

# Modern C++ Programming

## 13. TRANSLATION UNITS I

### LINKAGE AND ONE DEFINITION RULE

---

*Federico Busato*

2026-01-06

## **1** Basic Concepts

- Translation Unit
- Local and Global Scope
- Linkage
- Visibility

## 2 Storage Class and Duration

- Storage Duration
- Storage Class
- `static` Keyword
- Anonymous Namespace
- `extern` Keyword

## **3** Linkage of `const` and `constexpr` Variables

- Static Initialization Order Fiasco

## **4** Linkage Summary

## **5** Dealing with Multiple Translation Units

- Class in Multiple Translation Units

## 6 One Definition Rule (ODR)

- Global Variable Issues
- ODR - Point (3)
- `inline` Functions/Variables
- `constexpr` and `inline`

## 7 ODR - Function Template

- Cases
- `extern` Keyword

## **8** ODR - Class Template

- Cases
- `extern` Keyword

## **9** ODR Undefined Behavior and Summary

# Basic Concepts

---

# Translation Unit

## Header File and Source File

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

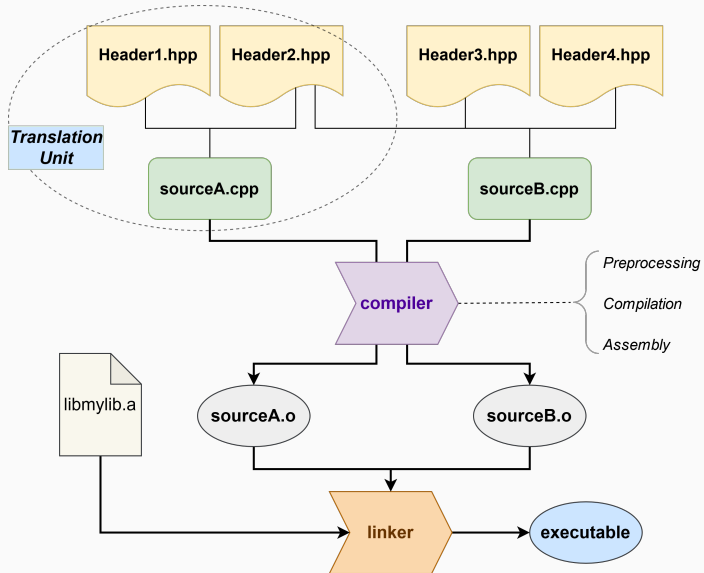
## Translation Unit

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

A *single translation unit* can be compiled into an object file, library, or executable program. The *final program or library* is the combination of all translation units



# Compile Process



# Local and Global Scope

## Scope

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

## Local Scope / Block Scope

Entities that are declared inside a function or a block are called local variables. Their memory address is not valid outside their scope

## Global Scope / File Scope / Namespace Scope

Entities that are defined outside all functions.  
They hold a single memory location throughout the life-time of the program

# Local and Global Scope

```
int var1;    // global scope

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

struct A {
    int var3; // depends on where the instance of 'A' is used
};
```

# Linkage

## Symbols

**Symbols** are named items (identifiers) in a program and they refer to entities such as variables, functions, classes

## Linkage

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

There are three ways to link symbols:

- *No linkage*
- *Internal linkage*
- *External linkage*

# Linkage

## No Linkage

**No linkage** refers to symbols in the local scope of declaration and not visible to the linker.

## Internal Linkage

**Internal linkage** refers to symbols accessible only in scope of a *single* translation unit. The same symbol name has a different memory address in distinct translation units.

## External Linkage

**External linkage** refers to symbols that are accessible *outside* a single translation unit. Symbols with external linkage have an *unique memory address* through the whole program. The same name in different translation units causes conflicts.

## Visibility

**Symbol visibility** determines whether a symbol is exported from a shared library or executable, making it available for dynamic linking.

The symbol table holds information about these symbols, including their names, addresses, linkage, and visibility.

Visibility attributes:

**default** The symbol is visible to external binaries and can be linked against.

**hidden** The symbol is not visible to external binaries; it is only accessible within the defining component.

Multiple libraries that export symbols with the same name could potentially cause undefined behavior at runtime.

