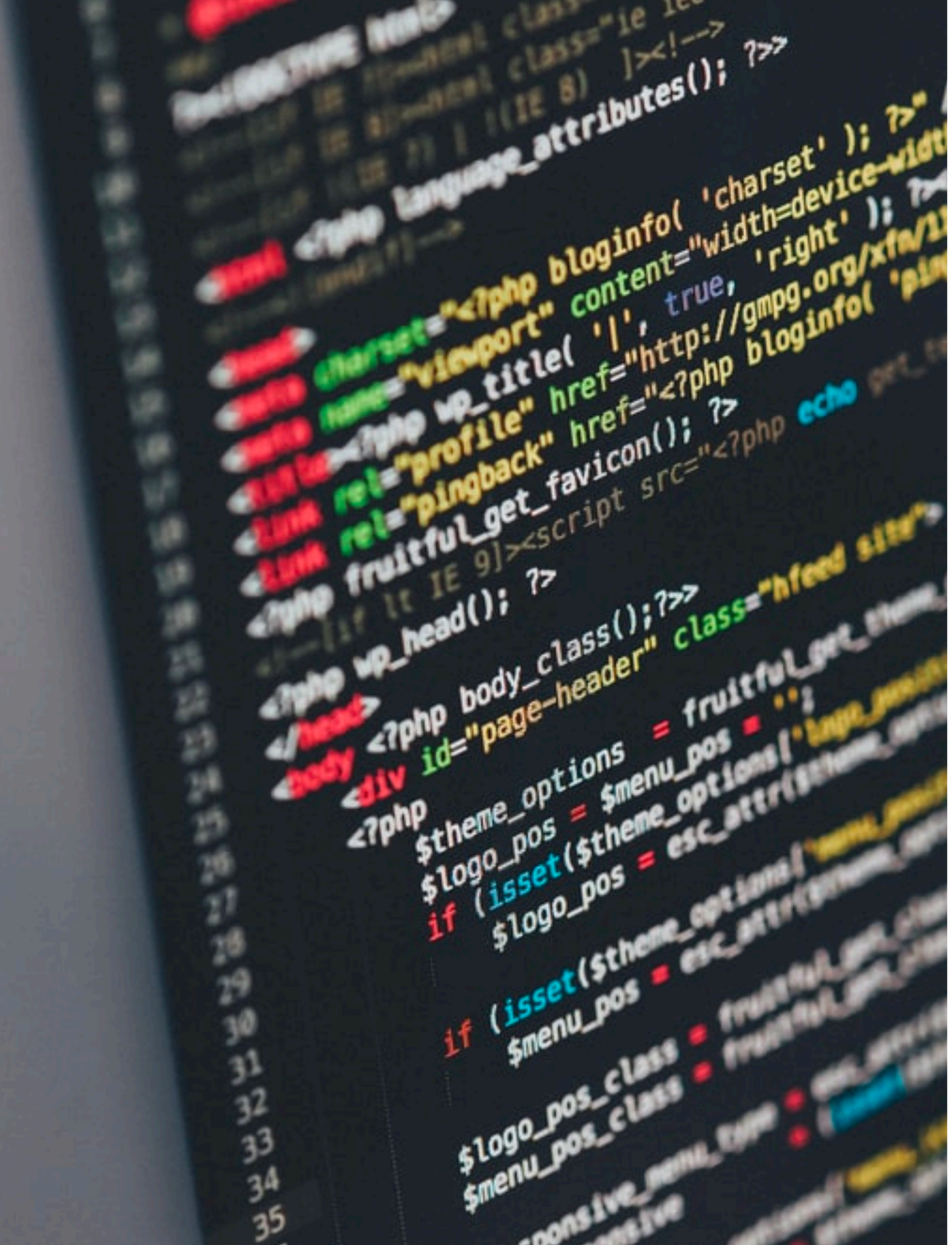
- Functions provide ***reusability.*** Once a function is defined, we can reuse it over and over again
- Functions also provide ***abstraction.*** A function's name hides the implementation of that function behind the scenes
- ***Program Readability -*** considered good practice to have main() be as few lines of code as possible.
- ***Modular Program Development -*** define functions that you will use ahead of time, then implement the functions

NOTE: This is a great way to break programs into smaller pieces! Think Program 2.

Program 2:

Start program 2 by completing small bits of functionality first!!

```cpp
#include <iostream>
#include <string>

using namespace std;
void display();
string capitalize(string);
char encryptToken(char, char);
char decryptToken(char, char);

int main() { return 0; }

void display() {
  // TODO: implement
}

string capitalize(string message) {
  // TODO: implement capitalize
}

char encryptToken(char original, char passcode) {
  // TODO: Implement
}
```



Challenge!

Implement a function that “flips” a coin by generating a random number and using it to return a string representing “heads” or “tails”

examples: heads_or_tails.cpp

A note on getline() and cin

- ***getline*** consumes the delimiter (i.e. '\n')
- ***cin*** reads up to whitespace, but does not consume it
- Need to be careful when using cin and getline together.

Best Practice

When possible, always use JUST cin or just getline to avoid these streaming problems

examples: getline.cpp

Memory of a C++ Program

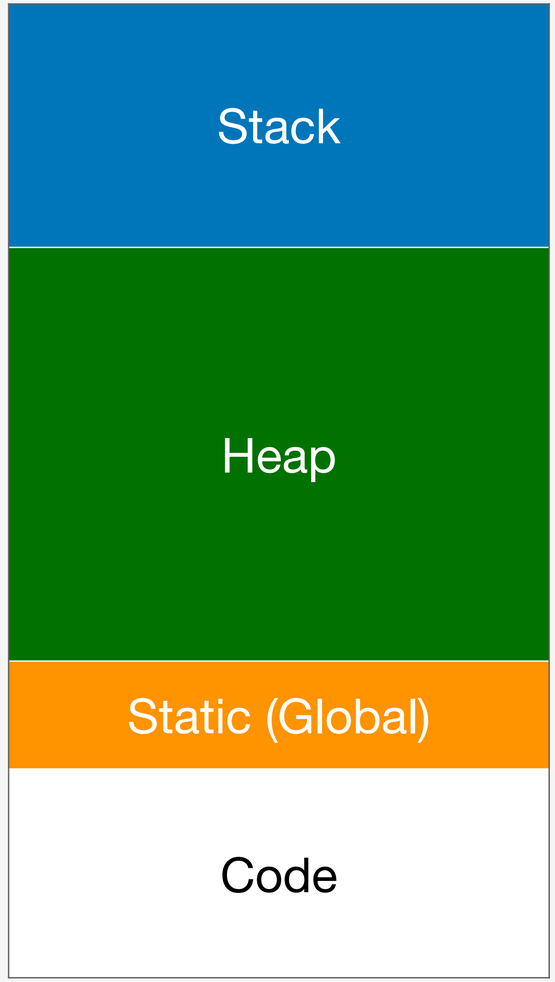
4 Main parts of memory

- Code
- Static (Global) Variables / Data
- Stack
- Heap

Heap and Stack vary dynamically

Code and Static/Global is fixed in size

Code is read-only



The Call Stack

The ***call stack*** is a data structure that is used to store information about the subroutines (or functions) of a computer program

