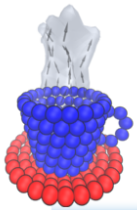


The Compilation Process

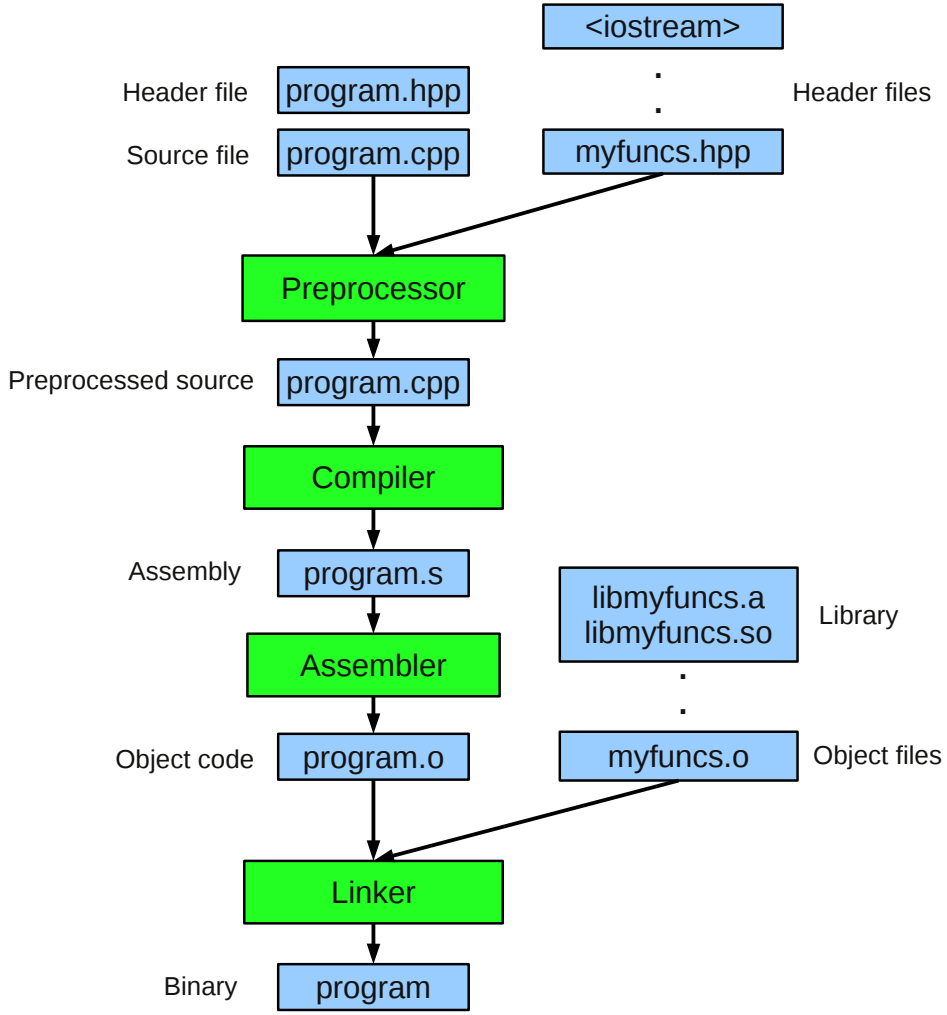
Olaf Lenz

Institut für Computerphysik
Universität Stuttgart

March 17-21, 2014

Separate Compilation

- So far, all programs were a single file
- Large software projects consist of thousands of lines of code
- Problem:
  - Bad for team projects
  - Compilation may take very long
  - Impractical to edit
- Better: split the project into several files, *compile* them one by one and *link* them together at the end

