An Introduction to Programming

through C++

with

Manoj Prabhakaran

Lecture 10

Anatomy of a Program

Pre-Processor, Scopes, Namespaces

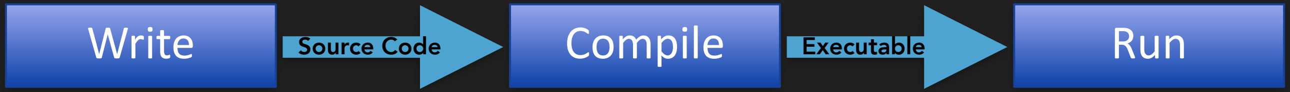
Based on material developed by Prof. Abhiram G. Ranade

Today

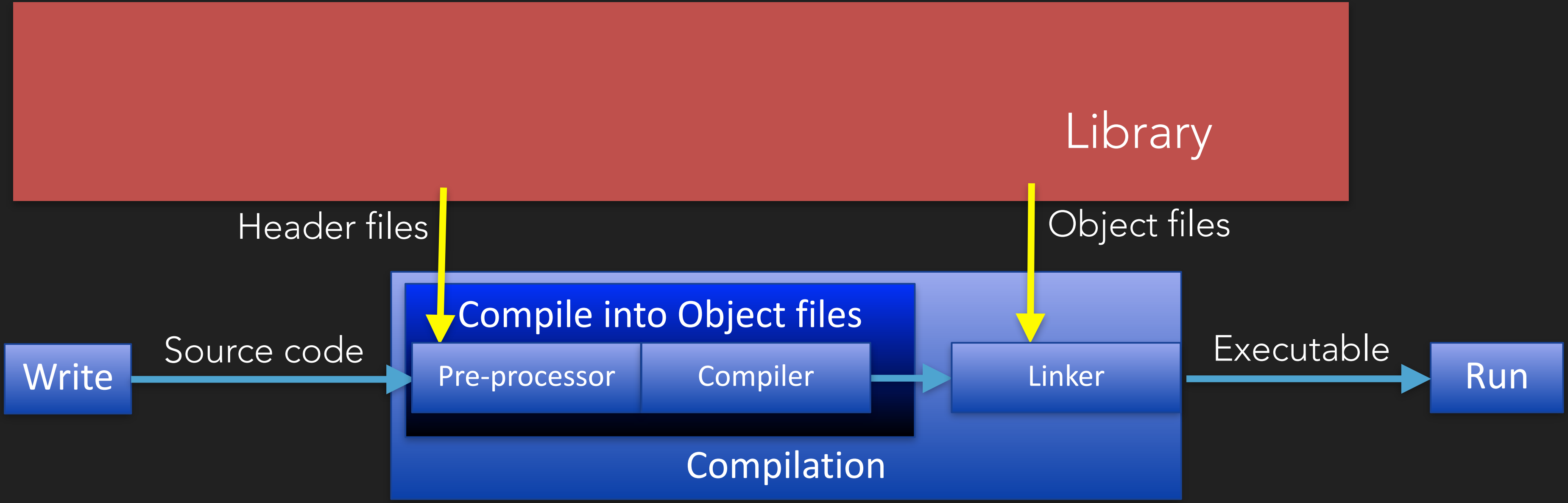
- Pre-processing
  - In particular, header files
- Scope of variables
- Namespaces

Reference: Chapter 11 (except 11.7)

