(8) C++ Abstractions

Nico Ludwig (@ersatzteilchen)

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

TOC

- Sources:
  - Bjarne Stroustrup, The C++ Programming Language
  - John Lakos, Large Scale C++ Software Design

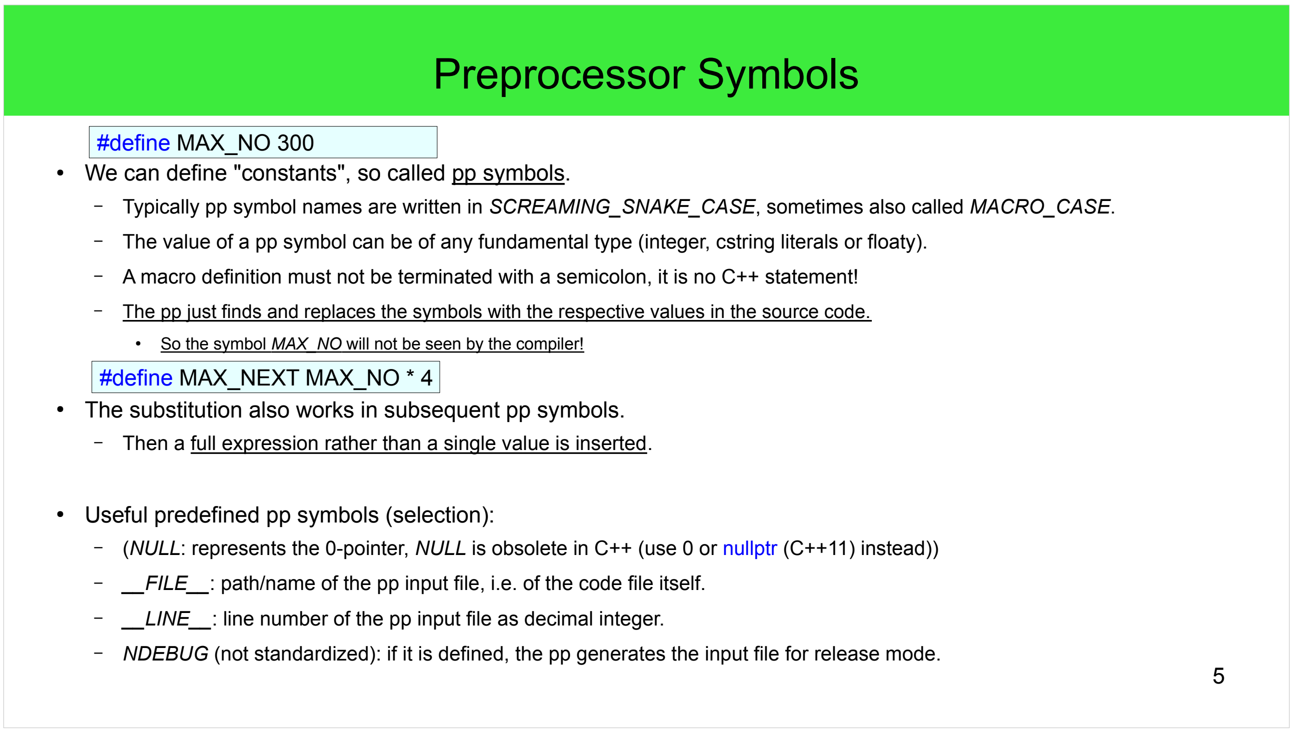
The Preprocessing Phase

- C/C++ programs are often spread into multiple files, combined with #includes.
  - This is downright required for large projects.
  - But it is also a good practice for small projects to keep a clear view.

- The resolution of #includes is a build procedure-phase: the preprocessing-phase.

- The C/C++ preprocessor (pp) has many responsibilities:
  - It resolves all #includes per c-/cpp-file (summarized as "c-file") recursively to a single file.
  - It executes conditional source code insertion.
  - It substitutes all pp symbols and expands all macros.
  - => The resulting files are called translation units (tus).

- The pp-phase can be controlled with pp-directives written in the C/C++ code.
  - Pp-directives start with a # (pound): #include ("pound-include"), #define etc.



Preprocessor Symbols

```
#define MAX_NO 300
```

- We can define "constants", so called pp symbols.
  - Typically pp symbol names are written in *SCREAMING_SNAKE_CASE*, sometimes also called *MACRO_CASE*.
  - The value of a pp symbol can be of any fundamental type (integer, cstring literals or floaty).
  - A macro definition must not be terminated with a semicolon, it is no C++ statement!
  - The pp just finds and replaces the symbols with the respective values in the source code.
    - So the symbol *MAX_NO* will not be seen by the compiler!

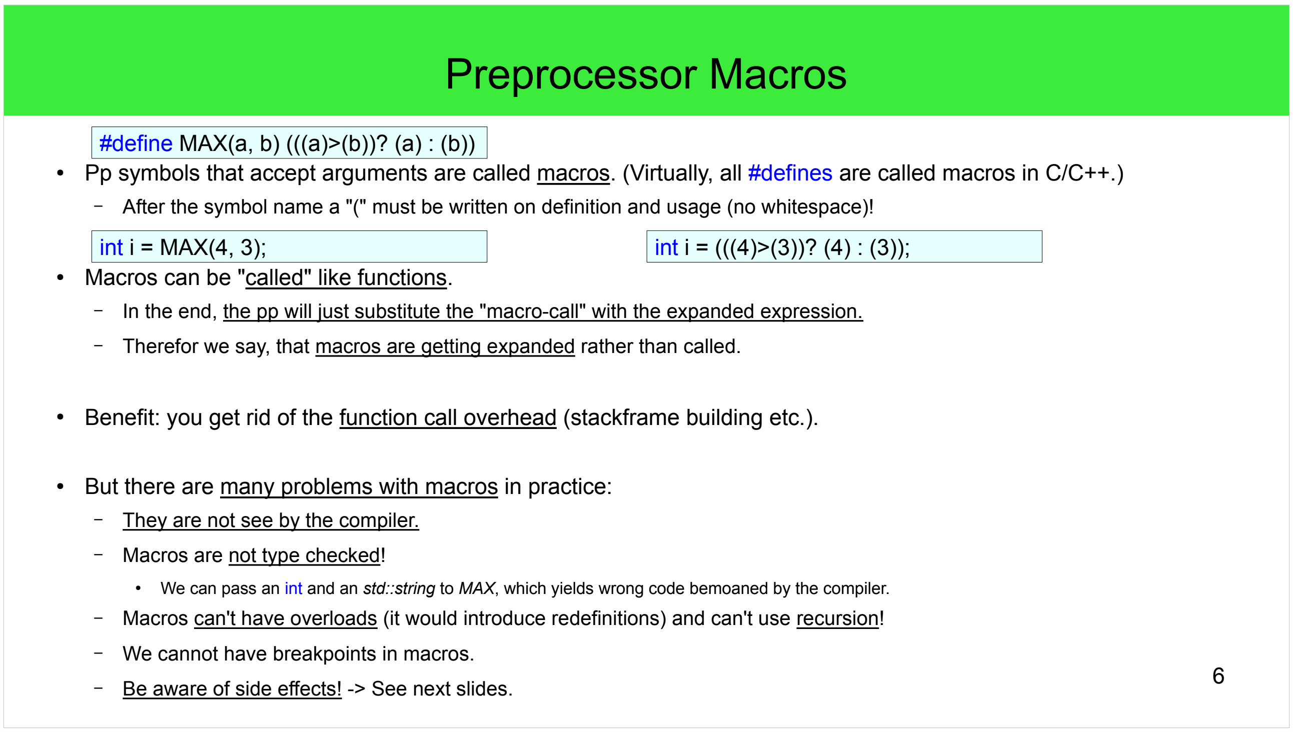
```
#define MAX_NEXT MAX_NO * 4
```

- The substitution also works in subsequent pp symbols.
  - Then a full expression rather than a single value is inserted.

- Useful predefined pp symbols (selection):
  - (*NULL*: represents the 0-pointer, *NULL* is obsolete in C++ (use 0 or nullptr (C++11) instead))
  - *__FILE__*: path/name of the pp input file, i.e. of the code file itself.
  - *__LINE__*: line number of the pp input file as decimal integer.
  - *NDEBUG* (not standardized): if it is defined, the pp generates the input file for release mode.

5

- The identifiers of pp symbols must be valid C/C++ identifiers.
- The '#'-symbol needs to be written in the first column of a line. Only one pp directive can be written per line.
- Pp symbols are only replaced when found on source code word boundaries, but never in string literals.
- We do not need to assign values to pp symbols on the #define directive. – We can "just" define them, this feature is typically used for #include guards, this topic will be discussed on the next slides.
- In C, the keywords and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq and wchar_t are defined as pp symbols.



