

Modern C++ Programming

10. TRANSLATION UNITS

Federico Busato

University of Verona, Dept. of Computer Science
2021, v3.09



1 Basic Concepts

- Translation Unit
- Local and Global Scopes

2 Linkage

- `static` and `extern` Keywords
- Internal/External Linkage Example
- Linkage of `const` and `constexpr`
- Linkage of `inline` Functions/Variables

3 Variables Storage

- Storage Class
- Storage Duration

4 Dealing with Multiple Translation Units

- One Definition Rule (ODR)
- Class in Multiple Translation Units
- Global Constants

5 Function Template

6 Class Template

7 Undefined Behavior and Summary

8 #include Issues

- Forward Declaration
- Include Guard
- Circular Dependencies
- Common Linking Errors

9 Namespace

- Namespace Functions vs. Static Methods
- Namespace Alias
- Anonymous Namespace
- `inline` Namespace

10 How to Compile

- Compile Strategies
- Compile with Libraries
- Build Static/Dynamic Libraries
- Find Dynamic Library Dependencies
- Analyze Object/Executable Symbols

Basic Concepts

Translation Unit

Header File and Source File

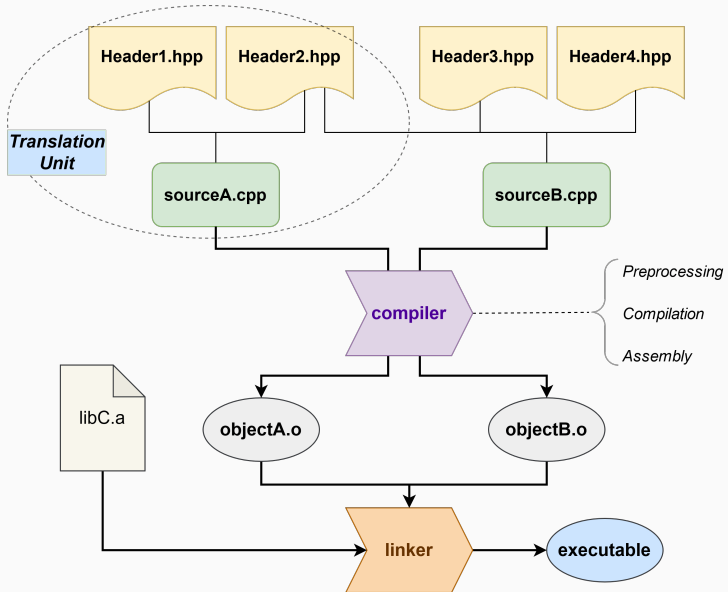
Header files allow to define interfaces (.h, .hpp, .hxx), while keeping the implementation in separated **source files** (.c, .cpp, .cxx).

Translation Unit

A **translation unit** (or *compilation unit*) is the basic unit of compilation in C++. It consists of the contents of a single source file, plus the contents of any header files directly or indirectly included by it

A single translation unit can be compiled into an object file, library, or executable program

Compile Process



Local and Global Scopes

Scope

The **scope** of a variable/function/object is the region of the code within the entity can be accessed

Local Scope

Variables that are declared inside a function or a block are called local variables (**local scope** or **block scope**)

Global Scope

Variables that are defined outside of all the functions and hold their value throughout the life-time of the program are global variables (**global scope** or **file scope**)

Local and Global Scopes

```
int var1;    // global scope

int f() {
    int var2; // local scope
}

struct A {
    int var3; // depends on where an instance of A is used
};

int main() {
    int var4; // local scope
}
```

Linkage

Linkage

Linkage

Linkage refers to the *visibility* of symbols to the linker

Internal Linkage

Internal linkage refers to symbols visible only in scope of a *single* translation unit

External Linkage

External linkage refers to entities that exist *outside* a single translation unit. They are accessible through the whole program, which is the combination of all translation units

No Linkage

No linkage refers to symbols visible only in the local scope of declaration

static and extern Keywords

`static` / *anonymous namespace-included global variables or functions* are visible only within the file (*internal linkage*)

- **Non-`static`** global variables or functions with the same name in different translation units produce name collision (or name conflict)

`extern` keyword is used to declare the existence of *global variables or functions* in another translation unit (*external linkage*)

- the variable or function must be defined in a one and only one translation unit

If, within a translation unit, the same identifier appears with both *internal* and *external* linkage, the behavior is undefined

static Variable Example

To not confuse with symbol visibility

```
#include <iostream>
using namespace std;

int f() {
    static int val = 1; // static
    val++;
    return val;
}

int main() {
    cout << f(); // print 1
    cout << f(); // print 2
    cout << f(); // print 3
}
```

