Lab Report for Object-oriented Programming course

Lab 3: Linkage

Wang, Chen
16307110064
School of Software
Fudan University

May 12, 2019

Contents

Chapter 1

Understanding Internal and External Linkage

1.1 Requirements

In this part, we are going to analyze the running result of the following short program and will try to make some subtle modifications whose change will result in the complete change of the result. The codes are shown below.

1.1.1 header.h

```
#ifndef LAB3_HEAER_H                    1
#define LAB3_HEAER_H                    2
                                        3
static int variable = 0;                4
extern int variable2;                   5
#endif //LAB3_HEAER_H                   6
```

1.1.2 file1.h

```
#ifndef LAB3_FILE1_H                    1
#define LAB3_FILE1_H                    2
void function1();                       3
                                        4
#endif //LAB3_FILE1_H                   5
```

1.1.3 file2.h

```
#ifndef LAB3_FILE2_H                    1
#define LAB3_FILE2_H                    2
void function2();                       3
                                        4
#endif //LAB3_FILE2_H                   5
```

1.1.4 file1.cpp

```
#include "header.h"
```

1.1.5 file2.cpp

```
#include "header.h"
void function2() {
    variable = 2;
    variable2 = 2;

}
```

1.1.6 main.cpp

```
#include <iostream>
#include "header.h"
#include "file1.h"
#include "file2.h"
int variable2;

int main() {
    function1();
    function2();

    std::cout << variable << std::endl;
    std::cout << variable2 << std::endl;
    return 0;
}
```

1.2 Execution result

The result of the execution is shown in the Figure 1.1 below.

1.3 Analysis for the result of static variable

From the C++ language standards, we can know that the *static* keyword has different implications and effects in two different occasions. In this project, the keyword appears before the definition of the global variable outside from any methods. In this case, the *static* keyword has the effect that the variable is only accessible in the specified file, i.e. having a *file scope*.

More specifically, in this project there are three **cpp** files in this project, each of them including the *header.h* header file where a static variable is declared. In this case, each of the three files has a version of the static variable *variable*. Therefore, the methods *function1* and *function2* will have no effect to the variable *variable* in the *main* method. Hence, the printing result is unchanged.

