Learning Objectives: Parameters

- **Demonstrate how to define a function with parameters**
- **Identify which value is assigned to each parameter**
- **Catch exceptions with a try, throw, and catch**
- **Define functions with the same name but different parameters**
- **Demonstrate how to declare a function with an array and vector as a parameter**

Passing Parameters

Parameters

If a function contains parameters within its definition, they are required to be present when the function is called. In the example below, the function, `Add()`, adds two integer parameters together. Parameters are the types or values located in between the parentheses. Multiple parameters are separated by commas.

```
void Add(int num1, int num2) {
    cout << num1 + num2 << endl;
}

int main() {
    Add(5, 7);
    return 0;
}
```

.guides/img/Parameters1

Copy and paste the following function into the text editor to your left between the lines `//add function definitions below this line` and `//add function definitions above this line`. **DO NOT** modify `main()` yet, or the code will not print correctly!

```
/**
 * This function adds two integers together
 *
 * @param num1 The first integer
 * @param num2 The second integer
 */
void Add(int num1, int num2) {
  cout << num1 + num2 << endl;
}
```

challenge

What happens if you:

- Change the function call in `main()` to `Add(5, "seven");`?
- Change the function call to `Add();` without any parameters?
- Change the function call to `Add(5, 10, 15);`?
- Change the entire function to...

```
void Add(int num1, int num2, int num3) {
  cout << num1 + num2 + num3 << endl;
}
```

---

important

IMPORTANT

- The **number** of arguments within `main()` should match the number of parameters specified in the function. If there are three parameters, then there should be three arguments as well.
- The argument **type** should also match the parameter type. If the function requires three integers, then the arguments should also consist of three integers. You cannot provide a string argument for an integer parameter, etc.

Order of Parameters

```
void AddSub(int num1, int num2, int num3) {
   System.out.println(num1 + num2 - num3);
}

int main() {
   AddSub(5, 10, 15);
   return 0;
}
```

.guides/img/Parameters2

Much like how C++ programs run code from left to right and then top to bottom, parameters are also read the same way. Because of this, the order of parameters is important. The first argument in the function call will be matched with the first parameter in the function header, the second argument from the function call will be the second parameter in the function header, etc. Copy the entire code below into the text editor and then click `TRY IT`. What do you predict the output will be?

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

/**
 * This function adds the first two integers together,
 * then subtracts the third integer
 *
 * @param num1 The first integer
 * @param num2 The second integer
 * @param num3 The third integer
 */
void AddSub(int num1, int num2, int num3) {
  cout << num1 + num2 - num3 << endl;
}

int main() {
  AddSub(5, 10, 15);
  return 0;
}
```

challenge

What happens if you:

- Change the function call in `main()` to `AddSub(10, 15, 5);`?
- Change the function call in `main()` to `AddSub(15, 5, 10);`?
- Change the function call in `main()` to `AddSub(10 + 5, 20 / 4, 5 * 2);`?

Checking Parameters

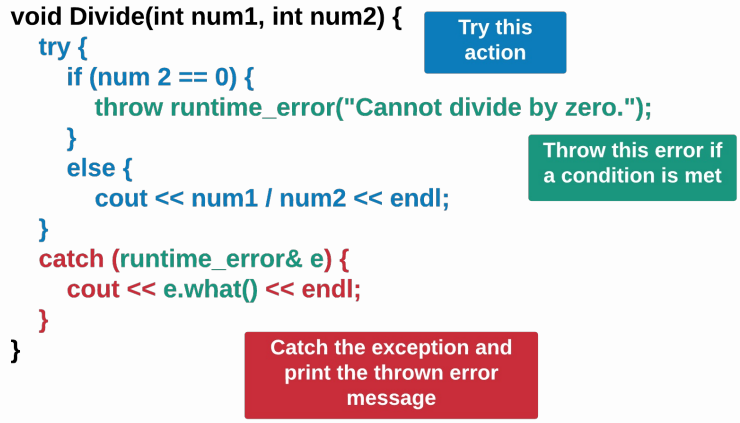
Checking Parameter Usage

Copy and paste the code below into the text editor and then `TRY IT`.