- A ***stack*** is a LIFO data structure (Last-in First-out)

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 int main() {
7   functionA();
8   return 0;
9 }
```

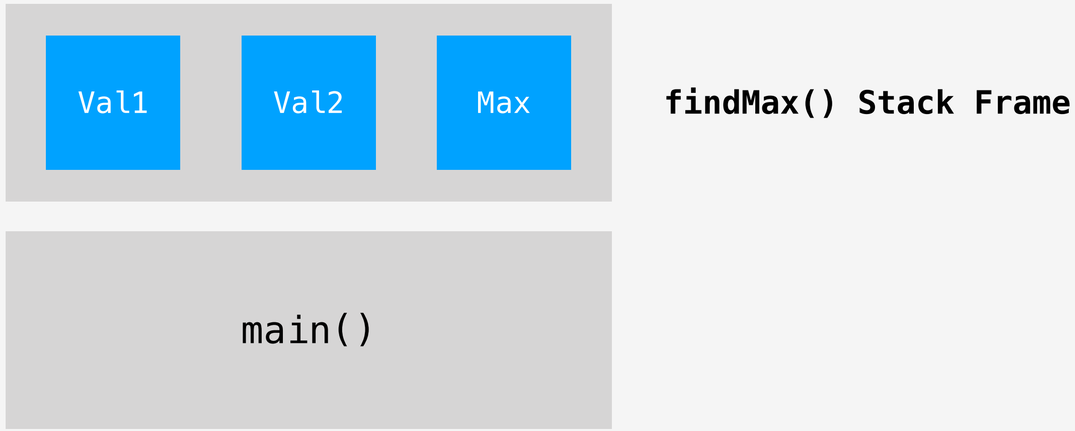
```
void functionA() {
  int value;
  value = functionB();
}
```

```
int functionB() {
  return 1 + 2;
}
```

The Call Stack

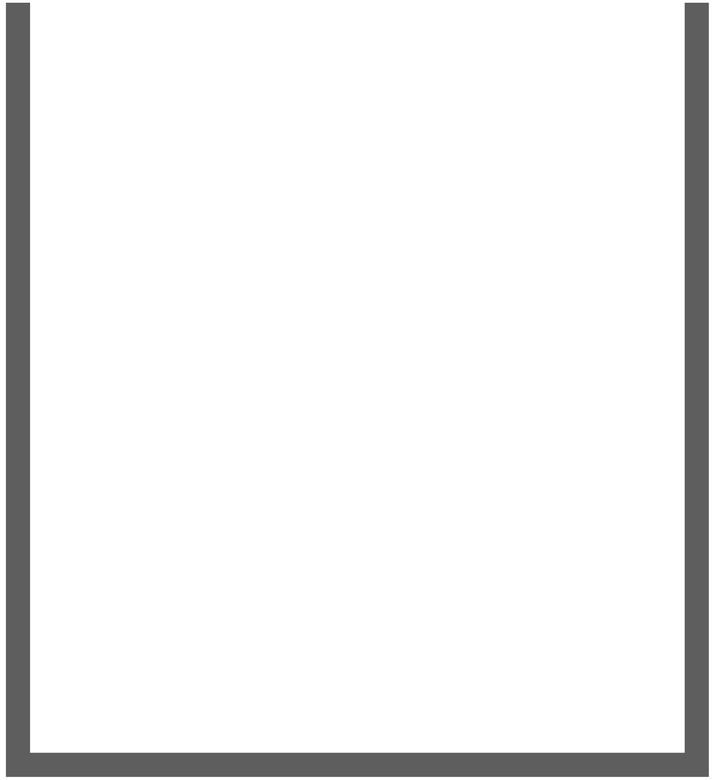
- Each function call in a program creates a new set of local variables
- Each function's context in memory is known as a ***stack frame***
- After a function completes execution, it's local variables and parameters are discarded.

```
int findMax(int val1, int val2) {
   int max;
   ...
   return max;
}
```



Example

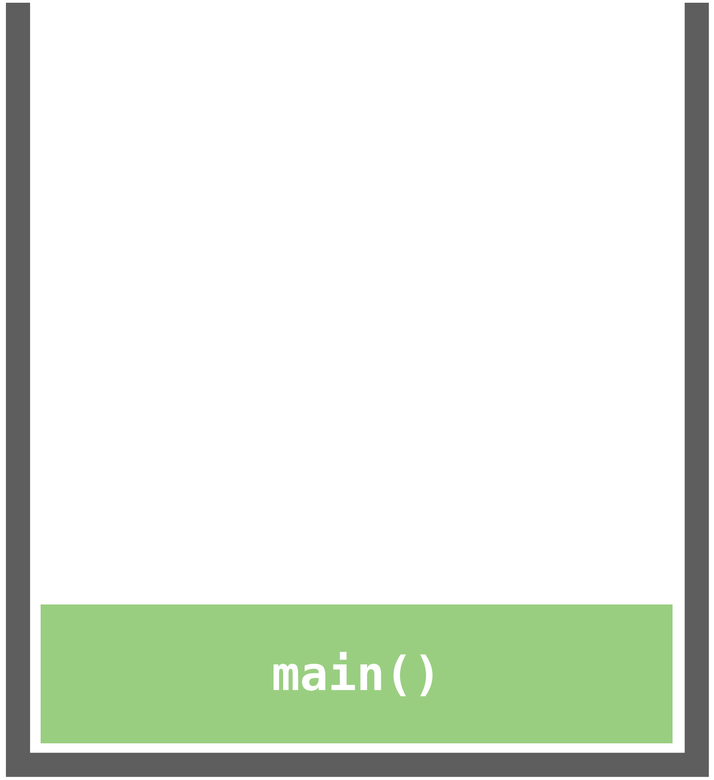
The Call Stack



