Concepts of C++ Programming

Lecture 3: Declarations/Definitions, Preprocessor, Linker

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2024/25

On “Internet”

Search engines/AI are **not** your friend
when it comes to C++!

Use high-quality sources.
Use the C++ reference.
Read the script of this lecture.

Compiler: Overview (1)


```
clang++ -E -o hello.ii hello.cpp
clang++ -o hello hello.ii
```

- Preprocessor transforms source code before actual compilation
- `clang++ -E` – stop after preprocessing

Preprocessor[33]

- Applies textual transformation before compilation
  - E.g., to conditionally exclude certain code paths from compilation
  - Preprocessor has no knowledge about "real" C++ language semantics
- Handles preprocessor directives: lines that begin with #
- Outputs program without directives

Use **carefully**, avoid if possible!

[33] https://en.cppreference.com/w/cpp/preprocessor

Preprocessor: `#include`[34]

- `#include "path/to/file"` – copy content from file at current position
- Literal textual inclusion ("copy-paste")

```
//--- magicNumber.inc
42

//--- magicNumber.cpp
int magic =
#include "magicNumber.inc"
  ;
```

- After preprocessing

```
// clang++ -E magicNumber.cpp
int magic =
42
  ;
```

[34] https://en.cppreference.com/w/cpp/preprocessor/include

Preprocessor: Include Path

- `#include "file"`
  - Search order: current directory, include path, system path
  - Convention: use for files in current project
- `#include <file>` – search include path, then system path
  - Search order: include path, system path
  - Convention: use for libraries and system includes

- Compiler flag: `-I<directory>` – add directory to include path
- CMake: `target_include_directories(target PUBLIC src/)`

- Typical: add root of project source to include path
  $\Rightarrow$ All files can be included by "absolute path"

Preprocessor: #define[36]

- `#define SOMENAME` – define a macro with the given name
- Can have an optional textual replacement
- `#undef` – undefined previously defined macro

```
#define EMPTY
#define return never
#define ANSWER 42
#define FUNC_DECL int getAnswer()
#undef return
FUNC_DECL { EMPTY return ANSWER; EMPTY }
// Preprocessed to:
// int getAnswer() { return 42; }
```

Don't use the preprocessor like this, this is primarily for illustration.[35]

[35] NB: Re-defining keywords is undefined behavior if the standard library is included.

[36] https://en.cppreference.com/w/cpp/preprocessor/replace

Preprocessor: `#define` – Example

Quiz: What does the function `f` return?

```
#define ONE 1
#define TWO (ONE + ONE)
#define FOUR TWO+TWO
#define SIXTEEN FOUR*FOUR
int f() { return (SIXTEEN + FOUR) * TWO + TWO; }
```

A. (compile error) B. 2 C. 26 D. 42

Don't use the preprocessor like this, this is primarily for illustration.

Preprocessor: Pre-defined Macros

- Some macros are pre-defined by the compiler
- Few are standardized, others vary between compilers
- Typically begin with double-underscore

Examples:

- `__cplusplus` – used C++ standard, e.g. 202302L
- `__FILE__` – name of the current file
- `__x86_64__` – defined if compiling for x86-64

- Compiler flag `-D<macroname>=<expansion>` – define a macro with the (optional) expansion

Preprocessor: Conditions[37] (1)

- `#if <expr>/#elif <expr>/#ifdef <macro>/#ifndef <macro>/#else/#endif` – conditionally compile part of code
  - Use cases: architecture-dependent code, code only for debug builds
- Expressions can use `defined(MACRO)` to test whether a macro is defined
- Preprocessor expressions *only* operate on macros!

```
#if defined(__x86_64__)
// x86-64-specific code goes here
#elif defined(__aarch64__)
// aarch64-specific code goes here
#else
// architecture-independent code goes here
#endif
```

[37] https://en.cppreference.com/w/cpp/preprocessor/conditional

Preprocessor: Conditions (2)

- `#error` – cause compilation to fail with given message

```
#if defined(__x86_64__) || defined(__aarch64__)
// x86-64 and aarch64 code goes here
#else
#error Unsupported architecture!
#endif

#if 0 // #if 0 can be used for comments, can be nested (unlike /* */)
void commentedOut() {}
#if 0
void moreCommentedCode() {}
#endif
#endif
```

Preprocessor: Conditions (3)

Quiz: What does the function `f` return?

```
int j = 5;
#if j * j == 25
int f() { return j * j; }
#else
int f() { return 20; }
#endif
```

A. (compile error) B. depends on `j` C. 20 D. 25

Don't use the preprocessor like this, this is primarily for illustration.