Internal/External Linkage Example

```
int          var1 = 3;  // external linkage
                        // (in conflict with variables in other
                        // translation units with the same name)

static int   var2 = 4;  // internal linkage (visible only in the
                        // current translation unit)

extern int    var3;     // external linkage
                        // (implemented in another translation unit)

void         f1() {}    // external linkage (could conflict)

static void   f2() {}   // internal linkage

namespace {           // anonymous namespace
void         f3() {}    // internal linkage
}

extern void   f4();     // external linkage
                        // (implemented in another translation unit)

int main() {}
```

Linkage of const and constexpr

`const` variable at global scope implies `static`

→ *internal linkage*

`constexpr` implies `const`, which implies `static`

→ *internal linkage*

note: the same variable has different memory addresses on different translation units

```
const      int var1 = 3;          // internal linkage
constexpr int var2 = 2;          // internal linkage

static const      int var3 = 3; // internal linkage (redundant)
static constexpr int var4 = 2; // internal linkage (redundant)

int main() {}
```

`inline`

`inline` specifier allows a function to be defined identically (not only declared) in multiple translation units (source file + headers)

- `inline` is one of the most misunderstood features of C++
- `inline` is a hint for the linker. Without it, the linker can emit “multiple definitions” error
- It can be applied for optimization purposes only if the function has *internal* linkage (`static` or inside an anonymous namespace)
- C++17 `inline` can be also applied to variables

Multiple definitions of the same `inline`-declared function with *external linkage* do not break the ODR rule (allowed behavior)

note: the same `inline` variable with *external linkage* has the same memory address on different translation units

```
inline      int var1 = 3;  // external linkage (C++17)
static inline int var2 = 3; // internal linkage (C++17)

inline      void f() {}   // external linkage
static      void g1() {}  // internal linkage
static inline void g2() {} // internal linkage
namespace {
    inline void g3() {}    // internal linkage
} // anonymous namespace -> same as static
```

`f()` :

- Can be defined in a header and included in multiple source files
- The linker removes all definitions except one
- Declaring `void f();` in a file that does not include the header is still valid because the function has *external* linkage

`g1(), g2(), g3()` :

- Can be defined in a header included in multiple source files
- The compiler replicates the code in each translation unit (the linker does not see these functions)
- Declaring `void g1();` in a file that does not include the header is no more valid because the function has *internal* linkage

Summary

No Linkage:

- Local **Variables, Functions, Classes**

Internal Linkage:

- **Variables:** Global `static`, `const`, `constexpr`
- **Functions*:** `static`, `constexpr`
- Anonymous `namespace` content

External Linkage:

- **Variables:** Global
- **Functions*:** `extern`, `template`
- `static` class data member (that are not `inline`)

* Windows (MSVC) treats function visibility in a different way
gcc.gnu.org/wiki/Visibility

Variables Storage

Storage Class

Storage Class Specifier

The **storage class** for a variable declaration is a type **specifier** that, *together with the scope*, governs its *storage duration* and its *linkage*

- Only one storage class specifier may appear in a declaration except that `thread_local`

Storage Class	Keyword	Scope	Duration Class	Linkage
Automatic	<code>auto</code> *	Local	Automatic	No linkage
Automatic	<code>auto</code> *	Global	Automatic	External
Register [†]	<code>register</code>	Local	Automatic	No linkage
Static	<code>static</code>	Local	static	No linkage
Static	<code>static</code>	Global	static	Internal
External	<code>extern</code>	Global	static	External
Thread Local*	<code>thread_local</code>	any	Thread Local	any

Storage Class Examples

```
int          v2;      // automatic storage
static      int v3 = 2; // static storage (global)
extern      int v4;    // external storage
thread_local int v5;    // thread local storage
thread_local static int v6; // thread local storage

int main() {
    int          v7;      // automatic storage
    auto         v8 = 3;   // automatic storage
    register int  v9;      // register storage (deprecated!)
    static int    v10;     // static storage (local)
    thread_local int v11;   // thread local storage
                        // (each thread has its own value)
    auto array = new int[10]; // automatic storage (array variable)
}
```

Storage Duration

The **storage duration** (or *duration class*) determines the *duration* of a variable, namely when it is created and destroyed

Storage Duration	Keyword	Allocation	Deallocation
Automatic	auto/no keyword	Code block start	Code end start
Static	static, global scope variable, extern	Program start	Program end
Dynamic	new/delete	Memory allocation	Memory deallocation
Thread	thread_local	Thread start	Thread end

Full Story:

http://en.cppreference.com/w/cpp/language/storage_duration

Automatic storage duration. Scope variables (local variable). register or stack (depending on compiler, architecture, etc.).

