

## Common Function Errors

There are two general classes of errors associated with functions – compiler and linker.

### Compiler

Compiler errors with functions usually result from a mismatch between the function invocation and the function prototype. The solution is a careful comparison between the two for any differences in function name, return type, number of parameters, or the types of parameters:

#### Function Name

A difference in the function name (such as spelling or capitalization) from that of the prototype will result in an undeclared identifier error. For example, if you had a prototype "void foo( )" and an invocation "Foo( )", you would receive the following error:

```
error C2065: 'Foo' : undeclared identifier
```

#### Return Type

If the function prototype and has a different return type from the invocation, you **may** receive a type conversion error. Using the previous example, if the function were invoked as:

```
int i = foo( ); // The return type is implied to be an int
```

you would receive the following error:

```
error C2440: 'initializing' : cannot convert from 'void' to 'int'
```

I say **may** because for some types a conversion can be performed, though the results may not be what you expect. An example would be a function that returns a double to an integer variable. We've already seen what happens in this case – the decimal portion is truncated. These types of logical errors are much more subtle and harder to find.

#### Number of parameters

Another common error is invoking a function with too many or too few arguments for the given prototype. If our function *foo* had a prototype that listed two parameters, but it was invoked with three, the resultant error would be:

```
error C2660: 'foo' : function does not take 3 parameters
```

It's left up to the programmer to determine how many parameters the function *does* expect.

### Parameter Types

Like return type, this error isn't always caught because conversions **do** exist for all the basic data types: int, float, double, char, and bool (though trying to convert to a bool will give a warning). However, strings, structs, arrays (any type), or mixing pass by value/pass by reference will give an error (confused, yet?). If the prototype specifies one type of parameter and the function is invoked with another, the error would be of the form:

```
cannot convert parameter <parameter number> from type <invocation type>
to <prototype type>
```

Often times these errors result from having the correct parameters, but in a different order than that in the prototype.

### **Linker**

If the prototype and the invocation match, but there is a difference between the prototype and the definition, a linker error will result. Like a compiler error, a linker error can be the result of a difference between function name, return type, number of parameter, or the types of parameters. Linker errors are actually easier because they all result in an identical error message – an "unresolved symbol" error. If a function is prototyped as "void foo(int x)", but the definition differs in one or more of the four areas, you will receive:

```
error LNK2001: unresolved external symbol "void __cdecl foo(int)"
```

Also, there is no type conversion problem with the linker – if a prototype calls for an int, but the function is defined with a double, an error will still be generated.

### **Other Types of Errors**

There are several other kinds of errors (compiler) that are common when working with functions.

Accidentally including a semicolon in the header of the function definition:

```
int foo(int x);
{
    // Function computation
}
```

will result in:

```
error C2447: missing function header (old-style formal list?)
```

An error like:

```
error C2601: 'foo' : local function definitions are illegal
```

means that instead of writing functions sequentially:

```
int main( )  
{  
  
}  
  
int foo()  
{  
  
}
```

you have nested them:

```
int main( )  
{  
    int foo()  
    {  
    }  
}
```

This is usually the result of a misplaced (or missing) French brace.

Finally, one of the most frightening errors to encounter is a runtime error that states that the program has performed an illegal operation and will be shut down. This is the type of error that occurs when attempting to access an array beyond its bounds. However, it also occurs if the program passes a filestream by value. While debugging array problems can be time consuming, a filestream that was passed by value can be seen with a quick look at the function prototypes. Always start with the easy solution – if you run your program and get this error, first verify that all your filestreams are passed by reference. Under **NO** circumstances should a filestream **EVER** be passed by value.