Side Effects

Arguments in the original C language were always passed *by value*. This was done to avoid *side effects*. Consider a little different version of the factorial function:


post_decrement.cpp

```
/*************************************************************
 * factorial
 *
 * Parameters: n: the integer whose factorial is to be computed
 *                (assert non-negative)

 * Modifies: nothing
 *
 * Returns: the factorial of n
 *************************************************************/

int factorial(int n){
    int result = 1;
    while (n > 1)
        result *= n--;
    return result;
}
```

The line

```
result *= n--;
```

actually contains two potential side effects. Let's focus first on the one caused by decrementing in the middle of an expression.

The line is equivalent to the following two lines

```
result *= n;
 n--;
```

That is the original, in one line, both

1. computes the next value of `result` (the *main effect*) as well as
2. decrements `n` (the *side effect*).

But when are we decrementing `n`? Notice in this second version it is very clear that `n` is *decremented after we used its previous value*.

Post- and Pre- Increment and Decrement

When we increment or decrement, there are actually two forms of the operators, known as *post*-decrement (or post-increment) and *pre*-decrement (or pre--increment). In the example above the decrement operator is said to be in the post position (it comes after the variable being decremented).

It is perfectly grammatically correct to put the decrement operator before the variable as:

```
result *= --n;
```

but now it means something different. Again, writing it as two lines we would need to write it this way—

```
n--;
result *= n;
```

Putting an increment/decrement operator in the pre- position means that *the increment/decrement operation is carried out before the value of the variable is used.*

Here is the same program ammended to use pre-decrement. Step through both versions in the teaching machine to see the difference


pre_decrement.cpp

```
/*************************************************************
 * factorial
 *
 * Parameters: n: the integer whose factorial is to be computed
 *                (assert non-negative)

 * Modifies: nothing
 *
 * Returns: the factorial of n
 *************************************************************/

int factorial(int n){
    int result = n;
    while (n > 2)
        result *= --n;
    return result;
}
```

While this second version works it is far less understandable than the first.

In fact, in general we regard incorporating incrementing into expressions as a side effect to be quite advanced programming

1. We recommend you not use the technique yourself. Instead, increment separately.
2. However, it is a sufficiently common part of the standard C++ *idiom* that we do expect you to be able to read and understand code that does use it.

Final Caveat

increment/decrement operators should never be incorporated into expressions on variables that are used more than once in the expression.

For example

```
    y = ++n * log(n);
```

should never be used. The `n` is incremented on the memory fetch part of the evaluation cycle and there is no way to predict which `n` optimizing compilers will decide to fetch first.

Pass-by-Value

We said there were potentially two side effects in the original piece of code. In decrementing `n` we are decrementing the value passed into us. In other words, we are changing it.

Pass-by-value was designed as a security feature to guard against just this problem. The `n` in the function refers to a local copy of the value of the original variable (which just happens to be called `n` in this program as well.

Here's the calling part of the original program:


post_decrement.cpp

```
int main(){
    int n;  // the number whose factorial is to be computed
    int fact; // its factorial

    cout << "A program to compute factorials." << endl;
    cout << "Please enter a non-negative integer: ";
    cin >> n;
    while ( n < 0) {
        cout << "Factorials can only be computed for "
             << "non-negative integers. Please enter again: ";
        cin >> n;
    }
    fact = factorial(n);
    cout << "The factorial of " << n << " is " << fact << endl;
    return 0;
}
```

Although we changed the value of `n` in the factorial function it doesn't affect the original `n` *because it is a different variable*. This is exactly what pass-by-value means. Instead of passing the original variable to the factorial function we pass just its value and make a new variable.

Thus, pass-by-value *always implies copying data*. It is this copy of the original variable which is decremented to `1` inside the factorial loop.

While this is secure (consider how surprised we would be if the line

```
 cout << "The factorial of " << n << " is " << fact << endl;
```

printed out `1` for `n` in the program above) there are times when it is not useful.

A Swap Function

Consider the following highly desirable function:


non_swap.cpp

```
 // A totally useless function!!!

void intswap(int x1, int x2){
    int temp;

    temp = x1;
    x1 = x2;
    x2 = temp;
}
```

The idea behind this swap routine is that it swaps the values of a pair of variables around. Swap routines are widely used. As we will find out, all routines to sort data into some sort of order work by comparing two pieces of data and, if they are out of order, swapping them.

If you run the routine TM you will find the swapping works just fine.

The only problem is, nothing happens from the calling routine's perspective.


non_swap.cpp

```
int main(){
    int a = 2;
    int b = 3;
    intswap(a,b);
    cout << "a is " << a << " and b is " << b << endl;
    return 0;
}
```

As we saw, the `intSwap` function merrily swaps `x1` and `x2,` but `a` and `b` are untouched because *`x1` is a copy of `a` and `x2` is a copy of `b`!*

In the case of the factorial function, this behaviour was a desirable security feature.

