7.2 — Passing arguments by value

BY ALEX ON JULY 23RD, 2007 | LAST MODIFIED BY ALEX ON MAY 10TH, 2018

Pass by value

By default, non-pointer arguments in C++ are passed by value. When an argument is **passed by value**, the argument's value is copied into the value of the corresponding function parameter.

Consider the following snippet:

```
#include <iostream>

void foo(int y)
{
    std::cout << "y = " << y << '\n';
}

int main()
{
    foo(5); // first call

    int x = 6;
    foo(x); // second call
    foo(x+1); // third call

    return 0;
}
```

In the first call to foo(), the argument is the literal 5. When foo() is called, variable y is created, and the value of 5 is copied into y. Variable y is then destroyed when foo() ends.

In the second call to foo(), the argument is the variable x. x is evaluated to produce the value 6. When foo() is called for the second time, variable y is created again, and the value of 6 is copied into y. Variable y is then destroyed when foo() ends.

In the third call to foo(), the argument is the expression x+1. x+1 is evaluated to produce the value 7, which is passed to variable y. Variable y is once again destroyed when foo() ends.

Thus, this program prints:

```
y = 5
y = 6
y = 7
```

Because a copy of the argument is passed to the function, the original argument can not be modified by the function. This is shown in the following example:

```
#include <iostream>

void foo(int y)
{
    std::cout << "y = " << y << '\n';

    y = 6;

    std::cout << "y = " << y << '\n';
} // y is destroyed here

int main()
{
    int x = 5;
    std::cout << "x = " << x << '\n';

```

```
17      foo(x);
18
19      std::cout << "x = " << x << '\n';
20      return 0;
21  }
```

This snippet outputs:

```
x = 5
y = 5
y = 6
x = 5
```

At the start of main, x is 5. When foo() is called, the value of x (5) is passed to foo's parameter y. Inside foo(), y is assigned the value of 6, and then destroyed. The value of x is unchanged, even though y was changed.

Function parameters passed by value can also be made const. This will enlist the compiler's help in ensuring the function doesn't try to change the parameter's value.

Pros and cons of pass by value

Advantages of passing by value:

- Arguments passed by value can be variables (e.g. x), literals (e.g. 6), expressions (e.g. x+1), structs & classes, and enumerators. In other words, just about anything.
- Arguments are never changed by the function being called, which prevents side effects.

Disadvantages of passing by value:

- Copying structs and classes can incur a significant performance penalty, especially if the function is called many times.

When to use pass by value:

- When passing fundamental data type and enumerators, and the function does not need to change the argument.

When not to use pass by value:

- When passing arrays, structs, or classes.

In most cases, pass by value is the best way to accept parameters of fundamental types when the function does not need to change the argument. Pass by value is flexible and safe, and in the case of fundamental types, efficient.


7.3 -- Passing arguments by reference


Index


7.1 -- Function parameters and arguments