Preprocessor Macros

```
#define MAX(a, b) (((a)>(b))? (a) : (b))
```

- Pp symbols that accept arguments are called macros. (Virtually, all #defines are called macros in C/C++.)
  - After the symbol name a "(" must be written on definition and usage (no whitespace)!

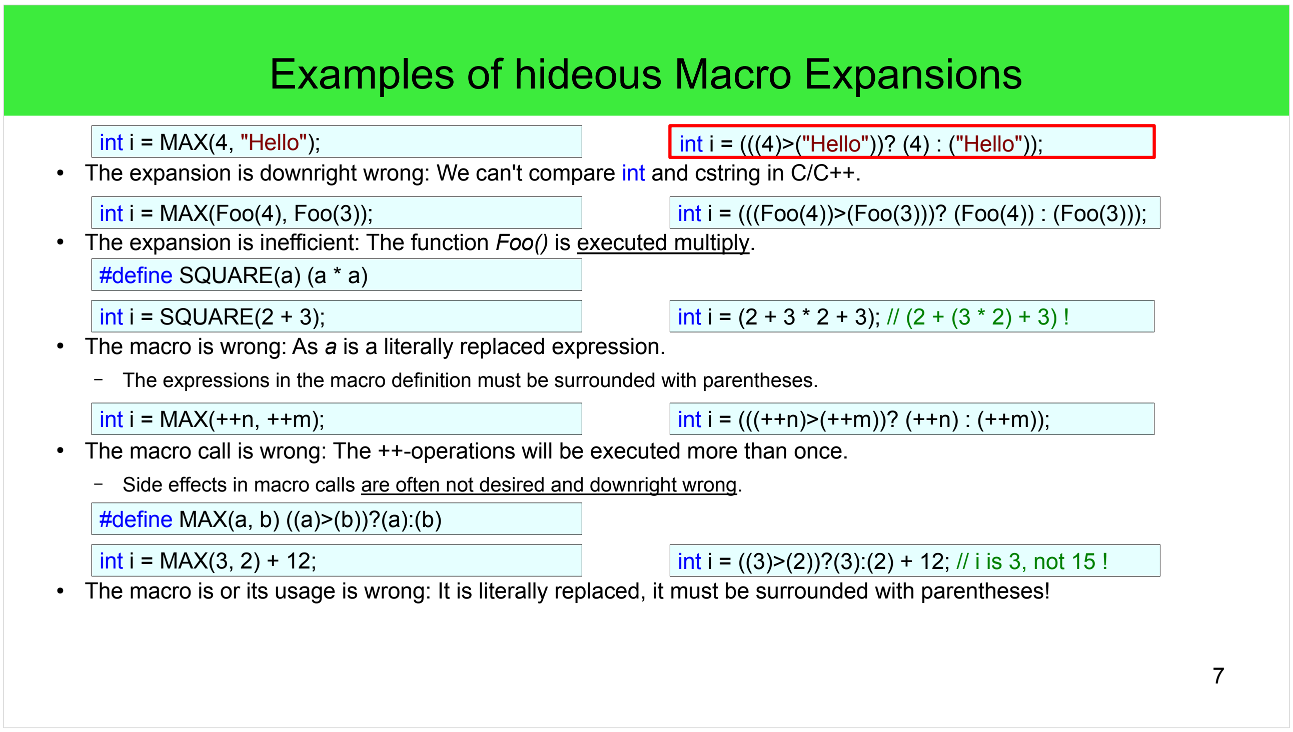
```
int i = MAX(4, 3);
```

```
int i = (((4)>(3))? (4) : (3));
```

- Macros can be "called" like functions.
  - In the end, the pp will just substitute the "macro-call" with the expanded expression.
  - Therefor we say, that macros are getting expanded rather than called.
- Benefit: you get rid of the function call overhead (stackframe building etc.).
- But there are many problems with macros in practice:
  - They are not see by the compiler.
  - Macros are not type checked!
    - We can pass an int and an *std::string* to *MAX*, which yields wrong code bemoaned by the compiler.
  - Macros can't have overloads (it would introduce redefinitions) and can't use recursion!
  - We cannot have breakpoints in macros.
  - Be aware of side effects! -> See next slides.

6

- Multiline macros can be defined by trailing each line with the '\'-symbol.
- There are even more problems with macros:
  - If we get compilation errors and they point to code using an unexpanded macro, we have to do the substitution/expansion ourselves to understand what is going on. – We've to keep in mind that macro "calls" modify the source code. Most C/C++ compilers can generate a version of the c-file having all macros (and other pp symbols and #includes) expanded, this expanded file is called translation unit (it will be explained in short).
  - As macros substitute code, their usage often leads to more code being generated in opposite to using functions.



Examples of hideous Macro Expansions

`int i = MAX(4, "Hello");` `int i = (((4)>("Hello"))? (4) : ("Hello"));`

- The expansion is downright wrong: We can't compare int and cstring in C/C++.

`int i = MAX(Foo(4), Foo(3));` `int i = (((Foo(4))>(Foo(3)))? (Foo(4)) : (Foo(3)));`

- The expansion is inefficient: The function *Foo()* is executed multiply.

`#define SQUARE(a) (a * a)`

`int i = SQUARE(2 + 3);` `int i = (2 + 3 * 2 + 3); // (2 + (3 * 2) + 3) !`

- The macro is wrong: As *a* is a literally replaced expression.
  - The expressions in the macro definition must be surrounded with parentheses.

`int i = MAX(++n, ++m);` `int i = (((++n)>(++m))? (++n) : (++m));`

- The macro call is wrong: The ++-operations will be executed more than once.
  - Side effects in macro calls are often not desired and downright wrong.

`#define MAX(a, b) ((a)>(b))?(a):(b)`

`int i = MAX(3, 2) + 12;` `int i = ((3)>(2))?(3):(2) + 12; // i is 3, not 15 !`

- The macro is or its usage is wrong: It is literally replaced, it must be surrounded with parentheses!

7

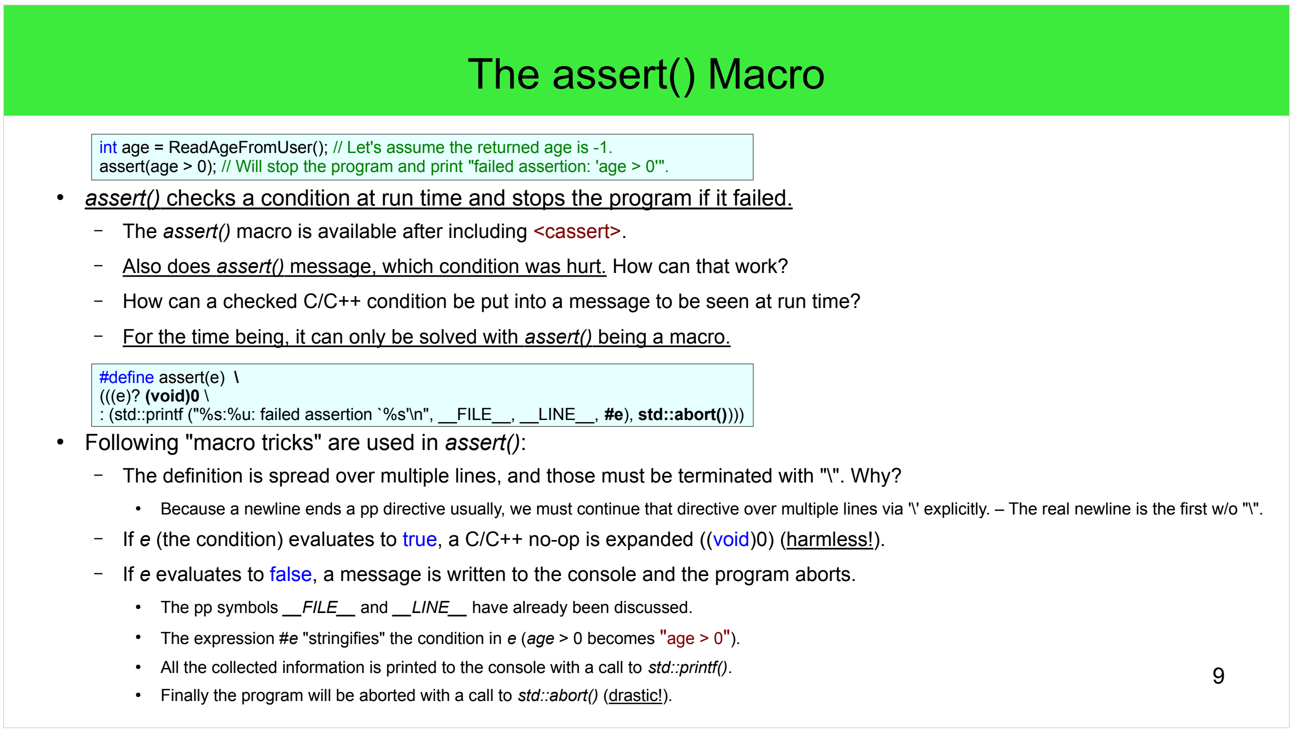
- In the end macros look innocent but can introduce goofy bugs. We've to keep in mind that macros are being literally replaced/expanded by the pp. The pp doesn't know about the C/C++ programming language at all (it knows what C/C++ string-literals and comments are)! – It is only a simple "text-replacer". In C++ there are better alternatives:
  - inline functions, const constants and templates.
  - Macro mechanisms in other languages like Dylan or Lisp are often much sophisticated, because they do not replace text, but work on a syntax-tree of the base language.
- But there are some useful and even required applications of macros in C/C++ and we'll examine them now.

