Programming Techniques for Scientific Simulations I

Preprocessing/Compiling/Linking

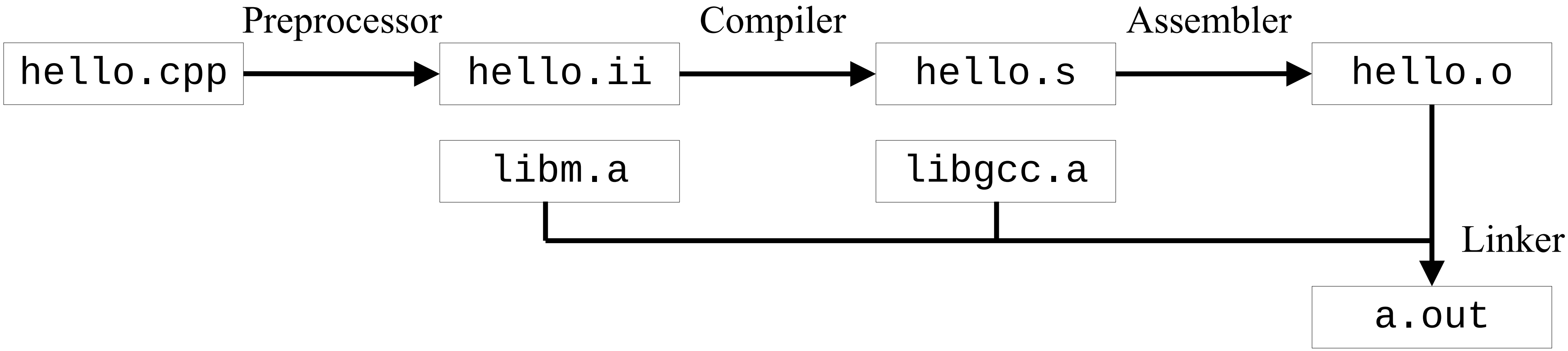
Steps when compiling a program

- What happens when we type the following?

```
$ c++ hello.cpp
```

- Observe the steps by adding some extra flags:

```
$ c++ --verbose -save-temps hello.cpp
```



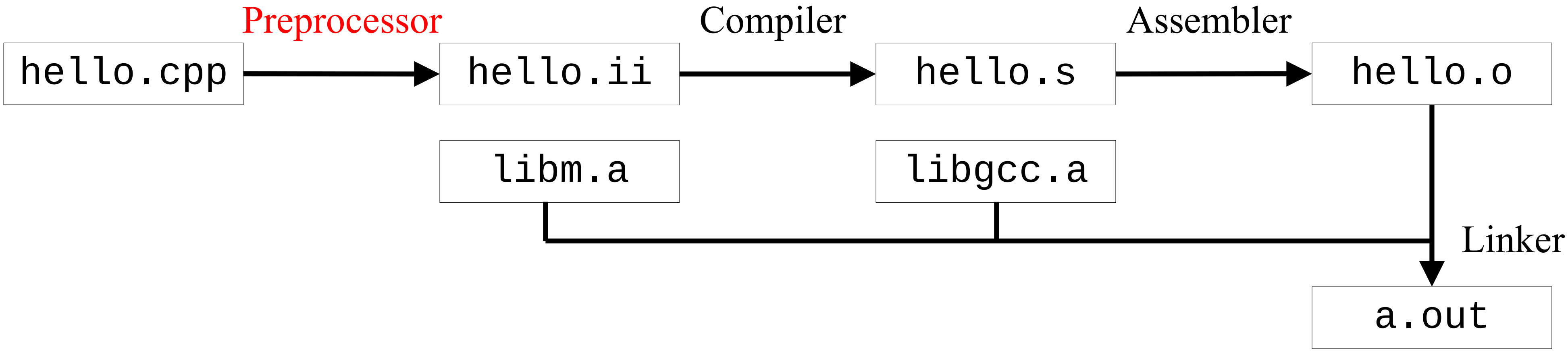
Steps when compiling a program

- What happens when we type the following?

```
$ c++ hello.cpp
```

- Observe the steps by adding some extra flags:

```
$ c++ --verbose -save-temps hello.cpp
```