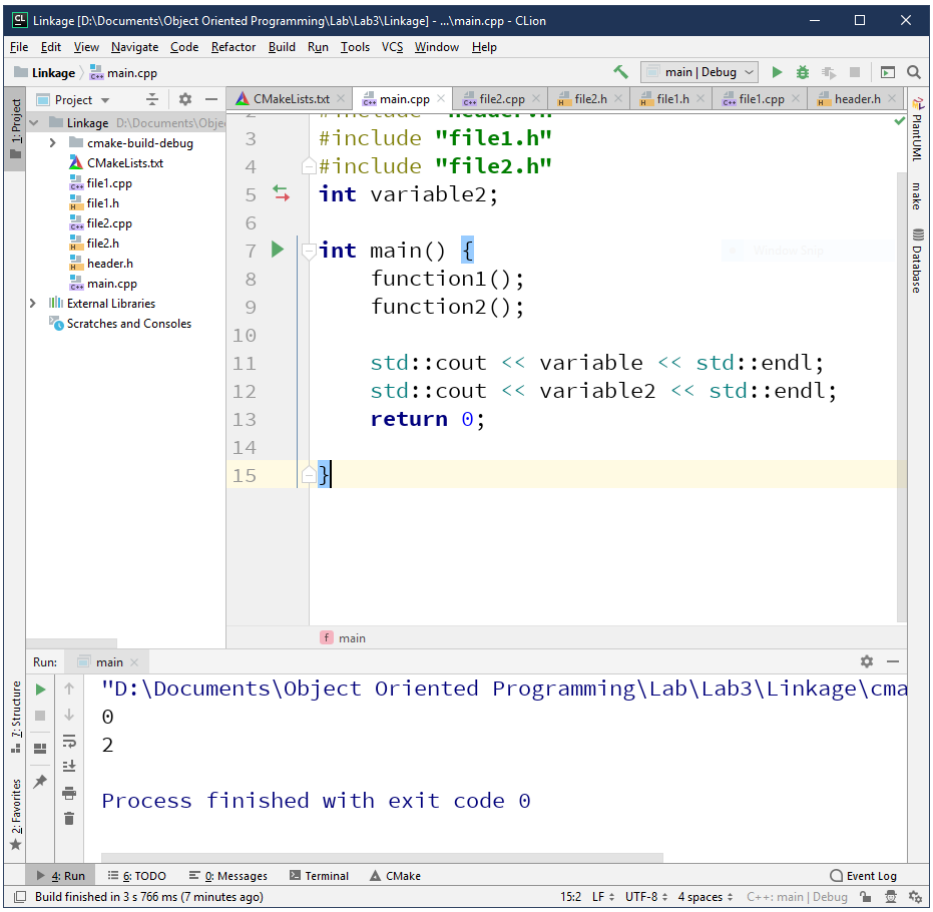

Figure 1.1: Execute Results

1.4 Analysis for the result of global variable

As to another part of this lab, the printing result of *variable2*, is a typical example of a global variable. Nevertheless, one slightly point making the analysis more complex is that the *external* keyword is used to make the global variable declared and defined at different places.

More specifically, in this project, the variable *variable2* is declared in *header.h* with an *external* keyword, indicating that this variable will be defined somewhere else in this project. Then in the file *main.cpp* this variable is defined. Therefore, this is a typical example of global variable and all the functions are modifying the same version of the variable. Therefore, the *main* function will print the result after being changed in the other functions.

1.5 Task 1.2: Duplicate variable definition

The result of the change is shown in the Figure 1.2 below, indicating an compiling error.

In this example, we have two places both defining the variable *variable2*. Therefore, the compiler will report a duplicate definition error.

1.6 Task 1.3: External declaration without definition

The result of the change is shown in the Figure 1.3 below, indicating an compiling error.

As per have stated above, an *extern* keyword indicates that this variable is defined somewhere else in this file, hence it will cause a compiling error if no definition is made through out the project file.