Why are Macros required?

- The answer is simple: macros allow writing code that can't be written in C/C++!

- Sorry?
  - Well, macros can generate C/C++ code.
  - And the pp "sees" and operates on the code, before it is passed to the compiler.
  - => Virtually the pp solves problems the C/C++ languages can't solve.
    - E.g. a module system (h-files) or the "literalization" of code.

- Tips
  - We should generally avoid using (and writing) macros!
    - We should use alternatives... – C++, and esp. C++11 provides many better alternatives!
  - If we need to write a macro we've to mind using parentheses and we've to think about multiple evaluation of arguments.
  - Macros can also be used to create not only "function-like" or "const-like", they can expand to any (code) fragment!
    - This can be very useful, but must be used with a lot of care!



The assert() Macro

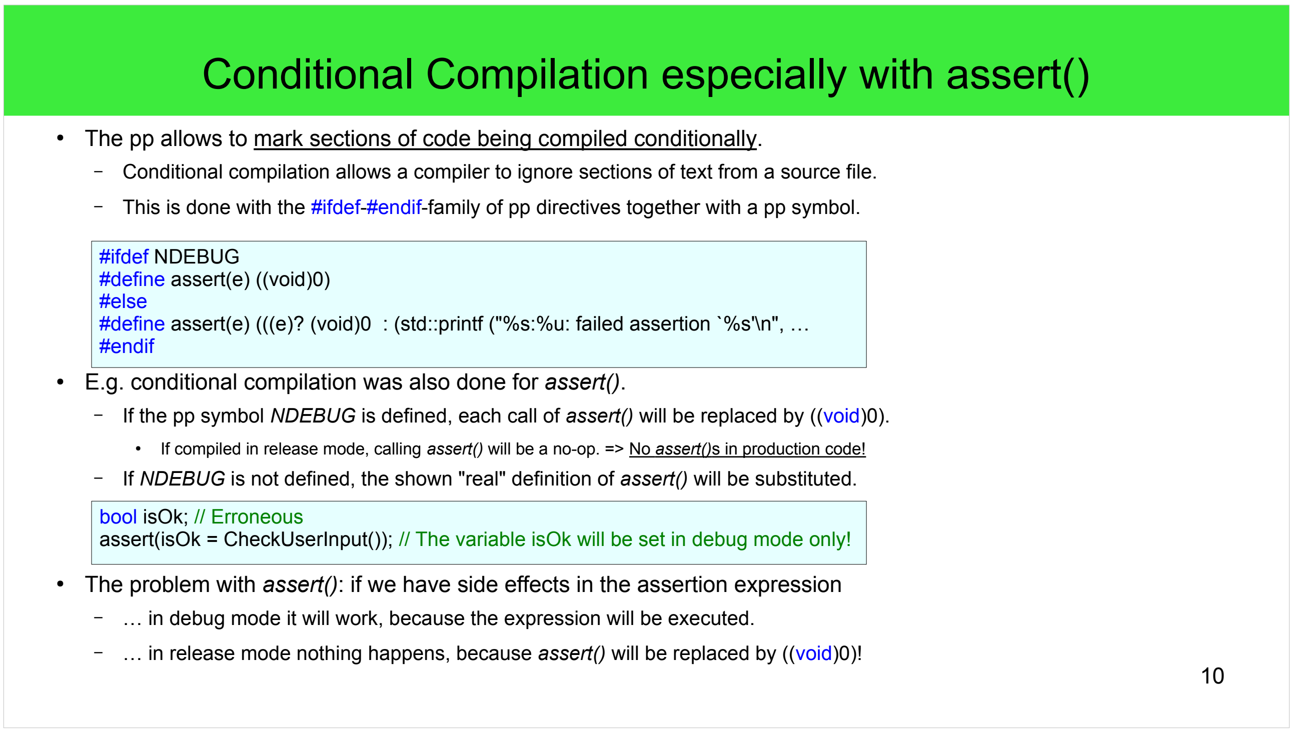
```
int age = ReadAgeFromUser(); // Let's assume the returned age is -1.
assert(age > 0); // Will stop the program and print "failed assertion: 'age > 0'".
```

- *assert()* checks a condition at run time and stops the program if it failed.
  - The *assert()* macro is available after including <cassert>.
  - Also does *assert()* message, which condition was hurt. How can that work?
  - How can a checked C/C++ condition be put into a message to be seen at run time?
  - For the time being, it can only be solved with *assert()* being a macro.

```
#define assert(e)  \
(((e)? (void)0 \
: (std::printf ("%s:%u: failed assertion `%s'\n", __FILE__, __LINE__, #e), std::abort())))
```

- Following "macro tricks" are used in *assert()*:
  - The definition is spread over multiple lines, and those must be terminated with "\". Why?
    - Because a newline ends a pp directive usually, we must continue that directive over multiple lines via '\' explicitly. – The real newline is the first w/o "\".
  - If *e* (the condition) evaluates to true, a C/C++ no-op is expanded ((void)0) (harmless!).
  - If *e* evaluates to false, a message is written to the console and the program aborts.
    - The pp symbols __*FILE*__ and __*LINE*__ have already been discussed.
    - The expression #*e* "stringifies" the condition in *e* (*age* > 0 becomes "age > 0").
    - All the collected information is printed to the console with a call to *std::printf()*.
    - Finally the program will be aborted with a call to *std::abort()* (drastic!).

- Macros can not live in namespaces, therefor no *std*-prefix is required for calling *assert()*.
- *assert()* is not meant to be an alternative to control flow, but shall stop a program forcibly.
- The *assert()* macro is a good example, as what it does can not be expressed with a C/C++ function!
- Why is the C++ code making up the *assert()* macro not highlighted?
  - Because it is "not yet" C++ code, it is only the body of the macro that could be expanded to real source code in future.
- (void)0 is a legal C/C++ statement, it just does nothing. The *assert()* macro is a prominent use of (void)0.
- *std::abort()* doesn't call dtors and it doesn't execute functions registered with *std::atexit()*!
- Here we can also spot the usage of the ,-operator (sequence operator). In an expression using the ,-operator the result of the expression after the last comma is the result of the whole expression.
- Assertions are usually enabled by default. Many compiler vendors offer the macro *NDEBUG* (not standardized), which can be defined to disable assertions.
- Macros can also be defined via the command line, when starting the compiler. – Defining a macro means, that macro-constant is being defined.
  - This means a c-file might not be compilable until we define a macro on the command line! – This should indicate how risky macros can be.



Conditional Compilation especially with assert()

- The pp allows to mark sections of code being compiled conditionally.
  - Conditional compilation allows a compiler to ignore sections of text from a source file.
  - This is done with the #ifdef-#endif-family of pp directives together with a pp symbol.

```
#ifdef NDEBUG
#define assert(e) ((void)0)
#else
#define assert(e) (((e)? (void)0  : (std::printf ("%s:%u: failed assertion `%s'\n", …
#endif
```

