# CS225 : Data Structures and Software Engineering
# Parameter Passing, Reference Variables, and `const`

Jason Zych

# Chapter 1

# Parameter passing

## 1.1 Arguments and Parameters

The symbol "&" has a few different purposes, depending on where it occurs in the code. As we have already seen in the pointer section, "&" can be an operator meaning "address of". However, it only has that function when it appears in front of a variable name inside a block of code. The symbol "&" may also appear after a type in a function signature, and in those situations, the "&" indicates a parameter that is a *reference variable*.

In order to accurately explain what a reference variable does (at least, in the context of passing parameters to functions), it is necessary to also understand how parameters can be passed to functions *without* using reference variables. So, let's examine three different functions, and how we would pass a parameter to each one of them.

```
// Example 1
int IntMinus1(int oldVal)
{
   oldVal = oldVal - 1;
   return oldVal;
}

// Example 2
int IntMinus2(int* oldVal)
{
   (*oldVal) = (*oldVal) - 2;
   return (*oldVal);
}

// Example 3
int IntMinus3(int& oldVal)
{
   oldVal = oldVal - 3;
   return oldVal;
}
```

Assume also that we have an integer, `myInt`, that we want to pass into the above functions, and

that we have a different integer, `secondInt`, that we will use to store the returned value of the above functions. Initially, only `myInt` has a value assigned to it:

```
int myInt = 31;
int secondInt;
```

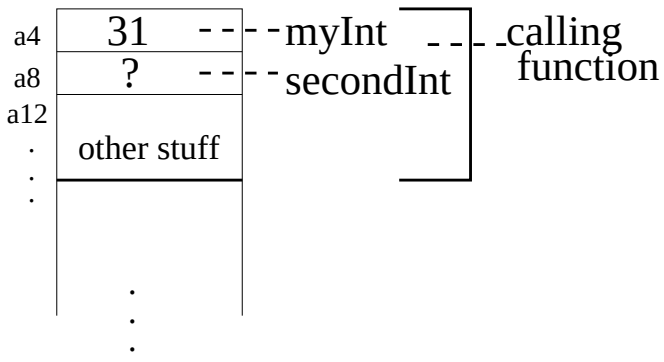


Figure 1.1: Initial memory configuration

For example 1, we would call the function as follows:

```
secondInt = IntMinus1(myInt);
```

Note that the function `IntMinus1` has an integer variable as a parameter, and thus the value we need to pass to this function is an integer value. When we pass `myInt` to `IntMinus1` we are indeed passing it an integer value, because `myInt` holds an integer value. So, all seems to be correct. However, this does raise the question, how are `oldVal` (in the function `IntMinus1`) and `myInt` related? The answer is, passing `myInt` to the function `IntMinus1` results in a *copy* of `myInt` being created in memory for the function to use. It is this copy that is given the name `oldVal`.

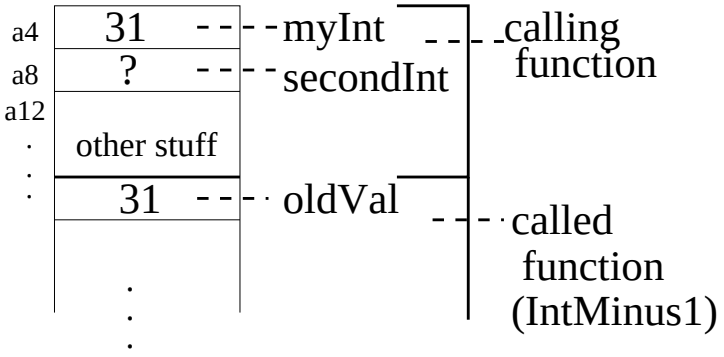


Figure 1.2: Memory while inside `IntMinus1` (pass by value)

In this case, any alterations that are done to the parameter `oldVal` from within the function `IntMinus1` are done *to the copy* of "31" (which is stored in `oldVal`), not to the original (which is stored in `myInt`). The original remains *unchanged*. For this reason, this parameter-passing

technique is known as "pass by value", because it is only the value that is passed, and only the value that is used. The actual variable that was passed in was irrelevant, because what was used was a *copy* of that variable – i.e. a copy of the value inside the variable. All that was imporant was the value inside the object. So, above, `secondInt` would hold the new value, since that was returned from the function. However, `myInt` would remain unchanged, since the subtraction done to it was done only to a copy of `myInt`, called `oldVal`, and not to `myInt` itself. (And, of course, once we return from `IntMinus1`, that function's stack frame is gone and `oldVal` no longer exists.)