Now suddenly its a bug. How can we get around it?

Pass-by-Reference

C++ inherited C's pass-by-value design. But C++ added a new operator `&` called the *reference operator* to take care of occasions when we really would like to use the original variables in a function and not just the values.

Here's a new version of the `intSwap` function that uses this *pass-by-reference* technique.


int_swap.cpp

```
 // A useful function!!!

void intswap(int& x1, int& x2){
    int temp;

    temp = x1;
    x1 = x2;
    x2 = temp;
}
```

The only thing that has changed is the declaration of the two function arguments.

Now, instead of `x1` and `x2` being `int`s, they are formally called `references to int`s. In the syntax of C++ a `reference to int` (or `reference to double` or `reference to an object of class string`) is *a distinctly different type* from an `int` (or `double` or `object of class string`).

the variables are the function parameters and they are copies of the original variables.

Note that the call to the pass-by-reference (useful) version of the `intSwap` function is identical to the (useless) pass-by-value version.


int_swap.cpp

```
int main(){
    int a = 2;
    int b = 3;
    intswap(a,b);
    cout << "a is " << a << " and b is " << b << endl;
    return 0;
}
```

Notice, there is *no way you can tell by looking at a call* whether the arguments are being passed by value or by reference. You *have* to look at the function declaration to know.

```
void intSwap(int& x1, int& x2);
```

is the declaration for a function that passes its arguments by reference while

```
int factorial(int n);
```

is the declaration for a function that passes its arguments by value.

Pass-by-Reference for Multiple Outputs

We have said that arguments represent the *inputs* to a function while its return value (if there is one)

represents its *output*.

You have already discovered that there are times when the limitation of a single output can be awkward.

However, arguments that are passed by reference can be both inputs and outputs.

Consider the following example:


triangle_area.cpp

```
int main(){
    float tBase;
    float tHeight;

    cout << "A program to calculate the area of a triangle." << endl;

    getValues(tBase, tHeight);
    cout << "The area is " << tBase * tHeight / 2 << endl;
    return 0;
}


void getValues(float& base, float& height) {
    string prompt;
    prompt = "base";
    base = getFloat(prompt);
    prompt = "height";
    height = getFloat(prompt);
}
```

As usual we like to modularize our problem using functions. As we have often done on in-class examples we hide the ugly details of prompting for and retreiving values in a function.

And how often, when faced with that very problem, have we not said or thought how nice it would be to return multiple values at once?

Well we can't actually return multiple values. But we do use pass-by-reference to have the `getValues` function set both the `base` and the `height` for us.

To make sure it works we have to make sure we give it variables for arguments when we make the call and not just values.

Const References

The display above doesn't actually show the `getFloat` function implementation or declaration. If you've run the example in the TM already you may have noticed something strange about them. Here's the implementation.

RUN

triangle_area.cpp

```
float getFloat(const string& what){
    float theInput;

    cout << "\nPlease input a value for " << what << ": ";
    cin >> theInput;
    return theInput;
}
```

What's up with the const keyword in the function prototype

```
float getFloat( const string& what)
```

In the first place why not just pass the string by value as we would normally do? After all we don't want to change it. It's just input for the function.

Remember that pass-by-value means a copy gets made. In this case a full copy of the string object (although you won't see it reflected in your simple memory model on the TM—it's done behind the scenes in another part of memory called the heap that we won't talk about until the advanced course). Making such a copy is potentially very expensive because strings can be very large.

1. It's expensive because it takes *time* to make the full copy.
2. It's expensive because it takes *extra memory to hold* the copy.

Passing-by-reference suppresses copying. The only thing passed into the function is the reference to the original object (which is about the size of an `int` or `long`).

However, there is an implication when we pass by reference that the variable will be modified.

The `const` tells the compiler (and more importantly, any clients who are going to use our function) that although `prompt` is being passed by reference *it will not be modified*. In other words, it is read-only.

Summary

1. pass-by-value means a copy is made of the original data.
2. copying the original data means that any original variables "passed" to a function can not be modified.
3. pass-by-reference allows variables passed to a function as arguments to be modified by the function.
4. pass-by-reference gives us a way for a function to "return" multiple values.
5. pass-by-reference suppresses copying
6. Putting const in front of a reference parameter signifies it will not be modified by the function.
7. For primitive parameters (e.g. `double`, `int`, `char`) always use pass-by-value unless a parameter is to be modified.
8. For objects, always use `const` pass-by-reference (unless a parameter is to be modified in which

case drop the `const`).

This page last updated on Monday, March 22, 2004