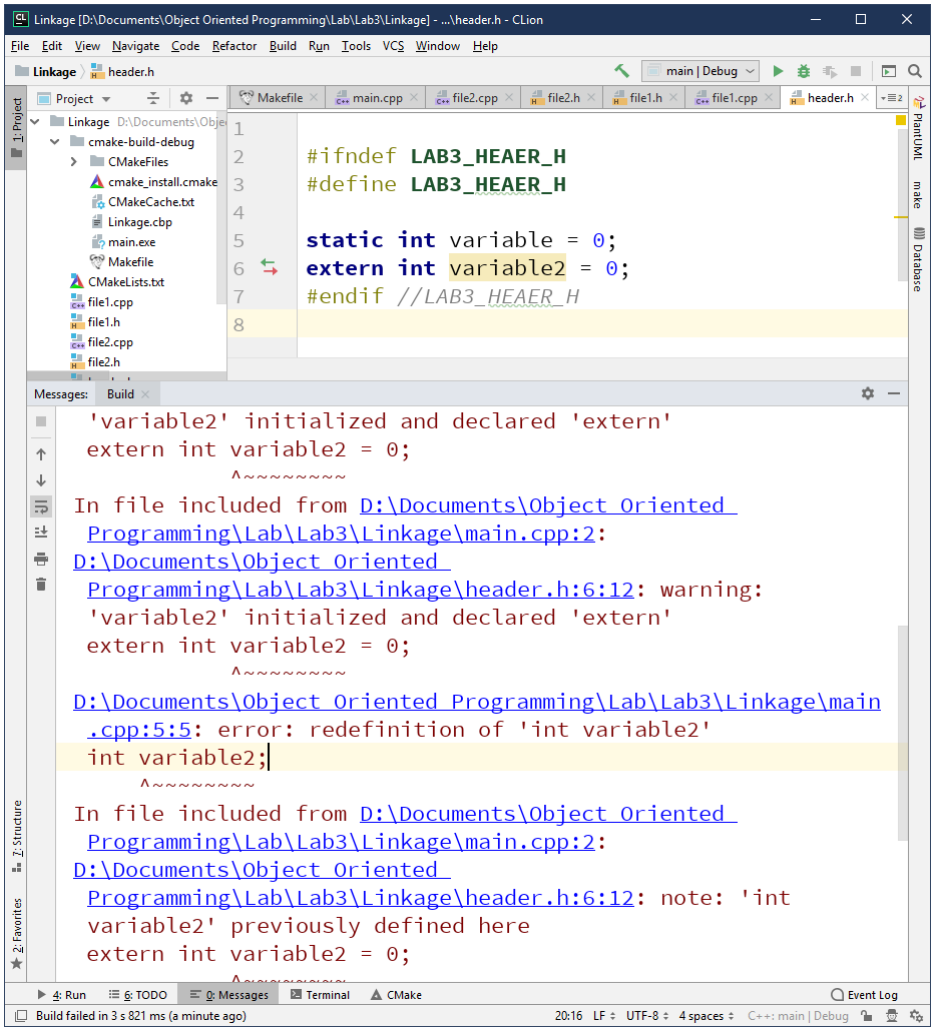

Figure 1.2: Execute Results of Task 1.2

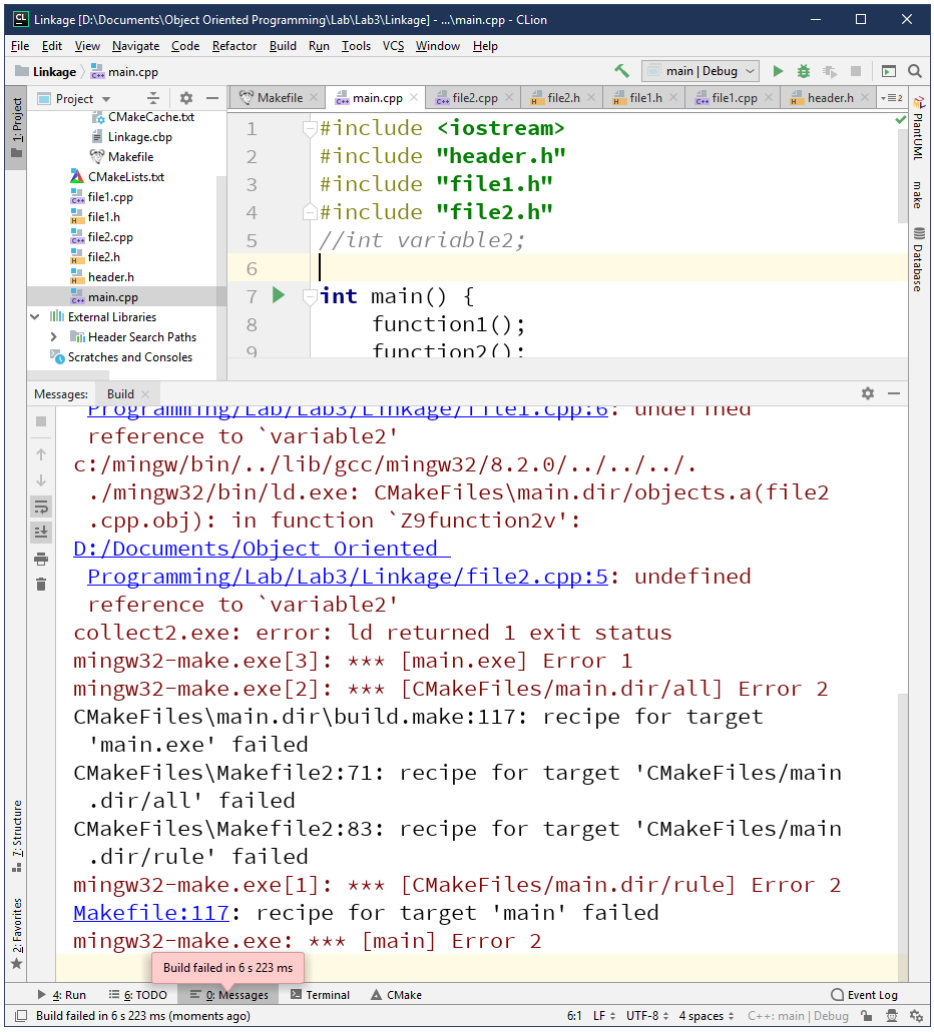

Figure 1.3: Execute Results of Task 1.3

Chapter 2

Understanding Name Space

2.1 Task 2.1 Duplicate variable naming in separate name spaces

In this task, we are going to use the name space mechanism, create two variables with the same name but different values and print them.

Here in my implementation, I declared two name spaces called **a** and **b** separately and then a variable named **a** in each name space but different value. Then in the *main* method, these two variables are printed separately, we can see the result that they have different values printed. The result is shown in the Figure 2.1 below.

2.2 Task 2.2 Extended namespace

In this task, I will need to duplicately declare the namespace of the same name in two files. The space of the first file defines "ab=1; cd=2", and the space of the second file is defined, "ab=3;bc=4" Finally, print the value in the namespace in the main function.