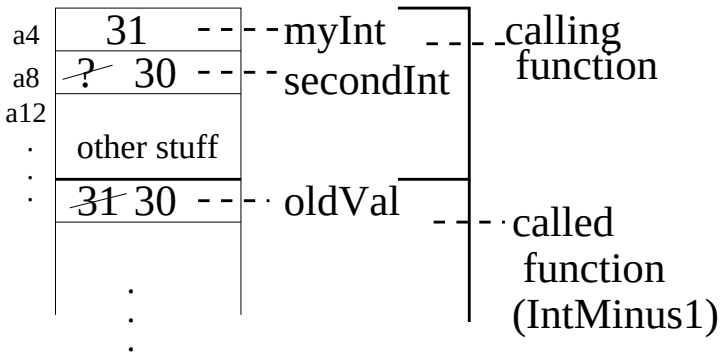


Figure 1.3: Changing `oldVal` does not change `myInt` (pass by value)

Note that we could have achieved the same results by simply calling the function as follows:

```
secondInt = IntMinus1(31);
```

In this case, `oldVal` would still be assigned the value "31", because that was the value that was passed to the function `IntMinus1`. The only difference here would be that in the first case, we passed to `IntMinus1` a variable which *held* the value 31, and in the second case, we passed the actual value 31. From the standpoint of `IntMinus1`, though, there is *no* difference between the two. Either way, `IntMinus1` receives the value "31" and stores it in the integer variable `oldVal`. Since the variable we are passing to the function is irrelevant (since all we really use is the value), we don't even need a variable at all, and this is seen in the above example where we pass in a value directly.

With pass-by-value, you always have the choice of passing in a literal value to the function, or just passing in some expression and letting the machine evaluate the expression to get the value and then send that value to the function. But regardless of which way you choose, it appears the same to the called function (`IntMinus1` in this case) – a new value is given to the called function and the called function stores that value in its parameter variable, and hence a copy is made of the value and alterations are made only to the copy.

So, if a programmer wanted to actually *change* a variable's value from within a function, one way to do that would be to use the technique in the second of our three examples, which we will call *pass-by-pointer*. Such a function would be called as follows:

```
secondInt = IntMinus2(&myInt);
```

Note that, since `IntMinus2`'s parameter variable is of type "integer pointer", we must pass in a value that an integer pointer variable can hold – namely, the address of an integer object. If the calling function had a variable that was an integer pointer – for example, if we had said

```
int* myIntPtr = &myInt;
secondInt = IntMinus2(myIntPtr);
```

then that would be okay as well. The first case corresponds to our passing in of "31" directly, since in this first case we are directly passing in an address (the expression `&myInt` will result in an actual machine address, as we have previously seen). In the second case, we store the address in a variable of the appropriate type (an "integer pointer" variable in this case) and then pass that variable to the function `IntMinus2`, just as we passed `myInt` (which held the value 31) to `IntMinus1` in the first example. And again, no matter which way we choose to send a value to the called function, the called function sees things the same way.

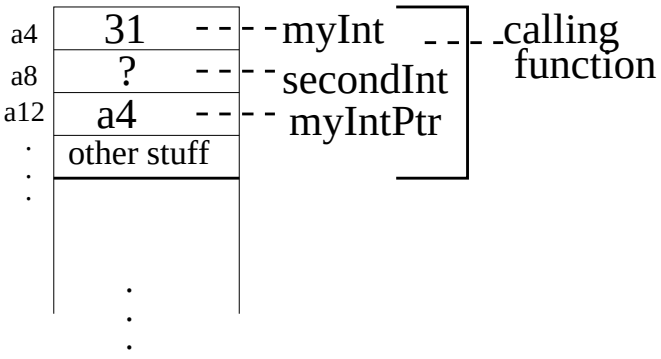


Figure 1.4: Memory before calling `IntMinus2` ("pass by pointer")

Just as before, the value we pass to the function is passed-by-value. However, here, it is the *address* that is our value, not the integer 31. So, what we make a copy of is the address – i.e. we create a new integer pointer variable to store the address we are passing to `IntMinus2`, just as we previously created an integer variable to store the integer we were passing to `IntMinus1`. When we pass `myIntPtr` to `IntMinus2`, the value (the address) in `myIntPtr` cannot be changed because we make a *copy* of that address to store in `IntMinus2`'s parameter variable `oldVal`.

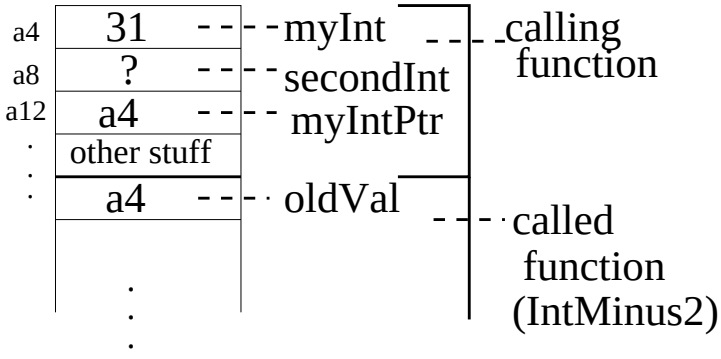


Figure 1.5: Memory while inside `IntMinus2` (pass by pointer)

All references to `oldVal` in the function refer to the newly created copy of the address, not to the original address itself. If we change `oldVal`, the parameter "integer pointer" variable,

to hold a different address, the original "integer pointer" variable (`myIntPtr`) still holds the address of `myInt`. Or, in the first case here, where we directly passed the address of the integer object, there was no original "integer pointer" variable in the first place.

So, just like example 1, we have passed in a value (integer in example 1, address in example 2), and a copy of that value is created for the function to use, and so alterations done in the copy are not done to the original value sitting in the original variable. The difference is that in example 2, we have the address of the original integer, and so we *can* alter the integer. That distinction is important:

- In example 1, we passed an integer, made a copy of that integer, and performed alterations to the copy. Those alterations were *not* made to the original integer.

- In example 2, we passed an integer address, made a new integer pointer variable to hold that address, and performed alterations to the copied address in the new pointer variable, rather than to the address in any original pointer variable that we may have passed in. However, by *using the address in the new pointer variable* to get to the original integer, we *can* make alterations to the actual integer. It is the *address* that must remain unchanged here, and the original pointer variable that holds that address. But we *can* change the actual integer.

This alteration of the integer *via the use of the address we already have* is what is going on in example 2. The line

```
(*oldVal) = (*oldVal) - 2;
```

dereferences the pointer variable `oldVal` to get the memory cell at **a4**. Then, we say that the value in this cell should be set equal to the current value in this cell minus 2. Thus, the value in cell **a4** – which happens to be the value of the variable `myInt` – has now been decreased by 2.