```
/**
 * This function divides one integer by the other
 *
 * @param num1 The first integer
 * @param num2 The second integer
 */
void Divide(int num1, int num2) {
  cout << num1 / num2 << endl;
}

int main() {
  Divide(5, 0);
  return 0;
}
```

You'll notice that the code produces an **exception**. An exception occurs when an operation cannot be successfully completed because a rule is broken. For example, dividing by `0` results in an *undefined* answer. Thus when you try to divide `5` by `0`, you get an exception as a response. Not all exception messages are created equal. Some are more clear than others. However, you may choose to clearly define an exception by using a `try`, `throw`, and `catch`.



.guides/img/TryCatchException

```
/**
 * This function divides one integer by the other
 *
 * @param num1 The first integer
 * @param num2 The second integer
 */
void Divide(int num1, int num2) {
  try {
    if (num2 == 0) {
      throw runtime_error("Cannot divide by zero.");
    }
    else {
      cout << num1 / num2 << endl;
    }
  }
  catch (runtime_error& e) {
    cout << e.what() << endl;
  }
}

int main() {
  Divide(5, 0);
  return 0;
}
```

challenge

What happens if you:

- Change the function call to `Divide(5, 2);`?
- Change the function call to `Divide(0, 2);`?
- Change the function call to `Divide(14.5, 2);`?
- Change the function call to `Divide(14.5, "2");`?

important

IMPORTANT

It's important to note that when arguments are passed as parameters, C++ tries to implicitly **cast** the arguments as the specified parameter type(s) first. In the example above, both `14.5` and `2` get cast as `int`s. Thus, `14.5` loses its trailing decimal places and becomes `14`. On the other hand, the string `"2"` cannot be implicitly cast as an `int` causing the system to fail to compile. **Note** that you can only catch C++ **exceptions**, not compilation **errors**.

`runtime_error()` is one example of an exception that can be used to produce a specified error message. `e` is the variable name for which you are calling the exception by. In C++, exceptions are caught by reference, not value. Thus, `runtime_error& e` is preferred over `runtime_error e`.

For a list of other exceptions, visit: C++ Exceptions.

Parameter Types

Function with Different Parameters

In C++, you are allowed to define functions with the *same* name as long as the parameters are *different* in quantity or type. Copy the code below and `TRY IT`.

```
/**
 * This function adds two integers together
 *
 * @param num1 The first integer
 * @param num2 The second integer
 */
void Add(int num1, int num2) {
  cout << num1 + num2 << endl;
}

/**
 * This function adds three integers together
 *
 * @param num1 The first integer
 * @param num2 The second integer
 * @param num3 The third integer
 */
void Add(int num1, int num2, int num3) {
  cout << num1 + num2 + num3 << endl;
}

int main() {
  Add(3, 14);
  return 0;
}
```

challenge

What happens if you:

- Change the function call to `Add(3, 14, 9);`?
- Change the function call to `Add(3, 14, 9, 2);`?

The two `Add()` functions above differ in the number of parameters they have. Here is an example of two functions with the same name but different parameter types.

```
/**
 * This function adds two integers together
 *
 * @param num1 The first integer
 * @param num2 The second integer
 */
void Add(int num1, int num2) {
  cout << num1 + num2 << endl;
}

/**
 * This function prints an integer followed
 * by a string
 *
 * @param num1 The integer
 * @param num2 The string
 */
void Add(int num1, string num2) {
  cout << num1 << num2 << endl;
}

int main() {
  Add(3, 14);
  return 0;
}
```

challenge

What happens if you:

- Change the function call to `Add(3, "14");`?
- Change the function call to `Add("14", 3);`?

Alternative Parameters

Alternative Parameter Types

Function parameters do not necessarily need to belong to one of the four commonly used data types (`int`, `string`, `double`, `bool`). In fact, parameters can be arrays/vectors and even objects. For now, we will not focus on objects, which will be covered in a future module.

```
/**
 * This function prints all values of an array
 *
 * @param array A string array
 */
void PrintArray(string array[], int size) {
  for (int i = 0; i < size; i++) {
    cout << array[i] << endl;
  }
}

int main() {
  string names[] = {"Alan", "Bob", "Carol"};
  int len = sizeof(names) / sizeof(names[0]);
  PrintArray(names, len);
  return 0;
}
```

challenge

What happens if you:

- Change `string names[] = {"Alan", "Bob", "Carol"};` to `string names[3];`?
- Add `names[0] = "Alan";` to the line below `string names[3];`?
- Change the first function parameter of `string array[]` to `string* array`?

Why Doesn't the Code Below Work?

```cpp
/**
 * This function prints all values of an array
 *
 * @param array A string array
 */
void PrintArray(string array[]) {
  for (int i = 0; i < sizeof(array) / sizeof(array[0]); i++) {
    cout << array[i] << endl;
  }
}

int main() {
  string names[] = {"Alan", "Bob", "Carol"};
  PrintArray(names);
  return 0;
}
```

important

IMPORTANT

When an array is passed as a function argument in C++, the system treats the array as a pointer that points to the first element within the array. Thus, the parameter `string array[]` is the same as `string* array`. Due to this, knowledge of the size of the array is lost. This is why it is a good practice to include an integer parameter for functions involving arrays so that the size can be calculated and stored before those functions are called.