```
1  #include <cstdlib>
2  #include <iostream>
3  #include <string>
4
5  using namespace std;
6
7  void display();
8  void formatString(string);
9  char update(char);
10
11 int main() {
12   display();
13   string example{"This is a test"};
14
15   formatString(example);
16
17   return 0;
18 }
19
20 void display() { cout << "Welcome!!" << endl; }
21
22 void formatString(string text) {
23   const unsigned int size = text.size();
24   for (int i = 0; i < size; i++) {
25     text.at(i) = update(text.at(i));
26   }
27 }
28
29 char update(char token) { return toupper(token); }
30
```

Example

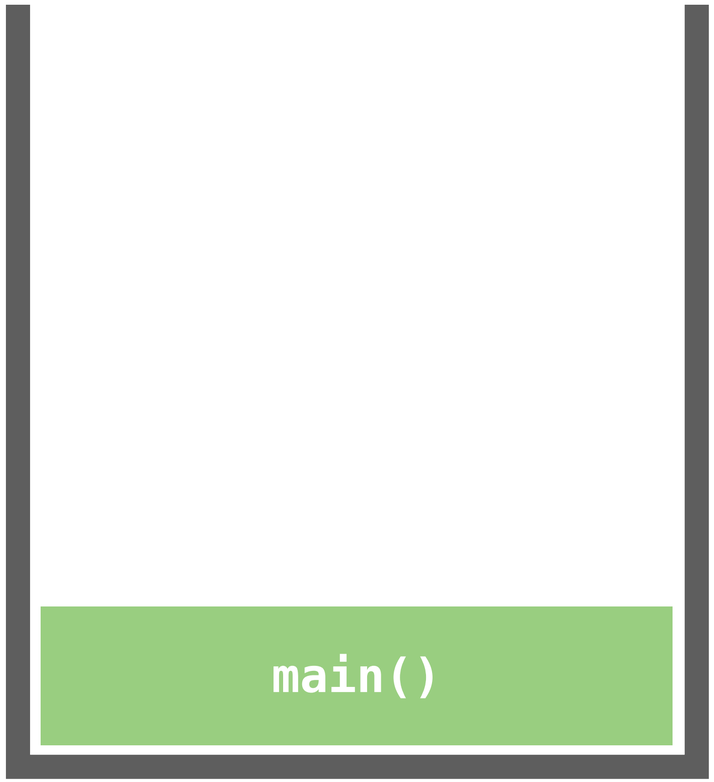
The Call Stack




```
1  #include <cstdlib>
2  #include <iostream>
3  #include <string>
4
5  using namespace std;
6
7  void display();
8  void formatString(string);
9  char update(char);
10
11 int main() {
12   display();
13   string example{"This is a test"};
14
15   formatString(example);
16
17   return 0;
18 }
19
20 void display() { cout << "Welcome!!" << endl; }
21
22 void formatString(string text) {
23   const unsigned int size = text.size();
24   for (int i = 0; i < size; i++) {
25     text.at(i) = update(text.at(i));
26   }
27 }
28
29 char update(char token) { return toupper(token); }
30
```

Example

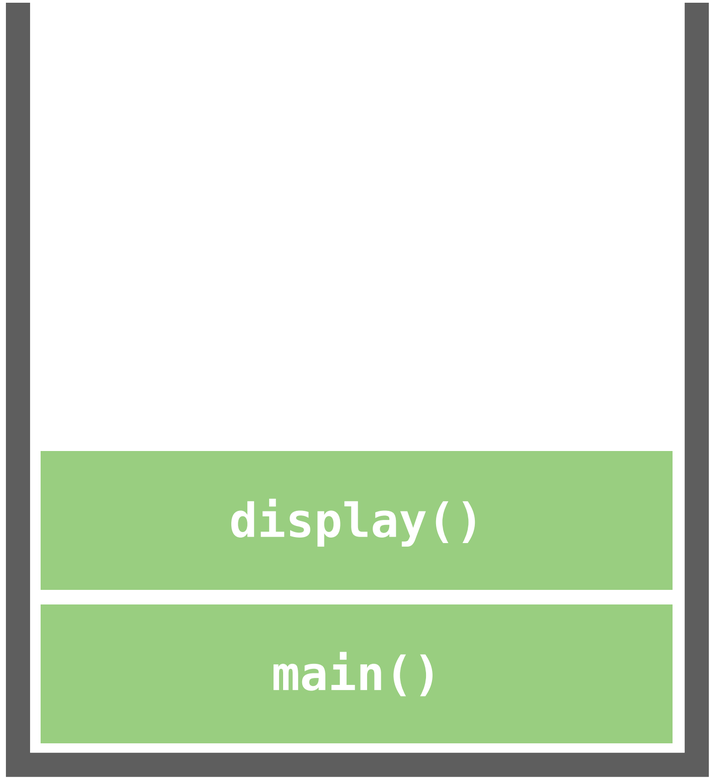
The Call Stack




```
 1 #include <cstdlib>
 2 #include <iostream>
 3 #include <string>
 4
 5 using namespace std;
 6
 7 void display();
 8 void formatString(string);
 9 char update(char);
10
11 int main() {
12   display();
13   string example{"This is a test"};
14
15   formatString(example);
16
17   return 0;
18 }
19
20 void display() { cout << "Welcome!!" << endl; }
21
22 void formatString(string text) {
23   const unsigned int size = text.size();
24   for (int i = 0; i < size; i++) {
25     text.at(i) = update(text.at(i));
26   }
27 }
28
29 char update(char token) { return toupper(token); }
30
```

Example

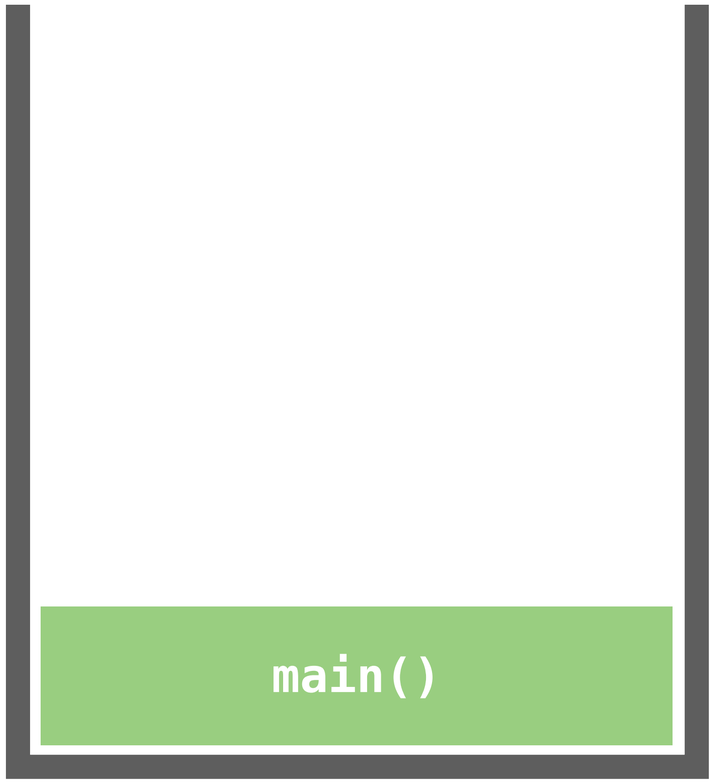
The Call Stack




