Introduction

This document provides a basic introduction to function parameters, overloading, and default arguments. For any given topic discussed in this document in regards to function parameters and overloading, there are additional important details not discussed because this is a basic introduction. There are numerous official online resources for C++ that can be consulted for these details. Links to some of these resources, specifically Microsoft's website, C++ Reference, and the ISO C++ Standard, are provided at the end in the **Resources** section.

- Microsoft's website and C++ Reference provide a general reference for any C++ related topic. Microsoft's website is a bit easier to understand and more beginner friendly.
- The ISO C++ Standard contains the C++ Core Guidelines which is a discussion on the current C++ best practices and standards. Rather than being a general reference, it's more of a list of best practices in all different situations.
- Microsoft's website and C++ Reference are what should be used by a beginner for general C++ learning. The ISO C++ Core Guidelines should be used by someone already well versed in C++ who is looking for a more detailed and nuanced discussion about C++ best practices.

Note: For the purposes of demonstration, the std namespace is being used by including the statement using namespace std. This is just to make code easier to read. For example, it allows string to be explicitly written instead of std::string to refer to a C++ string, and it allows *cout* to be explicitly written instead of std::*cout*. It is okay to use the std namespace for the purposes of learning, teaching, or demonstration, but in real C++ code it is not a good practice.

Table of Contents

Function Parameters in C

Before discussing function parameters in C++, it is important to review how they work in C as well as the basics of variables and pointers.

Recall the following definitions:

- **Variable:** a named location in memory
- **Pointer:** a variable that holds the address of another variable. Using the definition of a variable, it can be rephrased as:
  a named location in memory that holds the address of another variable.

In C, there are two methods that can be used with function parameters - **pass by value** and **pass by reference**.

- In **pass by value**, a variable is passed as an argument to a function, and a copy of the value stored in the variable is passed to the local variable that is the formal parameter. The variable cannot be modified by the function - it can only modify the local variable that is the formal parameter.
- In pass by reference, the address of a variable is passed as an argument to a function, and that address is stored in the local variable that is the formal parameter (in this case a pointer). Since the local variable has the address of the variable, the function has the ability to modify the variable. Any and every time a function needs to change the value stored in a variable passed to it as an argument, this method must be used.

Take the following example function that is supposed to add 2 to an integer

```
// pass by value
void add_2_v1(int x) {
    x = x + 2;
}

// pass by reference
void add_2_v2(int *x) {
    *x = *x + 2;
}

int main(int argc, char* argv[]) {
    int x = 4;
    add_2_v1(x);     // pass by value, x is still 4
    add_2_v2(&x);    // pass by reference, x is now 6
    return 0;
}
```

The function *add_2_v1* uses pass by value and does not modify the variable x in main. What happens is that a variable named x gets initialized in the main function with a value of 4. When the function *add_2_v1* is called with x as an argument, a copy of the value stored in x (in this case 4) is stored in the local variable x in the function. This x is a completely different x from the x in the main function. The line

```
x = x + 2
```

therefore modifies the local variable x and not the variable x in the main function. Thus, x remains unchanged.

So, the solution to this is to make a function like *add_2_v2* which uses pass by reference. Instead of passing a copy of the value in x, the address of x is passed to x. In this case, x is not an integer variable but rather a pointer to an integer variable. So, instead of holding the value 4, x holds the address of x. Then, the dereference operator "*" is used to modify x. The dereference operator essentially means go to the address that is stored in the variable being dereferenced.

Let's say x in the main function had an address of A

- x evaluates as the place in memory holding the value 4
- &x evaluates as A, the address of the place in memory holding the value 4
- x evaluates as A, the address of the place in memory holding the value 4
- *x evaluates as x, the place in memory holding the value 4

So, the line

*x = *x + 2

Is equivalent to

x = x + 2

It adds 2 to *x*, which once again is the x value in the main function. So as it can be seen, in order for a function to modify a variable in C, the function must use pointers aka use pass by reference.

Function Parameters in C++

In C++, there is an additional way a function can modify the value of a variable. It's still perfectly valid for a function in C++ to modify the value of a variable in the way previously described using pointers and addresses. However, this new way in C++ is simpler to implement and is the preferred method. It also has another use case (preventing unnecessary copy operations) talked about on the next page).

