Separate compilation

Programação (L.EIC009)

José Proença (FCUP) & **João Bispo** (FEUP) – *slides by Eduardo R. B.*

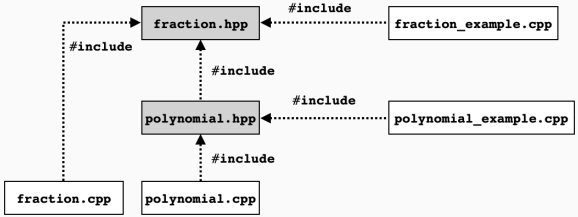
Header vs. implementation files

In C/C++ we usually divide the source code in two types of source code files.

Header files contain just declarations, e.g., of types (structs, classes, ...) or function prototypes. They typically have a `.hpp` or `.h` extension (C++ library files are an exception to this, they have no extension, e.g., `iostream`).

Implementation files (also sometimes called translation units) contain the actual implementations. They typically have a `.cpp` or `.cxx` extension (or just `.c` for C source code). Related functionality is often split in several source files (even the code of a single class).

Example



The example provided at GitHub corresponds to the examples of the previous class, but organised for separate compilation

- the `fraction` class, declared in `fraction.hpp` and implemented in `fraction.cpp`;
- the `polynomial` class, declared in `polynomial.hpp` and implemented in `polynomial.cpp`;
- two test programs, `fraction_example.cpp` and `polynomial_example.cpp`; and
- a `CMakeLists.txt` that automates the separate compilation process.

Example (cont.)

The header files just contain declaration, for instance within `fraction.hpp`:

```
...
  class fraction {
  public:
    // no code!
    fraction(int n, int d = 1);
    ...
    int numerator() const;
    ...
  private:
    int num, den;
    ...
  };
```

Example (cont.)

The implementation files contain the actual code, for instance within `fraction.cpp`:

```
#include "fraction.hpp"
...
fraction::fraction(int n, int d) : num(n), den(d) {
  reduce();
}
...
int fraction::numerator() const {
  return num;
}
...
```

- The implementation file must include the header file (otherwise it won't compile!). Other code just needs to include the header file.
- A header file should include only the header files that it requires. For instance, `polynomial.hpp` includes `fraction.hpp` and `vector` because it has method declarations that use those classes.
- Library headers are included using `#include <filename>`, local headers are included using `#include "filename"`.
- Header files should employ "header guards" so that repeated inclusion is not a problem (see slides from previous class).
- `using` directives should not be used in a header file to avoid potential duplication of symbol names ("namespace pollution"). These directives would also affect other files that include the header file.

Separate compilation process

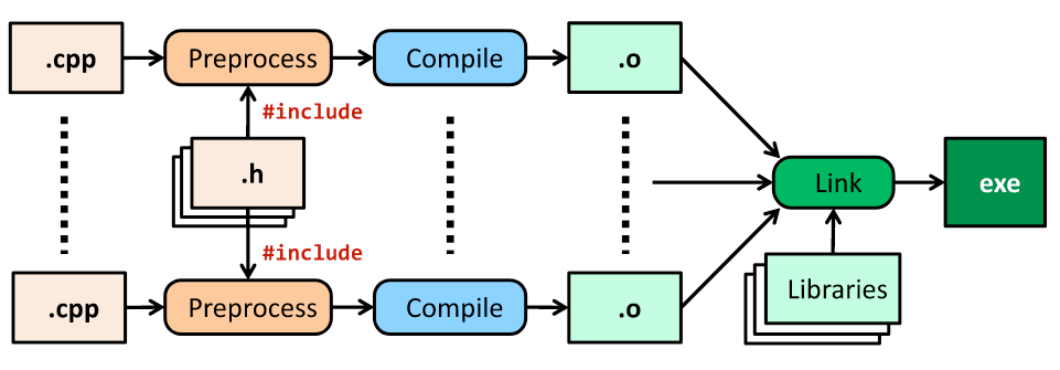
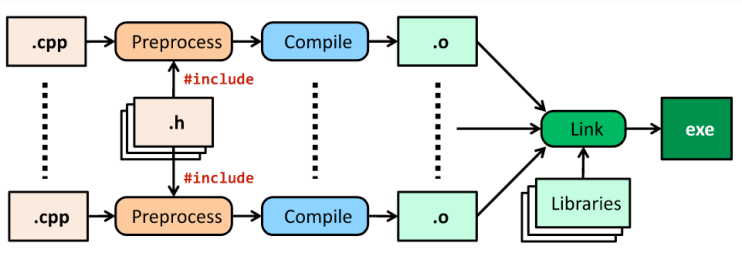


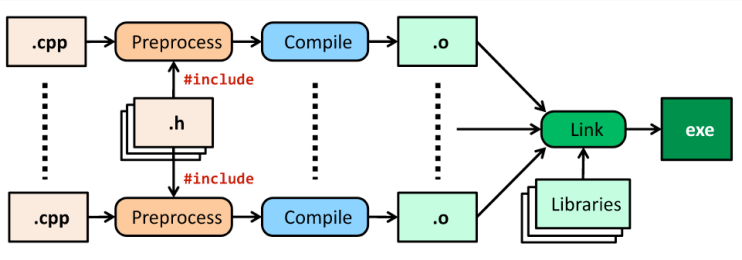
Image source: hackingcpp.com, by André Müller

Separate compilation process (cont.)



The C++ preprocessor "copy-pastes" header files defined in `#include` directives. The preprocessor supports more directives, such as the `#ifndef` or `#define` used in header guards.