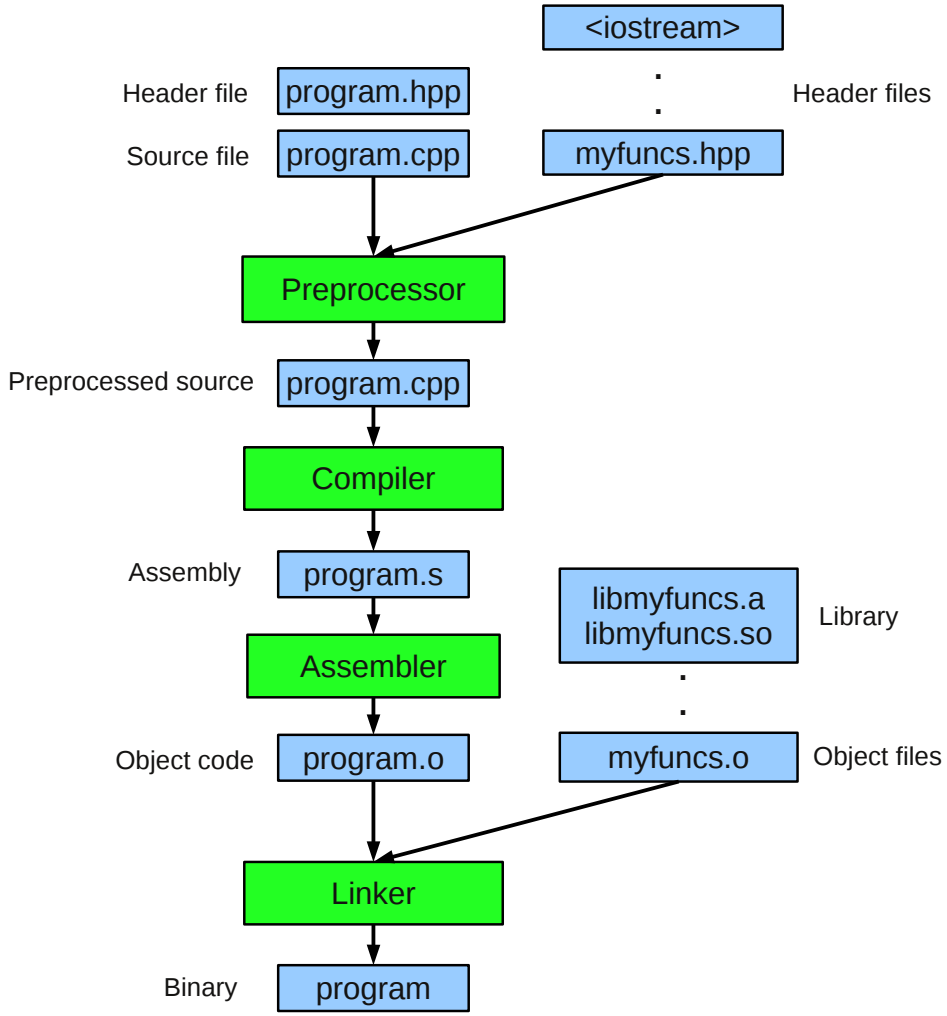


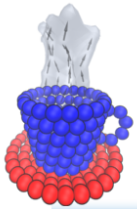
Stage 1: Preprocessor Inclusions

```
g++ -E <source>
```

```
#include FILE
```

- Preprocess FILE, output `.cpp`-file
- Pure text replacement: `#include FILE` is replaced by the contents of FILE
- Outputs a single C++ file





Stage 1: Preprocessor Macros

- `#define NAME REPLACEMENT` defines a *macro* with the given NAME
- Whenever the macro name turns up in the file, it is replaced by the REPLACEMENT
- This *was* used for constants (*e. g.* `M_PI`)
- Preprocessor Conditionals

```
#ifdef MACRO
.
.
#else
.
.
#endif
```

