Passing Parameters by Reference and Value

Thomas Merkh
tmerkh55@gmail.com

Section 2, Week 7

May 5, 2020
Overview

1. Passing Parameters

2. Const specification

3. Next Time
Last time we discussed optional arguments for functions. As a refresher:

- Functions can have *default values* specified for some or all of their parameters.
- If not called with a value for each of these optional parameters, the default values will be used.
- Functions are abstract objects in C++ and can also be passed around just as variables can be. This is a *functional-programming*-like aspect of C++.
Last time we discussed optional arguments for functions. As a refresher:

- Functions can have *default values* specified for some or all of their parameters.
- If not called with a value for each of these optional parameters, the default values will be used.
- Functions are abstract objects in C++ and can also be passed around just as variables can be. This is a *functional-programming*-like aspect of C++.

Today, we will learn the important topic of parameter passing. Specifically, we will see that there are (at least) two different ways to pass parameters to functions, each with their own consequences.
1 Passing Parameters

2 Const specification

3 Next Time
When you run an executable file

Other than the first week, we haven’t discussed in detail much on “what’s going on under the hood” when we compile and run a program. Discussing this may help you understand references and pointers, which are the upcoming topics.
When you run an executable file

Other than the first week, we haven’t discussed in detail much on “what’s going on under the hood” when we compile and run a program. Discussing this may help you understand references and pointers, which are the upcoming topics. When you run an executable file:

- The OS load the program into memory, and begins executing each command sequentially.
- As objects are initialized, memory is allocated for different data members of the program, i.e. variables, constants, etc.
- For example, initializing `int k = 5;` allocates a 4 byte chunk of memory called `k`’s address, to store the value 5.
Other than the first week, we haven’t discussed in detail much on “what’s going on under the hood” when we compile and run a program. Discussing this may help you understand references and pointers, which are the upcoming topics. When you run an executable file:

- The OS load the program into memory, and begins executing each command sequentially.
- As objects are initialized, memory is allocated for different data members of the program, i.e. variables, constants, etc.
- For example, initializing `int k = 5;` allocates a 4 byte chunk of memory called `k`’s address, to store the value 5.

Okay, so what is the big deal?
Memory Address Operator

Since the variable $k$ has both a *value* 5 and *an address* (e.g. 0056F7B8) in computer memory, this gives us two options on how we can pass around the variable $k$. Then the question becomes, how do we access the address of $k$?

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

int main()
{
    int k = 5;
    cout << "The address of k is: " << &k << endl;
    return 0;
}
```

This outputs "The address of $k$ is 0056F7B8".
Since the variable \( k \) has both a \textit{value} 5 and \textit{an address} (e.g. 0056F7B8) in computer memory, this gives us two options on how we can pass around the variable \( k \). Then the question becomes, how do we access the address of \( k \)? The answer is the \& operator. Consider the following code:

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

int main() {
    int k = 5;
    cout << "The address of k is: " << &k << endl;
    return 0;
}
```

This outputs "The address of \( k \) is 0056F7B8".
A program works by allocating memory for each of the variables, fetching the values from those addresses, doing some computations, and storing them in possibly different memory addresses.

Note: A reference is not an object. Anything which has an allocated space in memory and can hold data is considered an object in C++.

Question

Consider storing a 1000 by 1000 HD image, call this image myImage. This image requires well over 1 million values to fully specify each pixel color. If we want to pass this image to the function IsThisACat(image_data_type), do we need to pass 1000000+ values?
Of course not! Passing this image *by value* consists of sending 1000000+ values over to the function’s location to do some computations. Since the image lives somewhere in memory, we can simply pass its address to the function, called passing *by reference*, and doing so requires us to transfer only the hexadecimal address of the image. The following code shows the difference.
The following code shows the difference between passing by value and passing by reference, to achieve the same objective:

def double avg(double a, double b)
    cout << 'avg a address: ' << &a << endl;
    return (a + b) / 2.0;
}

double avgRef(double &a, double &b)
    cout << 'avgRef a address: ' << &a << endl;
    return (a + b) / 2.0;
}

int main()
{
    double a = 6;
    double b = 9;
    cout << 'The address of a is: ' << &a << endl;
    cout << avg(a, b) << endl;
    cout << avgRef(a, b) << endl;
    return 0;
}
This code outputs:

The address of a is: 00FBF8B4
avg a address: 00FBF7C0
avgRef a address: 00FBF8B4

Notice that the address of a in the avgRef function is the same as a’s original address! On the other hand, the address of the variable a inside of avg is different, why?
This code outputs:

The address of a is: 00FBF8B4
avg a address: 00FBF7C0
avgRef a address: 00FBF8B4

Notice that the address of a in the avgRef function is the same as a’s original address! On the other hand, the address of the variable a inside of avg is different, why?

If you said, because a copy of the value of a was created when pass a into the function avg, and then a temporary variable a with a new memory address was used for the computation in avg, you’d be correct!
What is the main point here?

- When passing by reference, modifications made to arguments passed in to the called function are retained when the function concludes.
- When passing by value, modifications made to arguments passed in do not effect the values in the code block calling the function.
- Passing by reference is almost always more efficient than passing by value, since only a memory address needs to be sent to the called function.
- The syntax is nearly the same, where all we need to do is change a function’s parameter from (type variable) to (type &variable).
Passing Parameters

Const specification

Next Time
The const Qualifier

Perhaps you’ve already encountered a variable with const int type. This is our first instance of a so-called compound type\(^1\), since the keyword const is combined with a primitive type like int to create a const int.

What effect does this have on the variable? And why would we use it?

\(^1\)Technically, a reference or pointer to an primitive type is also considered a compound type.
The const Qualifier

Perhaps you’ve already encountered a variable with const int type. This is our first instance of a so-called *compound type*¹, since the keyword const is combined with a primitive type like int to create a const int.

What effect does this have on the variable? And why would we use it?

- If a variable has the const qualifier, the value of the variable can not be changed
- const objects have most of the same operations as nonconst objects, as long as those operations do not change the variable’s value
- At compilation time, the compiler literally replaces all instances of a const variable with its value

¹Technically, a reference or pointer to an primitive type is also considered a compound type.
Because `const` types can not be reassigned, they **must** be initialized when declared. For example, writing `const int k;` would throw an error.

Later in the class, we will start writing C++ programs which are split across several files. It's important to know that:

- A `const` is *local* to the file its initialized in. Initializing it in `foo.cpp` means it will not be defined in `bar.cpp`, even if those files are linked!
- This led C++ creators to creating the `extern` qualifier, which makes the compiler aware that this `const` type variable is initialized in a linked file.
Passing a reference to a `const`

Consider the function prototypes:

```c
void DoWork(int n);
void DoWork(const int &n);
```
Consider the function prototypes:

```c
void DoWork(int n);
void DoWork(const int &n);
```

- When we pass by value, a copy of the parameter we pass is created
- When passing a reference to a `const`, the original data is not copied
- **In both cases, the original data can not be modified from inside the function**
- Passing the reference is way more efficient than passing by value, so each method has its merits
An important point to make is that a reference to `const` restricts only what we can do through the reference. Consider

```c
int i = 42;
int &r1 = i;  // r1 bound to i
const int &r2 = i;  // r2 also bound to i,
// but can’t be used to change i
```

```c
r1 = 0;  // r1 is not const, i is now 0
r2 = 0;  // error: r2 is a reference to const
```

Compound types are not a simple as they may appear. Sometimes extra special care is needed to make sure a variable’s type is what you mean it to be.
1. Passing Parameters

2. Const specification

3. Next Time
Next Time

Next time we will see topics such as:
- Scope, Unit testing - modular programming, Debugger - need to learn how to use this for visual studio.

Office hours are on a request only basis. Please email me if you’d like to meet.