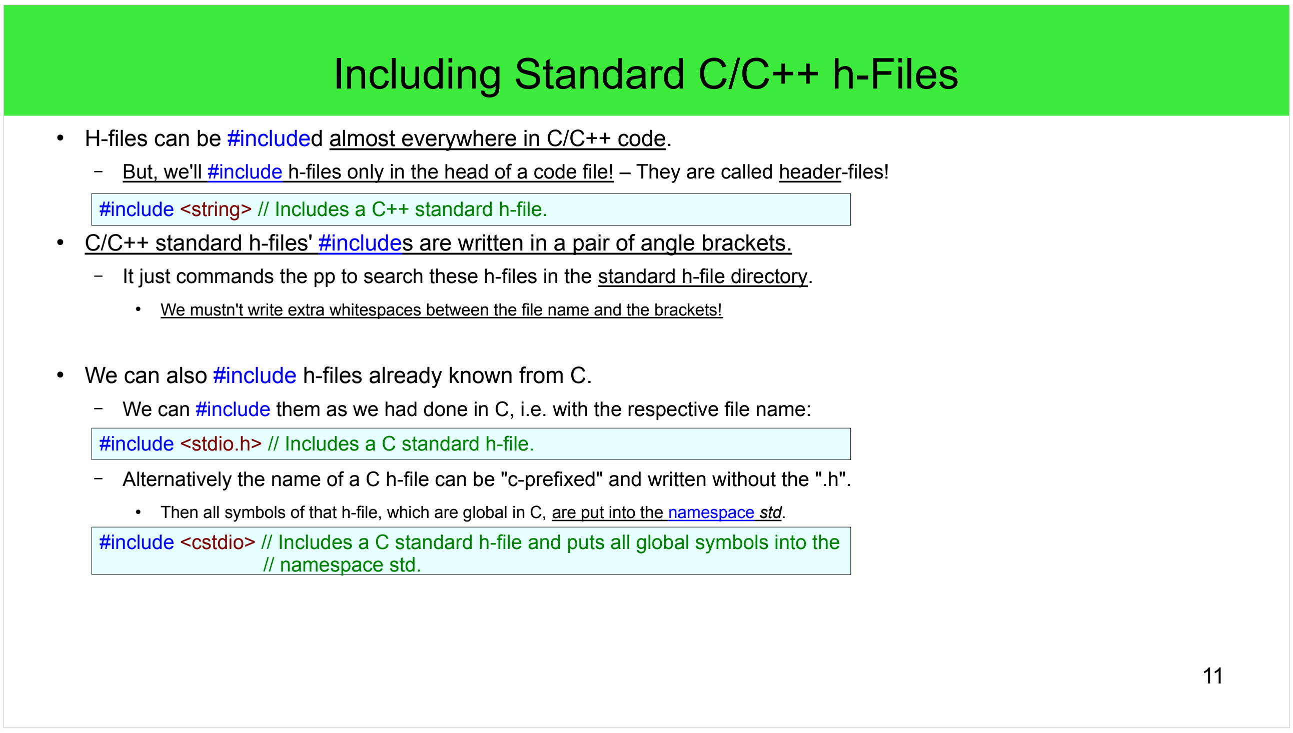
- E.g. conditional compilation was also done for *assert()*.
  - If the pp symbol *NDEBUG* is defined, each call of *assert()* will be replaced by ((void)0).
    - If compiled in release mode, calling *assert()* will be a no-op. => No *assert()*s in production code!
  - If *NDEBUG* is not defined, the shown "real" definition of *assert()* will be substituted.

```
bool isOk; // Erroneous
assert(isOk = CheckUserInput()); // The variable isOk will be set in debug mode only!
```

- The problem with *assert()*: if we have side effects in the assertion expression
  - … in debug mode it will work, because the expression will be executed.
  - … in release mode nothing happens, because *assert()* will be replaced by ((void)0)!

10

- Other examples of the #ifdef-#endif-family of pp directives: #ifndef, #if, #else, and #elif.
- The pp directive #if evaluates the content of a pp symbol or expression (an expression is something like *OPTION==3*), if it evaluates to true (i.e. not 0) the condition is fulfilled. But #ifdef evaluates to true if the pp symbol is just defined, independent of its content (or if it was not lately #undef'd).



Including Standard C/C++ h-Files

- H-files can be #included almost everywhere in C/C++ code.
  - But, we'll #include h-files only in the head of a code file! – They are called header-files!

```
#include <string> // Includes a C++ standard h-file.
```

- C/C++ standard h-files' #includes are written in a pair of angle brackets.
  - It just commands the pp to search these h-files in the standard h-file directory.
    - We mustn't write extra whitespaces between the file name and the brackets!

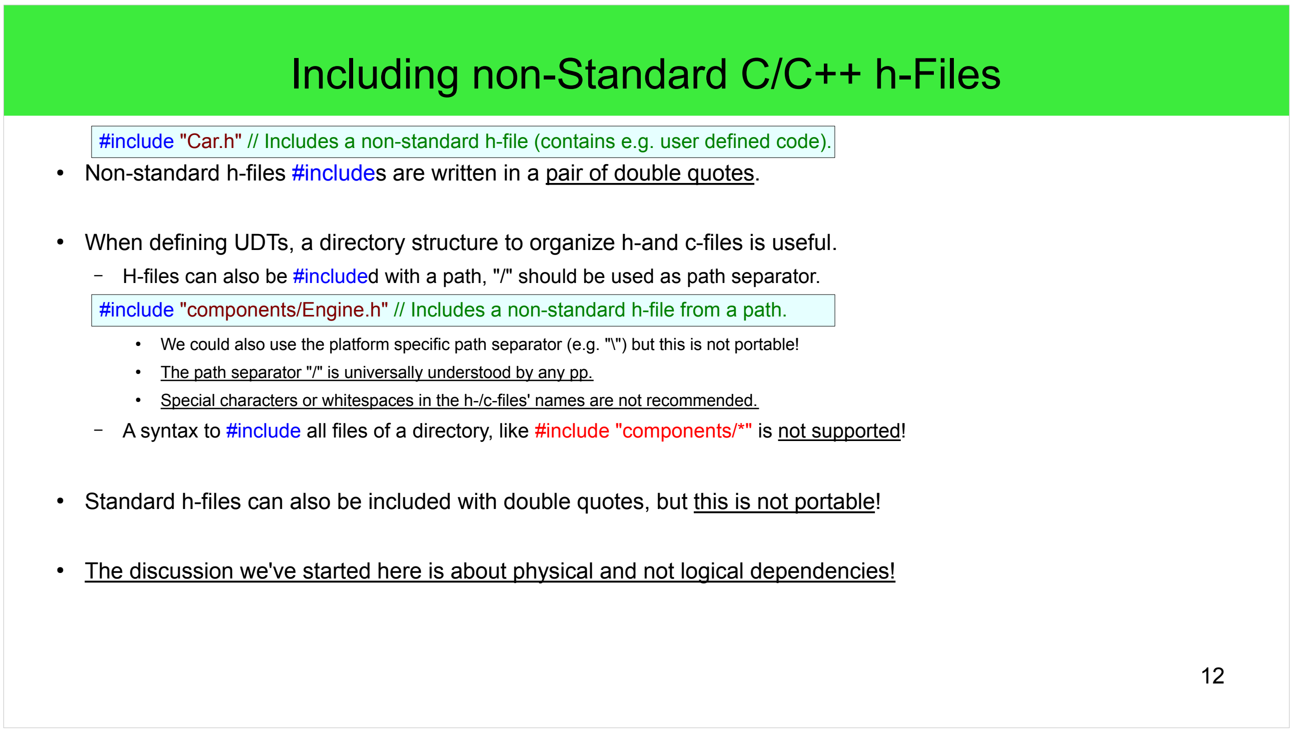
- We can also #include h-files already known from C.
  - We can #include them as we had done in C, i.e. with the respective file name:

```
#include <stdio.h> // Includes a C standard h-file.
```

  - Alternatively the name of a C h-file can be "c-prefixed" and written without the ".h".
    - Then all symbols of that h-file, which are global in C, are put into the namespace *std*.

```
#include <cstdio> // Includes a C standard h-file and puts all global symbols into the
                  // namespace std.
```

- Standard h-files need not to be provided as real files by a C++ distribution! – Instead the pp could copy the required contents to be inserted by the #includes from somewhere else into the code.