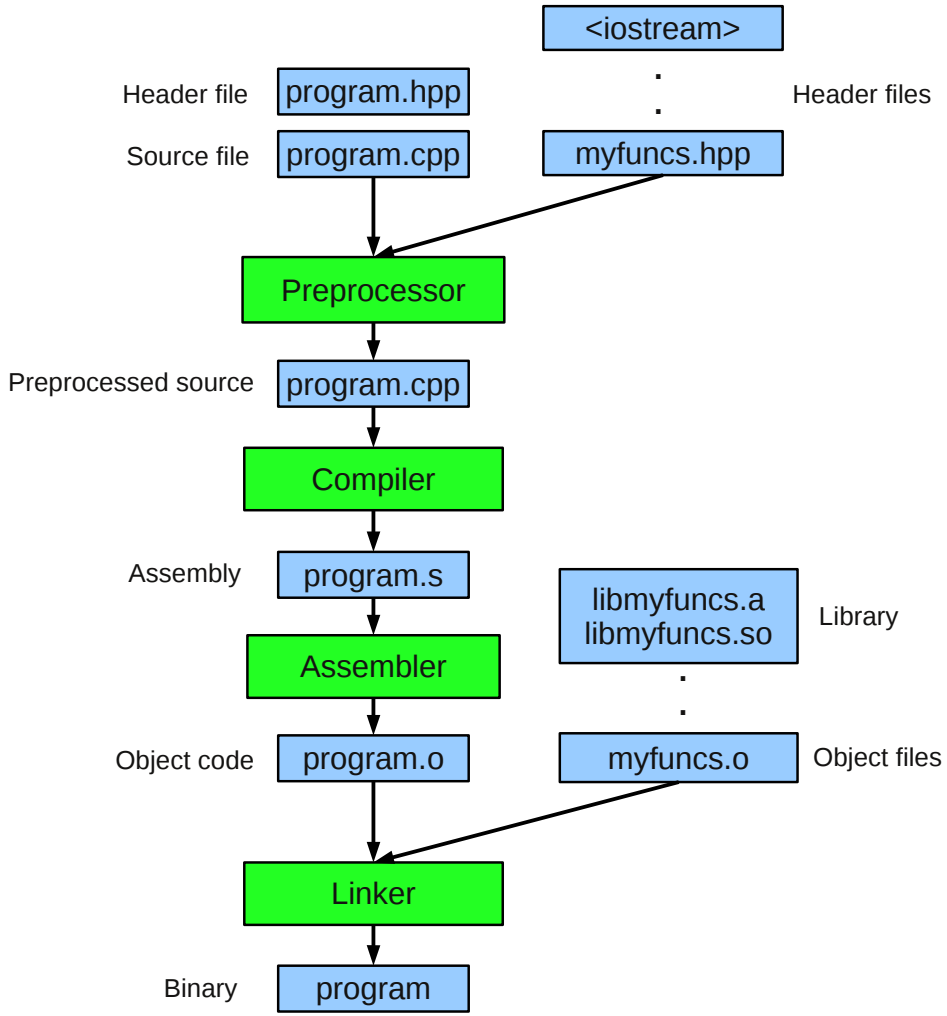
- Nowadays mostly used for compile time guards

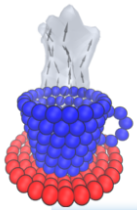
Declarations vs. Definitions

- Not all code is open source, so not all code should be there.
- ⇒ build object code from your code, and link it together.
- Problem: To be able to compile a function, some things must be known about all functions, variables and types that occur.

```
int main() {
  cout << "Hello, World!" << en
}
```

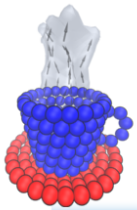
- What is `cout`, `<<` and `endl`?
- ⇒ Split function into declaration and definition





Declarations vs. Definitions

- All functions and variables (*symbols*) in C++ can be *declared* and have to be *defined* somewhere
- The *declaration* of a symbol introduce names to the compiler: “This function or this variable exists somewhere and here is what it should look like.”
- The *definition* tells the compiler: “Make this variable or function here.”
- Declarations can be repeated, definitions must be unique
- A *header file* (*e.g.* `.hpp`) should only contain declarations (it is included in other header and source files)
- A *source file* (*e.g.* `.cpp`) contains definitions as well as declarations
- All symbols declared in a header file should be defined in exactly one source file