`register` hints to the compiler to place the object in the processor registers (deprecated in C++11)

Static storage duration. The storage for the object is allocated when the program begins and deallocated when the program ends (`static` keyword at local or global scope)

Thread storage duration C++11. The object is allocated when the thread begins and deallocated when the thread ends. Each thread has its own instance of the object. (`thread_local` can appear together with `static` or `extern`)

Dynamic storage duration. The object is allocated and deallocated per request by using dynamic memory allocation functions (`new/delete`)

Storage Duration Examples

```
int          v1;          // static duration
static int   v2 = 4;      // static duration
extern int    v3;         // static duration

void f() {
    int      v4;          // automatic duration
    auto     v5 = 3;      // automatic duration
    static int v6;        // static duration
    auto array = new int[10]; // dynamic duration (allocation)
} // array, v1, v2, v3, v6 variables deallocation (from stack)
  // the memory associated with "array" is not deallocated!!

int main() {
    auto array = new int[10]; // dynamic duration (allocation)
    delete[] array;          // dynamic duration (deallocation)
}

// main end: v1, v2, v3, v6 deallocation
```

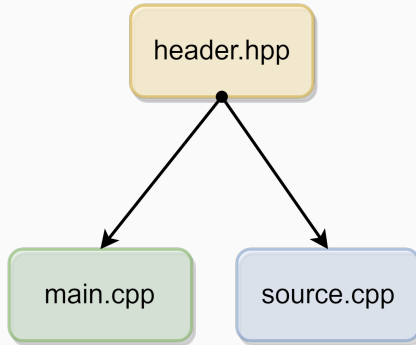
Dealing with Multiple Translation Units

One Definition Rule (ODR)

- (1) In any **(single) translation unit**, a template, type, function, or object, *cannot* have more than one definition
 - *Compiler error* otherwise
 - Any number of declarations are allowed
- (2) In the **entire program**, an object or non-inline function *cannot* have more than one definition
 - *Multiple definitions linking error* otherwise
 - Entities with *internal linkage* in different translation units are allowed, even if their names and types are the same
- (3) A template, type, or inline functions/variables, can be defined in more than one translation unit. For a given entity, each definition must be the same
 - *Undefined behavior* otherwise
 - Common case: same header included in multiple translation units

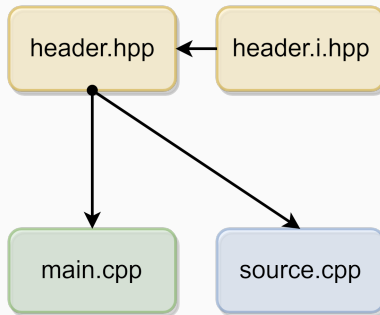
One Definition Rule - Code Structure 1

- one header, two source files → two translation units
- *the header is included in both translation units*



One Definition Rule - Code Structure 2

- two headers, two source files → two translation units
- one header for declarations (.hpp), and the other one for implementations (.i.hpp)
- *the header and the header implementation are included in both translation units*



* separate header declaration and implementation is not mandatory but it could help to better organize the code

One Definition Rule (Example, points (1), (2))

header.hpp:

```
void f();
```

main.cpp:

```
#include "header.hpp"
#include <iostream>

// external linkage
int      a = 1;

extern int b;
// internal linkage
static int c = 2;

int main() {
    std::cout << a; // print 1
    std::cout << b; // print 5
    std::cout << c; // print 2
    f();           // print 4
}
```

source.cpp:

```
#include "header.hpp"
#include <iostream>
// linking error !!
// (multiple definitions)
// int      a = 2;

int      b = 5;    // ok
// internal linkage
static int c = 4;  // ok

void f() {          // definition
    std::cout << b; // print 5
}
```

header.hpp:

```
inline void f() {} // the function is inline (no linking error)

template<typename T>
void g(T x) {}      // the function is a template (no linking error)

using var_t = int; // types can be defined multiple times (no linking error)
```

main.cpp:

```
#include "header.hpp"

int main() {
    f();
    g(3); // g<int> generated
}
```

source.cpp:

```
#include "header.hpp"

void h() {
    f();
    g(3); // g<int> generated
}
```