Including non-Standard C/C++ h-Files

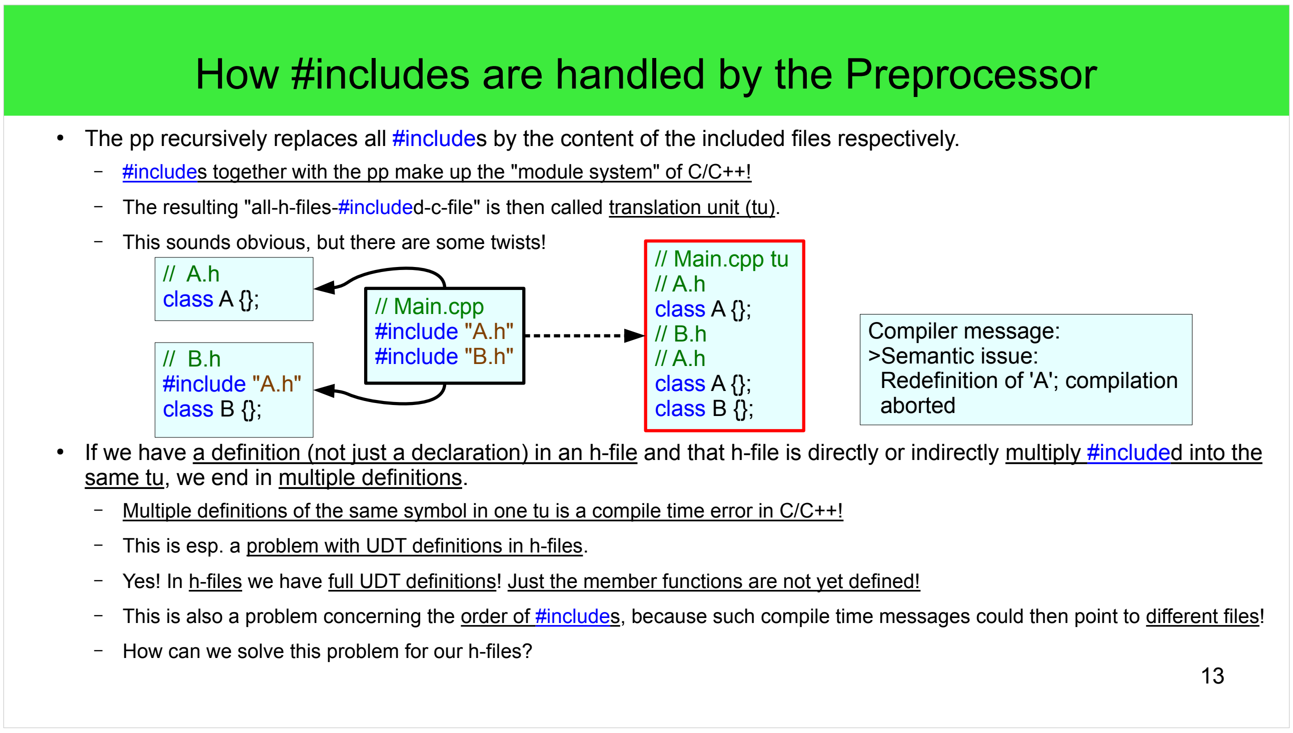
```
#include "Car.h" // Includes a non-standard h-file (contains e.g. user defined code).
```

- Non-standard h-files #includes are written in a pair of double quotes.
- When defining UDTs, a directory structure to organize h-and c-files is useful.
  - H-files can also be #included with a path, "/" should be used as path separator.

```
#include "components/Engine.h" // Includes a non-standard h-file from a path.
```

    - We could also use the platform specific path separator (e.g. "\") but this is not portable!
    - The path separator "/" is universally understood by any pp.
    - Special characters or whitespaces in the h-/c-files' names are not recommended.
  - A syntax to #include all files of a directory, like #include "components/*" is not supported!
- Standard h-files can also be included with double quotes, but this is not portable!
- The discussion we've started here is about physical and not logical dependencies!

- Why is the platform specific path separator (e.g. '\') not portable?
  - A pp on a Unix machine could probably not read such a path.
- Where is a non-standard h-file searched?
  - It depends on the pp settings or on the settings of the make system. First it is usually searched relative to the working directory and then relative to additionally set include paths.
- Standard h-files can also be included with double quotes. Why is this frowned upon?
  - It makes them less distinguishable from non-standard #includes.



How #includes are handled by the Preprocessor

- The pp recursively replaces all #includes by the content of the included files respectively.
  - #includes together with the pp make up the "module system" of C/C++!
  - The resulting "all-h-files-#included-c-file" is then called translation unit (tu).
  - This sounds obvious, but there are some twists!

```
//  A.h
class A {};
```

```
//  B.h
#include "A.h"
class B {};
```

```
// Main.cpp
#include "A.h"
#include "B.h"
```

```
// Main.cpp tu
// A.h
class A {};
// B.h
// A.h
class A {};
class B {};
```

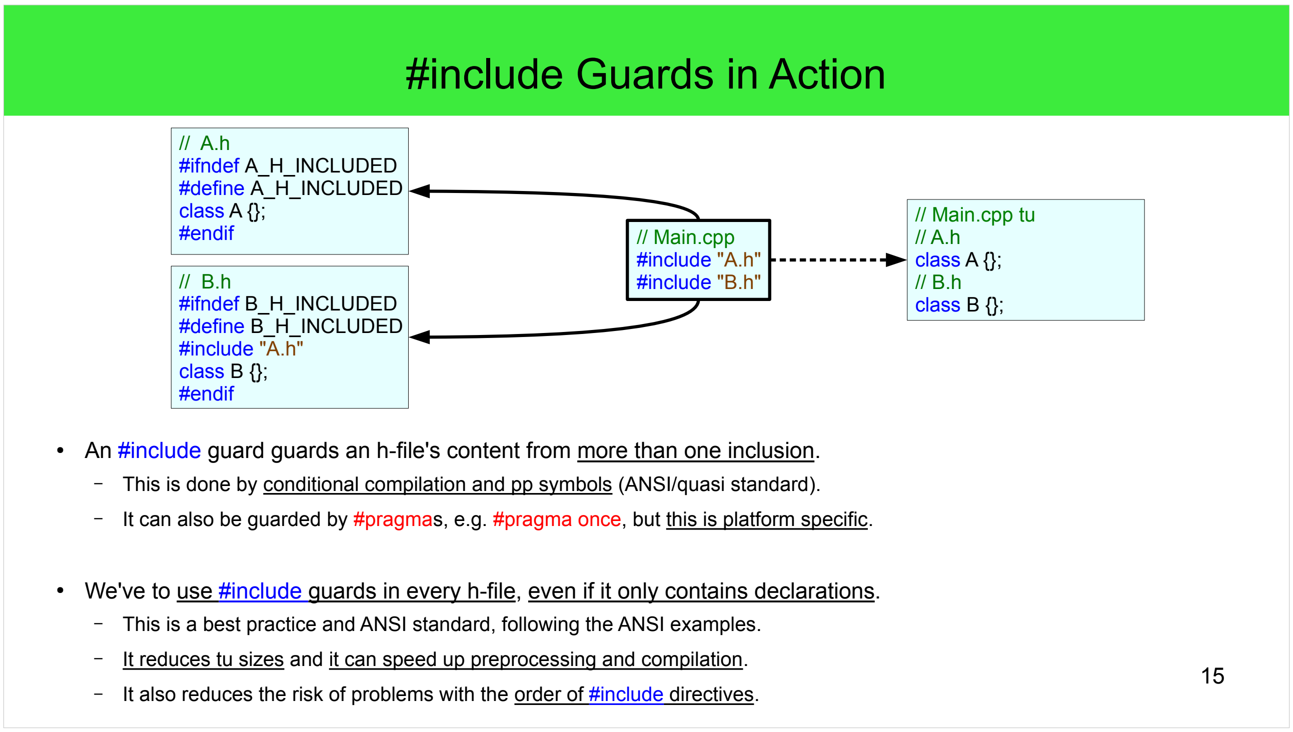
```
Compiler message:
>Semantic issue:
  Redefinition of 'A'; compilation
  aborted
```

- If we have a definition (not just a declaration) in an h-file and that h-file is directly or indirectly multiply #included into the same tu, we end in multiple definitions.
  - Multiple definitions of the same symbol in one tu is a compile time error in C/C++!
  - This is esp. a problem with UDT definitions in h-files.
  - Yes! In h-files we have full UDT definitions! Just the member functions are not yet defined!
  - This is also a problem concerning the order of #includes, because such compile time messages could then point to different files!
  - How can we solve this problem for our h-files?

- In reality, the comments are stripped from the code before the pp creates the tu. The comments have been left away in the example tu to get a clearer view of what is going on.

The Solution: #include Guards

- The solution for functions is simple: only put declarations into h-files.
  - Declarations can be safely written multiply in a tu.
  - But definitions can be only written once in a tu! – This is called the One Definition Rule (ODR).
- This can't be done for UDT definitions, as they need to reside in h-files generally.
  - Because only when another code file #includes that h-file that UDTs can be used.
- The solution are so called #include guards.
  - Surround the complete content of the h-file "A.h" with an #ifndef/#endif block.
  - The #ifndef directive should check a pp symbol that is unique in your project and belongs to the h-file ("A.h"). E.g. we can call the symbol *A_H_INCLUDED*.
  - The #ifndef block's first line #defines the symbol. (#define A_H_INCLUDED)
  - We've to repeat this modification for all h-files with different pp symbol names of course.
- Huh? Ok, let's rewrite our last example with #include guards.