The way C++ created this new method is that it gave an additional meaning to the *&* operator. In C, the *&* operator always means the **address of** operator (with the exception of the bitwise *&* operator) and it's used when passing the address of a variable as an argument to a function, or assigning the address directly to a pointer like

```
int x = 4;
add_2_v2(&x);
int *pX = &x;
```

In C++, the *&* operator can also be used in the formal parameters in which case it is the **reference** operator. This means that instead of a non-pointer variable holding a copy of the value of the argument passed to the function, it actually refers to the actual variable passed as an argument to the function, and since it refers to the actual variable passed as an argument to the function, the function can therefore modify the variable. If this is confusing, the following example should provide clarity.

```
// pass by reference originally in C
void add_2_v2(int* x) {
    *x = *x + 2;
}

// additional pass by reference way in C++
void add_2_v3(int& x) {
    x = x + 2;
}

int main(int argc, char* argv[]) {
    int x = 4;
    add_2_v2(&x);   // pass by reference in old C way, x is now 6
    add_2_v3(x);    // pass by reference in new C++ way, x is now 8
    return 0;
}
```

As seen in the above example, *add_2_v2* is the way of doing pass by reference in C, and this was already discussed in the previous section. Then, *add_2_v3* is the new way of modifying a variable in C++. It is simpler - there are no pointers or addresses used. The formal parameter

```
int& x
```

is saying that the local variable (the formal parameter) x is a reference to the variable *x* in the main function. In other words, both x and x refer to the same location in memory holding that value of 4. So, the statement

```
x = x + 2;
```

actually modifies the variable x in the main function.

Using Pass By Reference To Prevent Unnecessary Copy Operations

Aside from giving functions the ability to modify its parameters, pass by reference is also used to avoid unnecessary copy operations especially for objects. Recall, as previously discussed, that pass by value stores a copy of the argument in the formal parameter, whereas pass by reference means the formal parameter refers to the actual argument - there is no copy being made. When the formal parameter is an object, this means that with pass by value an entire object would have to be copied. In a general sense, this is not efficient, and for objects with lots of data, it is especially inefficient. For example, take the following functions which print out a vector of integers

```
// function definitions
void printVector1(vector<int> v) {
        for (auto i = v.begin( ); v != v.end( ); i++)
                cout << *i << endl;
}

void printVector2(const vector<int>& v) {
        for (auto i = v.begin( ); v != v.end( ); i++)
                cout << *i << endl;
}

// function calls
vector<int> v;
for (int i = 0; i < 1000000; i++)
        v.push_back(i);

printVector1(v);
printVector2(v);
```

As seen in the above example, v is a vector of 1,000,000 integers. When *printVector1* is called, it uses pass by value so v is a copy of v, and what prints out is that copy. So, a wasteful and unnecessary copy operation had to be done just to do the simple task of printing out a vector of integers. Whereas in *printVector2,* it uses pass by reference so v is a reference to v. Therefore, what prints out is actually v and there was no unnecessary copy operation. The const keyword is there because it is standard practice to include it in any function which is not going to modify an object. This helps prevent accidentally modifying the object because if some modification did happen, it would be treated as an error since const is present.

Function Overloading in C++

General Introduction

Function overloading in C++ is when two or more functions have the same name but have a different **function signature**, where a different function signature means one or more of the following:

- The number of formal parameters is not the same
  ```
  void foo(int n1);
  void foo(int n1, int n2);
  ```
- The type of at least 1 of the formal parameters is not the same
  ```
  void foo(int n1, int n2);
  void foo(double n1, int n2);
  ```
- Or some combination of both
  ```
  void foo(int n1);
  void foo(double n1, int n2);
  ```

Note that there are three changes that can be made to a function which would not result in overloaded functions. Rather, they would just result in invalid code. These are changing the return type, changing the formal parameters by using the const keyword, or changing by formal parameters by using C++ pass by value

- Different return types - invalid
  ```
  void foo(int n1);
  int foo(int n1);
  ```
- Formal parameters differ because of const - invalid
  ```
  void foo(const int n1);
  void foo(int n1);
  ```
- Formal parameters differ because of pass by value vs. pass by reference - invalid
  ```
  void foo(int& n1);
  void foo(int n1);
  ```

Why Function Overloading Exists in C++

The reason that C++ added function overloading was so that many different functions, all with different names, would not have to be created to do the same operation. C does not have function overloading. In C, to create a function that calculates the average of 2 integers or two floating point numbers, two functions with unique names would have to be created like

```
double averageInt(int n1, int n2);
double averageDouble(double n1, double n2);
```

This would get even more complicated if the average needed to be calculated for more than two numbers

```
double averageInt2Arg(int n1, int n2);
double averageInt3Arg(int n1, int n2, int n3);
double averageDouble2Arg(double n1, double n2);
double averageDouble3Arg(double n1, double n2, double n3);
```

Every time the average were to be calculated, the correct function would have to be chosen. By having function overloading in C++, the same function name can be used for any and all situations which just makes the process easier. The next page discusses the rules about how the compiler chooses which function to call.

Compiler Rules for Function Overloading

The way the compiler chooses which actual overloaded function to use is based on the following rules. The order matters - if the first rule is not met, the compiler will move onto the next rule.

1) Look for an exact match
2) Match by doing promotion within integer types or floating-point types. Promotion means type casting to larger byte versions like going from short to int or float to double. Note that for integer types, bool to int and char to int are also promotions.
3) Match by doing other conversions of predefined types, like converting from int to double.
4) Match using conversions of user defined types.
5) Match using ellipses (also called **variadic arguments**, discussed at the end of the page).