Preprocessor: Function-Like Macros

- Macros can have arguments, so they look like functions
- Again, purely textual replacement, no semantics
  - Wrap all parameters in parentheses to avoid precedence issues

```
#define MIN(a,b) ((a)<(b)?(a):(b))

int min3(int a, int b, int c) {
// Preprocessed to:
// return ((((a)<(b)?(a):(b)))<(c)?(((a)<(b)?(a):(b))):(c));
  return MIN(MIN(a, b), c);
}
```

Don't use the preprocessor like this, this is primarily for illustration.

Preprocessor: Function-Like Macros (Quiz)

Quiz: Why is this macro problematic?

```
#define MIN(a,b) ((a) < (b) ? (a) : (b))
```

A. One parameter is evaluated multiple times.
B. The unnecessary parenthesis make the code difficult to read.
C. The macro doesn't compute the minimum on unsigned integers.

- Don't do this — we'll cover modern replacements later

Preprocessor: Recommendations

Avoid if possible!

- Many pitfalls, code harder to read/analyze/debug
- Many use-cases have modern, safer C++ replacements (see later)

- No rule without exceptions...
- Some older code bases use preprocessor heavily
  - Primary reason we cover it so extensively here

Runtime Checks for Debugging: assert

- `assert(expr)` – abort program if assertion is false
- Use to check invariants

- When `NDEBUG` is defined, assert generates no code
- CMake automatically defines `NDEBUG` in release builds

```
#include <cassert>
double div(double a, int b) {
  assert(b != 0 && "divisor must be non-zero");
  return a / b;
}
```

assert – Implementation

- `assert(expr)` is a preprocessor macro

⇒ Expression gets *removed from source code* when `NDEBUG` is defined!

```
//--- /usr/include/assert.h (glibc) (excerpt) (code simplified for slide)

/* void assert (int expression);

   If NDEBUG is defined, do nothing.
   If not, and EXPRESSION is zero, print an error message and abort. */
#ifdef NDEBUG
# define assert(expr) ((void)(0))
#else
# define assert(expr) ((expr) ? (void)(0) : __assert_fail(#expr, /*...*/))
#endif
```

Multiple Source Files

- C++ source files know nothing about each other
  - Other than `#include`, which is just copy-paste

How do they know what functions other files define?

$\rightsquigarrow$ Explicit declarations

Declarations[38]

- Declarations introduce names
- Names must be declared before they can be referenced

- Variables: `int x;`
- Functions: `void fn();`
- Namespace: `namespace A { }`
- Using: `using A::x;`
- Class: `class C;`
- ...

[38] https://en.cppreference.com/w/cpp/language/declarations

Definition[40]

- A declared name can be used, but: most uses require[39] a *definition*
  - Reading/writing value or taking address of an object
  - Calling or taking address of function
- Most declarations are also definitions, with some exceptions
  - Function declaration without body
  - Variable declarations with `extern` and no initializer

[39] Formally called *odr-use*

[40] https://en.cppreference.com/w/cpp/language/definition

Function Declarations: Example

- Forward declaration necessary for cyclic dependencies

```
void bar(int n); // declaration, no definition

void foo(int n) { // declare + define foo
  std::println("foo");
  if (n > 0)
    bar(n - 1); // OK, bar declared above
}

void bar(int n) { // re-declare + define bar
  std::println("bar");
  if (n > 0)
    foo(n - 1); // OK, foo declared above
}
```

Variable Declarations: Example

```
extern int global; // declaration
int otherGlobal; // declararation + definition, zero-initialized

int readGlobal() {
  return global;
}

int global = 5; // re-declaration + definition
```

- The first declaration is rather useless, could move definition there

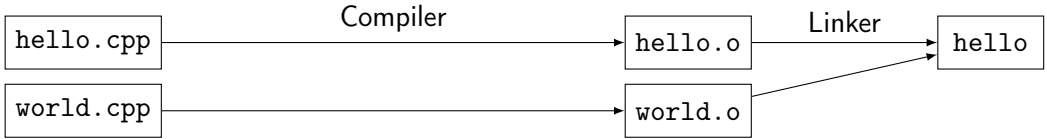
cv-Qualifier: `const` and `volatile`[41]

- Part of the type, can appear in variable declarations
- `const` – object cannot be modified
- `volatile` – object access has a side-effect
  - E.g., direct hardware access, communication with signal handlers

```
void function() {
  int a = 4;
  const int b = a;
  a = 0; // OK
  b = 10; // ERROR: assignment of read-only variable
  volatile int v = 5; // will not be optimized out
}
```