Compiling a Program

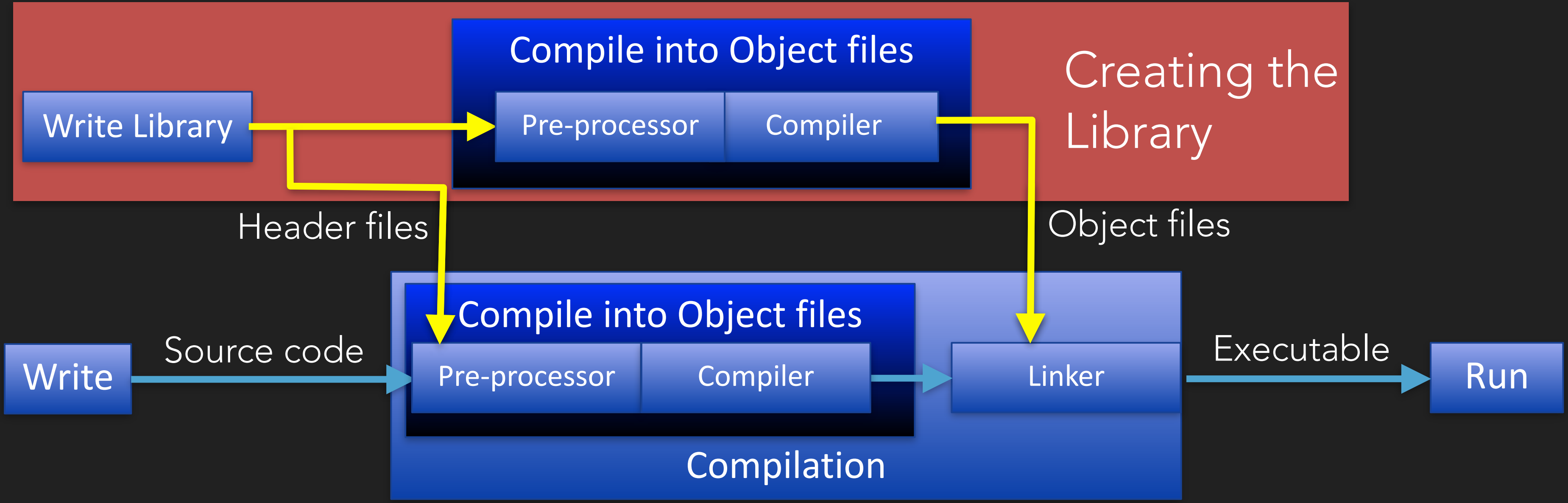


Compiling a Program



- *Header files* typically have the declarations of the functions (and more) in the library
- *Object files* are the binary compiled version of functions
    - It saves time to have the library functions pre-compiled

Compiling a Program



- *Header files* typically have the declarations of the functions (and more) in the library
- *Object files* are the binary compiled version of functions
  - It saves time to have the library functions pre-compiled

Pre-Processing Steps

- Some text transformations
    - Line ending with a \ is merged with the following line (example later)
    - Comments are stripped
        - Line comments: // comment till end of line
        - Block comments: /* comment spread
          over multiple lines */
- #include and other pre-processor directives (coming up) are processed, line-by-line
    - Processing one directive can result in the appearance of another directive. They are processed until no more directives are present.
        - But same directive is not applied twice (to avoid infinite invocations)

#include

Demo

numbers.h

```
int GCD(int, int);
int LCM(int, int);
bool coprimes(int,int);
bool covers(int w, int x);
bool PFE(int w, int x);
int reduce(int w, int x);
```