Take the following example:

```
// 4 overloaded functions
void foo(int n1);
void foo(double n1);
void foo(int n1, int n2);
void foo(double n1, double n2);
// function calls
foo(1);             // calls the first foo
foo(1.5);           // calls the second foo
foo(1, 2);          // calls the third foo
foo(1.5, 2.5);      // calls the fourth foo
```

The rules work in almost all situations, but certain situations can lead to an error. For example

```
// 2 overloaded functions
void foo(double n1, int n2);
void foo(int n1, double n2);
// function call
foo(2, 3);
```

This would lead to an error because the compiler would not know which overloaded function to select. Set up the function overloading in such a way so that this can't happen, or just avoid doing a function call that would lead to this error, though the former is definitely preferred - it's always better to guarantee an error cannot possibly happen.

Matching with ellipses

There is a special kind of function overload that uses **variadic arguments** denoted by ellipses.

```
void foo(...);
```

The ellipses mean that foo can take an unspecified amount of arguments of any type (including 0 arguments). It can also be used in conjunction with some preceding formal parameters.

```
void foo(int n1, …);
```

This foo means that the first argument is an integer followed by 0 or more arguments of any type. Note that passing an object as one of the arguments in a function that uses ellipses is undefined behavior.

The use of ellipsis has some pitfalls, so in general it's not recommended to use it unless the pitfalls are well understood and there is some unique situation requiring a need for it.

Default Arguments

Default arguments can be specified for call-by-value formal parameters. If any of the arguments are not included in the function call, the default values are used. For example,

```
void foo(int n1, int n2 = 8, int n3 = 4);

int main(int arg, char* argv[]) {
    foo(1, 2, 3);           // n1 = 1, n2 = 2, n3 = 3
    foo(1, 2);              // n1 = 1, n2 = 2, n3 = default value of 4
    foo(1);                 // n1 = 1, n2 = default value of 8, n3 = default value of 4
    foo( );                 // invalid since not all parameters have default arguments
    return 0;
}

void foo(int n1, int n2, int n3) {
    cout << n1 << endl;
    cout << n2 << endl;
    cout << n3 << endl;
}
```

Note how this looks similar to function overloading. If only the function calls were viewed, it couldn't be seen whether the functions being called were overloaded functions, or if they were for a function with default values.

There are a few rules with default arguments

1) Default arguments must show up wherever the function shows up first - if the function declaration came before the definition, they go in the declaration and not the definition (note how this is the case in the above example). If the definition comes first (such as an inline function, which has no declaration to begin with), they go in the definition.
2) Default arguments must always be in the right most positions. For example, the following is invalid
   ```
   void foo(int n1 = 1, int n2 = 8, int n3);
   ```
   For it to be valid, it would have to be changed to
   ```
   void foo(int n1, int n2 = 1, int n3 = 8);
   ```
   or if it was desired to not change the names of parameters that got default values
   ```
   void foo(int n3, int n1 = 1, int n2 = 8);
   ```
3) Going off of rule 2, when function calls are made, the arguments to be omitted must start from the right. The combination of rule 2 and 3 means the only way a parameter can have a default value is if all of the parameters to the right of it also have default values

Why Default Arguments Exist in C++

There are countless examples of where default arguments could be useful, and as with function overloading it is the responsibility of the programmer to determine when they are appropriate for use. For example, a person may or may not have a middle name. Therefore, default arguments could be helpful to use in a function that prints a person's name as seen in the example below. Notice how the example could also be implemented by using function overloading and not default arguments. For such a simple example, choosing one over the other wouldn't matter. Using default arguments is slightly easier in this case because only one function needs to be written instead of 2 overloaded functions where one takes 3 strings and the other takes 2 strings as arguments.

```
void printName(string firstName, string lastName, string middleName = "N/A");

int main(int arg, char* argv[]) {
    printName("John", "Smith", "Michael");      // prints John Michael Smith
    printName("John", "Smith");                 // prints John Smith
    return 0;
}

void printName(string firstName, string lastName, string middleName) {
    cout << firstName << " ";
    if (middleName != "N/A") {
        cout << middleName << " ";
    }
    cout << lastName << endl;
}
```

Resources

C++ References

- Microsoft: https://docs.microsoft.com/en-us/cpp/cpp/references-cpp?view=msvc-170
- C++ Reference: https://en.cppreference.com/w/cpp/language/reference

Function Overloading

- Microsoft: https://docs.microsoft.com/en-us/cpp/cpp/function-overloading?view=msvc-170
- C++ Reference: https://en.cppreference.com/w/cpp/language/overload_resolution

Default Arguments

- Microsoft: https://docs.microsoft.com/en-us/cpp/cpp/default-arguments?view=msvc-170
- C++ Reference: https://en.cppreference.com/w/cpp/language/default_arguments

ISO C++ Core Guidelines

Main Website: https://isocpp.org/
Core Guidelines: http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines