**Data Abstraction**

The only way to get massive increases in productivity is to leverage off other people’s code. That is, to use libraries. Thus, libraries are probably the most important way to improve productivity, and one of the primary design goals of C++ is to make library use easier.

If a program allocates and frees heap storage for a while, you can end up with a fragmented heap that has lots of memory free, but without any pieces that are big enough to allocate the size you’re looking for at the moment.

# Bad guesses

A header file must be included in any file that refers to a user defined data type because the compiler can’t even guess at what that structure looks like. However, it can guess at what a function looks like; this sounds like a feature but it turns out to be a major C pitfall.

It’s possible in C (but not in C++) to call a function that you haven’t declared.

# What's wrong?

C has a single name space for functions; that is, when the linker looks for a function name, it looks in a single master list. In addition, when the compiler is working on a translation unit, it can work only with a single function with a given name.

If there are two functions with same name in different libraries which we are using. C gives you an error, telling you there’s a type mismatch in the two different argument lists of the declared functions.

# The basic object

There are several other things that are different between C and C++. First, the declarations in the header files are required by the compiler. In C++ you cannot call a function without declaring it first. The compiler will issue an error message otherwise. This is an important way to ensure that function calls are consistent between the point where they are called and the point where they are defined.

C++ functions can be placed inside structs as “member functions". functions are inside the body of the struct. In addition, notice that the first argument from the C version of the library has been removed. In C++, instead of forcing you to pass the address of the structure as the first argument to all the functions that operate on that structure, the compiler secretly does this for you.

# What's an object?

With functions in the package, the structure becomes a new creature, capable of describing both characteristics (like a C struct does) and behaviors. In C++, an object is just a variable, and the purest definition is “a region of storage” (this is a more specific way of saying, “an object must have a unique identifier,” which in the case of C++ is a unique memory address). It’s a place where you can store data, and it’s implied that there are also operations that can be performed on this data.

# Abstract data typing

The ability to package data with functions allows you to create a new data type. This is often called encapsulation. The definition of a structure creates a new data type. You can perform operations defined for that data type by member functions. You create one by saying ‘struct\_name s’, just as you create a float by saying ‘float f’. A structure also has characteristics and behavior. Even though it acts like a real, built-in data type, we refer to it as an abstract data type, perhaps because it allows us to abstract a concept from the problem space into the solution space.

In addition, the C++ compiler treats it like a new data type, and if you say a function expects a structure variable, the compiler makes sure you pass a structure variable to that function. So the same level of type checking happens with abstract data types (sometimes called user-defined types) as with built-in types.

# Object details

**How big is an object, and what does it look like?**

The answer is “about what you expect from a C struct.” In fact, the code the C compiler produces for a C struct (with no C++ adornments) will usually look exactly the same as the code produced by a C++ compiler.

In early versions of the language, the size was zero, but an awkward situation arises when you create such objects: They have the same address as the object created directly after them, and so are not distinct. One of the fundamental rules of objects is that each object must have a unique address, so structures with no data members will always have some minimum nonzero size.

The last two sizeof statements show you that the size of the structure in C++ is the same as the size of the equivalent version in C. C++ tries not to add any unnecessary overhead.

# Header file etiquette

## Importance of header files

If you place all your function declarations in a header file, and include that header everywhere you use the function and where you define the function, you ensure a consistent declaration across the whole system. You also ensure that the declaration and the definition match by including the header in the definition file.

You need a header file:

* When you want user defined data type to be easily accessible to yourself and others.
* You want to separate the interface (the declaration) from the implementation (the definition of the member functions) so the implementation can be changed without forcing a re-compile of the entire system.
* Header file tells the compiler what is available in your library. You can use the library even if you only possess the header file along with the object file or library file; you don’t need the source code for the cpp file.

## The multiple-declaration problem

The compiler considers the redeclaration of a structure (User defined data type) to be an error, since it would otherwise allow you to use the same name for different types.

Both C and C++ allow you to redeclare a function, as long as the two declarations match, but neither will allow the redeclaration of a structure (User defined data type).

You’ll include several files that include the same header file. During a single compilation, the compiler can see the same header file several times. Unless you do something about it, the compiler will see the redeclaration of your structure and report a compile-time error.

The preprocessor directives #define, #ifdef, and #endif can be used to prevent such errors.

## A standard for header files

In each header file that contains a structure, you should first check to see if this header has already been included in this particular cpp file. You do this by testing a preprocessor flag. If the flag isn’t set, the file wasn’t included and you should set the flag (so the structure can’t get re-declared) and declare the structure.

#ifndef HEADER\_FLAG

#define HEADER\_FLAG

// Type declaration here...

#endif // HEADER\_FLAG

These preprocessor statements that prevent multiple inclusion are often referred to as include guards.

## Namespaces in headers

using directive eliminates the protection of that particular namespace, and the effect lasts until the end of the current compilation unit.

If you put a using directive (outside of a scope) in a header file, it means that this loss of “namespace protection” will occur with any file that includes this header, which often means other header files.

Thus, ifyou start putting using directives in header files, it’s very easy to end up “turning off” namespaces practically everywhere, and thereby neutralizing the beneficial effects of namespaces.

In short: don’t put using directives in header files.

## Using headers in projects

Put the declaration for each type or group of associated types in a separate header file, then define the functions for that type in a translation unit. When you use that type, you must include the header file to perform the declarations properly.

# END