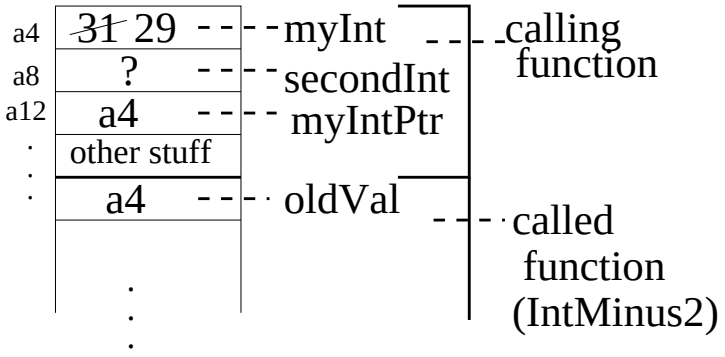


Figure 1.6: Memory after first line of `IntMinus2` (pass by pointer)

Now, the second line returns `*oldVal` – that is, we dereference `oldVal` to get the cell at **a4**, and return the value that is in that cell, which is 29. Thus, `secondInt` is set equal to 29. Just as with our call to `IntMinus1`, our call to `IntMinus2` has resulted in `secondInt` being given the desired value. The difference is that with `IntMinus2`, the original integer variable has been changed as well, whereas with `IntMinus1` it was not changed.

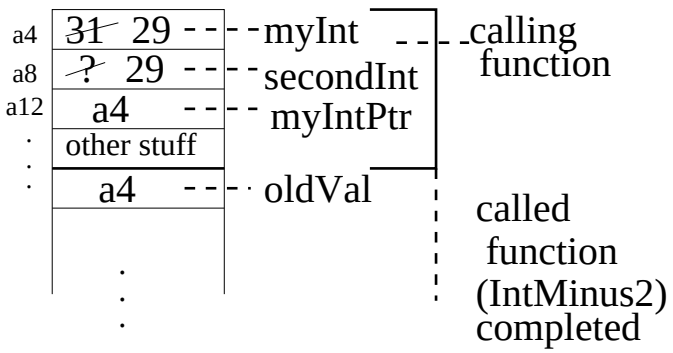


Figure 1.7: Memory when call to `IntMinus2` is completed (pass by pointer)

## 1.2 Reference variables

There are a few problems we face if those are the only two parameter-passing techniques open to us:

1. If we want to alter an object, we have to pass a pointer to an object and work through the pointer, rather than pass the object itself. This results in some messy syntax (note that the example 2 code is slightly messier than the example 1 code), and the messier syntax gets, the more error-prone it gets. A program could have incorrrect results simply because the programmer forgot a single dereference.

2. If we don't want to alter an object, and want to avoid the messy syntax of example 2, we could use example 1's pass-by-value technique. But in that case, we need to create a second copy of the object in memory. We aren't just using an already-existing object, we're making a new copy for our function to use. This results in extra time needed to copy the value, and it also results in extra memory being used to hold the copy (the extra memory being used in our examples is the cell needed for the variable `oldVal`). This extra time and extra memory might not seem like a big deal when our variables are of "small types" such as integers, but if we had an object of a larger type, taking up a lot of cells, then the time to copy those cells and the memory needed to store the copy could both be significant.

To solve these two problems, reference variables were created. This is the method used by example 3, a method we call *pass-by-reference*. You call the function as in example 1:

```
secondInt = IntMinus3(myInt);
```

except here, the type that the function actually accepts is not an integer, nor is it an integer pointer. Rather, it is an *integer reference*. Note the "&" after the type `int` in the function signature of the `IntMinus3` function. When the "&" appears after a type name in a function signature, it means that the following name is not a regular variable of that type, but rather a reference variable of that type. In `IntMinus3`, the variable `oldVal` is of type "integer reference", and not of type "integer".

So what *is* a reference variable? The idea is, the integer object `myInt` is used exactly as it exists, and the function simply reaches it through a different name, the name `oldVal`. The

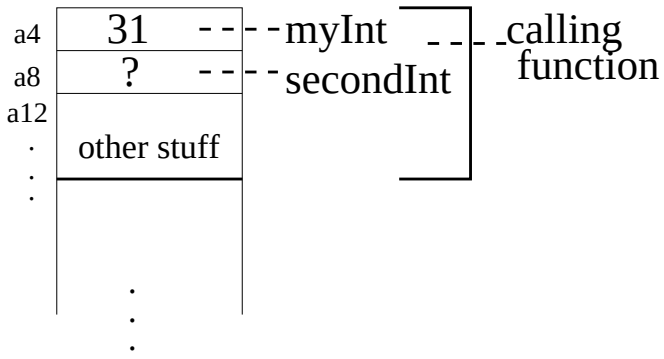


Figure 1.8: Memory before call to `IntMinus3` (pass by reference)