The C++ preprocessor

- Is a simple text processor, manipulating the source code
- Commands start with #

```
#define XXX
#define YYY 1
#define ADD(A,B) A+B
#undef ADD
#ifdef XXX
#else
#endif
#if defined(XXX) && (YYY==1)
#elif defined (ZZZ)
#endif
#include <iostream>
#include "square.h"
```

#define

- Defines a preprocessor macro

```
#define XXX "Hello"
std::cout << XXX;
```

→

```
std::cout << "Hello"
```

- Macro arguments are possible

```
#define SUM(A,B) A+B
std::cout << SUM(3,4);
```

→

```
std::cout << 3+4;
```

- Definitions on the command line possible

```
$ c++ -DXXX=3 -DYYY
```

equivalent to

```
#define XXX 3
#define YYY
```

#undef

- Undefine a macro

```
#define XXX 4
x = XXX;
#undef XXX
x = XXX;
```

→

```
x = 4;
x = XXX;
```

- Undefines on the command line are also possible

```
$ c++ -UXXX
```

  - Is the same as writing in the first line:

```
#undef XXX
```

Looking at preprocessor output

- Running only the preprocessor:

```
$ c++ -E
```

- Running the full compile process but storing the intermediate files

```
$ c++ -save-temps
```

- Look at the files `pre1.cpp` and `pre2.cpp`, then at the output of

```
$ c++ -E pre1.cpp
$ c++ -E pre2.cpp
$ c++ -E -DSCALE=10 pre2.cpp
```

#ifdef … #endif

- Conditional compilation can be done using #ifdef

```
#ifdef SYMBOL
something
#else
something_else
#endif
```

```
something

something_else
```

- Look at the output of

```
$ c++ -E pre3.cpp
$ c++ -DDEBUG -E pre3.cpp
```

#if … #elif … #endif

- Allows more complex instructions, e.g.

```
#if !defined (__GNUC__)
std::cout << " A non-GNU compiler";
#elif __GNUC__<=2 && _GNUC_MINOR < 95
std::cout << "gcc before 2.95";
#elif __GNUC__==2
std::cout << "gcc after 2.95";
#elif __GNUC__>=3
std::cout << "gcc version 3 or higher";
#endif
```

#error

- Allows to issue error messages

```
#if !defined(__GNUC__)
#error This program requires the GNU
compilers
#else
...
#endif
```

- Try the following

```
$ c++ -c pre4.cpp
```

`#include “file.h” #include <iostream>`

- Includes another source file at the point of invocation
- Try the following

```
$ c++ -E pre5.cpp
```

- `< >` brackets refer to system files, e.g. `#include <iostream>`

```
$ c++ -E pre6.cpp
```

- With `-I` you tell the compiler where to look for include files. Try

```
$ c++ -E pre7.cpp
$ c++ -E -Iinclude pre7.cpp
```