main.cpp

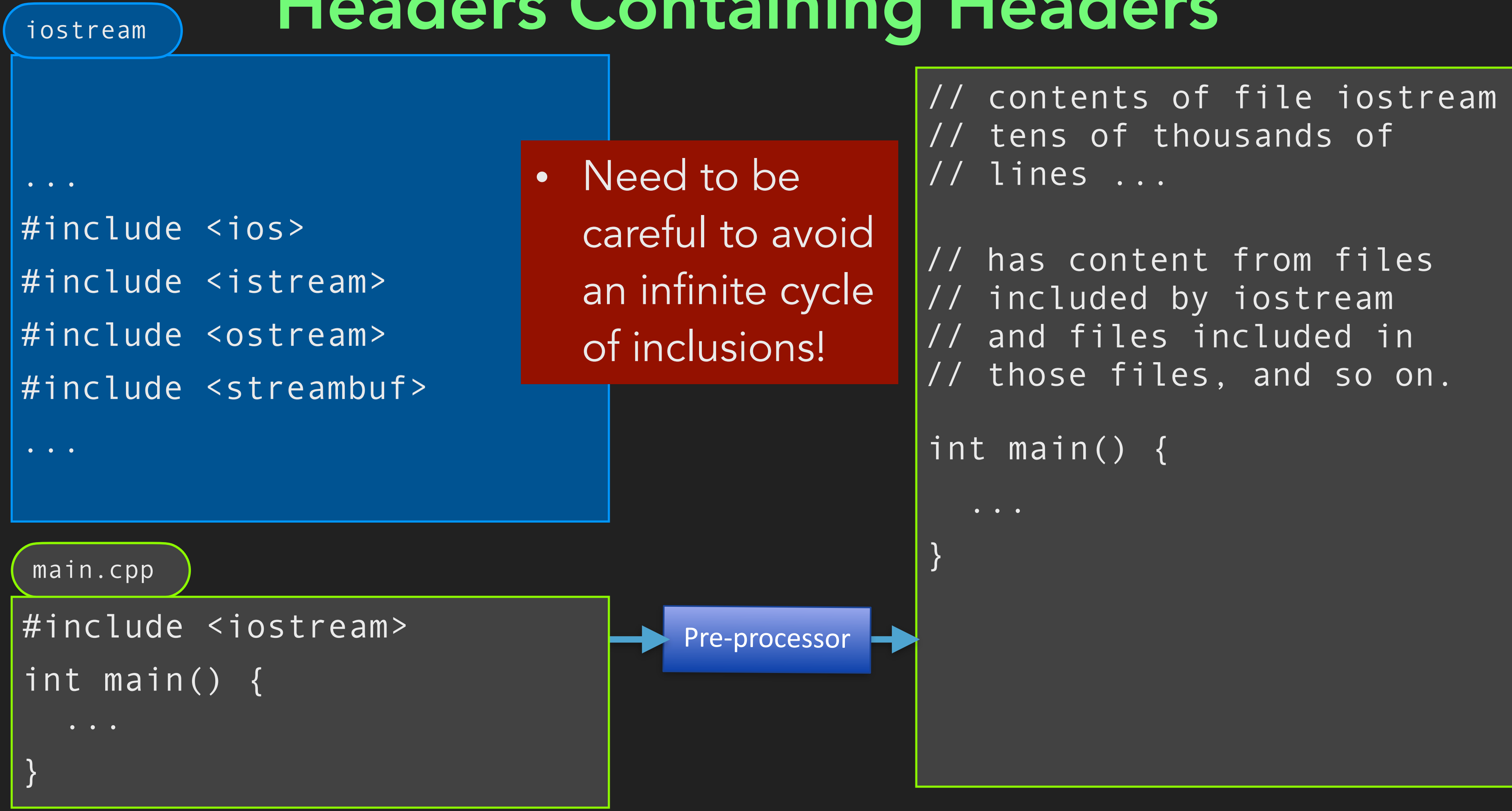
```
#include <iostream>
#include "numbers.h"

int main() {
   ...
}
```

```
$ g++ -E -P main.cpp
```

Pre-processor

```
// contents of file iostream
// tens of thousands of
// lines ...


int GCD(int, int);
int LCM(int, int);
bool coprimes(int,int);
bool covers(int w, int x);
bool PFE(int w, int x);
int reduce(int w, int x);


int main() {
   ...
}
```

Headers Containing Headers



iostream

```
...
#include <ios>
#include <istream>
#include <ostream>
#include <streambuf>
...
```

- Need to be careful to avoid an infinite cycle of inclusions!

main.cpp

```
#include <iostream>
int main() {
   ...
}
```

```
// contents of file iostream
// tens of thousands of
// lines ...

// has content from files
// included by iostream
// and files included in
// those files, and so on.

int main() {
   ...
}
```

