



Ben Saks



What Goes Where?

- C++ programs consist of *source files* and *headers*.
 - Source files and headers are usually text files, but need not be.
 - For simplicity, we'll speak as if they're always text files.
- Much of the text in C++ source and header files represents *declarations*.
 - The declarations establish the existence of *entities* such as functions, namespaces, objects, templates, types, and values.
- C++ has no specific rules about which declarations must go into source files and which must go into headers.
- However, it has lots of conventions, such as...



Ben Saks

What Goes Where?

- For a function, we typically:
 - *declare* it in a header, and...
 - *define* it in a corresponding source file.
- However, for a function that's inline, constexpr, or consteval, then:
 - *define* it in a header file, and...
 - that's it.
- Following these conventions helps you write “better” software:
 - software that's more reliable, maintainable, and efficient



Ben Saks

What Goes Where?

- A few general guidelines cover many common “what goes where?” situations.
- Unfortunately, there are lots of special cases that don’t fit the general guidelines.
- Mishandling these special cases can lead to:
 - build (compile or link) errors
 - non-portable or erroneous runtime behaviors
 - speed or space inefficiencies
- Experienced C++ programmers rely on their understanding of compiling and linking to deal with these cases.



Ben Saks



October 24-29

Declarations and Definitions

- A **declaration** is a program statement that says to the compiler:
 - “Here’s a name and some attributes for an entity.”
 - “It might be here, or it might be somewhere else.”
- A **definition** is a declaration that says:
 - “Here’s a name and the complete set of attributes for an entity that’s right here.”
- **All definitions are declarations.**
- **Not all declarations are definitions.**
- This distinction is important for understanding the code organization guidelines.



Declarations and Definitions

- This is a ***function declaration***:

```
int foo(int n);           // non-defining declaration
```

- It lacks a function body, so it isn't a definition.
- It tells the compiler how to generate code that ***calls*** this function.
- However, it doesn't tell the compiler how to generate the code for the ***function itself***.



Declarations and Definitions

- This function declaration is also a ***function definition***:

```
int foo(int n) {           // a definition
    return n + 5;
}
```

- It has a brace-enclosed body.
- It tells the compiler everything it needs to know to actually ***create the function***.



Declarations and Definitions

- In C++, an object declaration (outside a class) is also a ***definition unless it contains an extern specifier and no initializer***:

```
int i;           // definition
extern int j;    // non-defining declaration
extern int k = 42; // definition
```

- An ***object definition allocates storage*** for the object.
- Other ***object declarations*** don't.



Ben Saks

Declarations and Definitions

- In effect, a *non-defining declaration* for a function says:
 - “It exists, but not here.”
- A *definition* says:
 - “It exists, and here it is.”
- There’s a similar distinction for classes and templates.



Ben Saks

Outline

- Declarations and Definitions
- The Translation Process
- The One Definition Rule
- Explicit Template Specialization and Instantiation
- Putting It All Together



Ben Saks

Build Modes

- You can typically build a program as either a:
 - ***release build***: the program as intended for distribution, or
 - ***debug build***: the program augmented with symbolic information to facilitate debugging.
- We will focus on release builds.



Ben Saks

The Translation Process

- In most traditional toolchains, C++ programs are built (“translated”, as the Standard says) in three steps:
 1. Preprocessing
 2. Analysis and Code Generation
 3. Linking

The Translation Process

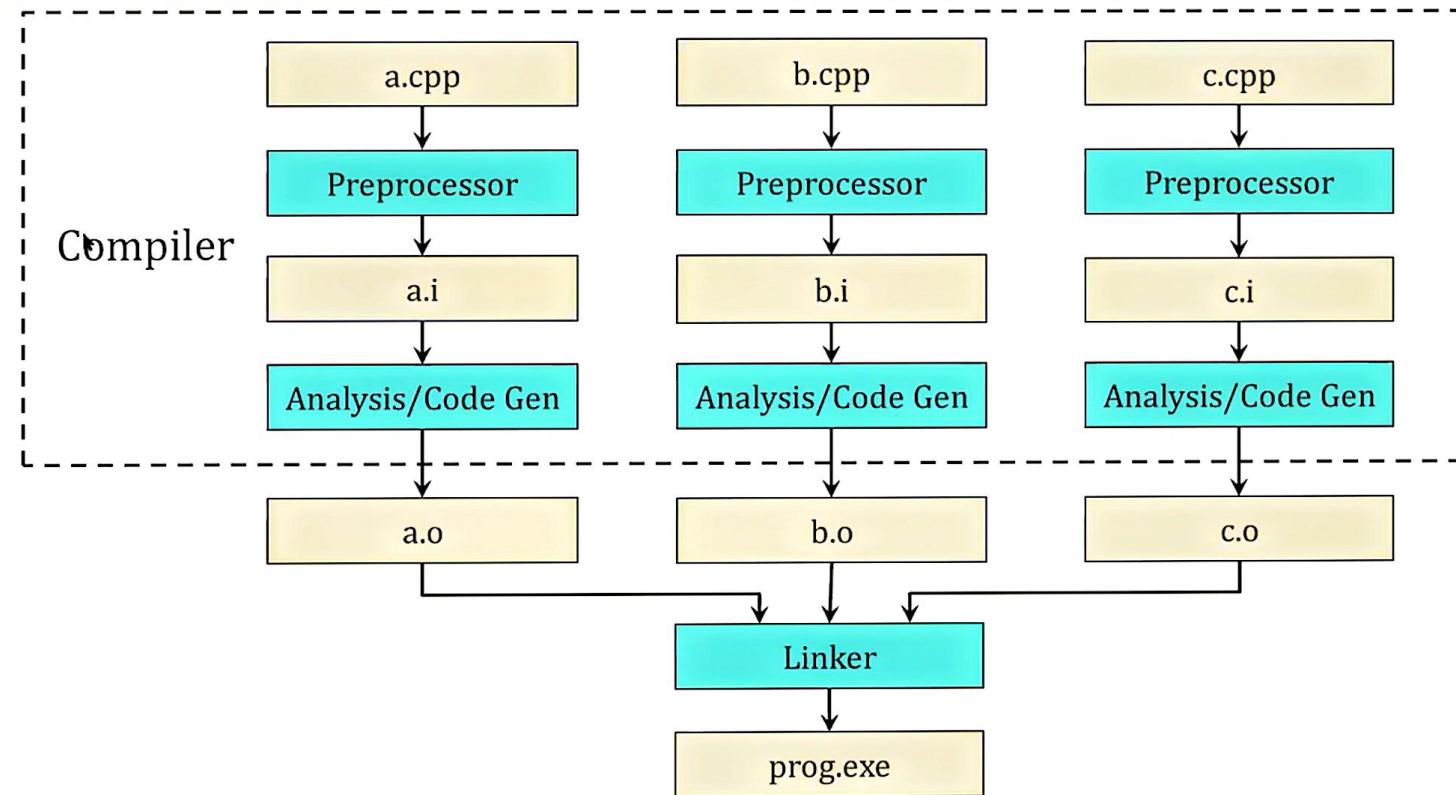
- In most traditional toolchains, C++ programs are built (“translated”, as the Standard says) in three steps:
 1. Preprocessing
 2. Analysis and Code Generation
 3. Linking
- Programmers often use the term “compiling” to describe this whole process.
- Sometimes they separate the process into:
 - “compiling” (preprocessing, analysis, and code generation)
 - “linking”.



