Separate compilation

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

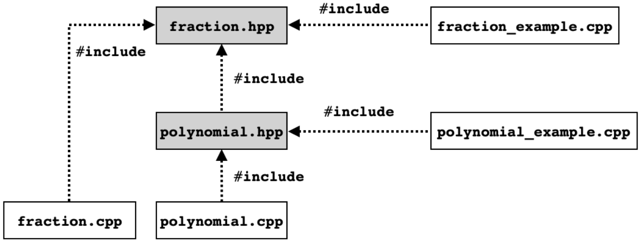
Header vs. implementation files

In C/C++ it is common to 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). It can be that related functionality can be 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 as follows:

- 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 custom `Makefile` 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;
}
...
```

Guidelines

- 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 other header files as necessary to guarantee self-containment. For instance, `polynomial.hpp` includes `fraction.hpp` and `vector` so that external code that uses `polynomial` objects just needs to include `polynomial.hpp`.
- 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 and potential duplication of symbol names ("namespace pollution"). Otherwise these directives will affect other files that include the header file.

Separate compilation process

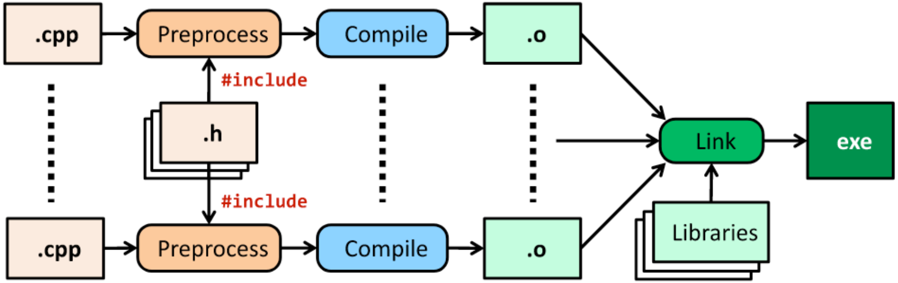
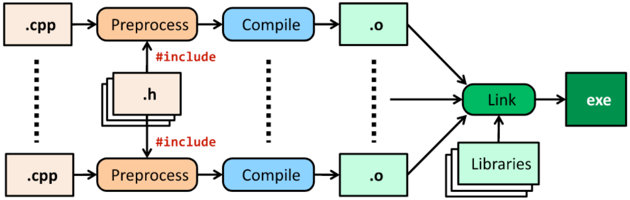


Image source: hackingcpp.com, by André Müller

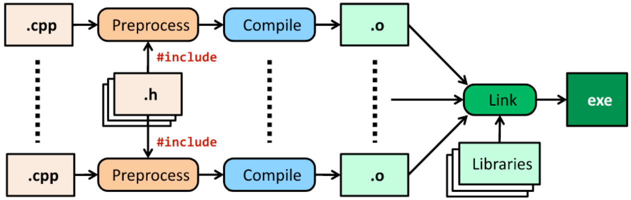
Separate compilation process (cont.)



Header files are included by implementation files through `#include` directives. The C++ preprocessor takes care of these directives by (literally) joining the contents of header files into the implementation files before compiling them.

Implementation files can be compiled separately to produce what are called **object files** (with an `.o` or `.obj` extension), files that contain compiled binary code. (The "object file" designation has nothing to do with object-oriented programming per se.)

Separate compilation process (cont.)



Object files can be combined to link an executable file, as long as one (and just one) of the object files contains a `main` definition. 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.

Separate compilation using GCC

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 executable file_1.o file_2.o ... file_n.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 executable file_1.cpp file_2.cpp ... file_n.o
```

Note: other compilers support analogous schemes.

Separate compilation using GCC (cont.)

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

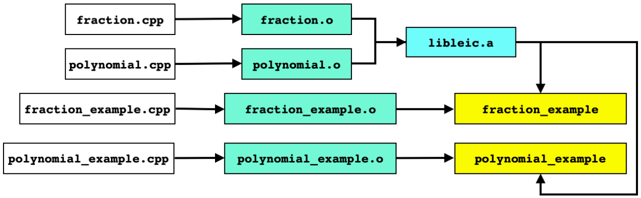
Static libraries are linked statically in (added to) the binary code of programs (in this case none of the object files should contain `main`):

```
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.)



Separate compilation proceeds by creating:

- 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`; and
- the `polynomial_example` executable from `libleic.a` and `polynomial_example.o`.

Example (cont.)

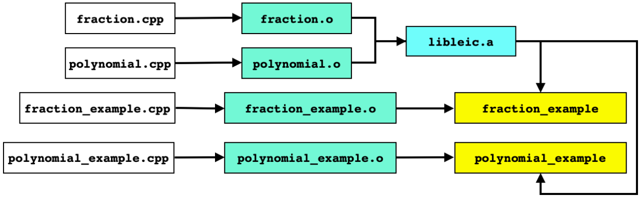
The corresponding `Makefile`:

```
libleic.a: fraction.o polynomial.o
  ar cr libleic.a fraction.o polynomial.o
%.o: fraction.hpp polynomial.hpp %.cpp
    $(CXX) $(CXXFLAGS) -c -o $*.o $*.cpp
fraction_example: libleic.a fraction_example.o
  $(CXX) $(CXXFLAGS) -o fraction_example \
            fraction_example.o libleic.a
polynomial_example: libleic.a polynomial_example.o
  $(CXX) $(CXXFLAGS) -o polynomial_example \
            polynomial_example.o libleic.a
```

Each `Makefile` rule has the form

```
target: dependencies
    action
```

Example (cont.)



```
$ make
g++ ...  -c -o fraction.o fraction.cpp
g++ ...  -c -o polynomial.o polynomial.cpp
ar cr libleic.a fraction.o polynomial.o
g++ ... -c -o fraction_example.o fraction_example.cpp
g++ ... -o fraction_example fraction_example.o libleic.a
g++ ... -c -o polynomial_example.o polynomial_example.cpp
g++ ...-o polynomial_example polynomial_example.o libleic.a
```