Alternative organization:

header.hpp:

```
inline void f();    // declaration
inline int gvalue;  // declar. (C++17)

template<typename T>
void g(T x);        // declaration

using var_t = int;  // type
#include "header.i.hpp"
```

header.i.hpp:

```
void f() {}         // definition
int gvalue = 3;     // def. (C++17)

template<typename T>
void g(T x) {}      // definition
```

main.cpp:

```
#include "header.hpp"

int main() {
    f();
    g(3); // g<int> generated
}
```

source.cpp:

```
#include "header.hpp"

void h() {
    f();
    g(3); // g<int> generated
}
```


header.hpp:

```
class A {  
public:  
    void f();  
    static void g();  
private:  
    int x;  
    static int y;  
};
```

main.cpp:

```
#include "header.hpp"  
#include <iostream>  
  
int main() {  
    A a;  
    std::cout << A.x; // print 1  
    std::cout << A.y; // print 2  
}
```

source.cpp:

```
#include "header.hpp"  
  
void A::f() {}  
void A::g() {}  
  
int A::x = 1;  
int A::y = 2;
```

header.hpp:

```
struct A {  
    int x1;  
    int x2 = 3;  
    int x3 { 4 };  
  
    static int y;    // zero-init  
    // static int y = 3; // compile error  
    //           must be initialized out-of-class  
  
    const int z = 3; // only in C++11  
    // const int z;    // compile error  
    //           must be initialized  
  
    static const int w1;    // zero-init  
    static const int w2 = 4; // inline  
    //           definition  
};
```

source.cpp:

```
#include "header.hpp"  
  
int A::x1 = 1;  
int A::y = 2;  
const int A::w1 = 3;
```

ODR Common Errors (Classes)

header.hpp:

```
struct A {  
    void f() {}; // declaration/definition inside struct (correct)  
    void g();    // declaration  
    void h();    // declaration  
};  
  
void A::g() {} // definition!! multiple definitions error
```

main.cpp:

```
#include "header.hpp"  
// linking error !!  
// multiple definitions of A::g()  
  
int main() {  
}
```

source.cpp:

```
#include "header.hpp"  
// linking error !!  
// multiple definitions of A::g()  
  
void A::h() { // definition, ok  
}
```

Global Constants

header.hpp:

```
#include <iostream>

struct A {
    A() { std::cout << "A()"; }
    ~A() { std::cout << '~A()'; }
};

// A          obj;          // linker error multiple definitions
const A       const_obj;    // "const" implies static (as "constexpr")
constexpr float PHI = 3.14f;
```

source1.cpp:

```
#include "header.hpp"

void f() { std::cout << &PHI; }
// address: 0x1234ABCD

// print "A()" the first time
// print "~A()" the first time
```

source2.cpp:

```
#include "header.hpp"

void f() { std::cout << &PHI; }
// print address: 0x3820FDAC !!

// print "A()" the second time!!
// print "~A()" the second time!!
```

Function Template

Function Template - Case 1

header.hpp:

```
template<typename T>
void f(T x) {}; // declaration and definition
```

main.cpp:

```
#include "header.hpp"

int main() {
    f(3);      // call f<int>()
    f(3.3f);   // call f<float>()
    f('a');    // call f<char>()
}
```

source.cpp:

```
#include "header.hpp"

void h() {
    f(3);      // call f<int>()
    f(3.3f);   // call f<float>()
    f('a');    // call f<char>()
}
```

Function Template Specialization - Case 2

header.hpp:

```
template<typename T>
void f(T x); // DECLARATION
```

main.cpp:

```
#include "header.hpp"

int main() {
    f(3);    // call f<int>()
    f(3.3f); // call f<float>()
    // f('a'); // linking error
} // specialization does not
   // exist
```

source.cpp:

```
#include "header.hpp"

template<typename T>
void f(T x) {} // DEFINITION

// template SPECIALIZATION
template void f<int>(int y);
template void f<float>(float y);
```

Function Template Specialization - Case 3a

header.hpp:

```
template<typename T>
void f(T x) {} // DECLARATION and DEFINITION

template<>
void f<int>(); // inform that the specialization exists in
              // another translation unit
```

source.cpp:

```
#include "header.hpp"

template<>
void f<int>(int x) {} // SPECIALIZATION
```


Function Template Specialization - Case 3b

C++11

header.hpp:

```
template<typename T>
void f(T x) {} // DECLARATION and DEFINITION

extern template void f<int>(); // inform that the specialization
                               // exists in another translation
                               // unit
```

source.cpp:

```
// #include "header.hpp" must be avoided

template<typename T>
void f(T x); // declaration;

template<>
void f<int>(int x) {} // SPECIALIZATION
```

Function Template Specialization - Case 3c

C++11

header.hpp:

```
template<typename T>
void f(T x); // DECLARATION

extern template void f<int>(); // inform that the specialization
                               // exists in another translation
                               // unit
```

source.cpp:

```
// #include "header.hpp" must be avoided

template<typename T>
void f(T x) {} // DECLARATION and DEFINITION

template<>
void f<int>(int x); // SPECIALIZATION
```

ODR Common Errors - Function Templates

header.hpp:

```
template<typename T>
void f();

// template<>           // linking error
// void f<int>() {}      // (multiple definitions) included twice
                        // full specializations are standard functions
```

main.cpp:

```
#include "header.hpp"

int main() {
// f<int>(); // linking error
} // f<int>() is not defined here
```

source.cpp:

```
#include "header.hpp"

template<typename T>
void f() {}
// error: valid only in this
// translation unit!!

void g() {
    f<char>(); // ok
}
```

Class Template

Class Template

header.hpp:

```
template<typename T>
struct A {
    T    x;    // declaration
    void f();  // declaration
};
#include "header.i.hpp"
```

header.i.hpp:

```
template<typename T>
T A<T>::x = 3; // definition

template<typename T>
void A<T>::f() {}
```

main.cpp:

```
#include "header.hpp"

int main() {
    A<int>    a1; // ok
    A<float>  a2; // ok
    A<char>   a3; // ok
}
```

source.cpp:

```
#include "header.hpp"

int g() {
    A<int>    a1; // ok
    A<float>  a2; // ok
    A<char>   a3; // ok
}
```

Class Template Specialization - Case 1

header.hpp:

```
template<typename T>
struct A {
    T    x;    // declaration
    void f();  // declaration
};
```

main.cpp:

```
#include "header.hpp"

int main() {
    A<int>  a1; // ok
    // A<char> a2; // compile error
}
```

source.cpp:

```
#include "header.hpp"

template<typename T>
int A<T>::x = 3; // definition

template<typename T>
void A<T>::f() {} // definition

// template specialization
template class A<int>;
```

Class Template Specialization - Case 2

C++11

header.hpp:

```
template<typename T>
struct A {
    void f(); // declaration
};

extern template class A<int>;
```

source.cpp:

```
#include "header.hpp"

template<typename T>
void A<T>::f() {} // definition

// template specialization
template class A<int>;
```

Undefined Behavior and Summary

Undefined Behavior - Example 1

main.cpp:

```
#include <iostream>
inline int f() { return 3; }

void g();

int main() {
    std::cout << f(); // print 3
    std::cout << g(); // print 3 !!
}
```

source.cpp:

```
// same signature and inline
inline int f() { return 5; }

int g() { return f(); }
```

The linker can *arbitrary* choose one of the two definitions of `f()`.
With `-O3`, the compiler can *inline* `f()` in `g()`, so now `g()` return 5

This issue is easy to detect in trivial examples but hard to find in large codebase

Solution: `static` functions (or `anonymous namespace`)

Undefined Behavior - Example 2

main.cpp:

```
#include <iostream>
template<typename T>
struct A {
    int f();
};
int g();

int main() {
    A<int> a;
    std::cout << a.f() << ", " << g(); // print 3, 3!!
}
```

source1.cpp:

```
//
template<typename T>
int A<T>::f() { return 3; }

template class A<int>;
```

source2.cpp:

```
template<typename T>
struct A {
    int f() { return 5; }
};
int g() {
    A<int> a; return a.f();
}
```

Undefined Behavior

Other ODR violations are even harder (if not impossible) to find, e.g. Diagnosing Hidden ODR Violations in Visual C++

Some tools for partially detecting ODR violations:

- `-detect-odr-violations` flag for gold/llvm linker
- `-Wodr -flto` flag for GCC
- Clang address sanitizer +
`ASAN_OPTIONS=detect_odr_violation=2` (link)

Another solution could be include all files in a single translation unit

Summary

- **header:** declaration of
 - structs/classes/types/alias
 - functions
 - template function/structs/classes
 - extern variables/functions
 - global const/constexpr variables
- **header implementation:** definition of
 - inline functions/variables
 - template functions/classes
- **source file:** definition of
 - functions
 - templates full specialization (function/class)
 - static global variables (+ declaration)
 - extern variables/functions