Ben Saks



The Translation Process



Copyright © 2021 by Ben Saks and Dan Saks



Ben Saks

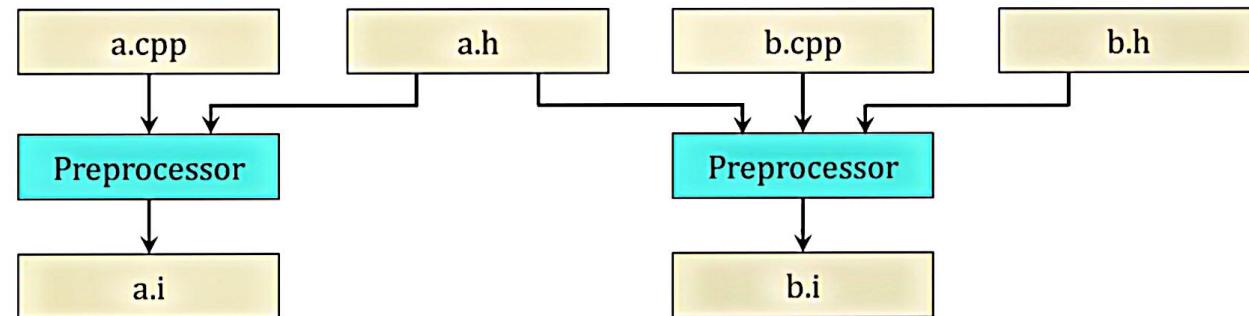
Preprocessing

- Let's take it step-by-step, starting with preprocessing...
- Each source file is preprocessed separately.
- The preprocessor performs text transformations, principally:
 - header inclusion
 - conditional compilation
 - macro substitution
- The preprocessor's output is called a ***translation unit***.
 - It's a concept that the Standard uses for descriptive purposes...



Ben Saks

Preprocessing



- A preprocessor need not generate the translation unit as a file.
 - It rarely does.
- Some compilers have options to actually generate translation units as files, typically with a *.i* or *.ii* extension.



Ben Saks

Analysis and Code Generation

- The compiler analyzes each translation unit.
- The compiled output is called an ***object file*** or ***object module*** (no relation to C++20 Modules).
- When stored in a file system, object files typically use a ***.o***, ***.oo***, or ***.obj*** extension.



Ben Saks

Analysis and Code Generation

- Broadly speaking, an object file contains:
 - ***data***: 0s and 1s that actually become part of the executable program, and
 - ***metadata***: information that the linker uses to combine object files into the executable program.



Analysis and Code Generation

- Broadly speaking, an object file contains:
 - ***data***: 0s and 1s that actually become part of the executable program, and
 - ***metadata***: information that the linker uses to combine object files into the executable program.
- The ***data*** is primarily:
 - ***machine instructions*** (the generated program code, more or less), and
 - ***values*** (for initialized program objects).





Analysis and Code Generation

- In release builds, the ***metadata*** in an object file is mostly symbols and values for:
 - ***function and object names***, and their associated addresses:
 - Some names represent “***defs***” (definitions).
 - Other names represent “***refs***” (non-defining declarations).
 - ***program section names*** and associated contents, such as:
 - “text” or “code” for machine instructions,
 - “literal” for initialized read-only data,
 - “data” for initialized read-write data, and
 - “bss” for uninitialized read-write data.



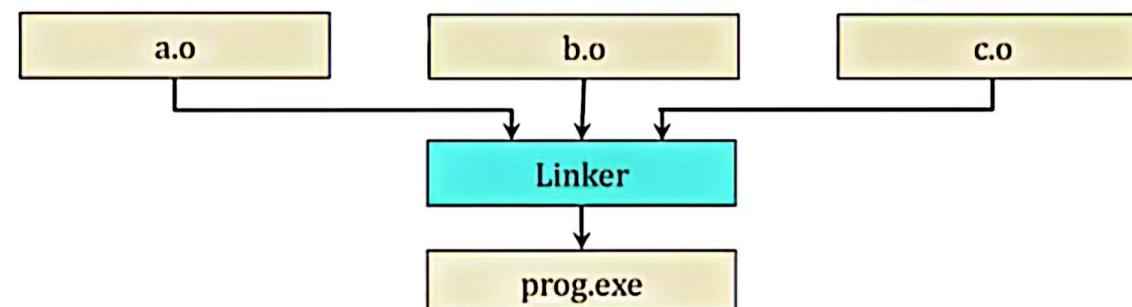
“Only Mostly Executable”

- Actually, the machine instructions in an object file are “only mostly executable”. [Goldman 1987]
- In some cases, an instruction contains an *external reference* to an entity defined in another translation unit.
 - The code generator puts a placeholder for the entity in the object file.
 - The linker makes the instruction executable by replacing the placeholder with the entity’s address.
- In addition, the machine instructions in an object file are often generated to be “relocatable”.



Ben Saks

- The linker resolves the external references by combining:
 - object files, and
 - libraries of previously-compiled object files.



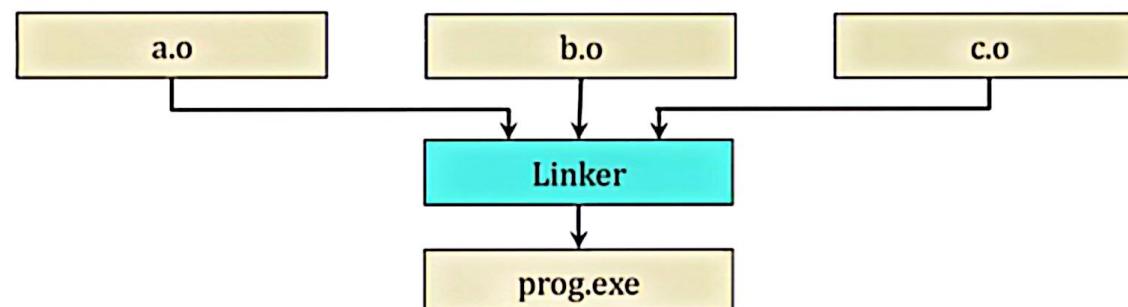
Copyright © 2021 by Ben Saks and Dan Saks



Ben Saks

Linking

- The linker resolves the external references by combining:
 - object files, and
 - libraries of previously-compiled object files.
- It may also handle relocating the values and code.





Ben Saks

The One Definition Rule

- In general, every entity that a program “uses” must also be “defined” in that program.
- The C++ Standard formalizes this notion using:
 - The **“One Definition Rule” (ODR)**, which states that no translation unit may define certain entities more than once.
 - The term **“odr-used”**, which spells out what uses of an entity require a program to define the entity exactly once.

titles/CC turned off

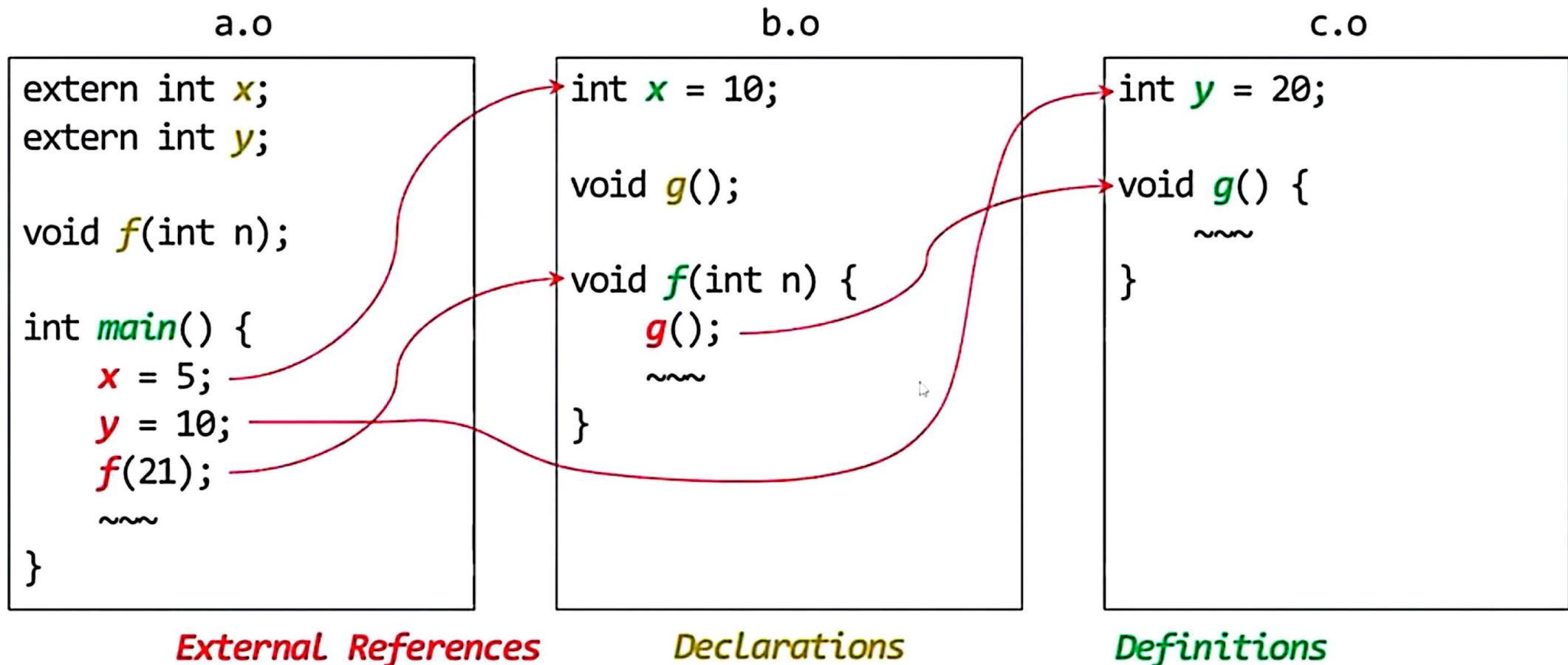


Ben Saks

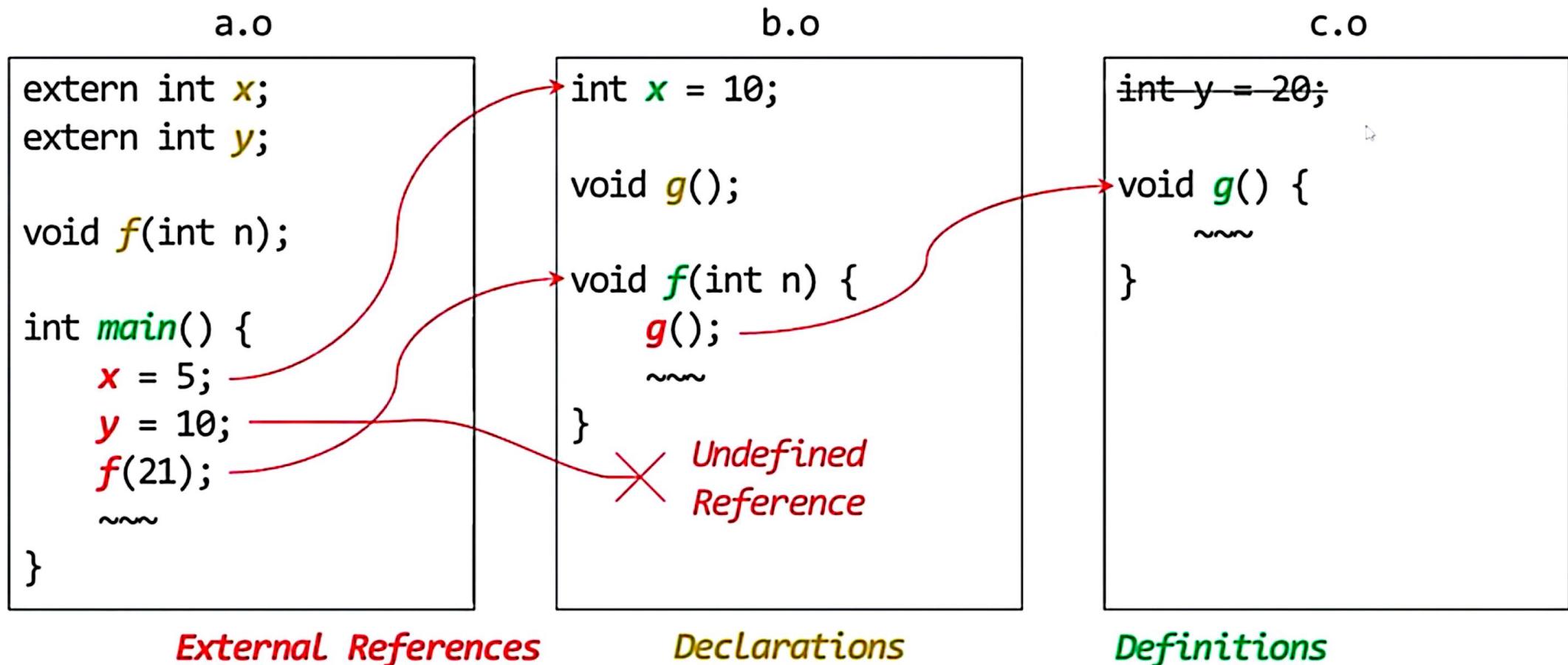
The One Definition Rule

- In general, every entity that a program “uses” must also be “defined” in that program.
- The C++ Standard formalizes this notion using:
 - The **“One Definition Rule” (ODR)**, which states that no translation unit may define certain entities more than once.
 - The term **“odr-used”**, which spells out what uses of an entity require a program to define the entity exactly once.
- The linker matches each external reference with its corresponding definition, as in...

The One Definition Rule



The One Definition Rule





Ben Saks

The One Definition Rule

- If the linker *can't find* a definition that matches an external reference, it will report an “undefined reference” error...



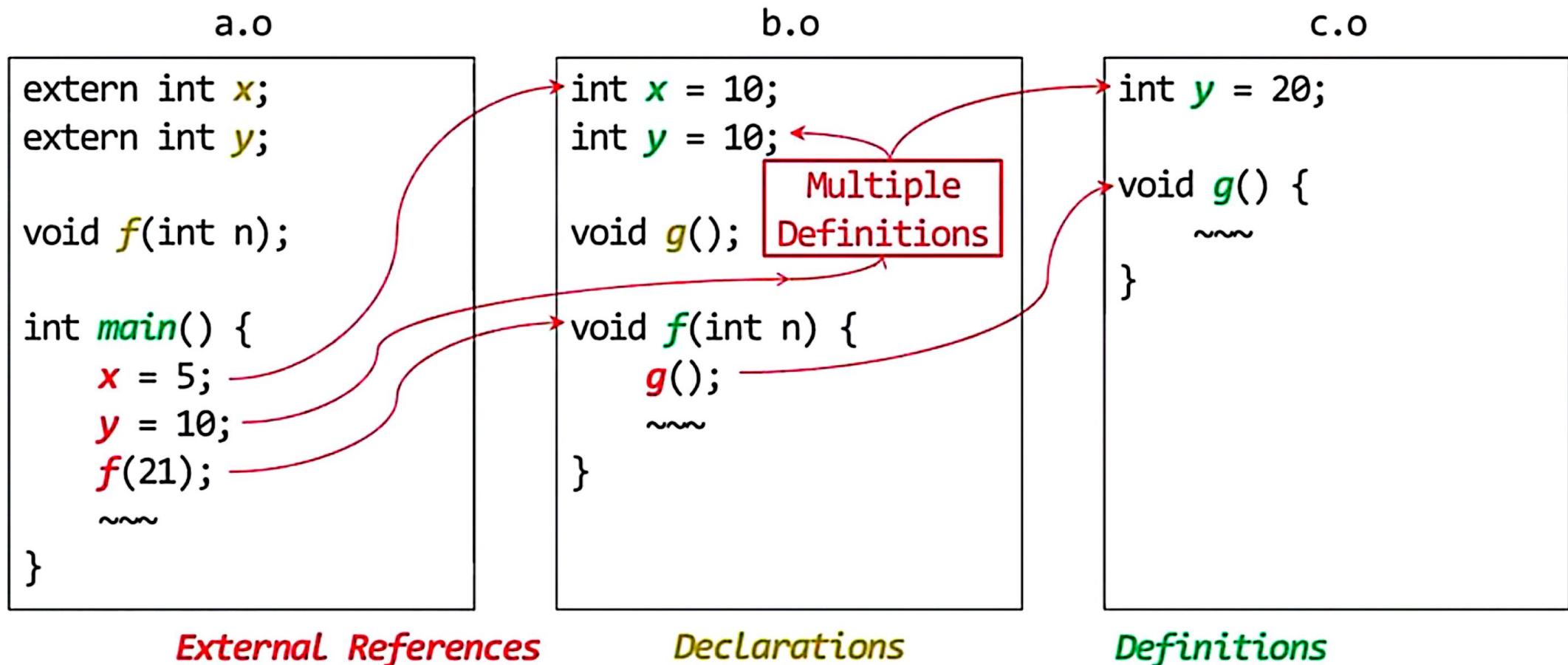
Ben Saks

The One Definition Rule

- If the linker finds ***more than one*** matching definition, it will report a “multiply-defined symbol” error...



The One Definition Rule





The One Definition Rule

- Both examples violate the ODR.
- In a generated executable, every entity will have exactly one definition.
- However, the translation process arrives at that one definition differently for different kinds of entities...



The One Definition Rule

- Some entities must be defined in exactly one translation unit:
 - non-inline variables
 - non-inline, non-template functions
 - `constexpr` and `consteval` functions are implicitly inline
 - Like other inline functions, they aren't included here.
- Many C++ programmers think of this as *the ODR*.
 - In fact, it's only one part of it.



Ben Saks

The One Definition Rule

- Some entities must be defined in exactly one translation unit:
 - non-inline variables
 - non-inline, non-template functions
 - `constexpr` and `consteval` functions are implicitly inline
 - Like other inline functions, they aren't included here.
- Many C++ programmers think of this as *the* ODR.
 - In fact, it's only one part of it.



ChatGPT ▾



🔑 جمع‌بندی ساده

قانون ODR برای این بخش می‌گه:

1. متغیر **global** معمولی → فقط یک تعریف در کل برنامه.
2. تابع معمولی (**غیر-inline** و **غیر-template**) → فقط یک تعریف در کل برنامه.
3. تابع **inline** / **constexpr** / **consteval** → می‌توانی در چند جا تعریف کنی، ولی همه تعاریف باید یکسان باشن.
4. اگر خلاف این کار کنی، لینکر خطای **multiple definition** می‌ده.

حالا می‌خوای من یه مثال واقعی از پروژه چند فایلی (دو فایل **cpp** + یک **hdr**) برات بسازم تا ببینی دقیقاً کجا درست کار می‌کنه و کجا خطای دهد؟



Ask anything



ChatGPT can make mistakes. Check important info. See [Cookie Preferences](#).



قانون ODR برای این بخش می‌گه:

1. متغیر **global** معمولی → فقط یک تعریف در کل برنامه.
2. تابع معمولی (غیر-**inline** و غیر-**template**) → فقط یک تعریف در کل برنامه.
3. تابع **inline** / **constexpr** / **consteval** → می‌توانی در چند جا تعریف کنی، ولی همه تعاریف باید یکسان باشن.
4. اگر خلاف این کار کنی، لینکر خطای **multiple definition** می‌دهد.



Header Entities and the ODR

- A different version of the ODR applies to entities that are usually defined in headers:
 - class and enumeration types
 - inline variables
 - inline functions (including `constexpr` and `consteval` functions)
 - function templates
 - class templates and their members
 - partial template specializations
- For these entities...



Ben Saks

Header Entities and the ODR

- These entities are allowed to have more than one definition if:
 - Each definition appears in a different translation unit.
 - All definitions are identical (i.e., consist of the same sequence of tokens).
- There are additional requirements, but you can usually meet them all by:
 - defining the entity in a single header...
 - ...that can be included by itself (i.e., it doesn't depend on including other headers).
- If the definitions meet these requirements, the program behaves as if there were a single definition for the entire program.



Header Entities and the ODR

- If the definitions don't meet these requirements, *the behavior is undefined*.
 - One way this can happen is if an object file isn't recompiled after a header file that it includes is changed.
- These requirements are hard to verify, so the Standard doesn't require toolchains to do so.
 - Many don't.



Ben Saks

Linking Header Entities

- The Standard doesn't specify *how* the compiler and linker should resolve the symbols for entities that are usually defined in header files.
- Recall that the code generator records definitions and non-defining declarations as metadata in the object files.
- A common solution is to generate slightly different metadata for these entities that lets the linker identify and remove duplicate definitions. [Template Instantiation]

titles/CC turned off



Ben Saks

Linking Header Entities

- The Standard doesn't specify *how* the compiler and linker should resolve the symbols for entities that are usually defined in header files.
- Recall that the code generator records definitions and non-defining declarations as metadata in the object files.
- A common solution is to generate slightly different metadata for these entities that lets the linker identify and remove duplicate definitions. [Template Instantiation]



Linking Header Entities

- For example, Microsoft Visual C++ marks some of these entities with a “pick any” flag.
- This tells the linker that:
 - it may see multiple definitions for an entity, and
 - it can simply choose one and discard the rest.
- These kinds of entities are sometimes referred to as “weak symbols”.
 - I use this term loosely.
 - Many toolchains use the term “weak symbols”, and there are some differences in behavior among them.

- We usually define templates in header files:

a.h

```
// class template definition
template <typename T>
class foo {
public:
    // member declaration
    void f(T t);
};

// member definition
template <typename T>
void foo<T>::f(T t) {
    ~~~
}
```



Ben Saks

all right
explicit
ZI | October 24-29

Explicit Template Specialization

- However, we treat ***explicit template specializations*** like non-inline entities:

a.h

```
// primary class template
template <typename T>
class foo;

// explicit specialization
template <>
class foo<int> {
public:
    // member declaration
    void f(int n);
};
```

a.cpp

```
// member definition for ...
// ... explicit specialization
void foo<int>::f(int n) {
    ~~~
}
```



Ben Saks



October 24-29

Explicit Template Specialization

- Why are explicit specializations different?
- A function template isn't a function.
 - It's a recipe for generating functions.
- A class template isn't a class.
 - It's a recipe for generating classes.
- The same rule applies for other template entities, such as variable templates.
- It holds for both primary templates and partially-specialized templates.

Copyright © 2021 by Ben Saks and Dan Saks



Explicit Template Specialization

- However, explicit template specializations aren't recipes for generating entities.
- For example:
 - an explicitly-specialized class template is a single (non-template) class, and
 - an explicitly-specialized function template is a single (non-template) function.
- Although it's accessed through the template syntax, it's treated like a non-template class in terms of "what goes where?"
- For example...

Explicit Template Specialization

a.h

```
// primary class template
template <typename T>
class foo {
public:
    // member declaration
    void f(T t);
};

// explicitly-specialized
// declaration
template <>
void foo<int>::f(int n);
```

a.cpp

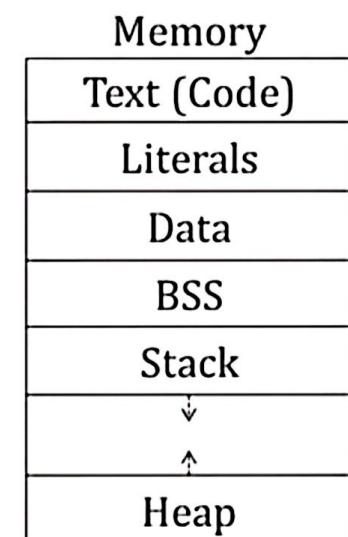
```
// member definition for ...
// ... explicit specialization
template <>
void foo<int>::f(int n) {
    ~~~
}
```



Ben Saks

Memory Maps

- Code generators and linkers usually separate entities into different program sections, such as:
 - “text” or “code” for machine instructions,
 - “literal” for initialized read-only data,
 - “data” for initialized read-write data, and
 - “bss” for uninitialized read-write data.
- The linker is often responsible for arranging these sections in memory.
- This arrangement is the program’s ***memory map***.





Ben Saks

Explicit Template Instantiation

- The linker usually decides where an entity generated from a template is defined.
 - For example, by keeping the definition in one object file and discarding the rest.
- If you need to control where a generated entity is defined, you can use ***explicit template instantiation***.
- Explicit template instantiation is another feature that affects “what goes where?”

where the storage for
those entities



Ben Saks

Controlling Memory Maps

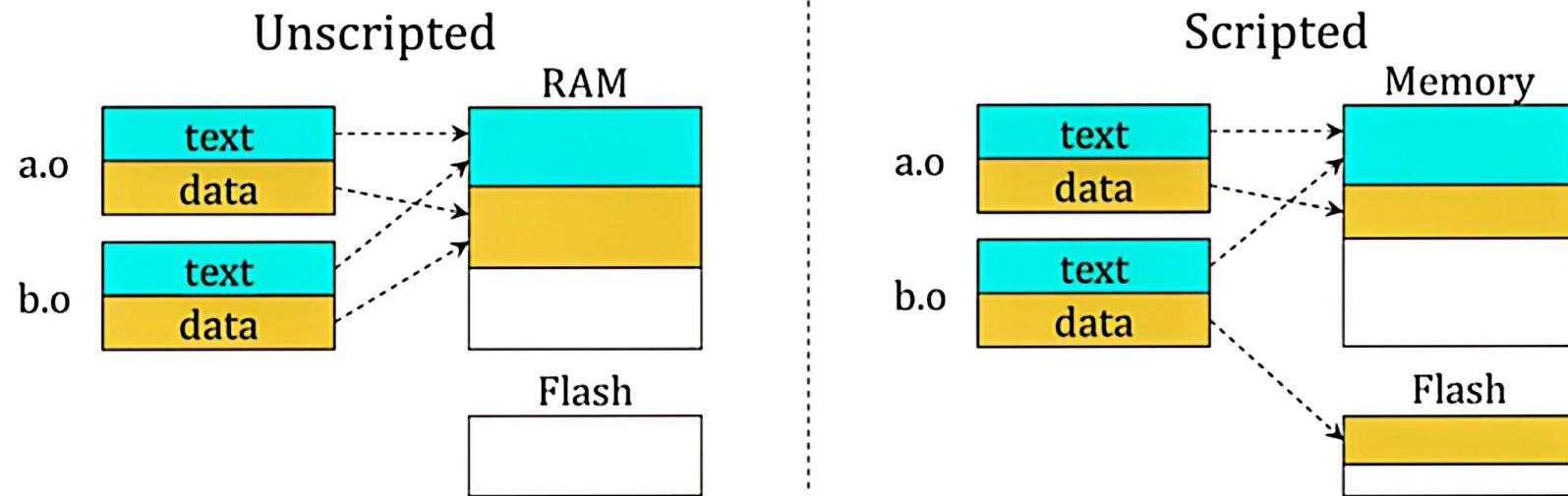
- Many linkers let you control how they generate the memory map through *linker scripts* or other tools.
- These scripts can be very valuable when working on platforms with multiple memory spaces.
- For example, many embedded systems have both RAM and Flash memory.
- Using a linker script, an embedded developer can load some entities into RAM and others into Flash.



Ben Saks

Controlling Memory Maps

- A linker script typically works at the object file level.
- You control the memory map by changing the mapping of one or more sections from an object file.





Controlling Memory Maps

- This level of control works fine for entities defined in source files:
 - non-inline, non-template functions
 - non-inline variables
 - explicitly-specialized templates
- It doesn't work very well for templates instantiated in multiple object files.
 - The linker will eliminate the duplicate definitions from all but one of the object files.
 - In general, you don't know which object file contains the definition that the linker will keep.
- You can use explicit template instantiation to choose which object file will contain a template definition...



Explicit Template Instantiation

a.h

```
// function template definition
template <typename T>
void swap(T &x, T &y) {
    T temp(std::move(x));
    x = std::move(y);
    y = std::move(temp);
}

// explicit function
// instantiation declaration
extern template
void swap<int>(int &x, int &y);
```

a.cpp

```
// explicit function
// instantiation definition
template
void swap<int>(int &x, int &y);
```



Ben Saks



Explicit Template Instantiation

- An explicitly-instantiated template is a single entity.
 - It's not a recipe for generating entities.
- As such, you're expected to treat it more like a non-template entity.
 - You *declare* it in a header (with the keyword `extern`).
 - You *define* it in a source file, ***but don't provide a body***.
 - The body comes from the earlier template definition.
- The compiler generates a single definition for the entity in that source file's translation unit.
 - It generates this definition regardless of whether it's actually used.
 - Other translation units see only the non-defining declaration for the entity.

Explicit Template Instantiation

- You can also ***explicitly instantiate*** individual member functions and entire classes.

a.h

```
// class template
template <typename T>
class foo {
public:
    void f(T t);
};

// explicitly class
// instantiation declaration
extern template
class foo<int>;
```

a.cpp

```
// explicit class
// instantiation definition
template
class foo<int>;
// no class body
```



Ben Saks

Explicit Template Instantiation

- Explicitly instantiating an entire class generates a definition for every member, regardless of whether that member is actually used.



Ben Saks

Putting It All Together

- Here are some examples of how these principles apply in practice.
- First up, a *non-inline function*...

Non-Inline Function

a.cpp

```
#include "entities.h"

int add(int x, int y) {
    return x + y;
}

void foo(int m) {
    cout << add(m, 10);
}
```

entities.h

```
int add(int x, int y);
```

Declaration: A non-inline function declaration in a header file.

b.cpp

```
#include "entities.h"

void bar() {
    int n;
    cin >> n;
    cout << add(n, 5);
}
```

Declarations

Definitions

Internal References

External References

Non-Inline Function

a.cpp

```
#include "entities.h"

int add(int x, int y) {
    return x + y;
}

void foo(int m) {
    cout << add(m, 10);
}
```

entities.h

```
int add(int x, int y);
```

Definition: A non-
inline function
definition in a
source file.

b.cpp

```
#include "entities.h"

void bar() {
    int n;
    cin >> n;
    cout << add(n, 5);
}
```

Declarations

Definitions

Internal References

External References

Non-Inline Function

a.cpp

```
#include "entities.h"

int add(int x, int y) {
    return x + y;
}

void foo(int m) {
    cout << add(m, 10);
}
```

entities.h

```
int add(int x, int y);
```

Internal Reference:
Refers to a definition in
the same translation
unit. Resolved by the
code generator, not the
linker.

b.cpp

```
#include "entities.h"

void bar() {
    int n;
    cin >> n;
    cout << add(n, 5);
}
```

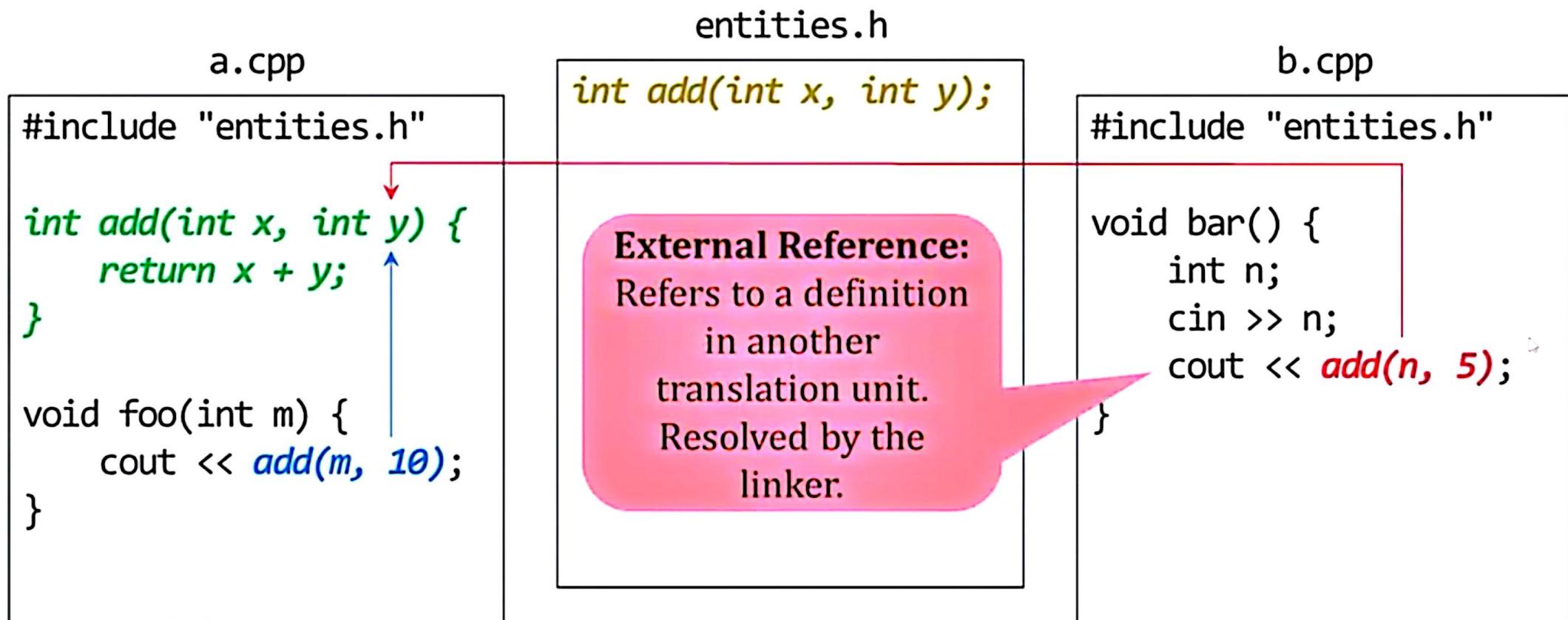
Declarations

Definitions

Internal References

External References

Non-Inline Function



Declarations

Definitions

Internal References

External References



Ben Saks

Putting It All Together

- Next, an *inline function that isn't actually inlined*.
 - If a function call *is* inlined, there's no need to refer to code defined elsewhere – the code is right there, inline.
- In this case, the function that's declared inline is recursive.
 - Recursive functions are hard to inline if the arguments aren't known at compile time.
 - Here, the arguments are run-time values.

a.cpp

```
#include "entities.h"

void foo(int m) {
    cout << tri(m);
}
```

entities.h

```
inline
int tri(int x) {
    return x > 0
        ? x + tri(x - 1)
        : 0;
}
```

Definition: An inline function definition in a header file.

b.cpp

```
#include "entities.h"

void bar() {
    int n;
    cin >> n;
    cout << tri(n);
}
```

Declarations

Definitions

Internal References

External References

a.cpp

```
#include "entities.h"

int tri(int x) {
    return x > 0
    ? x + tri(x - 1)
    : 0;
}

void foo(int m) {
    cout << tri(m);
}
```

entities.h

```
inline
int tri(int x) {
    return x > 0
    ? x + tri(x - 1)
    : 0;
```

b.cpp

```
#include "entities.h"

int tri(int x) {
    return x > 0
    ? x + tri(x - 1)
    : 0;
}

void bar() {
    int n;
    cin >> n;
    cout << tri(n);
}
```