[41] https://en.cppreference.com/w/cpp/language/cv

Compiler: Overview (2) – Multiple Files




```
clang++ -c -o hello.o hello.cpp
clang++ -c -o world.o world.cpp
clang++ -o hello hello.o world.o
```

- Compiler generates object file with machine code
  - One compile invocation compiles one *translation unit*
  - May contain references to not-yet-defined functions/globals
- Linker combines object files into executable
  - Resolve all undefined references

Multiple Files

```
//--- foo.cpp
int globalVar = 7;
int foo() { return 6; }

//--- main.cpp
#include <print>
extern int globalVar;
int foo();
int main() {
  std::println("{}", globalVar * foo());
  return 0;
}

$ clang++ -std=c++23 -c -o foo.o foo.cpp
$ clang++ -std=c++23 -c -o main.o main.cpp
$ clang++ -o main main.o foo.o
$ ./main
42
```

Multiple Files: Undefined References

```
//--- foo.cpp
int bar();
int foo() { return 2 * bar(); }

//--- main.cpp
extern int undefinedGlobal;
int main() {
  return undefinedGlobal;
}
```

```
$ clang++ -std=c++23 -c -o foo.o foo.cpp
$ clang++ -std=c++23 -c -o main.o main.cpp
$ clang++ -o main main.o foo.o
/usr/bin/ld: main.o: in function ‘main’:
main.cpp:(.text+0x8): undefined reference to ‘undefinedGlobal’
/usr/bin/ld: foo.o: in function ‘foo()’:
foo.cpp:(.text+0x8): undefined reference to ‘bar()’
clang++: error: linker command failed with exit code 1 (use -v to see invocation)
```

One Definition Rule (ODR)[42]

- At most one definition of a name allowed *within one translation unit*
- Exactly one definition of every used function or variable must appear *within the entire program*

- (for more cases, exceptions, subtleties: see reference documentation)

NB: Some ODR violations make programs "ill-formed, no diagnostic required" — only the linker can diagnose such violations

[42] https://en.cppreference.com/w/cpp/language/definition#One_Definition_Rule

One Definition Rule: Examples (Multiple Definitions)

```
int i = 0; // OK: declaration + definition
int i = 1; // ERROR: redefinition
```

```
//--- a.cpp
int foo() { return 1; }

//--- b.cpp
int foo() { return 2; }
```

```
clang++ -std=c++23 -c -o a.o a.cpp
clang++ -std=c++23 -c -o b.o b.cpp
clang++ a.o b.o
/usr/bin/ld: foo.o: in function ‘foo()’:
b.cpp:(.text+0x0): multiple definition of ‘foo()’; a.o:a.cpp:(.text+0x0):
    first defined here
```

Header and Implementation Files

- Duplicating declarations into every file technically possible
- But: not maintainable, error-prone

Idea: split into *implementation* (`.cpp`) and *header* (`.h`) file:

- Header file: just declarations that should be usable in other files
  - Conceptually: "API" of logical unit
  - Also should include documentation
- Implementation file: definitions for names declared in header
  - Conceptually: "implementation" of the API

Use *preprocessor* to copy-paste declaration

Header and Implementation Files: Example

```
//--- sayhello.h
#include <cstdint>
/// Print "Hello!" to standard output.
void sayHello(std::uint64_t number);

//--- sayhello.cpp
#include "sayhello.h"
#include <cstdint>
#include <print>
void sayHello(std::uint64_t number) { std::println("Hello␣{}!", number); }

//--- main.cpp
#include "sayhello.h"
int main() { sayHello(1); return 0; }
```

Header Guards

- Header files include other headers they require
  - E.g., for defined data types (see later)
- Transitive includes: same header might be included multiple times!
- But: can cause problems due to redefinitions

$\rightsquigarrow$ Wrap entire header with `#ifdef` and unique identifier

```
//--- sayhello.h
#ifndef CPPLECTURE_HELLO_H
#define CPPLECTURE_HELLO_H

/// Print "Hello!" to standard output.
void sayHello();

#endif // CPPLECTURE_HELLO_H
```

- Non-standard alternative

```
//--- sayhello.h
#pragma once

/// Print "Hello!" to standard output.
void sayHello();
```

Header Files and `#include`

- Include (exactly) used header files at the beginning
  - In both, header and implementation file
  - Be careful about transitive includes
- Typically grouped by: (Example)
  1. Accompanying header file
  2. Project includes
  3. Library includes
  4. System includes

- Only include header files
- **Never** include implementation files!