Here in my implementation, I declared a name space called **a** in two separate files and then assigned different values to the variables as stated. Then in the *main* method, these three variables are printed separately, we can see the result that this project cannot be compiled due to duplicate variable definition in the same name space. The result is shown in the Figure 2.2 below.

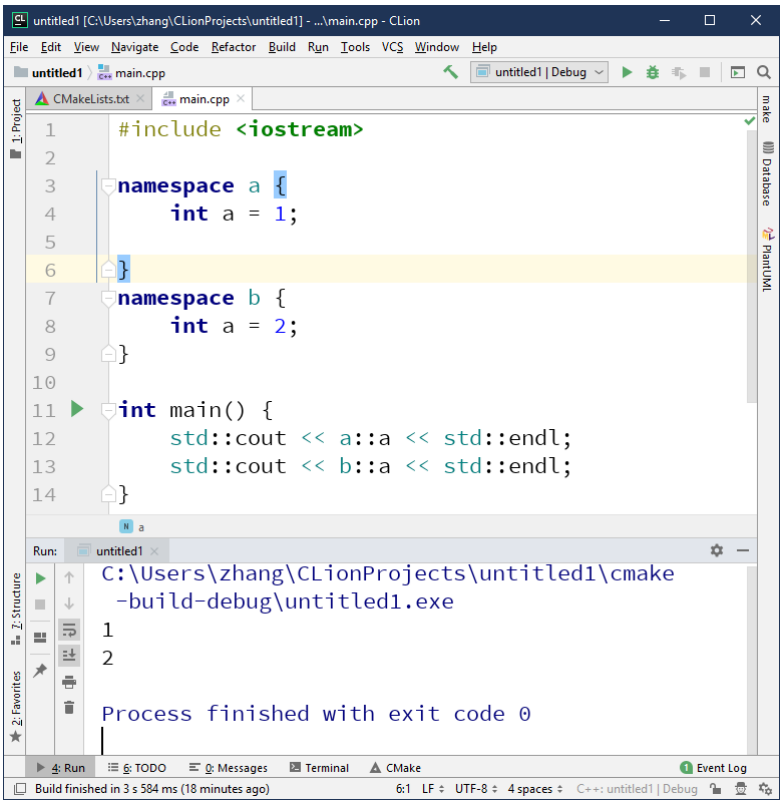

Figure 2.1: Execute Results of Task 2.1

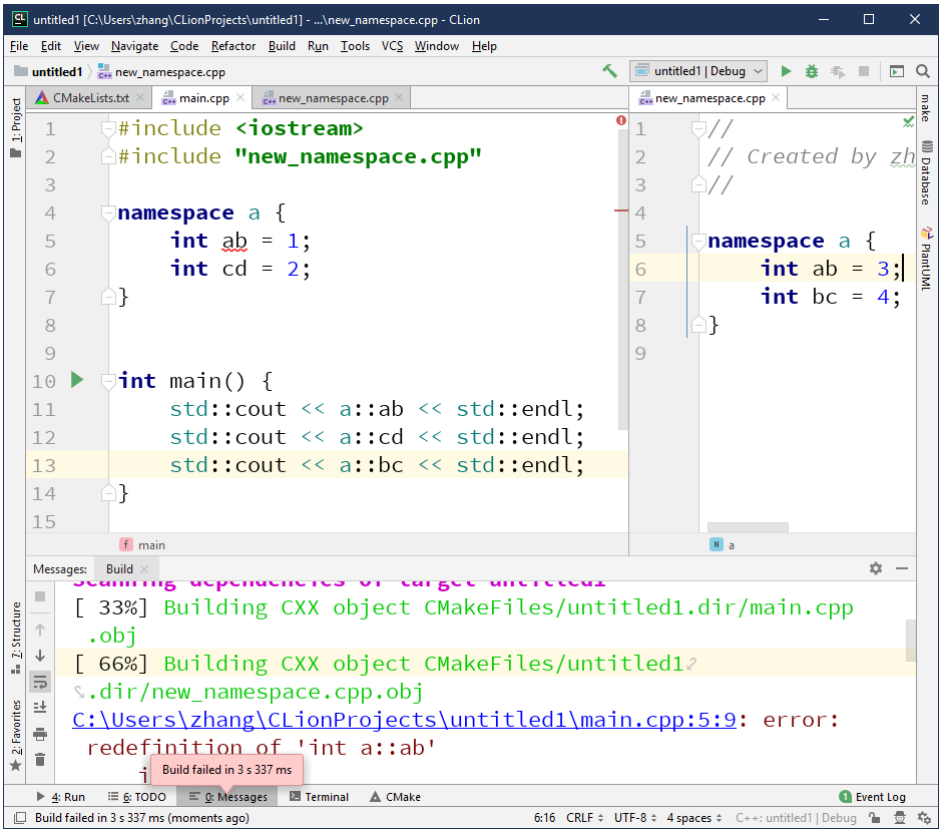

Figure 2.2: Execute Results of Task 2.2

Chapter 3

Compare with Typescript Name Space

3.1 Modules in TypeScript

***A note about terminology**: It's important to note that in TypeScript 1.5, the nomenclature has changed. "Internal modules" are now "namespaces". "External modules" are now simply "modules", as to align with ECMAScript 2015*[1] *'s terminology, (namely that* `module X {` *is equivalent to the now-preferred* `namespace X {` *).*

3.1.1 Introduction

Starting with ECMAScript 2015, JavaScript has a concept of modules. TypeScript shares this concept.

Modules are executed within their own scope, not in the global scope; this means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported using one of the *export* forms. Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported using one of the *import* forms.

Modules are declarative; the relationships between modules are specified in terms of imports and exports at the file level. Modules import one another using a module loader. At runtime the module loader is responsible for locating and executing all dependencies of a module before executing it. Well-known modules loaders used in JavaScript are the CommonJS module loader for Node.js and require.js for Web applications.

In TypeScript, just as in ECMAScript 2015, any file containing a top-level *import* or *export* is considered a module. Conversely, a file without any top-level *import* or *export* declarations is treated as a script whose contents are available in the global scope (and therefore to modules as well).

[1] Ecma International. (2019, May 5). ECMAScript 2015 Language Specification – ECMA-262 6th Edition. In *Ecma International*. Retrieved 08:23, May 11, 2019, from `http://www.ecma-international.org/ecma-262/6.0/`

3.1.2 Export

Exporting a declaration

Any declaration (such as a variable, function, class, type alias, or interface) can be exported by adding the *export* keyword.

Export statements

Export statements are handy when exports need to be renamed for consumers.

Re-exports