Headers Containing Headers

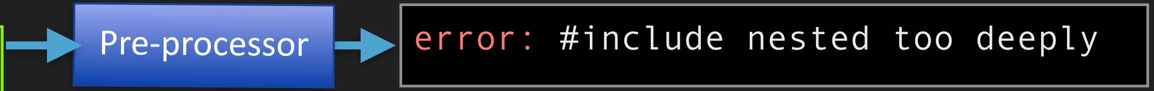
inc.h

```
#include "inc.h"
```

- Need to be careful to avoid an infinite cycle of inclusions!

main.cpp

```
#include "inc.h"
int main() {
   ...
}
```



```
error: #include nested too deeply
```

- There are pre-processor directives that can be used for conditional inclusion: coming up

#define

- #define VARIABLE *value*

  makes the pre-processor replace the text VARIABLE with the text *value* (when appearing as a “token” — e.g., not inside a string literal)

  ```
  #define DELTA 1e-6
  #define main_program int main()
  #define DEBUG_ENABLED
  ```

- “Macros” with parameters can be defined too.

```
#define CLOSE(x,y)  (abs((x)-(y)) <= DELTA)

#define repeat(X) for(int _RPT_i = 0, _RPT_n = X; \
                      _RPT_i < _RPT_n;  ++_RPT_i )
```

#ifdef and friends

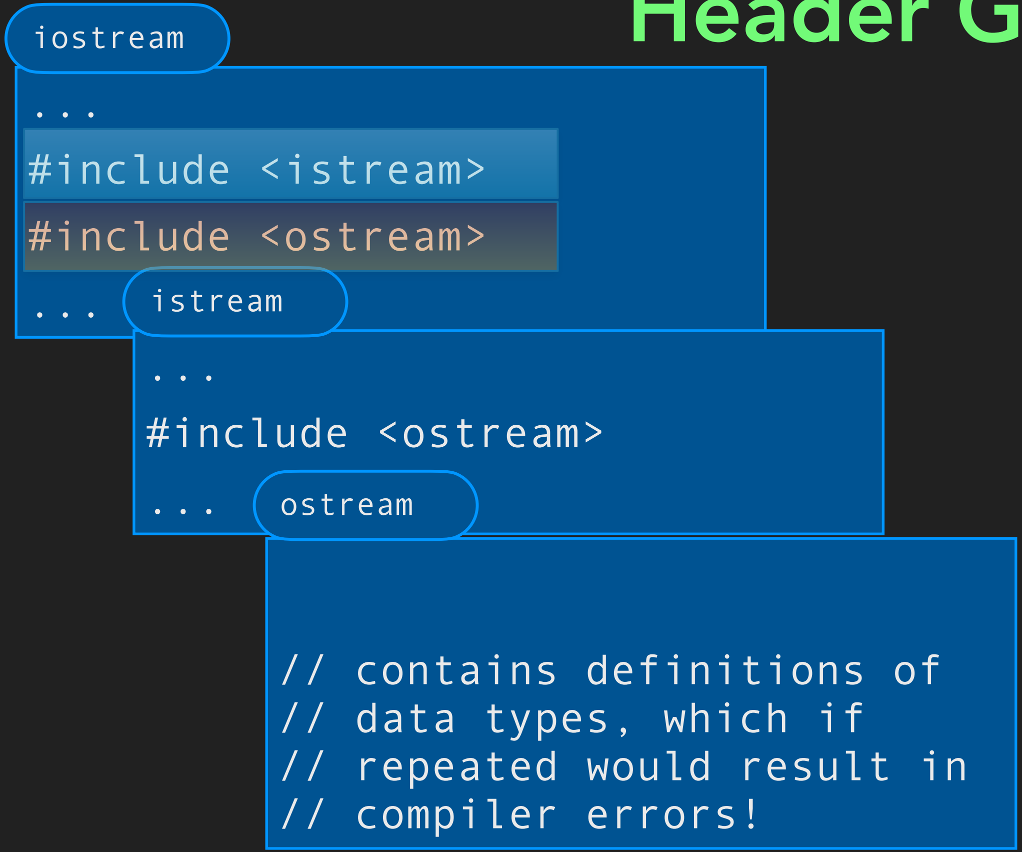
- #ifdef (alt: #if defined) or #ifndef (alt: #if !defined) to conditionally include code based on whether a macro has been defined

```
#define DEBUG_ENABLED  // value is optional
...
#ifdef DEBUG_ENABLED
#define LOG(x) cerr << x << endl
#else
#define LOG(x) // ignore
#endif
...
LOG("Some problem");
```