original name (`myInt` in this case) is within the scope of the calling function, so you can use that variable name in the calling function but you cannot use that variable name inside `IntMinus3`. Conversely, the variable name `oldVal` is within the scope of `IntMinus3`, so you can use that name inside `IntMinus3`, but you cannot use it inside the calling function – neither before the call to `IntMinus3` nor after that function call returns. But, *both* variable names refer to the *same* memory cell.

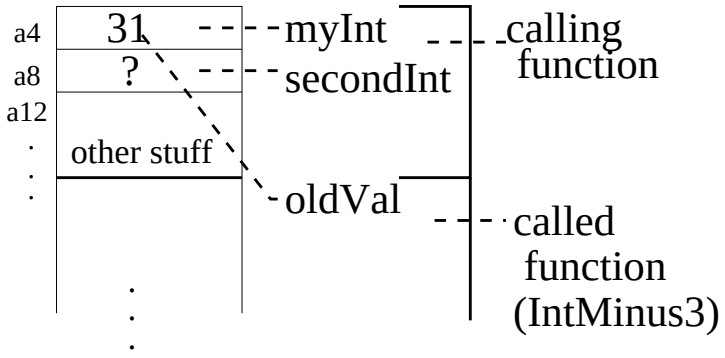


Figure 1.9: Memory when first entering `IntMinus3` (pass by reference)

One helpful way to think about this is to imagine that you and I are involved in espionage. You want to send packages to me by way of secret courier. Now, you have a number of secret couriers you could send to me, among them Bob, Susan, and Jim. I, on the other hand, have no idea who your couriers are or what their names are, but I know that they are your agents, so I decide I will just call them all "Agent". This is similar to our real-life function call example, where `IntMinus3` doesn't know the real name of the variable you are passing in, so it decides to just call all of the possible variables `oldVal`.

So, the first time you send me a package, you send Bob as the secret courier. And I answer my door, see a man with a top-secret package, and say, "Hello agent!". Bob, who is new to the spy business and somewhat nieve, says, "My name isn't Agent, it's B-" but before he can finish saying his real name, I reply, "I don't need to know your real name; I'm just going to call you Agent." At this point the Agent and I perform all the secret handshakes and utter the secret

code words and then he transfers the package to me.

The next time you send me a package, you send Susan as the secret courier. Again, I answer my door, and upon seeing someone carrying a top-secret package, I say, "Hello agent!". Susan, like Bob, is new to the spy business, and starts to say, "My name isn't Agent, it's Su-". But again, before she can finish stating her name, I say to her – as I said to Bob – "I don't need to know your real name; I'm just going to call you Agent." And again, we complete the handshakes and code words and then she transfers the package to me.

No matter who you send to my door next, I don't need to know their name. I can just call them "Agent", and your courier knows I am referring to him or her, and we can complete the transaction. I never need to know the real name of any Agent; instead, I give them a name of my own choosing – Agent – for the purposes of our interaction. This is the basic idea of reference variables. The function `IntMinus3` doesn't need to know the real name of any integer variable you send to it via a function call. The function simply says, "Aha! You are an integer variable! I will call you `oldVal`.", and then proceeds from there, running the `IntMinus3` code, using the name `oldVal` to refer to the variable that was sent to it by the calling function in the same way that I used the name "Agent" to refer to the courier that was sent to me by you.

This means, of course, that when you subtract 3 from the value of `oldVal`, you are also subtracting 3 from the value of `myInt`, since both names refer to the exact same memory cell. In this manner, we are able to alter our integer variable from within the `IntMinus3` function, but we don't need to use the many deference operations that we needed to use to accomplish this task in the `IntMinus2` function. Instead, our syntax, with the exception of one character – the "&" in the function signature after the type – is the same as the relatively simple syntax of `IntMinus1`. Thus, we get the best of both worlds – relatively clean syntax while still being able to alter variables from within the functions we send them to.