```
 1 #include <cstdlib>
 2 #include <iostream>
 3 #include <string>
 4
 5 using namespace std;
 6
 7 void display();
 8 void formatString(string);
 9 char update(char);
10
11 int main() {
12   display();
13   string example{"This is a test"};
14
15   formatString(example);
16
17   return 0;
18 }
19
20 void display() { cout << "Welcome!!" << endl; }
21
22 void formatString(string text) {
23   const unsigned int size = text.size();
24   for (int i = 0; i < size; i++) {
25     text.at(i) = update(text.at(i));
26   }
27 }
28
29 char update(char token) { return toupper(token); }
30
```

Example

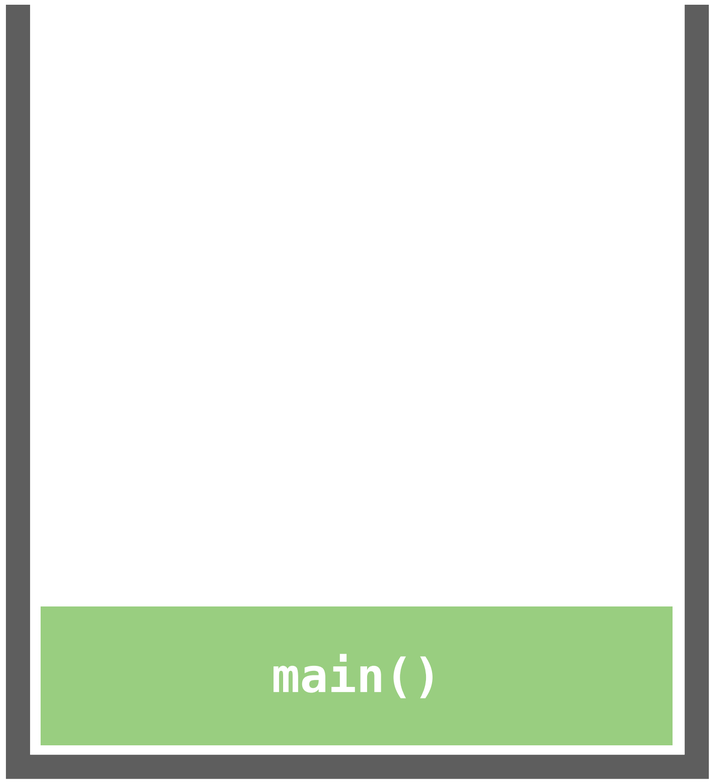
The Call Stack




```
 1 #include <cstdlib>
 2 #include <iostream>
 3 #include <string>
 4
 5 using namespace std;
 6
 7 void display();
 8 void formatString(string);
 9 char update(char);
10
11 int main() {
12   display();
13   string example{"This is a test"};
14
15   formatString(example);
16
17   return 0;
18 }
19
20 void display() { cout << "Welcome!!" << endl; }
21
22 void formatString(string text) {
23   const unsigned int size = text.size();
24   for (int i = 0; i < size; i++) {
25     text.at(i) = update(text.at(i));
26   }
27 }
28
29 char update(char token) { return toupper(token); }
30
```

Example

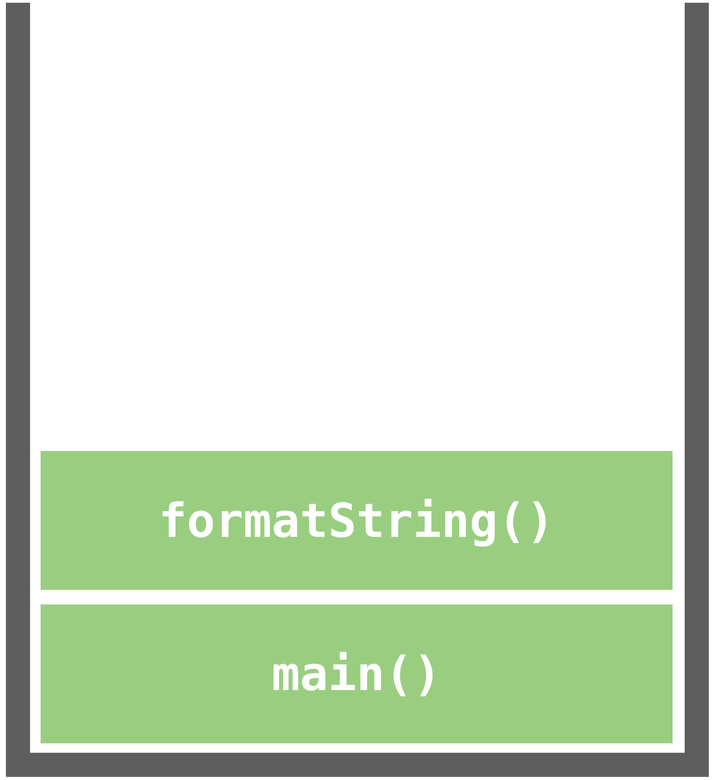
The Call Stack




```
1  #include <cstdlib>
2  #include <iostream>
3  #include <string>
4
5  using namespace std;
6
7  void display();
8  void formatString(string);
9  char update(char);
10
11 int main() {
12   display();
13   string example{"This is a test"};
14
15   formatString(example);
16
17   return 0;
18 }
19
20 void display() { cout << "Welcome!!" << endl; }
21
22 void formatString(string text) {
23   const unsigned int size = text.size();
24   for (int i = 0; i < size; i++) {
25     text.at(i) = update(text.at(i));
26   }
27 }
28
29 char update(char token) { return toupper(token); }
30
```

Example

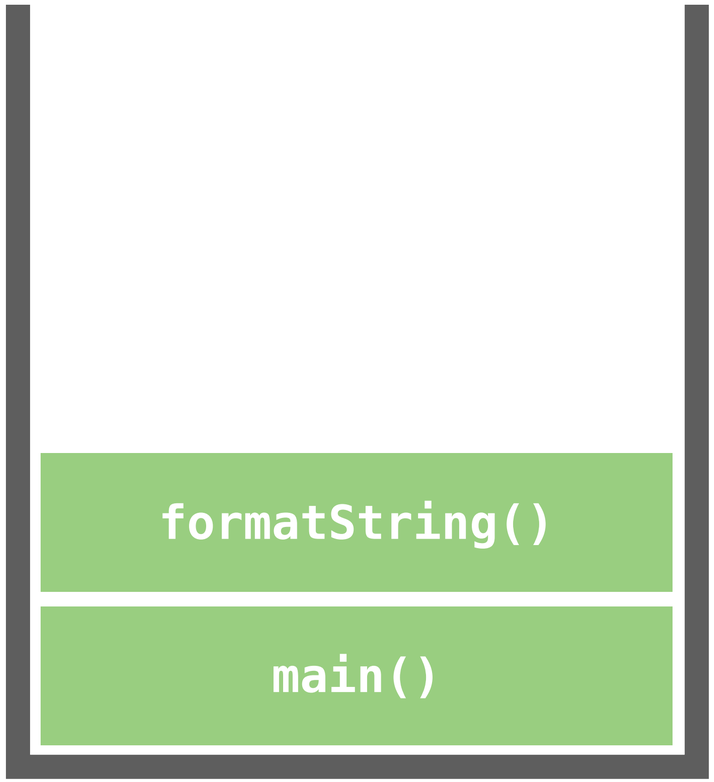
The Call Stack