#include Issues

Forward declaration is a declaration of an identifier for which a complete definition has not yet given

“*forward*” means that an entity is declared before it is used

Functions and **Classes** have external linkage by default

main.cpp:

```
void f(); // function forward declaration
class A;  // class forward declaration

class B {
    A* a; // ok, A* is declared
    // A b; // compiler error no definition (incomplete type)
}; // e.g. the compiler is not able to deduce the size of A

int main() {
    f(); // ok, f() is a function and not a variable
    // A a; // compiler error no definition (incomplete type)
}
```

source.cpp:

```
void f() {} // definition of f()
class A {}; // definition of A()
```

Advantages:

- Forward declarations can save compile time, as `#include` force the compiler to open more files and process more input
- Forward declarations can save on unnecessary recompilation.
`#include` can force your code to be recompiled more often, due to unrelated changes in the header

Disadvantages:

- Forward declarations can hide a dependency, allowing user code to skip necessary recompilation when headers change
- A forward declaration may be broken by subsequent changes to the library
- Forward declaring multiple symbols from a header can be more verbose than simply `#including` the header

Full Story:

google.github.io/styleguide/cppguide.html#Forward_Declarations

The `include guard` avoids the problem of multiple inclusions of a header file in a translation unit

`header.hpp`:

```
#ifndef HEADER_HPP // include guard
#define HEADER_HPP

... many lines of code ...

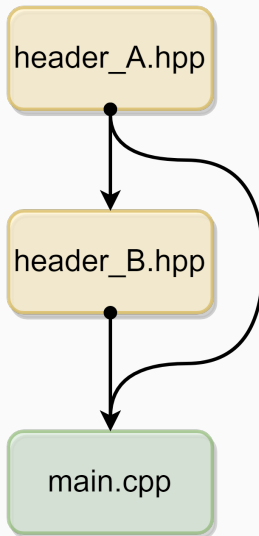
#endif // HEADER_HPP
```

`#pragma once` preprocessor directive is an alternative to the `include guard` to force current file to be included only once in a translation unit

- `#pragma once` is less portable but less verbose and compile faster than the `include guard`

The `include guard`/`#pragma once` should be used in every header file

Common case:



header_A.hpp:

```
#pragma once    // prevents "multiple definitions" linking error

struct A {
};
```

header_B.hpp:

```
#include "header_A.hpp" // included here

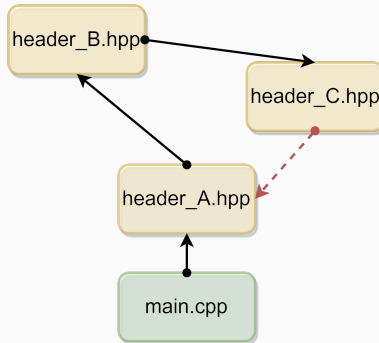
struct B {
    A a;
};
```

main.cpp:

```
#include "header_A.hpp" // .. and included here
#include "header_B.hpp"

int main() {
    A a; // ok, here we need "header_A.hpp"
    B b; // ok, here we need "header_B.hpp"
}
```

A **circular dependency** is a relation between two or more modules which either directly or indirectly depend on each other to function properly



Circular dependencies can be solved by using forward declaration, or better, by rethinking the project organization

header_A.hpp:

```
#pragma once
#include "header_B.hpp"

class A {
    B* b;
};
```

header_B.hpp:

```
#pragma once
#include "header_C.hpp"

class B {
    C* c;
};
```

header_C.hpp:

```
#pragma once
#include "header_A.hpp"

class C { // compile error "header_A" already included by "main.cpp"
    A* a; // the compiler cannot view the "class C"
};
```

header_A.hpp:

```
#pragma once
class B;    // forward declaration
           // note: does not include "header_B.hpp"

class A {
    B* b;
};
```

header_B.hpp:

```
#pragma once
class C;    // forward declaration

class B {
    C* c;
};
```

header_C.hpp:

```
#pragma once
class A;    // forward declaration

class C {
    A* a;
};
```

Common Linking Errors

Very common *linking* errors:

- **undefined reference**

Solutions:

- Check if the right headers are included
- Break circular dependencies with forward declarations

- **multiple definitions**

Solutions:

- `inline` function/variable definition or `extern` declaration
- Add `include guard/#pragma once` to header files
- Place template definition in header file and full specialization in source files

Namespace

The problem: Named entities, such as variables, functions, and compound types declared outside any block has *global scope*, meaning that its name is valid anywhere in the code