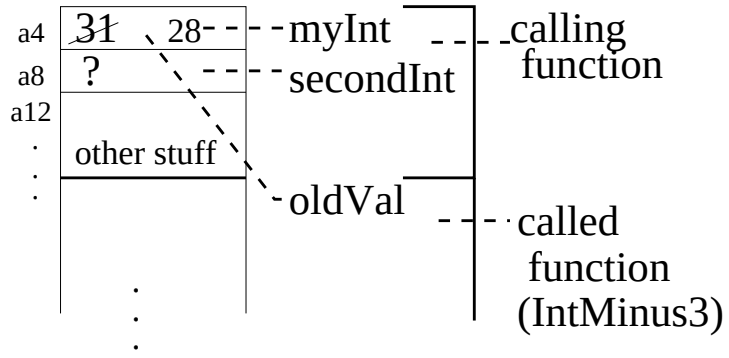


Figure 1.10: Memory after first line of `IntMinus3` (pass by reference)

This results in the clean syntax of pass-by-value, but with two benefits over the pass-by-value method:

1. We can now make changes inside the function (in this function, we are making them to the variable `oldVal`), and the changes are made to the original object. The end result of this function is that `secondInt` holds the decreased value, as always, but, as with example 2, `myInt` itself is decreased as well.

2. It is not necessary to create a copy of the original object. Under the "example 1" way

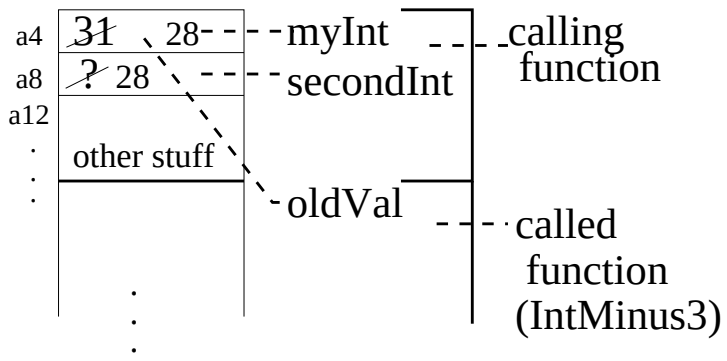


Figure 1.11: Memory after we return from `IntMinus3` (pass by reference)

of doing things, if we passed an object to a function, we had to make a copy of it. With reference variables, we use the actual object itself, instead of making a copy. This means we save time – the time needed to copy the data from the original object to the copy of that object – and we also save the memory that would be needed to store the copy.

Note that, in actual machine languge, reference variables might be implemented in a "pass-by-pointer" manner. That is, the true implementation of example 3 could be similar to example 2, only the obtaining of addresses and the dereferencing of those addresses would all be done automatically for you behind the scenes, rather than you having to do those things explicitly as in example 2. You can simply think in terms of the "renaming" abstraction we have already discussed. However, even if the compiler does indeed insert such code behind the scenes, there is still only the memory for one pointer needed – which for large objects is still much less memory than if you were to copy the entire object.

Note that, unlike with pass-by-value and pass-by-pointer, you have only one option of what to pass when using pass-by-reference. That is, you have to pass a variable. You cannot hardcode a value into the function call (as we did when we passed `31` to `IntMinus1` or when we passed `&myInt` to `IntMinus2`), because there is no memory location that is holding that value, and thus nothing to actually rename. Passing-by-reference is a renaming of a memory location, and thus you need to pass in a variable and not just hardcode a value into the function call. (If you *do* hardcode a value in the function call, some compilers will create a temporary internal variable to store that value, so that the passing-by-reference still makes sense. In those situations, the compiler will also likely warn you what is going on, so that *you* can decide whether to accept the situation or change it.)