Often modules extend other modules, and partially expose some of their features. A re-export does not import it locally, or introduce a local variable.

Optionally, a module can wrap one or more modules and combine all their exports using *export * from "module"* syntax.

3.1.3 Import

Importing is just about as easy as exporting from a module. Importing an exported declaration is done through using one of the *import* forms below:

1. Import a single export from a module
2. Import the entire module into a single variable, and use it to access the module exports
3. Import a module for side-effects only

Though not recommended practice, some modules set up some global state that can be used by other modules. These modules may not have any exports, or the consumer is not interested in any of their exports.

3.1.4 Default exports

Each module can optionally export a *default* export. Default exports are marked with the keyword *default*; and there can only be one *default* export per module. *default* exports are imported using a different import form. *default* exports are really handy. For instance, a library like jQuery might have a default export of *jQuery* or $, which we'd probably also import under the name $ or jQuery.

Classes and function declarations can be authored directly as default exports. Default export class and function declaration names are optional.

3.1.5 *export =* and *import = require()*

Both CommonJS and AMD generally have the concept of an *exports* object which contains all exports from a module.

They also support replacing the *exports* object with a custom single object. Default exports are meant to act as a replacement for this behavior; however, the two are incompatible. TypeScript supports *export =* to model the traditional CommonJS and AMD workflow.

The *export =* syntax specifies a single object that is exported from the module. This can be a class, interface, namespace, function, or enum.

When exporting a module using *export =*, TypeScript-specific *import module = require("module")* must be used to import the module.

3.1.6 Code Generation for Modules

Depending on the module target specified during compilation, the compiler will generate appropriate code for Node.js (**CommonJS**), require.js (**AMD**), **UMD**, **SystemJS**, or **ECMAScript 2015 native modules** (ES6) module-loading systems.

3.1.7 Optional Module Loading and Other Advanced Loading Scenarios

In some cases, you may want to only load a module under some conditions. In TypeScript, we can use the pattern shown below to implement this and other advanced loading scenarios to directly invoke the module loaders without losing type safety.