Typical Project Layout

```
+-- CMakeLists.txt
+-- src/
    +-- Module.cpp
    +-- Module.hpp
    +-- Printer.cpp
    +-- Printer.hpp
    +-- log/
        +-- Log.cpp
        +-- Log.hpp
        +-- LogEntry.cpp
        +-- LogEntry.hpp
    +-- main.cpp
```

- Source files and header files next to each other
- Entry points (`main()`) often separate
  - Typically small files $\rightsquigarrow$ easier testing
- CMakeLists defines
  - `add_executable` with all sources (`*.cpp`)
  - `target_include_directories(... src)`
- Alternative layouts exist

Tracking Changes in Source Code

```
//--- a.hpp
extern int globalA;
//--- a.cpp
#include "a.hpp"
int globalA = 10;
//--- square.hpp
#include "a.hpp"
int square(int n = globalA);
//--- square.cpp
#include "square.hpp"
void square(int n) {
  return n * n;
}
//--- main.cpp
#include "square.hpp"
// ...
```

Quiz: `a.hpp` changed. Which files to re-compile?

A. `a.hpp`
B. `a.cpp`
C. `a.cpp`, `square.cpp`
D. `a.cpp`, `square.cpp`, `main.cpp`
E. `a.hpp`, `a.cpp`, `square.cpp`, `main.cpp`

Tracking Changes in Source Code

- Incremental compilation: only recompile files that actually changed
  - Substantially reduces build time during development
- Detecting files that need recompilation is non-trivial
  - Transitive dependencies of header files
- Build systems like CMake use compiler to output list of used includes
  - If any of the files changed, the source file needs recompilation

Linkage

- Linkage of declaration: visibility across different translation units
- No linkage: name only usable in their scope
  - E.g., local variables
- Internal linkage: can only be referenced from same translation unit
  - Global functions/variables with `static`
  - const-qualified global variables without `extern`
  - Declarations in namespace without name (“anonymous namespace”)
- External linkage: can be referenced from other translation units
  - Global functions/variables (unless `static`)

Declaration Specifiers

- Variable/function declarations allow for additional specifier
- Controls storage duration *and* linkage

Specifier	Global Func/Variable	Local Variable
none	static + external	automatic + none
`static`	static + internal	static + none
`extern`	static + external	static + external
`thread_local`	thread + ext/int	thread + none

- And there's `inline` (it deserves it's own slide)

Declaration Specifiers – Example

```
//--- a.cpp
static int foo = 1;
int bar = 2;
static int add(int x, int y) { return x + y; }
int countMe() {
  static int counter = 0; // static storage duration, no linkage
  return counter++
}

//--- b.cpp
static int foo = 1; // OK
int bar; // ERROR: ODR violation

// OK: a.cpp's and b.cpp's add have internal linkage
static int add(int x, int y) { return x + y; }
```

Internal Linkage: Anonymous Namespaces

- Option A: Use `static` (only works for variables and functions)

```
static int foo = 1; // internal linkage
static int bar() { // internal linkage
    return foo;
}
```

- Option B: Use *anonymous namespaces* (preferred)

```
namespace {
int foo = 1; // internal linkage
int bar() { // internal linkage
    return foo;
}
} // end anonymous namespace
```

inline Specifier[43]

- `inline` – permit multiple definitions in different translation units
  - No direct relation to the inlining optimization!

```
//--- sum.h
#ifndef SUM_H
#define SUM_H

inline int sum(int a, int b) {
  return a + b;
}

#endif // SUM_H
```

```
//--- a.cpp
#include "sum.h"
// Now has definition of sum
// ...

//--- b.cpp
#include "sum.h"
// Now has definition of sum
// ...
```

- Linker keeps only one definition

[43] https://en.cppreference.com/w/cpp/language/inline

Declarations/Definitions, Preprocessor, Linker – Summary

- Preprocessor transforms source code before actual compilation
  - Use (almost) exclusively for header guards and header includes
- Use `assert()` for invariants, but be aware that it is a macro
- Declarations introduce names, but not necessarily define them
- Exactly one definition of used func/var required in program
- For multiple files, separate header and implementation files
- There must be exactly one definition of every used name
- Exceptions: internal linkage and inline functions

Declarations/Definitions, Preprocessor, Linker – Questions

- Why is the use of function-like macros problematic?
- What are state modifications inside `assert()` problematic?
- What is the difference between a declaration and a definition?
- How to declare functions and global variables?
- Why is the header guard important?
- Why is including C++ implementation files (`.cpp`) a bad idea?
- What does the `static` specifier do on local variables?
- What is the effect of an unnamed namespace?