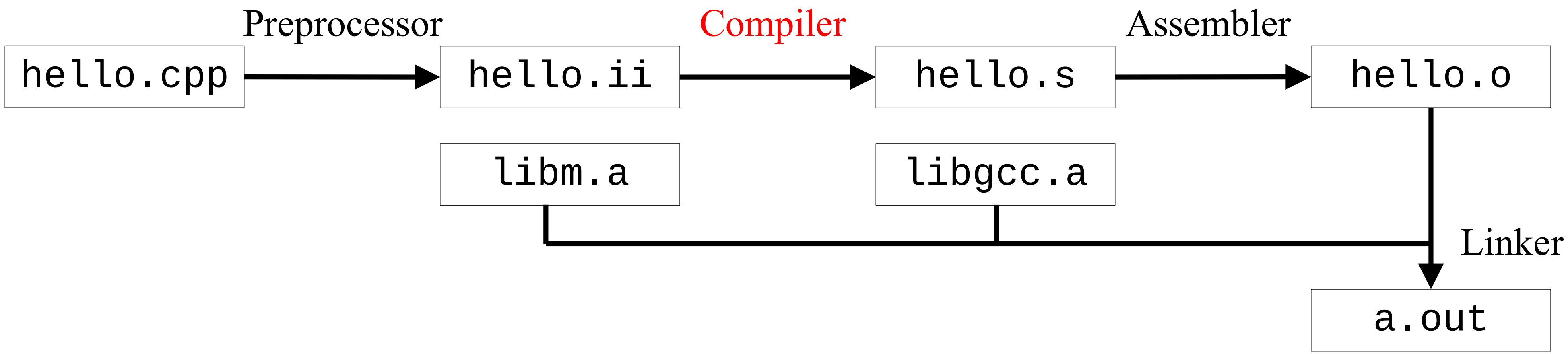
Steps when compiling a program

- What happens when we type the following?

```
$ c++ hello.cpp
```

- Observe the steps by adding some extra flags:

```
$ c++ --verbose -save-temps hello.cpp
```



Looking at the compilation output

- Let us look at the assembly code of a simple example

```
$ c++ -S -O0 functioncall.cpp
$ c++ -S –O3 functioncall.cpp
```

Invoke preprocessor and compiler

- Look at `functioncall.s` - What can you observe?
  - Can you observe automatic “inlining”?
- Try maybe with

Pipe, see e.g. https://swcarpentry.github.io/shell-novice/04-pipefilter/index.html

```
cat functioncall.s | c++filt | less
```

- Demangles C++ symbols… More on that later…
- Checkout the compiler explorer: https://godbolt.org/

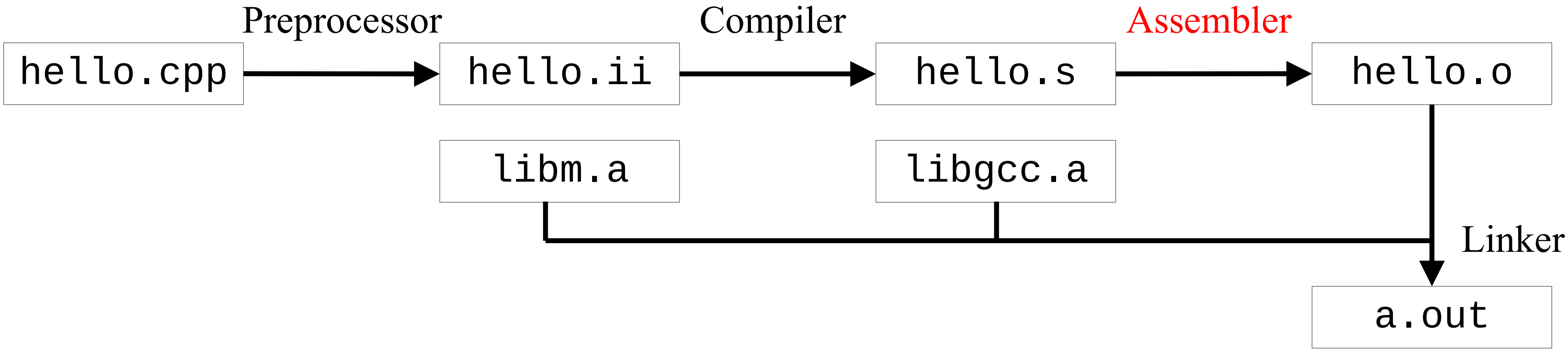
Steps when compiling a program

- What happens when we type the following?