Namespaces allow to group named entities that otherwise would have global scope into narrower scopes, giving them ***namespace scope*** (where *std* stands for “standard”)

Namespaces provide a method for preventing name conflicts in large projects. Symbols declared inside a namespace block are placed in a named scope that prevents them from being mistaken for identically-named symbols in other scopes

Namespace Functions vs. Static Methods

Namespace functions:

- Namespace can be extended anywhere (without control)
- Namespace specifier can be avoided with the keyword `using`

Static methods:

- Can interact with static data members
- Struct/Class cannot be extended outside their declarations

Static methods should define operations strictly related to object definition, otherwise *namespace* should be preferred

Defining a Namespace

```
#include <iostream>

namespace ns1 {
void f() {
    std::cout << "ns1" << endl;
}
} // namespace ns1

namespace ns2 {
void f() {
    std::cout << "ns2" << endl;
}
} // namespace ns2

int main () {
    ns1::f(); // print "ns1"
    ns2::f(); // print "ns2"
    // f();    // compile error f() is not visible
}
```

Namespace Conflits

```
#include <iostream>
using namespace std;
void f() {
    cout << "global" << endl;
}

namespace ns1 {
    void f() { cout << "ns1::f()" << endl; }
    void g() { cout << "ns1::g()" << endl; }
}

int main () {
    f();          // ok, print "global"
    // g();       // compile error g() is not visible

    using namespace ns1;
    // f();       // compile error ambiguous function name
    ::f();        // ok, print "global"
    ns1::f();     // ok, print "ns1::f()"
    g();          // ok, print "ns1::g()", only one choice
}
```

Nested Namespaces

```
#include <iostream>
using namespace std;
namespace ns1 {
    void f() { cout << "ns1::f()" << endl; }
namespace ns2 {
    void f() { cout << "ns1::ns2::f()" << endl; }
}
}

namespace ns1 { // the same namespace can be declared multiple times,
namespace ns2 { // and extended in multiple files
    void g() {}
}
}
```

C++17 allows nested namespace definitions:

```
namespace ns1::ns2 {
    void h()
}
```

Namespace Scope

```
#include <iostream>
using namespace std;
namespace ns1 {
    void f() { cout << "ns1::f()" << endl; }
}
namespace ns2 {
    void f() { cout << "ns1::ns2::f()" << endl; }
    void g() { cout << "ns1::ns2::g()" << endl; }
}

namespace ns1 {
    void g() {}           // ok
    // void f() {}       // compile error function name conflict with
                        // header.hpp: "ns1::f()"
}

int main() {
    ns1::f();             // ok, print "ns1::f()"
    ns1::ns2::f();        // ok, print "ns1::ns2::f()"
    using namespace ns1;
    g();                  // ok, print "ns1::ns2::g()"
}
```

Namespace Alias

Namespace alias allows declaring an alternate name for an existing namespace

```
namespace very_very_long_namespace {  
    void g() {}  
}  
  
int main() {  
    namespace ns = very_very_long_namespace; // namespace alias  
    ns::g();  
}
```

Anonymous Namespace

A namespace with no identifier before an opening brace produces an **unnamed/anonymous namespace**

Entities inside an anonymous namespace are used for declaring unique identifiers, visible in the same source file

Anonymous namespaces vs. static global entities

- Anonymous namespaces allow *type declarations*, and they are *less verbose*

main.cpp

```
#include <iostream>
namespace { // anonymous
    void f() { std::cout << "main"; }
}           // internal linkage

int main() {
    f();    // ok, print "main"
}
```

source.cpp

```
#include <iostream>
namespace { // anonymous
    void f() { std::cout << "source"; }
}

int g() {
    f();    // ok, print "source"
}
```

inline Namespace

inline namespaces is a concept similar to library versioning. It is a mechanism that makes a nested namespace look and act as if all its declarations were in the surrounding namespace

```
namespace ns1 {  
    inline namespace V99 {  
        void f(int) {}    // most recent version  
    }  
    namespace V98 {  
        void f(int) {}  
    }  
}  
  
using namespace ns1;  
  
int main() {  
    V98::f(1);    // call V98  
    V99::f(1);    // call V99  
    f(1);         // call default version (V99)  
}
```


How to Compile

Compile Strategies

Method 1

Compile all files together (naive):

```
g++ -Iinclude main.cpp source.cpp -o main.x
```