The compiler detects whether each module is used in the emitted JavaScript. If a module identifier is only ever used as part of a type annotations and never as an expression, then no *require* call is emitted for that module. This elision of unused references is a good performance optimization, and also allows for optional loading of those modules. The core idea of the pattern is that the *import id = require("...")* statement gives us access to the types exposed by the module. The module loader is invoked (through *require*) dynamically, as shown in the if blocks below. This leverages the reference-elision optimization so that the module is only loaded when needed. For this pattern to work, it's important that the symbol defined via an *import* is only used in type positions (i.e. never in a position that would be emitted into the JavaScript). To maintain type safety, we can use the *typeof* keyword. The *typeof* keyword, when used in a type position, produces the type of a value, in this case the type of the module.

3.2 Name space in TypeScript

3.2.1 Introduction

In this subsection I will try to outline the various ways to organize the code using namespaces (previously "internal modules") in TypeScript. As per alluded in previous section's note about terminology, "internal modules" are now referred to as "namespaces". Additionally, anywhere the module keyword was used when declaring an internal module, the namespace keyword can and should be used instead. This avoids confusing new users by overloading them with similarly named terms.

3.2.2 Namespacing

As we have more and more methods of the same or of similiar functions, we're going to want to have some kind of organization scheme so that we can keep track of our types and not worry about name collisions with other objects.

Instead of putting lots of different names into the global namespace, we can wrap up our objects into a namespace.

If we want the interfaces and classes in the namespace to be visible outside the namespace, we can preface them with export. Conversely, the implementation details can be left unexported and will not be visible to code outside the namespace.

3.2.3 Splitting Across Files

As our application grows, we'll want to split the code across multiple files to make it easier to maintain.

Multi-file namespaces

We can split our namespace across many files. Even though the files are separate, they can each contribute to the same namespace and can be consumed as if they were all defined in one place. Because there are dependencies between files, we'll add reference tags to tell the compiler about the relationships between the files.

3.2.4 Aliases

Another way that you can simplify working with namespaces is to use *import q = x.y.z* to create shorter names for commonly-used objects. Not to be confused with the *import x = require("name")* syntax used to load modules, this syntax simply creates an alias for the specified symbol. We can use these sorts of imports (commonly referred to as aliases) for any kind of identifier, including objects created from module imports.

Notice that we don't use the *require* keyword; instead we assign directly from the qualified name of the symbol we're importing. This is similar to using *var*, but also works on the type and namespace meanings of the imported symbol. Importantly, for values, *import* is a distinct reference from the original symbol, so changes to an aliased *var* will not be reflected in the original variable.

3.3 Difference and relation between modules and name spaces in TypeScript

3.3.1 Introduction

In this section, I will try to clarify the difference on both concept and usage of modules and namespace in Typescript.

3.3.2 Using Namespaces

Namespaces are simply named JavaScript objects in the global namespace. This makes namespaces a very simple construct to use. They can span multiple files, and can be concatenated using *–outFile*. Namespaces can be a good way to structure your code in a Web Application, with all dependencies included as *¡script¿* tags in your HTML page.

Just like all global namespace pollution, it can be hard to identify component dependencies, especially in a large application.

3.3.3 Using Modules

Just like namespaces, modules can contain both code and declarations. The main difference is that modules **declare** their dependencies.

Modules also have a dependency on a module loader (such as CommonJs/Require.js). For a small JS application this might not be optimal, but for larger applications, the cost comes with long term modularity and maintainability benefits. Modules provide for better code reuse, stronger isolation and better tooling support for bundling.

It is also worth noting that, for Node.js applications, modules are the default and the recommended approach to structure your code.

Starting with ECMAScript 2015, modules are native part of the language, and should be supported by all compliant engine implementations. Thus, for new projects modules would be the recommended code organization mechanism.

3.4 Internal Linkage in C

If the declaration of a file-scope identifier for an object or a function contains the **storage-class-specifier** *static*, the identifier has internal linkage. Otherwise, the identifier has external linkage.

Within one translation unit, each instance of an identifier with internal linkage denotes the same identifier or function. Internally linked identifiers are unique to a translation unit.

3.5 External Linkage in C

If the first declaration at file-scope level for an identifier does not use the *static* storage-class specifier, the object has external linkage.

If the declaration of an identifier for a function has no **storage-class-specifier**, its linkage is determined exactly as if it were declared with the **storage-class-specifier** *extern*. If the declaration of an identifier for an object has file scope and no **storage-class-specifier**, its linkage is external. An identifier's name with external linkage designates the same function or data object as does any other declaration for the same name with external linkage. The two declarations can be in the same translation unit or in different translation units. If the object or function also has global lifetime, the object or function is shared by the entire program.