```
 1 #include <cstdlib>
 2 #include <iostream>
 3 #include <string>
 4
 5 using namespace std;
 6
 7 void display();
 8 void formatString(string);
 9 char update(char);
10
11 int main() {
12   display();
13   string example{"This is a test"};
14
15   formatString(example);
16
17   return 0;
18 }
19
20 void display() { cout << "Welcome!!" << endl; }
21
22 void formatString(string text) {
23   const unsigned int size = text.size();
24   for (int i = 0; i < size; i++) {
25     text.at(i) = update(text.at(i));
26   }
27 }
28
29 char update(char token) { return toupper(token); }
30
```

Example

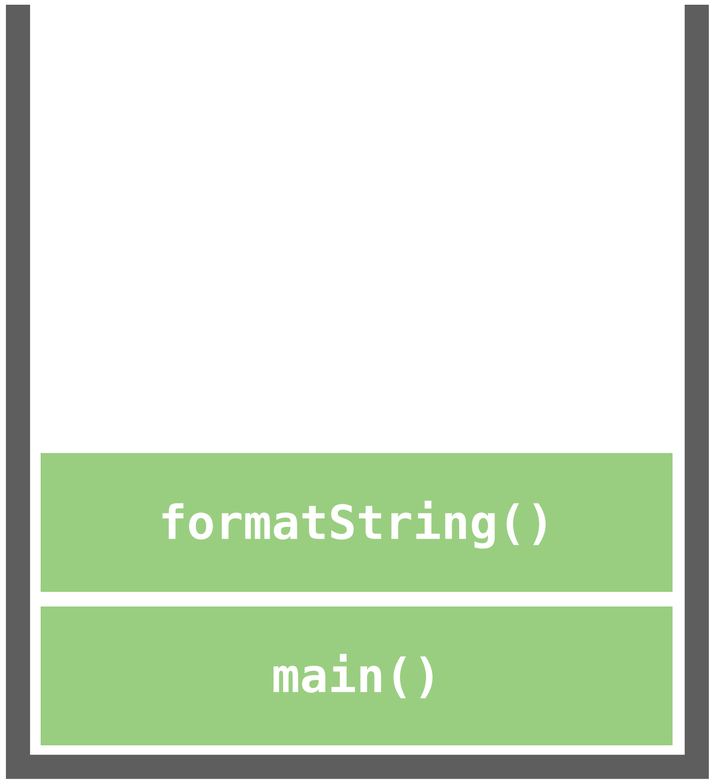
The Call Stack




```
 1 #include <cstdlib>
 2 #include <iostream>
 3 #include <string>
 4
 5 using namespace std;
 6
 7 void display();
 8 void formatString(string);
 9 char update(char);
10
11 int main() {
12   display();
13   string example{"This is a test"};
14
15   formatString(example);
16
17   return 0;
18 }
19
20 void display() { cout << "Welcome!!" << endl; }
21
22 void formatString(string text) {
23   const unsigned int size = text.size();
24   for (int i = 0; i < size; i++) {
25     text.at(i) = update(text.at(i));
26   }
27 }
28
29 char update(char token) { return toupper(token); }
30
```

Example

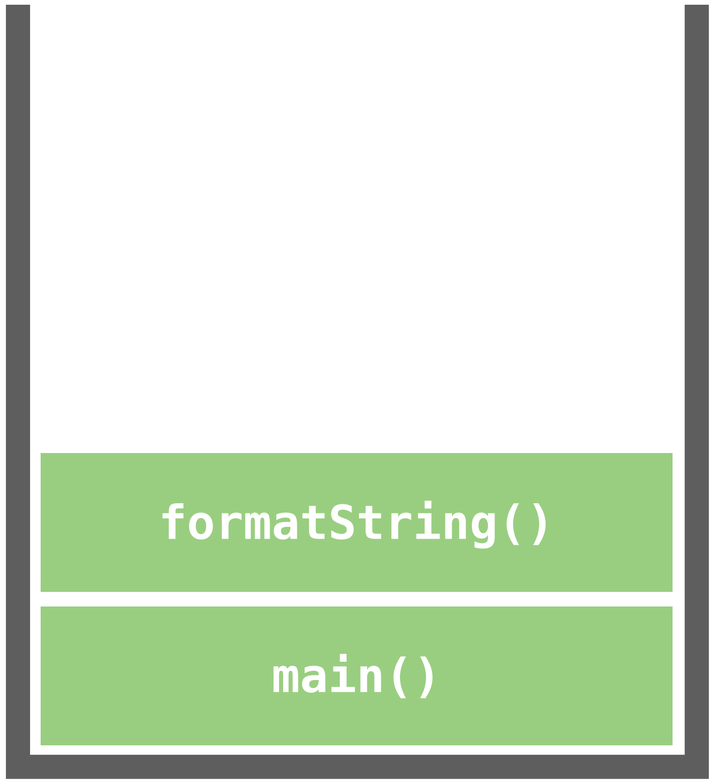
The Call Stack




```
 1 #include <cstdlib>
 2 #include <iostream>
 3 #include <string>
 4
 5 using namespace std;
 6
 7 void display();
 8 void formatString(string);
 9 char update(char);
10
11 int main() {
12   display();
13   string example{"This is a test"};
14
15   formatString(example);
16
17   return 0;
18 }
19
20 void display() { cout << "Welcome!!" << endl; }
21
22 void formatString(string text) {
23   const unsigned int size = text.size();
24   for (int i = 0; i < size; i++) {
25     text.at(i) = update(text.at(i));
26   }
27 }
28
29 char update(char token) { return toupper(token); }
30
```

Example

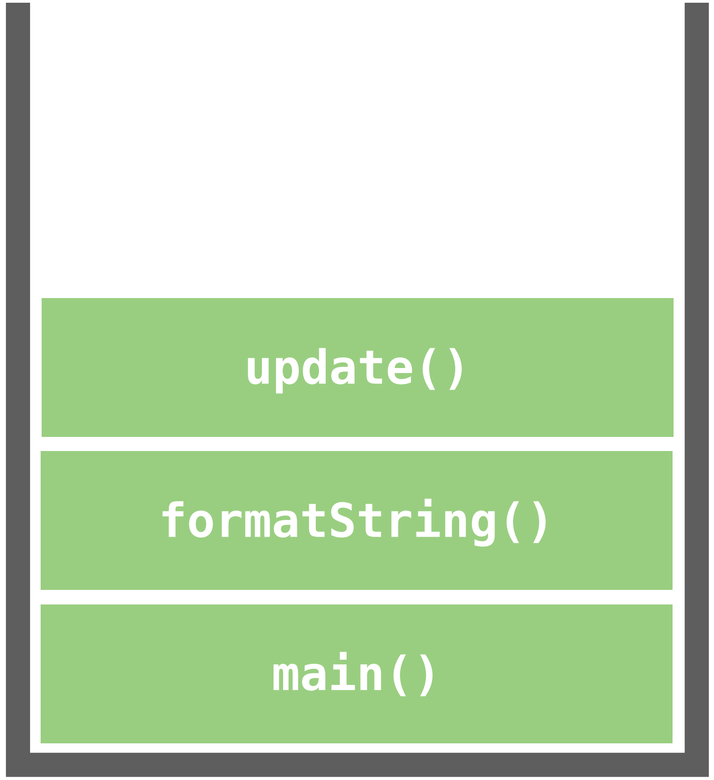
The Call Stack