- Also: #if *expression* where *expression* is an integer constant expression

```
#if __cplusplus < 201103  // __cplusplus gives C++ version
#error Please use -std=c++11 option while compiling // Compilation aborts
#endif
```

Header Guards



iostream

```
...
#include <istream>
#include <ostream>
...
```

istream

```
...
#include <ostream>
...
```

ostream

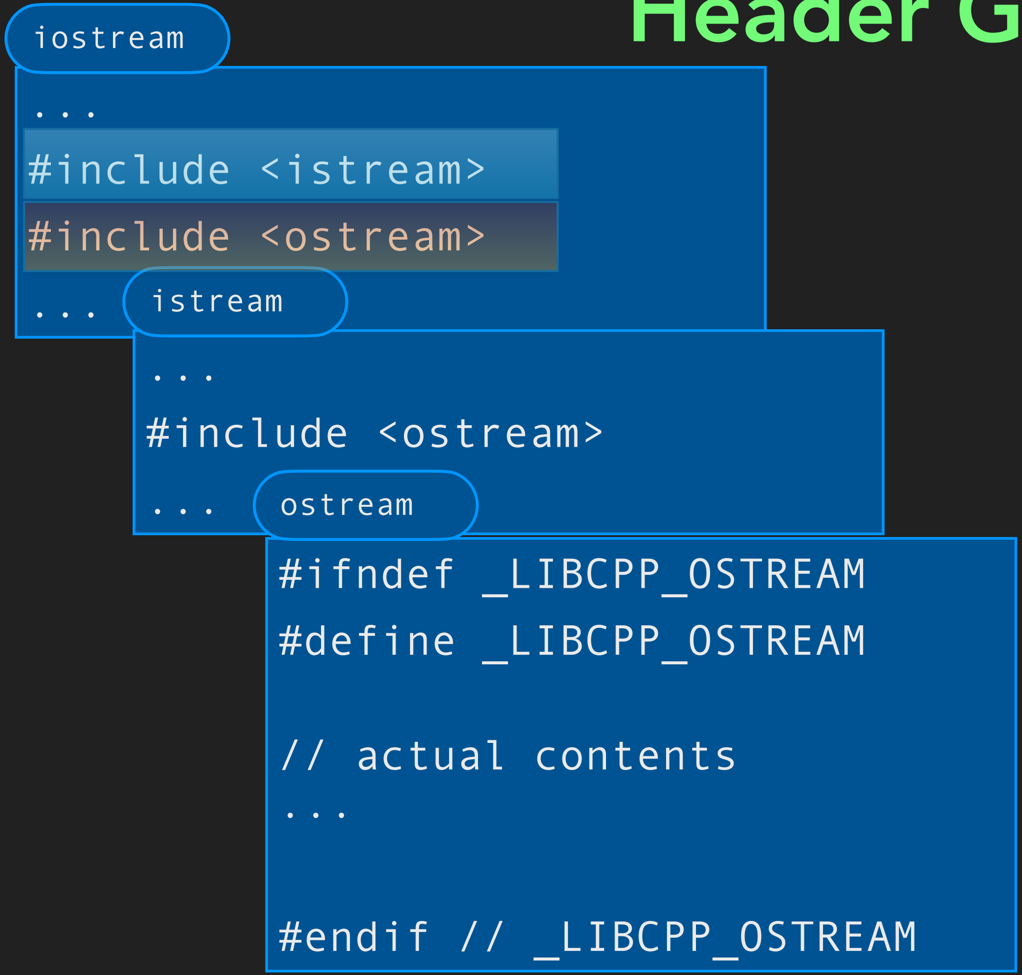
```
// contains definitions of
// data types, which if
// repeated would result in
// compiler errors!
```

```
// including <istream>
// including <ostream> as
// required in <istream>

// remaining contents of
// istream included

// including <ostream> as
// required in <iostream>
// remaining contents of
// <iostream> included
```

Stop!