Specify the **include path** to the compiler: `-I`

`-I` can be used multiple times

Method 2

Compile each *translation unit* in a file object:

```
g++ -c -Iinclude source.cpp -o source.o
```

```
g++ -c -Iinclude main.cpp -o main.o
```

Link all file objects:

```
g++ main.o source.o -o main.x
```

see CMake in “Debugging and Tools” lecture

C++ Libraries

A **library** is a package of code that is meant to be reused by many programs

A **static library** (.a) consists of routines that are compiled and linked directly into your program. If a program is compiled with a static library, all the functionality of the static library becomes part of your executable

- A static library cannot be modified without re-compile
- Increase the size of the binary

A **dynamic library**, also called a **shared library** (.so), consists of routines that are loaded into your application at run-time. If a program is compiled with a dynamic library, the library does not become part of your executable. It remains as a separate unit

- A dynamic library can be modified without re-compile
- Dynamic library functions are called outside the executable

Compile with Libraries

Specify the **library path** (path where search for static/dynamic libraries) to the compiler:

```
g++ -L<library_path> main.cpp -o main
```

-L can be used multiple times

Specify the **library name** (e.g. liblibrary.a) to the compiler:

```
g++ -llibrary main.cpp -o main
```

Linux/Unix Environmental variables:

- **LIBRARY_PATH** Specify the directories where search for *static* libraries at *compile-time*. Used by the compiler
- **LD_LIBRARY_PATH** Specify the directories where search for *dynamic/shared* libraries at *run-time*. Used by the program

Build Static/Dynamic Libraries

Static Library Creation

- Create object files for each translation unit (.cpp)
- Create the static library by using the **archiver** (**ar**) linux utility

```
g++ source1.c -c source1.o
g++ source2.c -c source2.o
ar rvs libmystaticlib.a source1.o source2.o
```

Dynamic Library Creation

- Create object files for each translation unit (.cpp). Since library cannot store code at fixed addresses the compile must generate *position independent code*
- Create the dynamic library

```
g++ source1.c -c source1.o -fPIC
g++ source2.c -c source2.o -fPIC
g++ source1.o source2.o -shared -o libmydynamiclib.so
```

Find Dynamic Library Dependencies

The `ldd` utility shows the shared objects (shared libraries) required by a program or other shared objects

```
$ ldd /bin/ls
linux-vdso.so.1 (0x00007ffcc3563000)
libselinux.so.1 => /lib64/libselinux.so.1 (0x00007f87e5459000)
libcap.so.2 => /lib64/libcap.so.2 (0x00007f87e5254000)
libc.so.6 => /lib64/libc.so.6 (0x00007f87e4e92000)
libpcre.so.1 => /lib64/libpcre.so.1 (0x00007f87e4c22000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007f87e4a1e000)
/lib64/ld-linux-x86-64.so.2 (0x00005574bf12e000)
libattr.so.1 => /lib64/libattr.so.1 (0x00007f87e4817000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f87e45fa000)
```

The `nm` utility provides information on the symbols being used in an object file or executable file

```
$ nm -D -C something.so
```

```
  w __gmon_start__
```

```
  D __libc_start_main
```

```
  D free
```

```
  D malloc
```

```
  D printf
```

```
# -C:  Decode low-level symbol names
```

```
# -D:  accepts a dynamic library
```

`readelf` displays information about ELF format object files

```
$ readelf --symbols something.so | c++filt
... OBJECT LOCAL DEFAULT 17 __frame_dummy_init_array_
... FILE LOCAL DEFAULT ABS prog.cpp
... OBJECT LOCAL DEFAULT 14 CC1
... OBJECT LOCAL DEFAULT 14 CC2
... FUNC LOCAL DEFAULT 12 g()
```

--symbols: display symbol table

`objdump` displays information about object files

```
$ objdump -t -C something.so | c++filt
... df *ABS*      ...  prog.cpp
...  0 .rodata    ...  CC1
...  0 .rodata    ...  CC2
...  F .text      ...  g()
...  0 .rodata    ...  (anonymous namespace)::CC3
...  0 .rodata    ...  (anonymous namespace)::CC4
...  F .text      ...  (anonymous namespace)::h()
...  F .text      ...  (anonymous namespace)::B::j1()
...  F .text      ...  (anonymous namespace)::B::j2()
```

--t: display symbols

-C: Decode low-level symbol names

References and Additional Material

- 20 ABI (Application Binary Interface) breaking changes every C++ developer should know
- Policies/Binary Compatibility Issues With C++
- 10 differences between static and dynamic libraries every C++ developer should know