```
 1 #include <cstdlib>
 2 #include <iostream>
 3 #include <string>
 4
 5 using namespace std;
 6
 7 void display();
 8 void formatString(string);
 9 char update(char);
10
11 int main() {
12   display();
13   string example{"This is a test"};
14
15   formatString(example);
16
17   return 0;
18 }
19
20 void display() { cout << "Welcome!!" << endl; }
21
22 void formatString(string text) {
23   const unsigned int size = text.size();
24   for (int i = 0; i < size; i++) {
25     text.at(i) = update(text.at(i));
26   }
27 }
28
29 char update(char token) { return toupper(token); }
30
```

Example

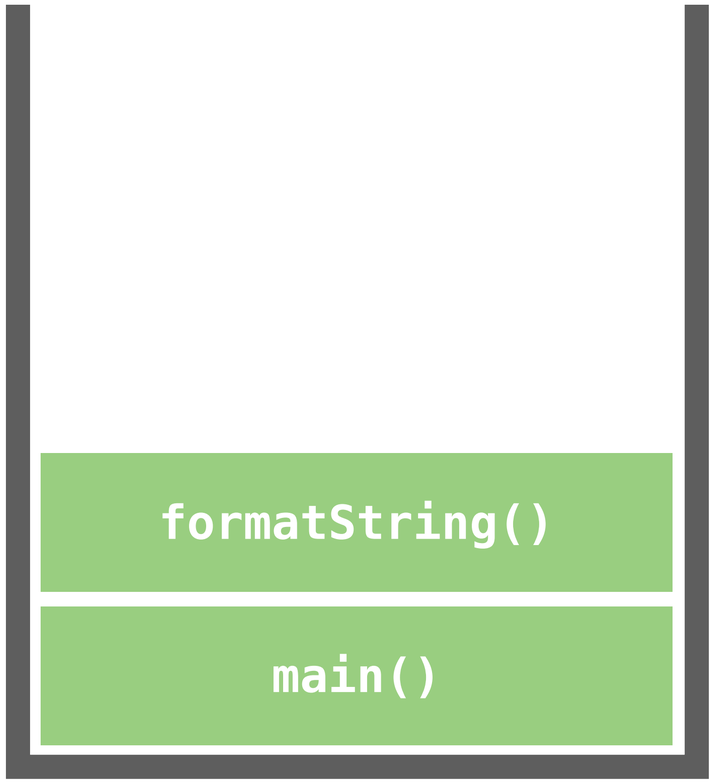
The Call Stack




```
1  #include <cstdlib>
2  #include <iostream>
3  #include <string>
4
5  using namespace std;
6
7  void display();
8  void formatString(string);
9  char update(char);
10
11 int main() {
12   display();
13   string example{"This is a test"};
14
15   formatString(example);
16
17   return 0;
18 }
19
20 void display() { cout << "Welcome!!" << endl; }
21
22 void formatString(string text) {
23   const unsigned int size = text.size();
24   for (int i = 0; i < size; i++) {
25     text.at(i) = update(text.at(i));
26   }
27 }
28
29 char update(char token) { return toupper(token); }
30
```

Example

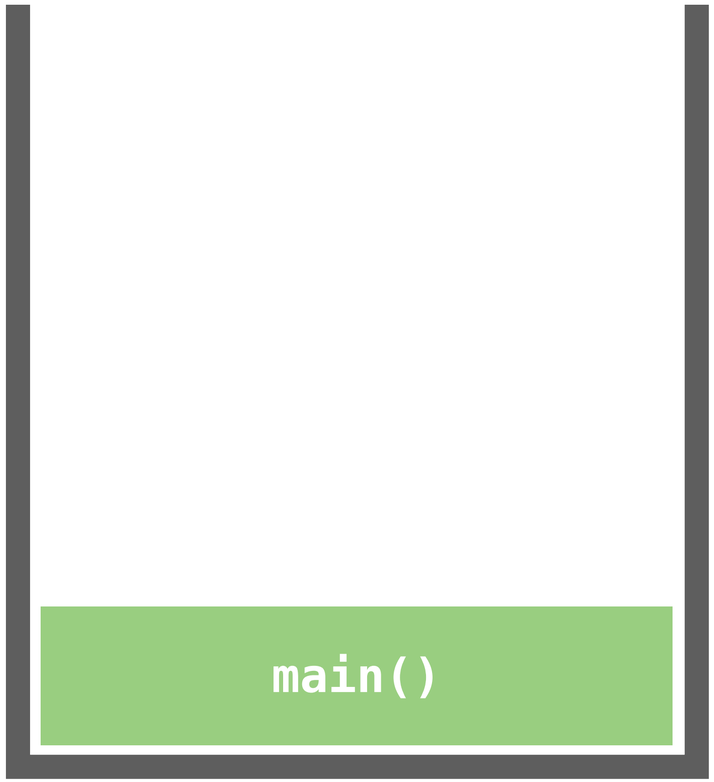
The Call Stack