*Symbol visibility* and *linkage* are related but distinct concepts.

- Symbols with *internal linkage* cannot be exported (*hidden visibility*).
- Symbols with *external linkage* are exported if they have *default visibility*.

GCC/Clang Default visibility: `default`. The compilers allow the opposite behavior with `-fvisibility=hidden`.

MSVC Default visibility: `hidden`.

Additionally, it is possible to control individual symbol visibility:

```
__attribute__((visibility("hidden")))
void internal_function(); // GCC/Clang

__attribute__((visibility("default")))
int exported_variable = 3;
```

```
__declspec(dllexport) // MSVC
void exported_function();

__declspec(dllimport)
void imported_function();
```

# Storage Class and Duration

---



## Storage Duration

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

Storage Duration	Allocation	Deallocation
<b>Automatic</b>	Code block start	Code block end
<b>Static</b>	Program start	Program end
<b>Dynamic</b>	Memory allocation	Memory deallocation
<b>Thread</b>	Thread start	Thread end

- **Automatic storage duration**. Local variables temporary allocated on registers or stack (depending on compiler, architecture, etc.).  
*If not explicitly initialized, their value is undefined*
- **Static storage duration**. The storage of an object is allocated when the program begins and deallocated when the program ends.  
*If not explicitly initialized, it is zero-initialized*
- **Dynamic storage duration**. The object is allocated and deallocated by using dynamic memory allocation functions ( `new/delete` ).  
*If not explicitly initialized, its memory content is undefined*
- **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

# Storage Duration Examples

```
int v1; // static duration

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

int main() {
    f();
}
// main end: v1 is deallocated
```

# 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 *linkage*

Storage Class	Notes	Scope	Storage Duration	Linkage
<i>no storage class</i>	local <code>var</code> decl.	Local	<i>automatic</i>	<i>No linkage</i>
<i>no storage class</i>	global <code>var</code> decl.	Global	<i>static</i>	<i>External</i>
<code>static</code>		Local	<i>static</i>	<i>Function Dependent</i>
<code>static</code>		Global	<i>static</i>	<i>Internal</i>
<code>extern</code>		Global	<i>static</i>	<i>External</i>
<code>thread_local</code>	C++11	any	<i>thread local</i>	<i>any</i>

# Storage Class Examples

```
int          v1;      // no storage class
static      int v2 = 2; // static storage class
extern      int v3;    // external storage class
thread_local int v4;    // thread local storage class
thread_local static int v5; // thread local and static storage classes

int main() {
    int          v6;      // auto storage class
    auto         v7 = 3;  // auto storage class
    static int    v8;      // static storage class
    thread_local int v9;    // thread local and auto storage classes
    auto array = new int[10]; // auto storage class ("array" variable)
}
```

## static Keyword for Local Variables

`static` *local variables* are allocated when the program begins, *initialized* when the function is called the first time, and deallocated when the program ends

```
int f() {  
    static int val = 1;  
    val++;  
    return val;  
}  
  
int main() {  
    cout << f(); // print 2 ("val" is initialized)  
    cout << f(); // print 3  
    cout << f(); // print 4  
}
```

## static Keyword for Global Variables

`static` *global variables* or *functions* are visible only within the translation unit where they are declared → *internal linkage*

- **Non-`static`** *global variables* or *functions* with the same name in different translation units produce *name collision* (or name conflict) → multiple definitions at link-time

```
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)

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

static void f2() {}    // internal linkage
```

A namespace with no identifier is called **unnamed/anonymous namespace**

Entities within an anonymous namespace have *internal linkage* and, therefore, are used for declaring unique identifiers, visible only in the same source file

## Anonymous namespace vs. global static functions/variables:

- Entities withing an *anonymous namespace* have the same properties of `static` declarations at global scope
- In addition, *anonymous namespaces* allow *type declarations* and *class definitions*
- *Anonymous namespaces* are less verbose than `static` variables/functions but, entities within an *anonymous namespace* are less visible if the scope contains many lines



main.cpp

```
#include <iostream>

namespace { // anonymous, internal linkage

void f() { std::cout << "main"; }

using my_int = int; // not possible
                  // with 'static'

} // namespace

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

source.cpp