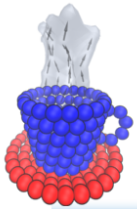
Function Declarations and Definitions

- Declaration

```
int func(int length, int width);
```

- Provides the signature (name, parameters and return type) of a function
- The names are ignored by the compiler
- Definition

```
int func(int length, int width) {
  // function code
  .
  .
}
```



Variable Declarations and Definitions

- Only interesting for *global variables*
- Declaration

```
extern int a;
```

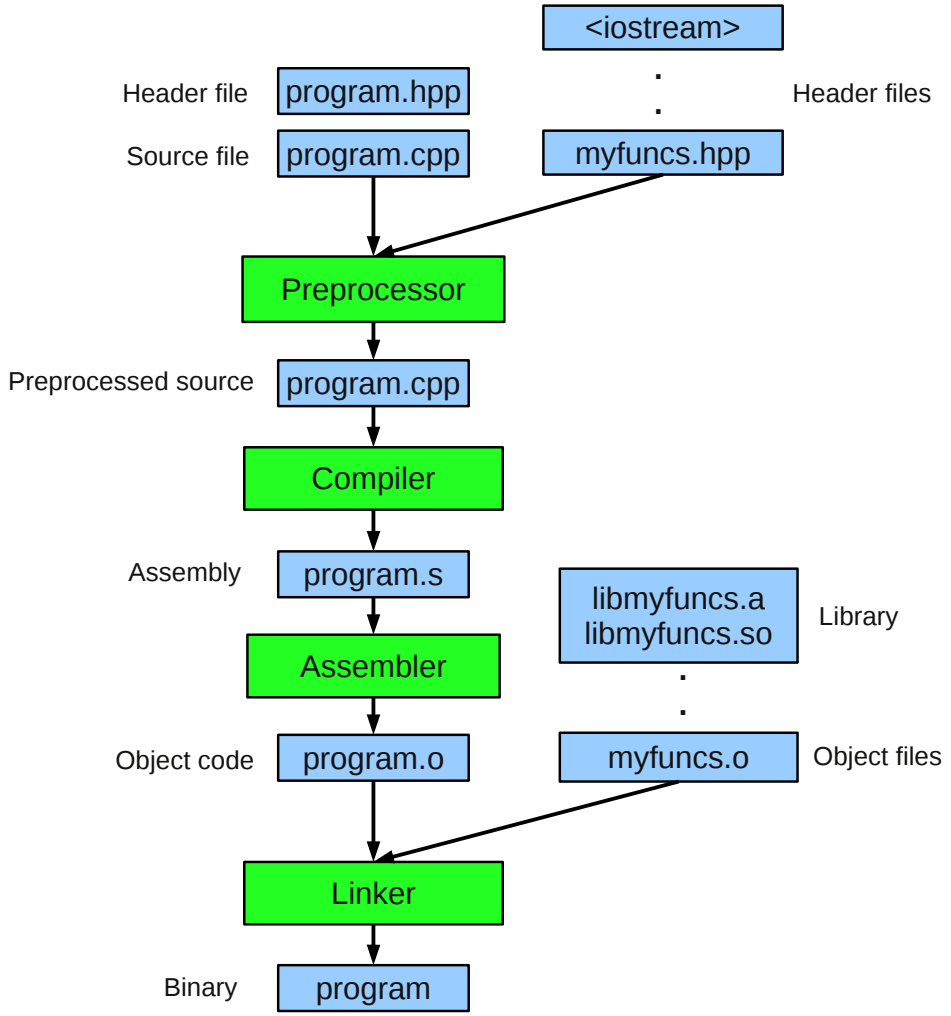
- Definition

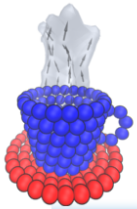
```
int a;
```

Stage 1: Preprocessor Inclusions 2

```
g++ -E -I<path> <source>
```

- The header files have to be in the *include path*
  - Usually called `.../include` on a Unix system.
  - The default include path contains *e.g.* `/usr/include`.
  - It can be extended in the compiler call by `-I <path>`.
  - **`#include`** `<file>` for system headers
  - **`#include`** `"file.hpp"` for own headers