#include Guards in Action

```
// A.h
#ifndef A_H_INCLUDED
#define A_H_INCLUDED
class A {};
#endif
```

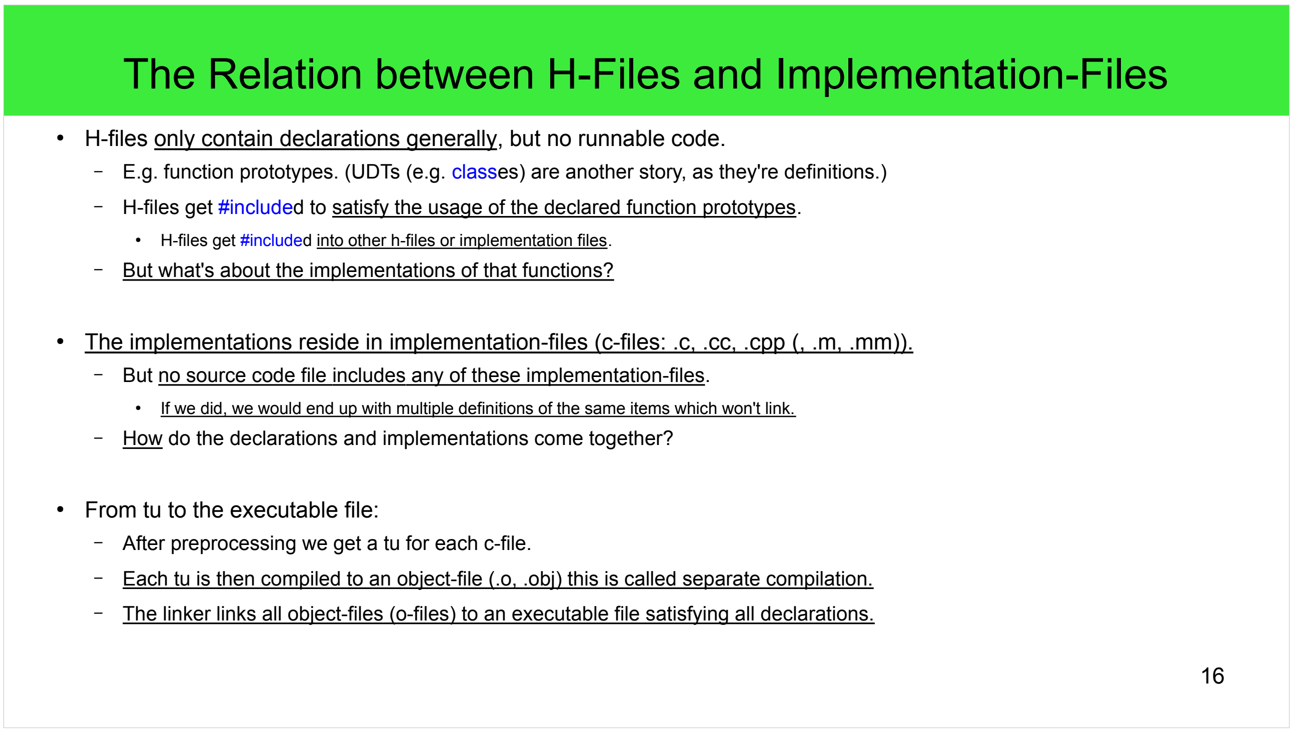
```
// B.h
#ifndef B_H_INCLUDED
#define B_H_INCLUDED
#include "A.h"
class B {};
#endif
```

```
// Main.cpp
#include "A.h"
#include "B.h"
```

```
// Main.cpp tu
// A.h
class A {};
// B.h
class B {};
```

- An #include guard guards an h-file's content from more than one inclusion.
  - This is done by conditional compilation and pp symbols (ANSI/quasi standard).
  - It can also be guarded by #pragmas, e.g. #pragma once, but this is platform specific.
- We've to use #include guards in every h-file, even if it only contains declarations.
  - This is a best practice and ANSI standard, following the ANSI examples.
  - It reduces tu sizes and it can speed up preprocessing and compilation.
  - It also reduces the risk of problems with the order of #include directives.

- Let's remember this aspect of pp symbols: we do not need to assign values to them on the #define directive. – We can "just" define them, this feature is typically used for #include guards.
- If we don't use #include guards consequently or forget to use them in a file in between, we'll pay that mistake with compiler errors, which are very difficult to catch. – Often we need to have the compiler system generate the tus of each implementation file in order to analyze these errors.
  - Most pps are clever enough to deal with cyclic #includes.
- There exist different ways to define #include guards, modern IDEs automatically add #include guards, when the user creates new h-files.
- Via the pp directive #pragma we can use extensions of the pp. This said, all #pragmas are compiler specific per se. The #pragma once can be found on most compilers meanwhile.
- #include guards: Why? – It also reduces tu sizes and it can speed up preprocessing and compilation.
- In Obj-C we also have to deal with h-files and implementation files. But an Obj-C compiler system needs no explicitly written #include guards (or more exactly: #import guards) as it guards the #imports automatically.



The Relation between H-Files and Implementation-Files

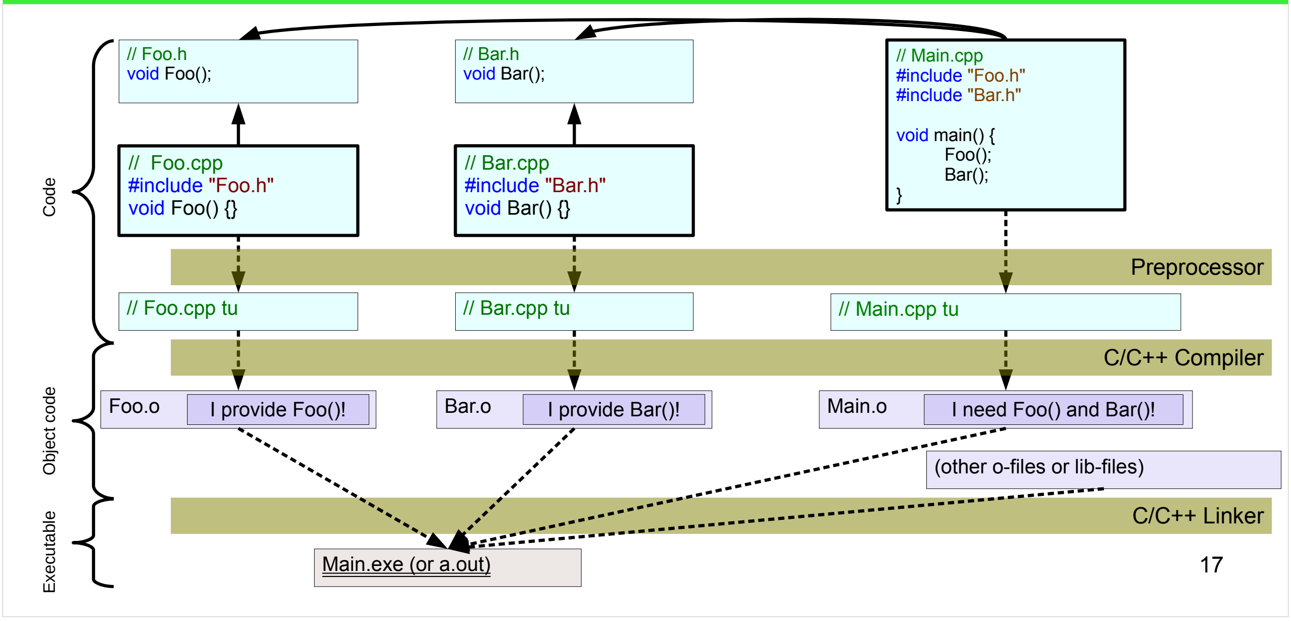
- H-files only contain declarations generally, but no runnable code.
  - E.g. function prototypes. (UDTs (e.g. classes) are another story, as they're definitions.)
  - H-files get #included to satisfy the usage of the declared function prototypes.
    - H-files get #included into other h-files or implementation files.
  - But what's about the implementations of that functions?

- The implementations reside in implementation-files (c-files: .c, .cc, .cpp (, .m, .mm)).
  - But no source code file includes any of these implementation-files.
    - If we did, we would end up with multiple definitions of the same items which won't link.
  - How do the declarations and implementations come together?