```
1  #include <cstdlib>
2  #include <iostream>
3  #include <string>
4
5  using namespace std;
6
7  void display();
8  void formatString(string);
9  char update(char);
10
11 int main() {
12   display();
13   string example{"This is a test"};
14
15   formatString(example);
16
17   return 0;
18 }
19
20 void display() { cout << "Welcome!!" << endl; }
21
22 void formatString(string text) {
23   const unsigned int size = text.size();
24   for (int i = 0; i < size; i++) {
25     text.at(i) = update(text.at(i));
26   }
27 }
28
29 char update(char token) { return toupper(token); }
30
```

Example

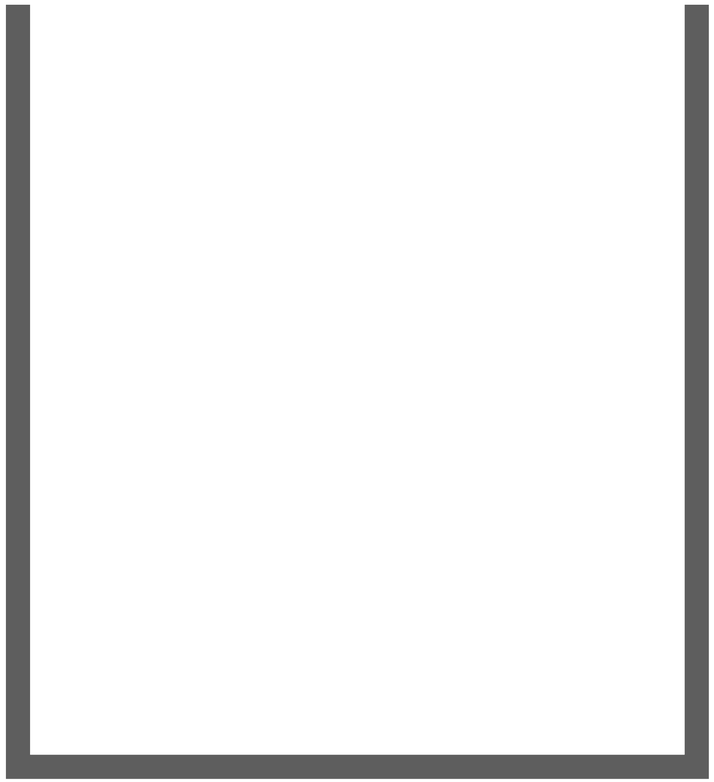
The Call Stack




```
1  #include <cstdlib>
2  #include <iostream>
3  #include <string>
4
5  using namespace std;
6
7  void display();
8  void formatString(string);
9  char update(char);
10
11 int main() {
12   display();
13   string example{"This is a test"};
14
15   formatString(example);
16
17   return 0;
18 }
19
20 void display() { cout << "Welcome!!" << endl; }
21
22 void formatString(string text) {
23   const unsigned int size = text.size();
24   for (int i = 0; i < size; i++) {
25     text.at(i) = update(text.at(i));
26   }
27 }
28
29 char update(char token) { return toupper(token); }
30
```

Example

The Call Stack



```
1  #include <cstdlib>
2  #include <iostream>
3  #include <string>
4
5  using namespace std;
6
7  void display();
8  void formatString(string);
9  char update(char);
10
11 int main() {
12   display();
13   string example{"This is a test"};
14
15   formatString(example);
16
17   return 0;
18 }
19
20 void display() { cout << "Welcome!!" << endl; }
21
22 void formatString(string text) {
23   const unsigned int size = text.size();
24   for (int i = 0; i < size; i++) {
25     text.at(i) = update(text.at(i));
26   }
27 }
28
29 char update(char token) { return toupper(token); }
30
```

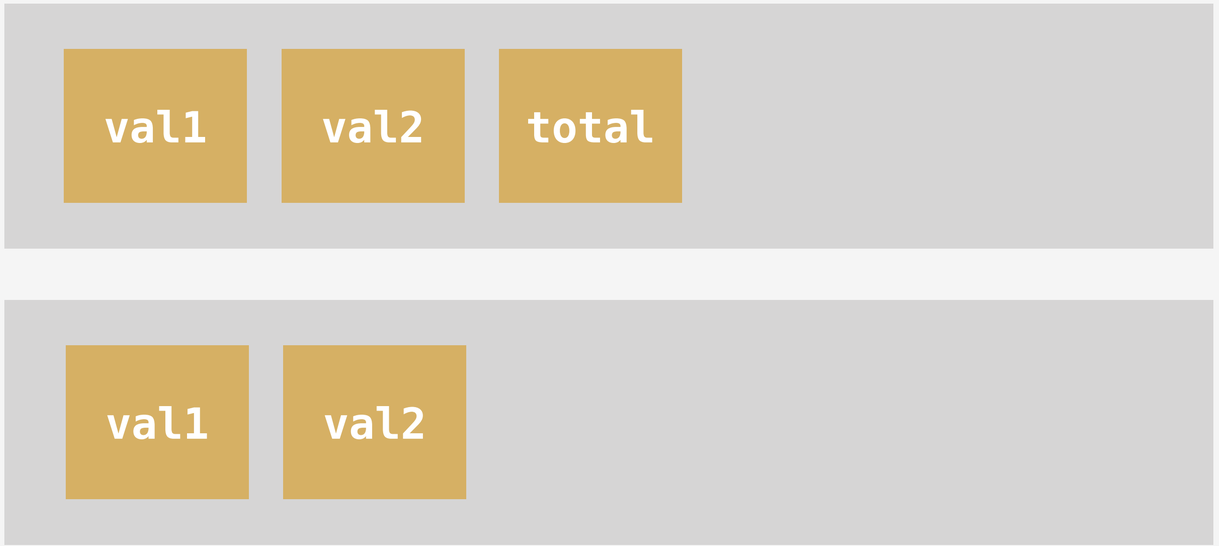
Pass by Value

C++ Functions pass parameters by ***value*** by default.

- ***Pass by value -*** creates a copy of the arguments passed in as parameters, then performs any operations on or using those local variables.

```
#include <iostream>

using namespace std;

void average(int val1, int val2) {
  double total = val1 + val2;
  cout << total / 2.0 << endl;
}

int main() {
  int val1{10}, val2{20};

  average(val1, val2);
  return 0;
}
```

Creates copies of `val1` and `val2`



examples: swapByValue.cpp

Pass by Value