Preprocessor Compile Time Guards

- If a header is included several times, this prevents multiply definitions of types

```
#ifndef __MYHEADER_HPP
#define __MYHEADER_HPP
.
// The actual code
.
#endif
```

- Or they can be used for conditional compilation, *e. g.* when a program can use a library if it is there, but can still work if not

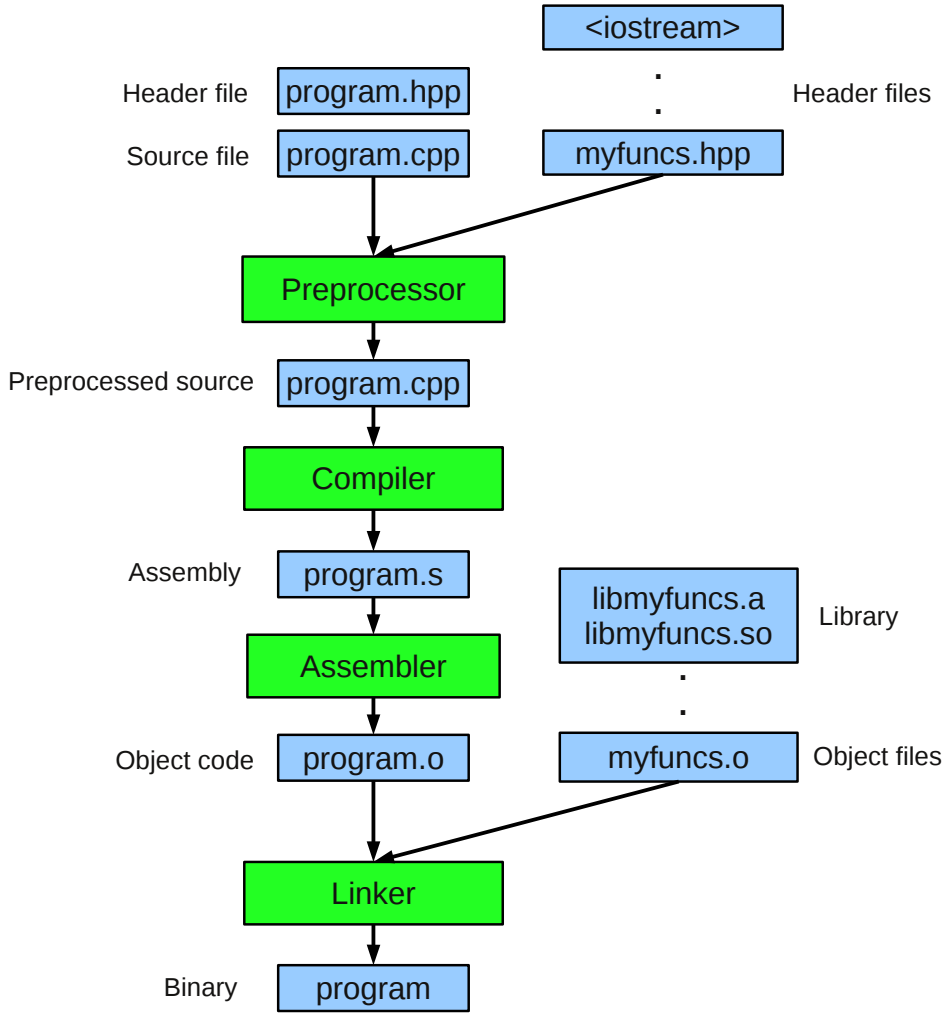
```
#ifdef FFTW
// code that uses FFTW
#else
// code without FFTW
#endif
```

Stage 2: Compiler

- Performs *static type checking*: do the types match?
- The *code generator* translates source code into machine code, function by function
- An *optimizer* optimizes the machine code
- Complains when a symbol is used that has not been declared
- . . . but does *not* complain when it is not defined!
- For each *defined symbol* in the code, it will store the generated machine code together with the symbol
- For each *undefined symbol* that is used in the code and that was only declared but not defined, it will store where it is used