The compiler translates implementation files to assembly and produces **object files** (with an `.o` or `.obj` extension). The "object file" designation is unrelated to the objects of object-oriented programming and predates them.



The linker combines all object files into a single executable file, as long as one (and just one) of the object files contains a `main` implementation. Along with object files, an executable may also require libraries (themselves typically consisting of several object files). For instance, C++ executables are always linked with the C++ standard library.

Why no declarations in header files?

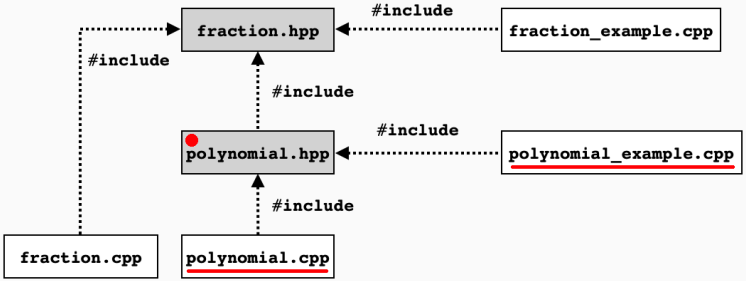
Consider `polynomial.hpp` contains an implementation of `evaluate()`. What happens if we compile `polynomial_example.cpp`?

Why no declarations in header files?

Consider `polynomial.hpp` contains an implementation of `evaluate()`. What happens if we compile `polynomial_example.cpp`?

```
ld.lld: error: duplicate symbol:
  leic::polynomial::evaluate(leic::fraction const&) const
>>> defined at .../prog2425_examples/10/polynomial.hpp:26
>>>            objects.a(polynomial_example.cpp.obj)
>>> defined at .../prog2425_examples/10/polynomial.hpp:26
>>>            libleic.a(polynomial.cpp.obj)
```

Why no declarations in header files? (cont.)



- Implementation in `polynomial.hpp` now appears in two different `.cpp` files.
- Header files are necessary, an implementation file can only use classes and functions that it knows about (i.e., through declarations).
- However, between all `.cpp` files of a program, there can be only one implementation for each declaration.

Separate compilation using GCC

U.PORTO

We can invoke g++ to produce an object file as follows:

```
g++ ... [other options] ... -c -o file.o file.cpp
```

After compiling a set of object files, we can link them into an executable (if one of the object files contains `main`):

```
g++ ... [other options] ... -o a.exe file1.o file2.o ... filen.o
```

We can also generate an executable directly, object files are implicitly generated and then erased at the end by the compiler (separate compilation is implicit):

```
g++ ... [other options] ... -o a.exe file1.cpp file2.cpp ... filen.cpp
```

Note: other compilers support similar schemes.

We can combine object files into a **library**. A library groups several object files.

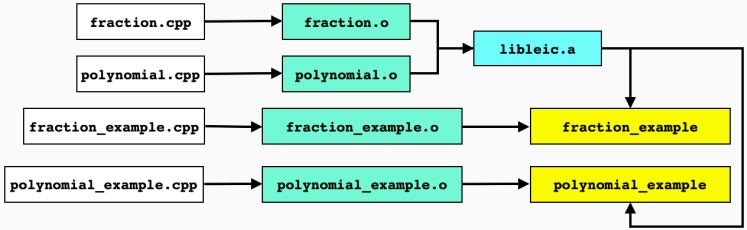
Static libraries are linked statically in (added to) the binary code of programs (in this case none of the object files should contain `main`):

```
ar cr libMyLib.a file_1.o file2_o ... file_n.o
g++ ... options ... -o executable executable.o libMyLib.a
```

There also **dynamic libraries** - also called shared objects or dynamically linked libraries (DLLs) in Windows. These are not linked together with the executable, but loaded at runtime when a program executes.

```
g++ -shared -o libMyLib.so file_1.o file2_o ... file_n
g++ ... options ... -o executable executable.o libMyLib.so
```

Example (cont.)



In this example, separate compilation creates:

- each `.o` file from the corresponding `.cpp` file, e.g., `fraction.o` from `fraction.cpp`
- the `libleic.a` static library from `fraction.o` and `polynomial.o`
- the `fraction_example` executable from `libleic.a` and `fraction_example.o`
- the `polynomial_example` executable from `libleic.a` and `polynomial_example.o`

Example (cont.)

The corresponding `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.15)
project(examples_10)

# Create library leic
add_library(leic fraction.cpp polynomial.cpp)

add_executable(fraction_example_2 fraction_example.cpp)
target_link_libraries(fraction_example_2 leic)

add_executable(polynomial_example_2 polynomial_example.cpp)
target_link_libraries(polynomial_example_2 leic)
```

Example (cont.)

```
$ cmake -B build && cd build && make
[ 14%] Building CXX object CMakeFiles/leic.dir/fraction.cpp.obj
[ 28%] Building CXX object CMakeFiles/leic.dir/polynomial.cpp.obj
[ 42%] Linking CXX static library libleic.a
[ 42%] Built target leic
[ 57%] Building CXX object CMakeFiles/.../fraction_example.cpp.obj
[ 71%] Linking CXX executable fraction_example_2.exe
[ 71%] Built target fraction_example_2
[ 85%] Building CXX object CMakeFiles/.../polynomial_example.cpp.obj
[100%] Linking CXX executable polynomial_example_2.exe
[100%] Built target polynomial_example_2
```