```
 1 #include <iostream>
 2
 3 using namespace std;
 4
 5 void swap(int val1, int val2) {
 6   int tempVal = val1;
 7   val1 = val2;
 8   val2 = tempVal;
 9 }
10
11 int main() {
12   int val1 = 100;
13   int val2 = 200;
14
15   cout << val1 << " - " << val2 << endl;
16   swap(val1, val2);
17   cout << val1 << " - " << val2 << endl;
18
19   return 0;
20 }
```

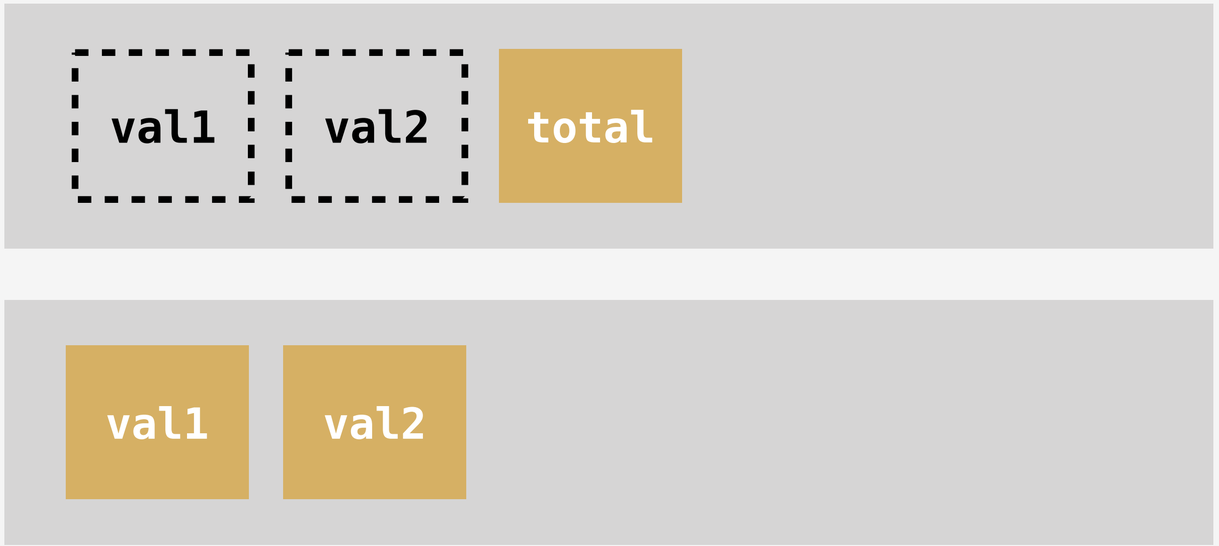
Pass by Reference

Pass by reference - passes along the arguments memory address for the variable and the function will have access to that argument variable directly

- Denote pass by reference by using the **&** operator

```
#include <iostream>

using namespace std;

void average(int &val1, int &val2) {
  double total = val1 + val2;
  cout << total / 2.0 << endl;
}

int main() {
  int val1{10}, val2{20};

  average(val1, val2);
  return 0;
}
```

Does ***NOT*** create a copy of `val1` and `val2`



examples: swapByReference.cpp

Pass by Reference

```
 1 #include <iostream>
 2
 3 using namespace std;
 4
 5 void swap(int &val1, int &val2) {
 6   int tempVal = val1;
 7   val1 = val2;
 8   val2 = tempVal;
 9 }
10
11 int main() {
12   int val1 = 100;
13   int val2 = 200;
14
15   cout << val1 << " - " << val2 << endl;
16   swap(val1, val2);
17   cout << val1 << " - " << val2 << endl;
18
19   return 0;
20 }
```

Pass by Value

- Great with simple data types and when not needing to mutate an object or variable
- Prevents side effects
- Copying values can be expensive for large data types

Pass by Reference

- Use when we want to “return” multiple values
- Great for working with large data types! No overhead of copying values
- Preferred method for efficiency!
- Use ***const*** keyword where appropriate!

Best Practice

Pass by reference if the object or variable needs to be modified

example: multiple_returns.cpp

Variable Reference

A variable ***reference*** is a type of variable that acts as an alias to another object or value.

- Can have references to const and non-const variables
- **&** - depending on usage can mean the “reference” operator or “address” operator.
- References must be initialized when created and cannot be re-assigned

Best Practice

When declaring a reference variable, put the ampersand next to the type to make it easier to distinguish it from the address-of operator.

example: reference.cpp

String and C-String Parameters

- Can pass strings and objects to functions as well
- ***String*** objects can be passed by value or by reference
- ***C-strings and other c++ arrays*** are automatically passed by pointer (essentially passed by reference)
  - This means that the function does NOT create a copy of a cstring.

```
// Function replaces spaces with hyphens
void StrSpaceToHyphen(char modString[]) {
   int i;      // Loop index

   for (i = 0; i < strlen(modString); ++i) {
      if (modString[i] == ' ') {
         modString[i] = '-';
      }
   }
}
```

example: string_params.cpp
strlen.cpp

Function Scope

- Variables declared inside a function are only visible to the compiler / program within that function (remember the call stack!)
- Cannot refer to variables out of scope

- Example:

```
void swapRef(int &val1, int &val2) {
  int tempVal = val1;
  val1 = val2;
  val2 = tempVal;
}
```

Default Parameters

Default Parameters - use default parameters to automatically assign a value to a parameter if a user does not provide a value for that argument in a function call

- Default parameters must appear in the last positions in the function declaration.
- Default values must be included in prototype ***before*** function is used

```
int add(int x, int y, int z = 0, int a = 0) {
  return x + y + z + a;
}

int main() {

  cout << add(1, 2) << endl;
  cout << add(1, 2, 3) << endl;
  cout << add(1, 2, 3, 4) << endl;

  return 0;
}
```

example: default_params.cpp

Default Parameters

```
int add(int x, int y = 0, int z = 0);

int main() {
  ...
}

int add(int x, int y, int z) {
  return x + y + z;
}
```

```
int add(int x, int y, int z);

int main() {
  …
}

int add(int x, int y = 0, int z = 0) {
  return x + y + z;
}
```

Wont Compile

Function Overloading

Function Overloading is a feature in C++ where two or more functions share the same name but have different parameter types.

- Cannot overload function non return type alone because it is not part of the unique signature used by compiler

```
print(1);
print(1.0);
print("hello");
```

example: function_overloading.cpp

The Preprocessor

The ***preprocessor*** is a tool that scans the program file before the rest of the compilation steps.

- Looks for any lines starting with ***#*** which indicates a ***preprocessor directive***
- ***#include*** - a preprocessor directive that tells the compiler to replace a line by the contents of a given filename or source.
- There are other preprocessor directives like ***#define***
- More on preprocessor directives: http://www.cplusplus.com/doc/tutorial/preprocessor/

#include

The ***#include*** preprocessor directive will include files from different locations depending on its use

- ***#include <library>*** - indicates importing a file from the c++ standard library! Usually located somewhere like (/usr/bin/g++/)
- ***#include "file.h" -*** will include a file within the same directory

- Separating our code into separate files can provide many benefits
  - Reduce code in our main.cpp
  - Reusability and organization

#include

Files in C++ are typically divided into 2 types:

- ***file.h - .h*** files or header files contain function or class declarations / prototypes
- ***file.cpp - .cpp*** files or implementation files are used for defining the function or class implementation.

- .cpp files are compile and the .h files are read in by the preprocessor and used during compilation
- This ensures that the function definitions are read in by the compiler ***before*** main();

example: functions/

Header File Guard

- Header file guards are preprocessor directives
- Cause compiler to only include contents of the file ONCE
- ***Always*** use header file guards when creating a new ***.h*** file

```
#ifndef FILENAME_H
#define FILENAME_H

// Header file contents

#endif
```