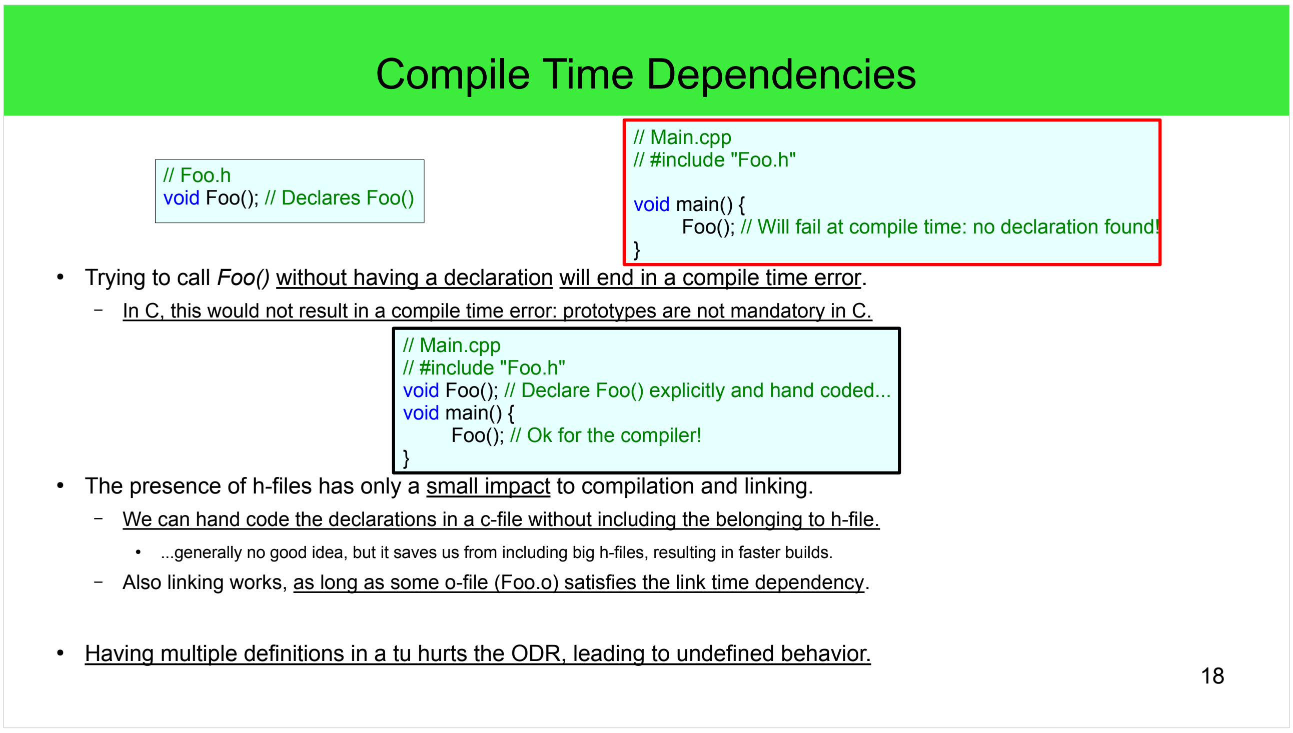
- From tu to the executable file:
  - After preprocessing we get a tu for each c-file.
  - Each tu is then compiled to an object-file (.o, .obj) this is called separate compilation.
  - The linker links all object-files (o-files) to an executable file satisfying all declarations.

- Separated compilation is possible because we have independent compile and link phases during the build process.

The full Build Process



- Notice how the c-files get separately compiled to o-files.
  - The compiler checks the C/C++ syntax and translates the C/C++ code into machine code.
- The relation, that Main.o has to the implementation of *Foo()* and *Bar()*, is called <u>link time dependency</u>.
  - The function of the linker is to "somehow satisfy the link time dependencies of the o-files". The object code of the implementation of *Foo()* could also reside in another o-file (not necessarily Foo.o) the linker can find.
- If we had called standard C/C++ functions in Main.cpp there would have been another link time dependency to one of the standard lib files that are installed with the compiler system.
  - In fact we have always dependencies to lib-files of the compiler system as we have to bind a standalone executable with code that makes it executable.
  - A lib-file is just an aggregation of a couple of o-files.



Compile Time Dependencies

```
// Foo.h
void Foo(); // Declares Foo()
```

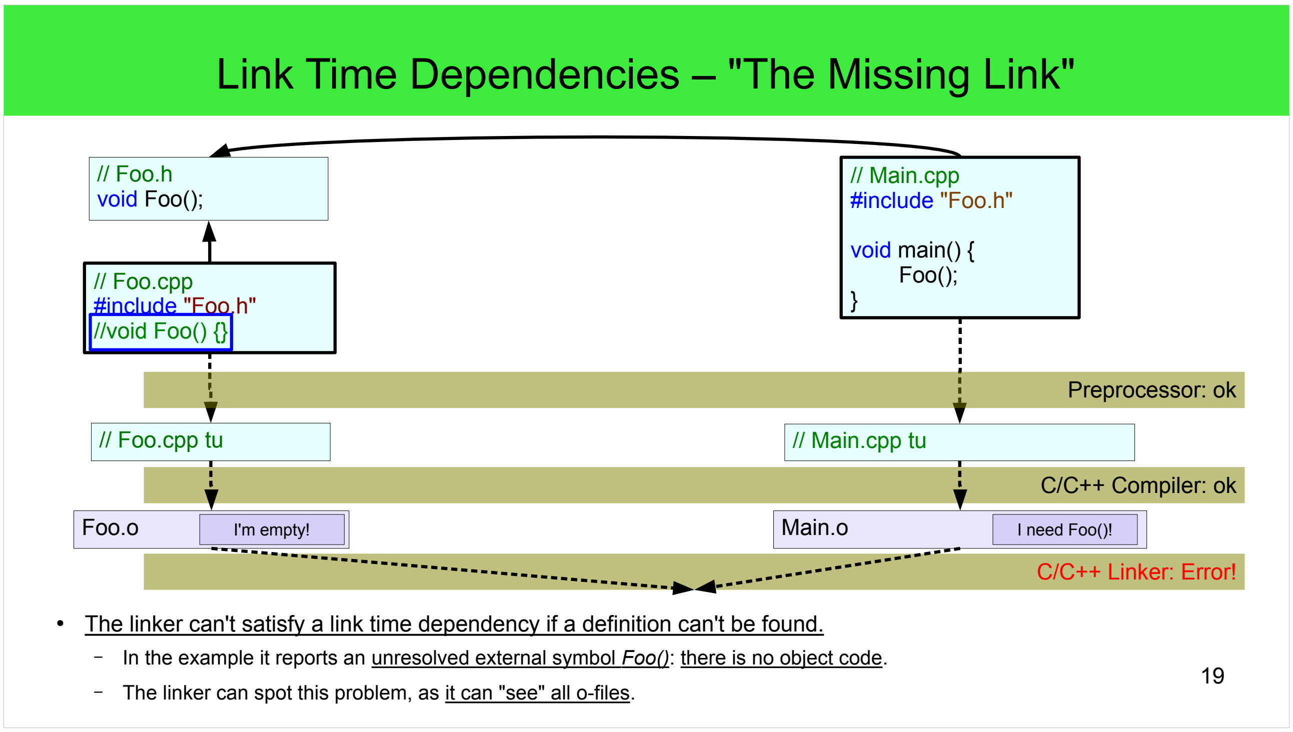
```
// Main.cpp
// #include "Foo.h"

void main() {
	Foo(); // Will fail at compile time: no declaration found!
}
```

- Trying to call *Foo()* without having a declaration will end in a compile time error.
  - In C, this would not result in a compile time error: prototypes are not mandatory in C.

```
// Main.cpp
// #include "Foo.h"
void Foo(); // Declare Foo() explicitly and hand coded...
void main() {
	Foo(); // Ok for the compiler!
}
```

- The presence of h-files has only a small impact to compilation and linking.
  - We can hand code the declarations in a c-file without including the belonging to h-file.
    - ...generally no good idea, but it saves us from including big h-files, resulting in faster builds.
  - Also linking works, as long as some o-file (Foo.o) satisfies the link time dependency.
- Having multiple definitions in a tu hurts the ODR, leading to undefined behavior.

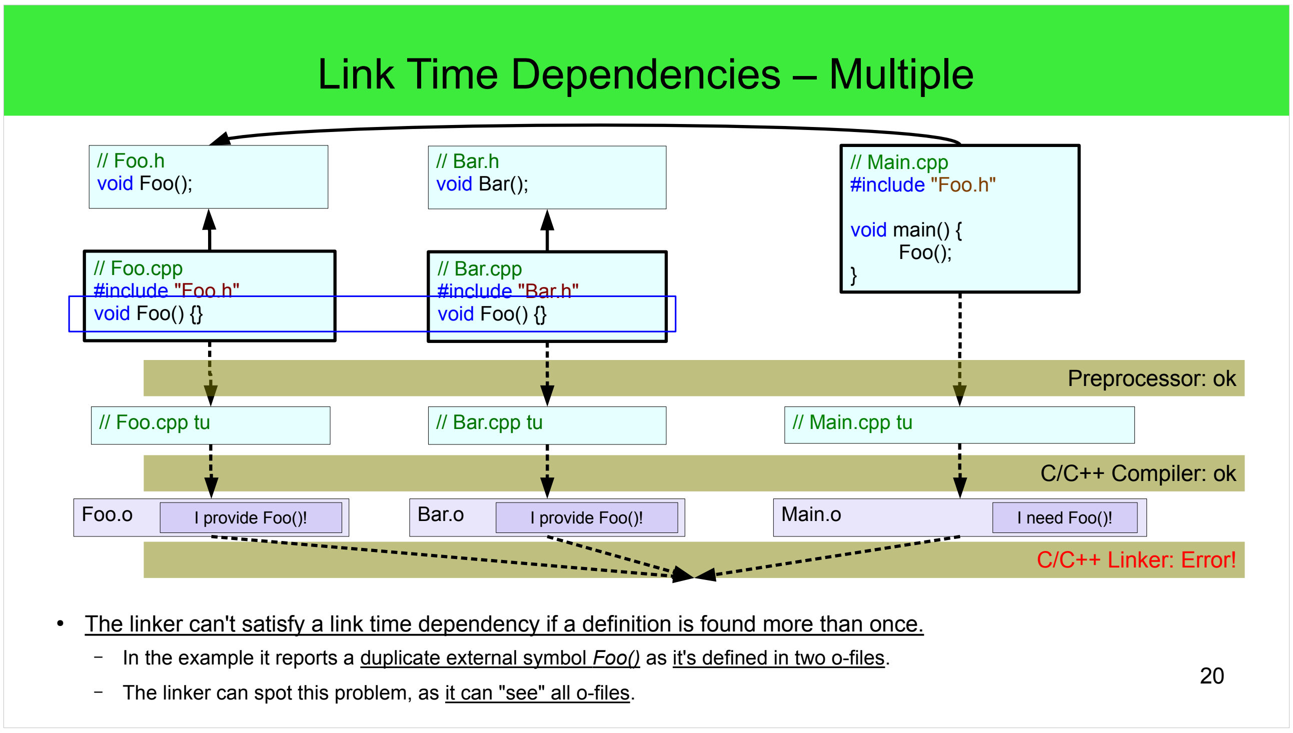
- Violations of the ODR can be spotted by the compiler.
- As already mentioned, it is simple to get code compiled in C! On missing function prototypes, a C compiler would infer the prototype from the arguments and the return type will always be set to int (but ignored return values are no problem). Maybe we get a couple of warnings from the compiler. – Also linking would work in this example, because some o-file (here Foo.o) will satisfy the link time dependency, where declarations or prototypes don't matter.
- The number of #includes affects the build time (preprocessing and compilation).
- Compilers can remove unused functions, then the o-file gets smaller.