## 1.3 Returning a value from a function

It is possible to return a value from a function using the same three ways that we passed values into a function:

1. You can *return-by-value*. In that case, you just have the regular return type listed – `int`, or `Coord`, or whatever. In those situations, whatever you happen to be returning, a temporary copy is made by the program behind the scenes, and the expression that uses your return type reads that copy, rather than the original value you tried to return.

2. You can *return-by-pointer*. That is, you can return the address of a value, instead of the value itself. As we have already mentioned, it is very dangerous to return the addresses of local variables!! However, if you had some more permanently-stored data, you might have a reason to return the address of that data. If you are indeed returning an address, then your return type would be a pointer – `int*` or `Coord*`, for example.

3. You can *return-by-reference*. In these cases, you'd have the & after your type, just as in the parameter list – that is, your return type could be something like `int&` or `Coord&`, for example. In this case, you would need to be returning a stored data of some kind – not just a value – so that there was some memory cell to rename. Also, just as with return-by-pointer, it would be very dangerous to return references to local variables, since those variables go away and so you are returning a new name for a memory cell that will soon cease to be under your control.

   The function call itself becomes the "new name" for the specific object that is returned by reference. For example, if some object `myVal` had a member function `Access()` that returned an integer member variable by reference, then we could do things like this:

   ```
   myVal.Access() = 5;
   ```

   The idea is, the function call itself becomes the "variable name" for that memory cell that was returned. We could then use that function call in ways similar to how we would use a variable name – i.e. we could read from that "renamed" cell, or we could write to it (as in the example above). This concept is admittedly a bit strange, but you will shortly see some more examples of this and get a feel for why and when we'd want to return something by reference.

## 1.4 The `const` keyword

You will often encounter the keyword `const` in function signatures. Depending on where it occurs, it can mean one of a number of things, though all the meanings are related in some way to declaring something to be constant. We will look at three of the more common usages here:

1. If it appears in front of a parameter:

   ```
   int MovePiece(const Piece& currentPiece); // perhaps from a
                                             //  chess program
   ```

   then it means that that parameter must remain constant for the life of the function. If you try to change the value of the parameter during the function, you will get a compiler error. Note, that "during the function" means "at any point in the program until you leave the function". This means that if you call a second function from inside this one, the parameter must remain constant in that function as well. Once you declare a parameter to be constant for a function, then no matter what you do in that function, and no matter what else you call while inside that function, the parameter must remain constant until the end of the function. This will often be the cause of compiler errors or warnings you may get relating to the `const` keyword. If you declare a parameter to be constant, and

then pass that parameter to a function that *can* change the parameter's value (never mind if it actually does, the point is that if the parameter is not `const` for that function, then it conceivably *could*), the compiler will complain.

This use of `const` is seen very frequently with reference variables. This may be puzzling at first...why would we pass something by reference, and then make it constant? Doesn't passing it by reference mean we want to change it? The answer is that in situations like that we are taking advantage of the second and third properties of reference variables, but not the first. We don't want to change the argument to that function, but neither do we want to use up time and memory creating and storing an entire second copy of it. So instead, we could make the original argument (rather than a copy of it) available to the function by using pass-by-reference, but by simply marking the parameter as being a constant for the life of the function, we ensure that the parameter – which is also the original, because we are using pass-by-reference – does not get altered by mistake. This way, we save copying time and memory while at the same time still preventing alteration of the original variable from inside the function.

2. If `const` appears at the end of the function signature of a class's member function:

```
int Print() const;  // print the information of a Coord object,
                    //  for example
```

then it means that the object you are calling the function on must remain constant until the end of the function. So, any member functions of a class that are declared `const` in this manner cannot write to the member variables of that class. Such member functions can still read the member variables. The functions just can't write to those member variables. This usage of `const` often appears in functions that are designed to read and return member variables of an object; since you are only reading the variables, there is no need to change them, so you make those functions `const` to be sure that you don't accidentally do this (because, again, if you then do try and write to those variables anyway, the compiler will complain).