3.6 Name space in C++

3.6.1 Introduction

A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries. All identifiers at namespace scope are visible to one another without qualification. Identifiers outside the namespace can access the members by using the fully qualified name for each identifier, for example *std::vector¡std::string¿ vec;*, or else by a using

Declaration for a single identifier (*using std::string*), or a using Directive for all the identifiers in the namespace (*using namespace std;*). Code in header files should always use the fully qualified namespace name.

3.6.2 *using* directives

The *using* directive allows all the names in a *namespace* to be used without the **namespace-name** as an explicit qualifier. Use a using directive in an implementation file (i.e. *.cpp) if you are using several different identifiers in a namespace; if you are just using one or two identifiers, then consider a using declaration to only bring those identifiers into scope and not all the identifiers in the namespace. If a local variable has the same name as a namespace variable, the namespace variable is hidden. It is an error to have a namespace variable with the same name as a global variable.

3.6.3 Declaring namespaces and namespace members

Typically, you declare a namespace in a header file. If your function implementations are in a separate file, then qualify the function names.

A namespace can be declared in multiple blocks in a single file, and in multiple files. The compiler joins the parts together during preprocessing and the resulting namespace contains all the members declared in all the parts. An example of this is the std namespace which is declared in each of the header files in the standard library.

Members of a named namespace can be defined outside the namespace in which they are declared by explicit qualification of the name being defined. However, the definition must appear after the point of declaration in a namespace that encloses the declaration's namespace.

3.6.4 The global namespace

If an identifier is not declared in an explicit namespace, it is part of the implicit global namespace. In general, try to avoid making declarations at global scope when possible, except for the entry point main Function, which is required to be in the global namespace. To explicitly qualify a global identifier, use the scope resolution operator with no name, as in *::SomeFunction(x);*. This will differentiate the identifier from anything with the same name in any other namespace, and it will also help to make your code easier for others to understand.

3.6.5 The std namespace

All C++ standard library types and functions are declared in the *std* namespace or namespaces nested inside *std*.

3.6.6 Nested namespaces

Namespaces may be nested. An ordinary nested namespace has unqualified access to its parent's members, but the parent members do not have unqualified access to the nested namespace (unless it is declared as inline).

Ordinary nested namespaces can be used to encapsulate internal implementation details that are not part of the public interface of the parent namespace.

3.6.7 Inline namespaces (C++ 11)

In contrast to an ordinary nested namespace, members of an inline namespace are treated as members of the parent namespace. This characteristic enables argument dependent lookup on overloaded functions to work on functions that have overloads in a parent and a nested inline namespace. It also enables you to declare a specialization in a parent namespace for a template that is declared in the inline namespace.

You can use inline namespaces as a versioning mechanism to manage changes to the public interface of a library. For example, you can create a single parent namespace, and encapsulate each version of the interface in its own namespace nested inside the parent. The namespace that holds the most recent or preferred version is qualified as inline, and is therefore exposed as if it were a direct member of the parent namespace. Client code that invokes the Parent::Class will automatically bind to the new code. Clients that prefer to use the older version can still access it by using the fully qualified path to the nested namespace that has that code.

The inline keyword must be applied to the first declaration of the namespace in a compilation unit.

3.6.8 Namespace aliases

Namespace names need to be unique, which means that often they should not be too short. If the length of a name makes code difficult to read, or is tedious to type in a header file where using directives can't be used, then you can make a namespace alias which serves as an abbreviation for the actual name.

3.7 Difference and relation between TypeScript module and C++ internal linkage

There is no obvious relationship between module in TypeScript and internal linkage in C++. Nevertheless, the usage of internal linkage has something similar to the variables defined inside a module in TypeScript.

An internal linking variable in C++ has its scope inside the file or the translation unit; while any variable inside a module in TypeScript has its scope inside this module and invisible to any member outside the module unless they are explicitly using one of the *export* forms. Hence this indicates both the similarity and the difference between a module in TypeScript and internal linkage in C++: both are visible to the specified scope while a variable in a module in TypeScript can be explicitly exported while an internal linking variable in C++ can never be exported again.

3.8 Difference and relation between TypeScript module and C++ external linkage

As stated in the previous section above, we can see the similarity and difference between the module in TypeScript and external linkage in C++. The module itself has nothing to do with the external linkage in C++ while the variables

in a module in TypeScript can have similar behavior as the external linking variables in C++.