```
#include <iostream>

namespace { // anonymous, internal linkage

void f() { std::cout << "source"; }

using my_int = unsigned; // no conflicts

} // namespace

int g() {
    f(); // print "source", no conflicts
}
```

`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 one and only one translation unit
- it is redundant for functions
- it is necessary for variables to prevent the compiler to associate a memory location in the current translation unit

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

## External Linkage Example

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

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

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

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

# Linkage of `const` and `constexpr` Variables

---

# Linkage of `const` and `constexpr` Variables

`const` variables have *internal linkage* at global scope

`constexpr` variables imply `const`, which implies *internal linkage*

*note:* the same variable has different memory addresses on different translation units (code bloat)

```
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() {}
```

In C++, the order in which global variables are initialized at runtime is not defined. This introduces a subtle problem called *static initialization order fiasco*

source.cpp

```
int f() { return 3; } // run-time function

int x = f();          // run-time evaluation
```

main.cpp

```
extern int x;
int      y = x; // run-time initialized

int main() {
    cout << y;    // print "3" or "0" depending on the linking order
}
```

source.cpp

```
constexpr int f() { return 3; } // compile-time/run-time function

constexpr int x = f();          // compile-time initialized (C++20)
```

main.cpp

```
constexpr extern int x;         // compile-time initialized (C++20)
int y = x; // run-time initialized

int main() {
    cout << y; // print "3"!!
}
```

# Linkage Summary

---



## No Linkage: Local variables, functions, classes

- `static` local variable address depends on the linkage of its function

## Internal Linkage:

(not accessible by other translation units, no conflicts, different memory addresses)

- **Global Variables:**
  - `static`
  - *non-inline, non-template, non-specialized, non-extern* `const` / `constexpr`
- **Functions:** `static`
- Anonymous `namespace` content, even structures/classes

## External Linkage:

(accessible by other translation units, potential conflicts, same memory address)

### ▪ Global Variables:

- no specifier, or `extern`
- `template/specialized` C++14 (no conflicts for `template`, see ODR)
- `inline` `const` / `constexpr` C++17 (no conflicts, see ODR)

### ▪ Functions:

- no specifier (no conflicts with `inline`, see ODR), or `extern`
- `template/specialized` (no conflicts for `template`, see ODR)

Note: `inline`, `constexpr` (which implies `inline` for functions) functions are not accessible by other translation units even with *external linkage*

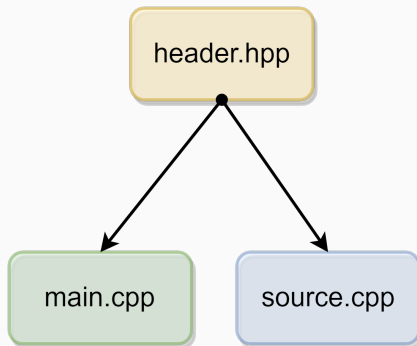
### ▪ Enumerators, Classes and their *static*, *non-static* members

# Dealing with Multiple Translation Units

---

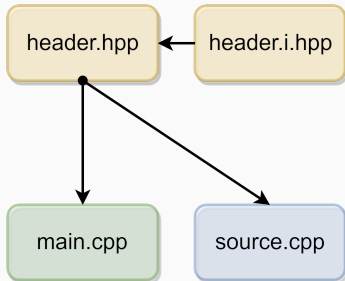
# Code Structure 1

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



## 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

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::y = 2;  
// int A::x = 1; // non-static data member  
// cannot be defined out-of-line
```

header.hpp:

```
struct A {  
    static int      y1;      // zero-init  
    // static int    y2 = 3; // compile error  
    //               must be initialized out-of-class  
    inline static int y3 = 4; // inline initialization  
  
    const int z = 3; // C++11 and later  
    // const int z;   // compile error  
    //               must be initialized  
  
    static const int w1;      // zero-init  
    static const int w2 = 4;  // inline-init  
};
```

source.cpp:

```
#include "header.hpp"  
  
int      A::y1 = 2;  
const int A::w1 = 3;
```