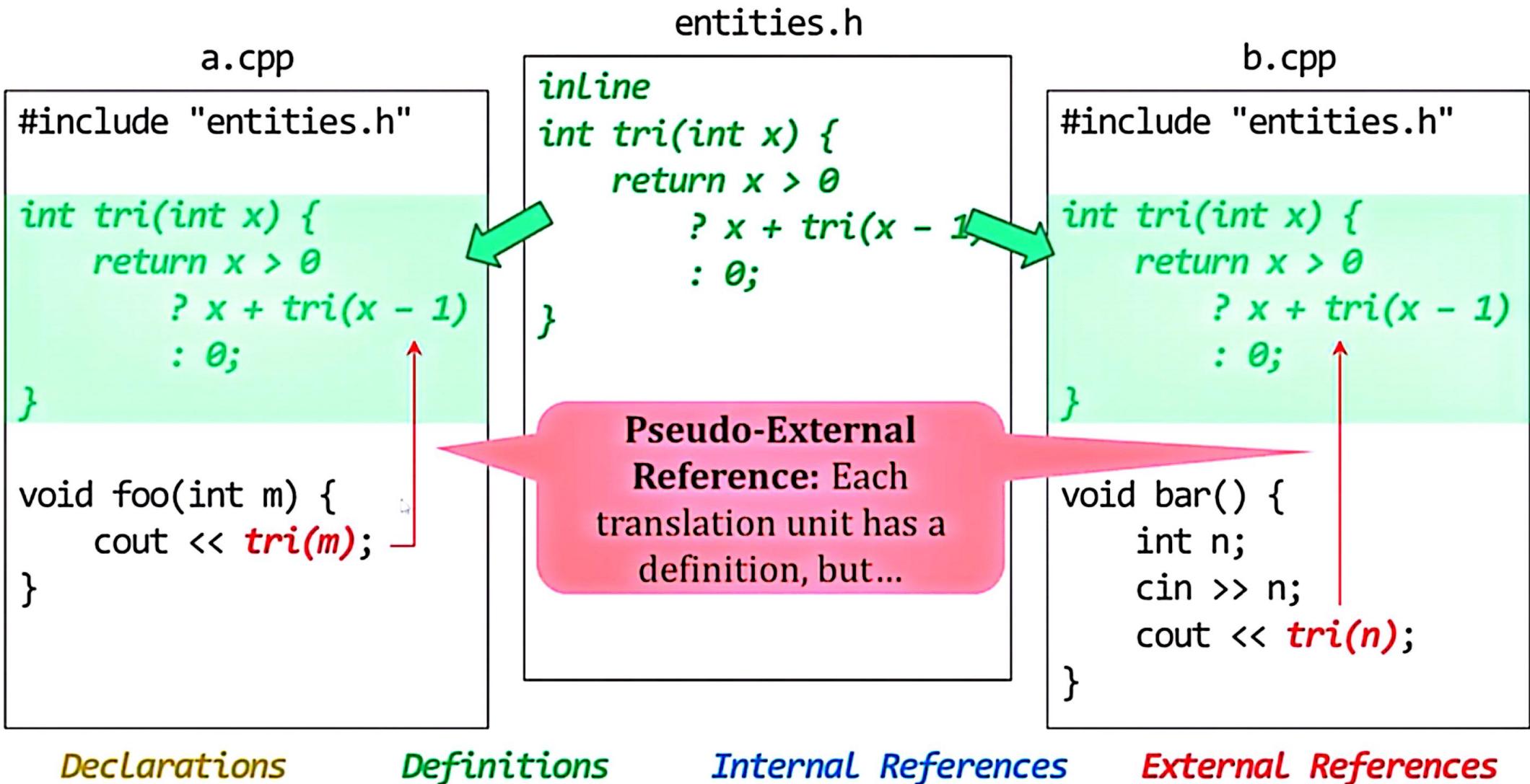
Each translation unit gets its own copy of the definition.

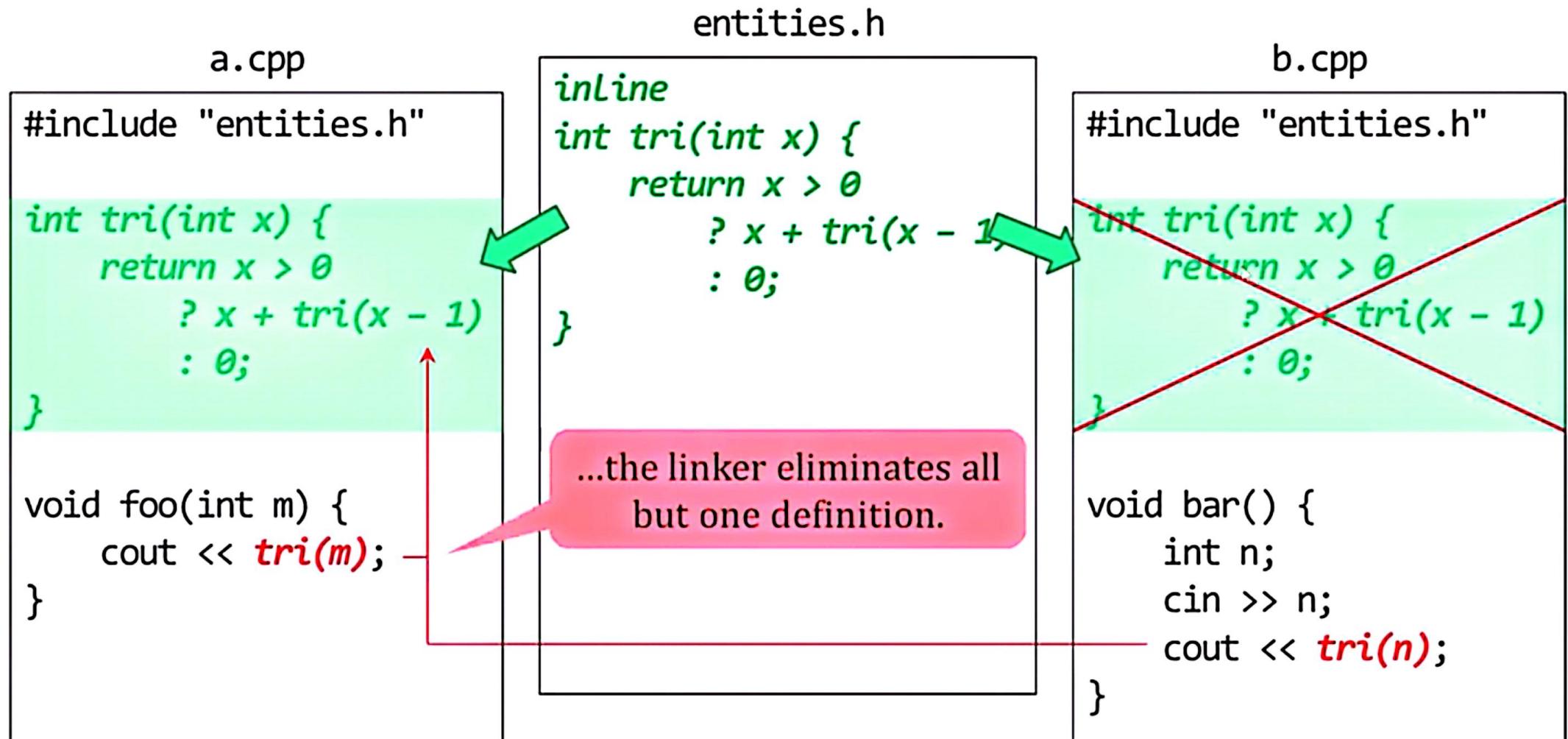
Declarations

Definitions

Internal References

External References





Declarations

Definitions

Internal References

External References