Link Time Dependencies – "The Missing Link"

- The linker can't satisfy a link time dependency if a definition can't be found.
  - In the example it reports an unresolved external symbol *Foo()*: there is no object code.
  - The linker can spot this problem, as it can "see" all o-files.

- If the linker can't find a definition of *main()*, but was told to create a standalone executable, we'll also get a link time error: unresolved external symbol.



Link Time Dependencies – Multiple

- The linker can't satisfy a link time dependency if a definition is found more than once.
  - In the example it reports a duplicate external symbol *Foo()* as it's defined in two o-files.
  - The linker can spot this problem, as it can "see" all o-files.

20

- A function can only be defined once, period. – But we have to discuss the phenomenon of linkage in a future lecture, it weakens this rule somewhat.
- What kind of errors happen in consideration of *Bar()*?
  - None. As *Bar()* will never be called there is neither a compile time nor a link time dependency that could be hurt.

Why separate Compilation?

- When implementations of functions change, only their c-files need recompilation.
  - Because new o-files are created, the linker needs to relink everything of course.

- When an h-file changes, every including c-file needs to be recompiled.
  - Changes in h-file have a bigger impact to compilation than changes in c-files.
    - Changes in h-files can be very costly in large projects.
    - Changes in comments of a h-file do of course also change the h-file! Ugh!

- E.g. inline functions in h-file can create a costly compile time dependency.
  - Changes in the implementation of *Date::SetMonth()* force all includers to recompile!

```
// Date.h (contents hidden)
class Date { // (members hidden)
        int month;
public:
        // inline member function:
        int SetMonth(int month) {
            this->month = month;
        }
};
```

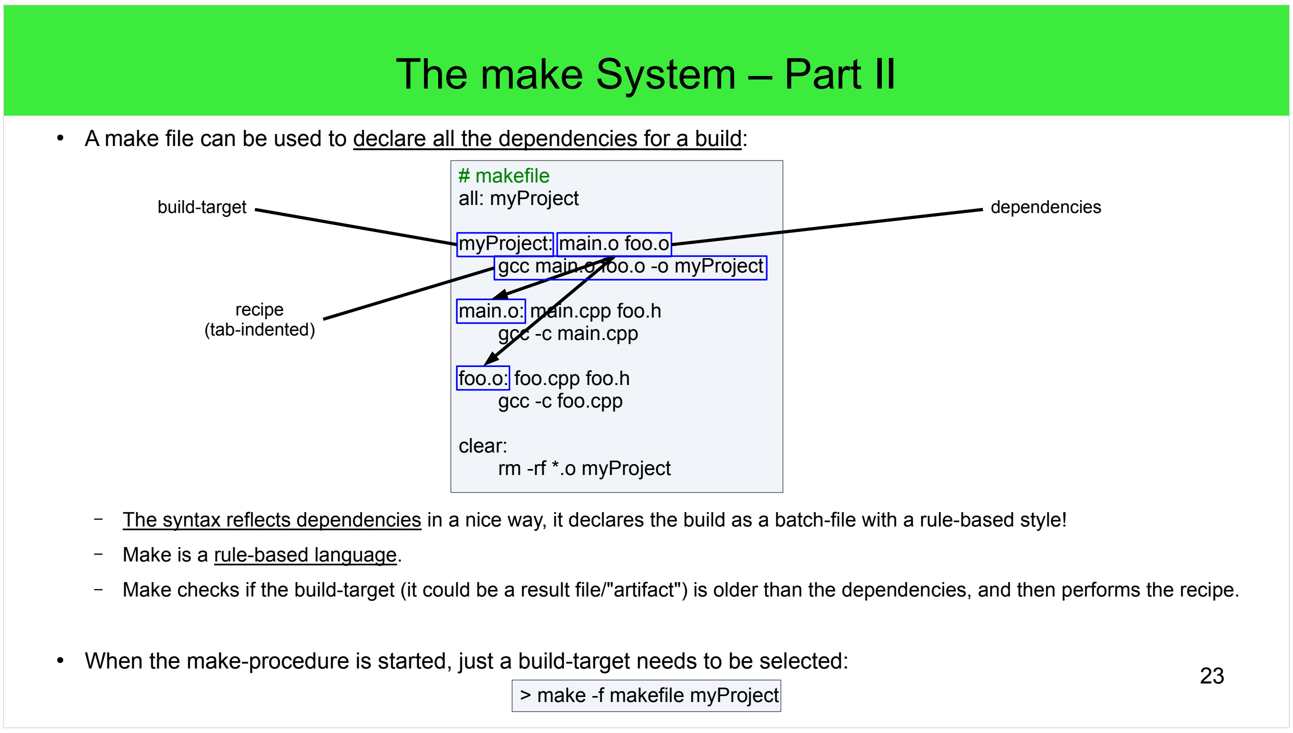
The make System – Part I

- The chain of rebuilding:
  - A tu is marked as being modified, if its code, esp. if any #included file was modified!
    - => A modified tu could stem from a modified h-file or c-file.
  - If a tu was modified, it needs recompilation, producing a new o-file.
  - If a new o-file was produced, the whole executable needs relinking.

- Where is the information about the build structure defined?
  - The dependencies between h-files and c-files and the order of compilation needs to be defined somewhere.
  - The linker needs to know, which o-files should be linked to get the result.

- Modern IDEs take care for these relations automatically.

- But we should also learn how to do it without an IDE: with so called make files.
  - Make files are executed on the command line! – No IDE is required!



The make System – Part II

- A make file can be used to declare all the dependencies for a build:

```
# makefile
all: myProject

myProject: main.o foo.o
	gcc main.o foo.o -o myProject

main.o: main.cpp foo.h
	gcc -c main.cpp

foo.o: foo.cpp foo.h
	gcc -c foo.cpp

clear:
	rm -rf *.o myProject
```

build-target
dependencies
recipe
(tab-indented)

  - The syntax reflects dependencies in a nice way, it declares the build as a batch-file with a rule-based style!
  - Make is a rule-based language.
  - Make checks if the build-target (it could be a result file/"artifact") is older than the dependencies, and then performs the recipe.

- When the make-procedure is started, just a build-target needs to be selected:

```
> make -f makefile myProject
```

- There also exist alternative build systems like ninja and Cmake.
- Actually, a makefile uses two "programming" languages:
  - The build-targets are written in makefile syntax.
  - The recipes, which contain commands that start the compiler or echo something to the console use the dialect of the shell, in which the makefile is executed. Many makefile-systems allow to specify a specific interpreter for the recipes.
  - To tell recipe-syntax from makefile-syntax, the receipt parts must be intended by tabs (not spaces)!
- We could simply execute the command *make* on the command line in this case. Then make will search for a file named "makefile" in the local directory and it will execute the command "all" to make the standard "all" target (which depends on the target "myProject" in the presented make file).
- Each target will only be built, if the target is older than any of its dependencies.
- Running *make* in a Cygwin environment:
  - Install the packages *core-gcc*, *gcc-g++* and *make*.

Thank you!