# One Definition Rule (ODR)

---



# 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

# ODR - Point (1), (2)

header.hpp:

```
void f(); // DECLARATION
```

main.cpp:

```
#include "header.hpp"
#include <iostream>
int      a = 1; // external linkage
// int      a = 7; // compiler error, Point (1)

extern int b;

static int c = 2; // internal linkage

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

source.cpp:

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

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

void f() { // DEFINITION
    // std::cout << a; // 'a' is not visible
    std::cout << b; // print 5
    std::cout << c; // print 4
}
```

## Global Variable Issues - ODR Point (2)

header.hpp:

```
#include <iostream>
struct A {
    A() { std::cout << "A()"; }
    ~A() { std::cout << "~A()"; }
};
// A          obj;          // linking error multiple definitions, Point (2)
const A      const_obj{}; // "const/constexpr" implies internal linkage
constexpr float PI = 3.14f;
```

source1.cpp:

```
#include "header.hpp"

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

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

source2.cpp:

```
#include "header.hpp"

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

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

## Common Class Error - ODR Point (2)

header.hpp:

```
struct A {  
    void f() {}; // inline DEFINITION  
    void g();    // DECLARATION  
    void h();    // DECLARATION  
};  
void A::g() {} // DEFINITION
```

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
```

## ODR - Point (3)

**ODR Point (3):** A template, type, or inline functions/variables, can be defined in more than one translation unit

- The linker removes all definitions of an `inline` / `template` entity except one
- All definitions must be identical to avoid undefined behavior due to arbitrary linking order
- `inline` / `template` entities have a *unique memory address* across all translation units
- `inline` / `template` entities have the *same linkage* as the corresponding variables/functions without the specifier

## inline

`inline` specifier allows a function or a variable (in C++17) to be identically defined (not only declared) in multiple translation units

- `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
- `inline` entities cannot be *exported*, namely, used by other translation units even if they have *external linkage* (related warning: `-Wundefined-inline`)
- `inline` doesn't mean that the compiler is forced to perform function *inlining*. It just increases the optimization heuristic threshold

```
void f() {}  
inline void g() {}
```

f() :

- Cannot be defined in a header included in multiple source files
- The linker issues a “*multiple definitions*” error

g() :

- Can be defined in a header and included in multiple source files

## constexpr and inline

`constexpr` functions are implicitly `inline`

`constexpr` variables are not implicitly `inline`. C++17 added `inline` variables

```
void                f1() {} // external linkage
                       // potential multiple definitions error

constexpr void      f2() {} // external linkage, implicitly inline
                       // multiple definitions allowed

constexpr int       x = 3;  // internal linkage
                       // different files allows distinct definitions
                       // -> different addresses, code bloat

inline constexpr int y = 3; // external linkage unique memory address
                       // -> potential undefined behavior

int main() {}
```



header.hpp:

```
inline void f() {} // the function is marked 'inline' (no linking error)
inline int v = 3; // the variable is marked 'inline' (no linking error) (C++17)

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(5); // g<int> generated
}
```

## Alternative organization:

header.hpp:

```
inline void f();    // DECLARATION  
inline int v;       // DECLARATION
```

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

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

main.cpp:

```
#include "header.hpp"  
  
int main() {  
    f();  
    g(3); // g<int> generated  
}
```

header.i.hpp:

```
void f() {}         // DEFINITION  
int v = 3;          // DEFINITION
```

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

source.cpp:

```
#include "header.hpp"  
  
void h() {  
    f();  
    g(5); // g<int> generated  
}
```

# ODR - 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>()
}
```

`f<int>()` , `f<float>()` , `f<char>()` are generated two times (in both translation units)

## Function Template - 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
} // the specialization does not exist
```

source.cpp:

```
#include "header.hpp"

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

// template INSTANTIATION
template void f<int>(int);
template void f<float>(float);
// any explicit instance is also
// fine, e.g. f<int>(3)
```

# Function Template and Specialization

header.hpp:

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

main.cpp:

```
#include "header.hpp"

int main() {
    f<char>(); // use the generic function
    f<int>();  // use the specialization
}
```

source.cpp:

```
#include "header.hpp"

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

# Function Template - extern Keyword

C++11

header.hpp:

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

main.cpp:

```
#include "header.hpp"

extern template void f<int>();
// f<int>() is not generated by the
// compiler in this translation unit

int main() {
    f<int>();
}
```

source.cpp:

```
#include "header.hpp"

void g() {
    f<int>();
}
// or 'template void f<int>();'
```

# ODR Function Template Common Error

header.hpp:

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

// template<>       // linking error
// void f<int>() {} // multiple definitions -> included twice
// full specializations are like standard functions
// it can be solved by adding "inline"
```

main.cpp:

```
#include "header.hpp"

int main() {}
```

source.cpp:

```
#include "header.hpp"

// some code
```



# ODR - Class Template

---

# Class Template - Case 1

header.hpp:

```
template<typename T>
struct A {
    T            x = 3;  // "inline" DEFINITION
    static inline T y = 3; // "inline" DEFINITION (C++17)
    void         f() {}; // "inline" DEFINITION
};
```

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 - Case 2

header.hpp:

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

main.cpp:

```
#include "header.hpp"

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

header.i.hpp:

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

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

source.cpp:

```
#include "header.hpp"

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

## Class Template - Case 3

header.hpp:

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

main.cpp:

```
#include "header.hpp"

int main() {
    A<int>  a1; // ok
// A<char> a2; // linking error
}
// 'f()' is undefined
// while 'x' has an undefined
// value for A<char>
```

source.cpp:

```
#include "header.hpp"

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

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

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

# Class Template - extern Keyword

C++11

header.hpp:

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

source.cpp:

```
#include "header.hpp"

extern template class A<int>;
// A<int> is not generated by the
// compiler in this translation unit
int main() {
    A<int> a;
}
```

source.cpp:

```
#include "header.hpp"

// template INSTANTIATION
template class A<int>;

// or any instantiation of A<int>
```

# ODR Undefined Behavior and Summary

---

# Undefined Behavior - inline Function

main.cpp:

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

void g();

int main() {
    std::cout << f(); // print 3
    std::cout << g(); // print 3!!
}                      // not 5
```

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 could *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` or `anonymous namespace`

# Undefined Behavior - Member Function

header.hpp:

```
#include <iostream>

struct A {
    int f() { return 3; }
};

int g();
```

main.cpp:

```
#include "header.hpp"

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

source.cpp:

```
struct A {
    int f() { return 5; }
};

int g() {
    A a;
    return a.f();
}
```



# Undefined Behavior - Function Template

header.hpp:

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

int g();
```

main.cpp:

```
#include "header.hpp"

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

source.cpp:

```
template<typename T>
int f() {
    return 5;
}

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

# Undefined Behavior

Other ODR violations are even harder (if not impossible) to find, see 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 included all files in a single translation unit

# Declarations and Definitions in Header and Source Files Summary

---

**Declarations:** entities can be *declared* multiple times as long as all declarations are equivalent

Declarations are typically used in

**Header** As interface to expose symbols of one or more source files

**Source File** Directly make available external symbols in the current source file

# Definitions in Headers

- ✓ `inline` variables/functions
- ✓ `template` variables/functions/structs/classes
- ✓ `const/constexpr` variables and `constexpr` functions
  - Should be marked `inline` in C++17
- ⚠ `static` or entities in *anonymous namespace* → binary bloating
- ✗ *non-inline, non-template, non-specialized* variables/functions/structs/classes  
→ multiple definitions

Entities intended to be **exported**:

- ✓ *non-inline* / `extern` functions/variables
- ✓ structs/classes
- ✓ `template` , *in particular* full specialized functions/variables/structs/classes
- ✗ `inline` variables/functions → potential conflicts/undefined behavior
- ✗ `static` / *anonymous namespace* variables/functions → not visible externally

Entities that are **internal only**:

- ✓ Any entity marked `static` or in an *anonymous namespace*
- ✓ `const/constexpr` variables and `constexpr` functions
- ✗ Any other entity → multiple definitions/undefined behavior