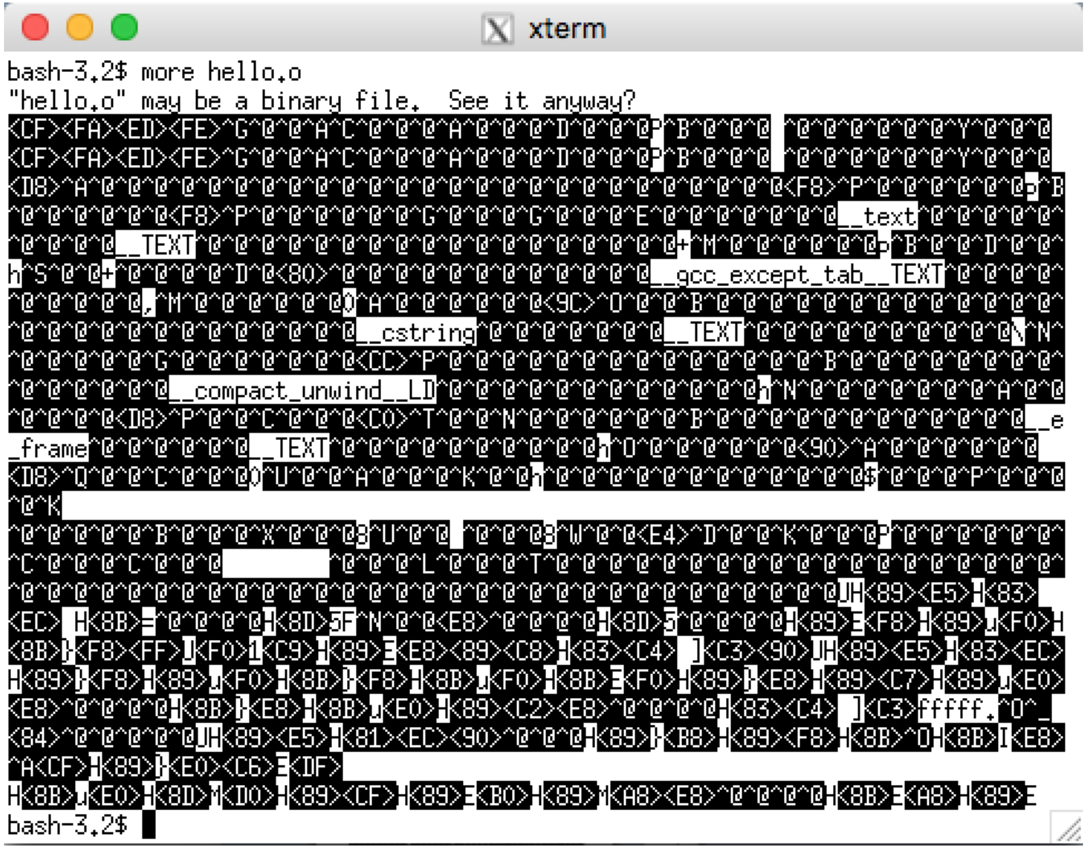
```
$ c++ hello.cpp
```

- Observe the steps by adding some extra flags:

```
$ c++ --verbose -save-temps hello.cpp
```



Looking at the assembler output



Steps when compiling a program

- Summary

American Standard Code for Information Interchange, i.e. a text file

  - Preprocessor --> _.ii ASCII intermediate file
  - Compiler --> _.s ASCII intermediate file
  - Assembler --> _.o (relocatable) object file
  - Linker --> _.exe executable object file

Segmenting programs

- Programs can be
  - split into several files
  - compiled separately
  - and finally linked together
- However functions **defined** in another file have to be declared before use!
- The **function declaration** is similar to the definition
  - but has no body
  - parameters need not be given names
- Easiest solution are header files. Help maintain consistency

- `square.hpp`

```cpp
double square(double);
```

- `square.cpp`

```cpp
#include "square.hpp"
double square(double x) {
  return x*x;
}
```

- `main.cpp`

```cpp
#include <iostream>
#include "square.hpp"
int main() {
  std::cout << square(5.)
            << std::endl;
  return 0;
}
```

Compiling and linking

- Compile the file `square.cpp`, with the `-c` option (no linking)

```
$ c++ -c square.cpp
```

- Compile the file `main.cpp`, with the `-c` option (no linking)

```
$ c++ -c main.cpp
```

- Link the object files

```
$ c++ main.o square.o
```

- Link the object files and name it, e.g `square`

```
$ c++ main.o square.o -o square
```

Include guards

- Consider file `grandfather.h`

```
struct foo {
  int member;
};
```

- and file `father.h`

```
#include "grandfather.h"
```

- and finally `child.cpp`

```
#include "grandfather.h"
#include "father.h"
```

- What happens here?

```
$ c++ -c child.cpp
```

Include guards

- Consider file `grandfather.h`

```
#ifndef GRANDFATHER_H
#define GRANDFATHER_H
struct foo {
  int member;
};
#endif /* GRANDFATHER_H */
```

- and file `father.h`

```
#include "grandfather.h"
```

- and finally `child.cpp`

```
#include "grandfather.h"
#include "father.h"
```

- Works!

```
$ c++ -c child.cpp
```

Assert in header `<cassert>`

- Are a way to check preconditions, postconditions and invariants
- `<cassert>` looks something like

```
#ifdef NDEBUG
#define assert(e)       ((void)0)
#else
#define assert(e)
/* implementation defined */
...
#endif
```

- If the expression is false the program will abort and print the expression with a notice that this assertion has failed
- Try it

```
$ c++ assert.cpp
$ c++ -DNDEBUG assert.cpp
```

Libraries

- Collection of useful functions
- Come in two kinds
  - Static libraries aka archives: **`lib*.a`** (`a`: archive)
    At link time, only the used functions from the archive are copied into the executable. (Win: *.lib)
  - Shared libraries: **`lib*.so`** (`so`: shared object)
    The functions from the library are not copied into the executable. Instead the library is loaded only once into memory where it can be used by any executable. Hence shared.
    (Win: Dynamic-Link Library *.dll)

Making a static library

- Compile the sources into object files

```
$ c++ -c square.cpp
```

- Pack the `*.o` object files into a static library with

```
$ ar -crs libsquare.a square.o
```

See man page!

Name of library

Object file(s) to put into the library

- `ar` creates an archive, more than one object file can be specified
  - The name must be `libsomething.a`

Will see why when linking!

- Voila! We have a static library!

Making a shared library

- Compile the sources into Position Independent Code (PIC) object files

```
$ c++ -fPIC -c square.cpp
```

- Pack the *.o object files into a shared library with

```
$ c++ -shared -fPIC -o libsquare.so square.o
```

- The compiler creates a shared object file, more than one object file can be specified
  - The name must be `libsomething.so`

Will see why when linking!

- Voila! We have a shared library

How to use libraries

- Compiling the main

```
$ c++ -c –Ilib main.cpp
```

- After compilation the object files are linked

```
$ c++ -o square main.o -Llib -lsquare
```

- If there are undefined functions (e.g. `square`) the libraries are searched for the function, and the needed functions linked with the object files
  - `-I` specifes the directory where the header file is located
  - `-L` specifes the directory where the library is located
  - `-lsomething` specifies looking in the library `libsomething.a`
- Note that the order of libraries is important
  - if `libA.a` calls a function in `libB.a`, you need to link in the right order: `-lA -lB`

Documenting your library

- After you finish your library, document it with
  - **Synopsis** of all functions, types and variables declared
  - **Semantics**
    - What does the function do?
  - **Preconditions**
    - What must be true before calling the function
  - **Postconditions**
    - What you guarantee to be true after calling the function if the preconditions were true
  - **Dependencies**
    - What does it depend on?
  - **Exception guarantees** (will be discussed later)
  - **References** or other additional material

Example documentation

- The function square:
  - **Synopsis**: `double square(double);`
  - **Semantics**: `square` calculates the square of x
  - **Preconditions**: the square can be represented in a double
    ```
    std::abs(x) <= std::sqrt(std::numeric_limits<double>::max())
    ```
  - **Postconditions**: the square root of the return value agrees with the absolute value of x within floating point precision
    ```
    std::sqrt(square(x)) - std::abs(x) <= std::abs(x)*std::numeric_limits<double>::epsilon()
    ```
  - **Dependencies**: None.

    We will discuss such things later…
  - **Exception guarantees**: no-throw.
  - **References**: None.

After a while it becomes tedious...

- Consider

```
$ c++ -c a.cpp
$ c++ -c b.cpp
$ c++ -c c.cpp
...
$ c++ main.o a.o b.o c.o ... –I... -L... –l...
```

- Change something
  - Need to keep track of who depends on who!
- Or: Build systems!!!
  - make
  - CMake
  - (SCons, …)