The external linkage makes use of the variables defined somewhere else; similarly, when importing an module, we are also utilizing the variables defined somewhere else. However, there are still several distinct differences between the external linkage and the module import. As have been described in the several previous sections, importation in TypeScript can have several forms and could be used in freedom at a rather large scale. That is, we can specify whether we want to import a variable, a function, or several of them or the entire module. We can also specify any one of them an alias when importing, while in C++ we can only link one function or variable at a time, unable to change the name of it.

3.9 Difference and relation between TypeScript module and C++ name space

A module has great similarity with C++ name space – they are designed with the same purpose, actually. Both of them are designed to cope with the problem of naming conflict. Both variables in a module and variables in a name space could be accessed via the corresponding module or name space.

However, they have some more distinctions in detail.

1. Each module in TypeScript is at file level. That is, any file containing a top-level *import* or *export* is considered a module. Conversely, any file without a top-level *import* or *export* is not a module. However, in C++, name space does not have to have relationship with file. In a file, there can be several name spaces without interacting with each other.

2. In C++, name space has a weak relationship with files, that is, they are actually decoupled from each other. Any file can declare using any name space. When including files, the compiler will not care about which name space the file is using. And we can call functions by addressing their name space without importing any specific name space. However, in TypeScript, a module is coupled with a file and a file with *import* or *export* is a module. Since there is no including directive in TypeScript, importing an module is the process of importing an file. In addition, we can specify what contents the import and what alias we want to use.

3. Unlike C++, where during the process of importing, all the contents of the imported file are copied into the destination, TypeScript can choose what contents to import and what to be left.

4. In TypeScript, there is the mechanism of a *default export*, where such mechanism is missing in C++. However, it is not needed because C++ importing allows everything accessible.

3.10 Difference and relation between TypeScript name space and C++ internal linkage

Just like the relationship between TypeScript module and C++ internal linkage, TypeScript name space and C++ internal linkage are two different objects but the variables inside a TypeScript name space and the variables declared with internal linkage has something in common.

A variable defined inside a TypeScript can only be accessed inside the name space so as to avoid collision, and the internal linking variables are accessible in its scope. However, there are some distinctions between the scopes of the two things. A name space can go across various files and the functions and variables can be exported towards outside the name space. Nevertheless, in C++ internal linkage, the scope is strictly defined to be inside the translation unit and no more addition modifier is allowed.

3.11 Difference and relation between TypeScript name space and C++ external linkage

Just like the relationship between TypeScript module and C++ external linkage, TypeScript name space and C++ external linkage are two different objects but the variables inside a TypeScript name space and the variables declared with external linkage has something in common.

Variables inside a name space in TypeScript can be shared and accessible to the codes with the same name space, or exported so that can be used in other name spaces. However, in C++, the variables only need to be declared to be *external* variables and then it can be used if it is actually defined somewhere else.

3.12 Difference and relation between TypeScript name space and C++ name space

The name space in TypeScript and name space in C++ have great in common, especially from the perspectives of purpose of design and functions. Therefore, the similarity of TypeScript name space and the C++ name space is omitted here and only differences are mentioned.

In TypeScript name space, visibility mechanism is utilized so that developers can choose what are visible to the codes outside the name space and what are visible. However, in C++ everything in a name space is visible to the codes declaring using the same name space.

Bibliography

[1] Wikipedia contributors. (2019, May 5). TypeScript. In *Wikipedia, The Free Encyclopedia*. Retrieved 08:23, May 11, 2019, from `https://en.wikipedia.org/w/index.php?title=TypeScript&oldid=895568490`

[2] Wikipedia contributors. (2019, May 9). JavaScript. In *Wikipedia, The Free Encyclopedia*. Retrieved 08:24, May 11, 2019, from `https://en.wikipedia.org/w/index.php?title=JavaScript&oldid=896215789`

[3] Wikipedia contributors. (2019, April 24). C++. In *Wikipedia, The Free Encyclopedia*. Retrieved 08:25, May 11, 2019, from `https://en.wikipedia.org/w/index.php?title=C%2B%2B&oldid=893975016`

[4] Wikipedia contributors. (2019, February 27). Object-oriented programming. In *Wikipedia, The Free Encyclopedia*. Retrieved 10:22, March 23, 2019, from `https://en.wikipedia.org/w/index.php?title=Object-oriented_programming&oldid=885274966`

[5] Wikipedia contributors. (2019, March 18). Namespace. In *Wikipedia, The Free Encyclopedia*. Retrieved 08:27, May 11, 2019, from `https://en.wikipedia.org/w/index.php?title=Namespace&oldid=888324103`