3. If it appears at the beginning of the return type:

```
const Coord& Foo(); // return type is const Coord&
```

   then it means that whatever is returned is constant. The purpose of this syntax is to protect a reference variable, as in the example above. This function (`Foo()`) returns a reference to a `Coord` object. In a read-only situation, this works wonderfully; we only need to read a value, so why return a copy of the value when it's easier to just use that actual value (via a reference)? The problem is that some badly written code may then try to *write* to whatever is returned. So, in the situation above, whatever `Coord` object is returned via the reference variable is constant if we access it through this reference. We may change this object some other way, but the particular reference our calling expression uses to access this returned data will *not* allow alteration of the object it refers to. (Just as with return-by-reference, return-by-const-reference requires some examples to better illustrate, and we will show you those soon.)

The phrase *const correctness* refers to the idea of using `const` everywhere it should be used. That is, any parameter that you know will remain constant, put a `const` in front of it. Any member function that won't alter the object, put a `const` in front of it. Any return value that you pass back by reference, if that reference should not allow you to change what it refers to, then put a `const` in front of the return type. The big problem with using `const` is that, more or less, it has to be used correctly everywhere, or not used at all. For example, if we have a function:

```
void Foo(const Coord& myVal)
{
   myVal.Print();
}
```

then, since `myVal` is supposed to remain constant until the function `Foo` has ended, it will not be permissible to call `Print()` off of `myVal` unless `Print()`, a member function, has been marked `const`. Otherwise, the compiler will give you an error message, something like "trying to call non-const member function on const object". Now, perhaps `Print()` doesn't change `myVal` at all. That does not matter! When compiling the line `myVal.Print();`, the compiler won't try and figure out whether `Print()` can change `myVal`. It will simply note if `Print()` is marked `const`, and will complain if it isn't. So if you're going to try and mark the parameter `myVal` as `const`, then putting the `const` at the end of the declaration of `Print()` (whatever file that declaration is in) will also be necessary.

This can raise problems for you if you use someone else's library code for your own program, and that library code does not make correct use of `const`. So, sometimes you would *want* to use `const` correctly yourself, but need to come up with some bizarre hacks in a few places because someone else's code, which you are using, doesn't do things correctly. The ideal situation is when `const` is used correctly in all the code you are using written by someone else, and then you can correctly use `const` yourself as well, and thus take advantage of the compiler checks that that will give you, without worrying about some bizarre `const` problem giving you trouble.

## 1.5 Computer science parameter-passing terminology

One last remark needs to be made, this one on definitions. When speaking of programming languages in general, there are usually two types of parameter-passing concepts that people speak of: pass-by-value (the original argument used to assign a parameter cannot be changed from inside the function) and pass-by-reference (the original argument used to assign a parameter *can* be changed from inside the function). The concept illustrated in our first example was pass-by-value. The second technique is pass-by-value from the standpoint of the pointer parameter itself, but with respect to the item the pointer points to, it is a pass-by-reference technique, since you can change the value of what is pointed to from within the function (as we did in example 2). Likewise, the third technique – the use of reference variables – can also allow us to change the value of an original argument to a function from within the function, and so it, too, can be considered a pass-by-reference technique.

So, when we talk about "pass-by-value" and "pass-by-reference" in the general "theory of programming languages" sense, we would have to say that method 1 is "pass-by-value", and methods 2 and 3 are *both* "pass-by-reference". And so to make it clear what method we are referring to, if you wanted to be precise you could call the three methods:

1. pass-by-value
2. pass-by-reference using pointer variables
3. pass-by-reference using reference variables

If you use those descriptions, then it is clear that methods 2 and 3 are both "pass-by-reference" methods, in the language-theory sense of "pass-by-reference". And yet we can also tell the two apart, also based on their description.

However, usually we are not too concerned about corresponding with exact language-theory definitions, and instead simply want to be able to tell one method from another. So, the shorthand we already used, namely

1. pass-by-value
2. pass-by-pointer
3. pass-by-reference

will serve us well enough. Just keep in mind that there are really more accurate descriptions for those last two techniques that match better with the formal theory terms for parameter-passing techniques.