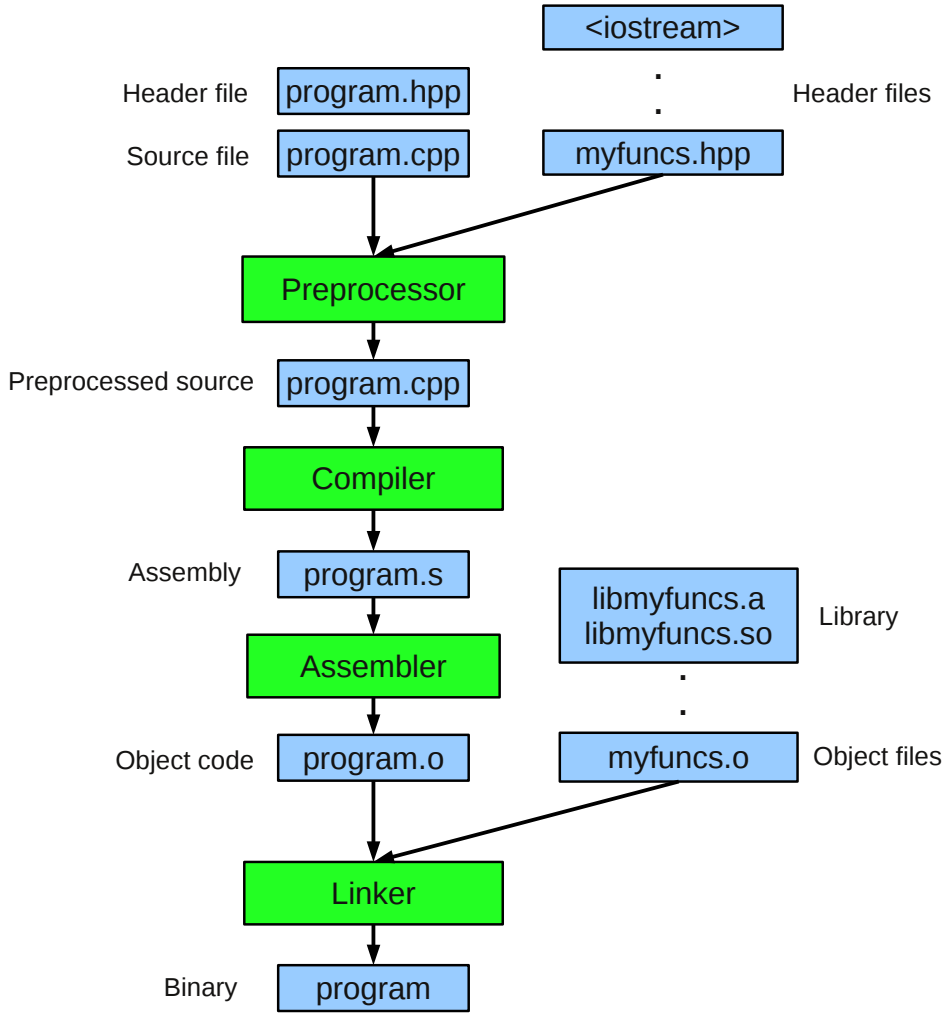
Stage 2: Compiler 2

- It generates `.o`-files (*object code*-files) (nothing to do with OOP objects!)
- `g++ -c FILE`: Preprocess and compile FILE, output `.o`-file
- `nm -C FILE.o`: Shows defined and undefined symbols in `.o`-file



Stage 3: Linker

- *Links* several `.o`-files together into a single executable file
- Starts with the function `main`
- Recursively *resolves* all undefined symbols
  - puts the code of used symbols into the executable
  - puts the addresses of the symbol where the symbol is used
- Fails when a symbol cannot be resolved



Libraries

- `ar` can put several object files together into a *library*
- The name is `lib<name>.a` or `lib<name>.so` (*e. g.* `libgsl.a`)
- A library file should come together with a set of header files (*e. g.* gsl.h)
- A library file can be linked together with other object files via the compiler option `-l<name>` (*e. g.* `-lgsl`)