Cannot redeclare same variables, data types (structs) default arguments, etc.!

Header Guards



iostream

```
...
#include <istream>
#include <ostream>
...
```

istream

```
...
#include <ostream>
...
```

ostream

```
#ifndef _LIBCPP_OSTREAM
#define _LIBCPP_OSTREAM

// actual contents
...


#endif // _LIBCPP_OSTREAM
```

```
// including <istream>

// including <ostream> as
// required in <istream>


// define _LIBCPP_OSTREAM
// and include contents of
// ostream

// remaining contents of
// istream included

// _LIBCPP_OSTREAM is defined
// so #ifndef,,,#endif skipped

// remaining contents of
// <iostream> included
```

Header Guards

inc.txt

```
#ifndef _INC_DONE
  #ifdef _INC_ALMOST_DONE
    #define _INC_DONE
  #else
    #define _INC_ALMOST_DONE
  #endif
hello
#include "inc.txt"
bye
#endif
```

Exercise: Explain how this happens

main.cpp

```
Testing preprocessor.
Not a valid program.
#include "inc.txt"
```

Pre-processor

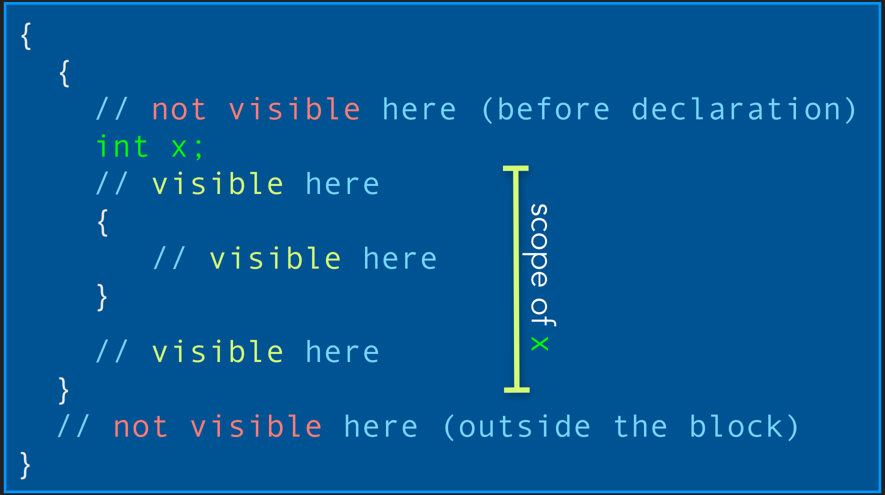
```
Testing preprocessor.
Not a valid program.
hello
hello
bye
bye
```

Source File after Pre-Processing

- After pre-processing a source file has any number of:
    - Declarations (global variables and functions)
    - Struct definitions
    - Function definitions (and templates)
    - (More later)
- A function definition has:
    - Return type, function name, and parameter list
    - Followed by statements enclosed in { … }
    - Different kinds of statements (declaration with or without initialisation, *expression*; , conditional statement, conditional loop statement, break, continue, and return statements, …)
- Compiler produces a single object file for each such source file

Scope of Variables

- In C++, a variable can be used only where its declaration is "visible"
  - Visible only <u>within the "block"</u> it is declared in
  - And only <u>after</u> it is declared
  - Scope of a variable: region in the code where it is visible



```
{
  {
    // not visible here (before declaration)
    int x;
    // visible here
    {
       // visible here
    }

    // visible here
  }
  // not visible here (outside the block)
}
```

- A variable cannot be declared twice within the same block
  - However can declare a new variable with the same name (but possibly a different type) in a "sub-block"
  - In its scope, the new variable "shadows" the old one

Scope of Variables

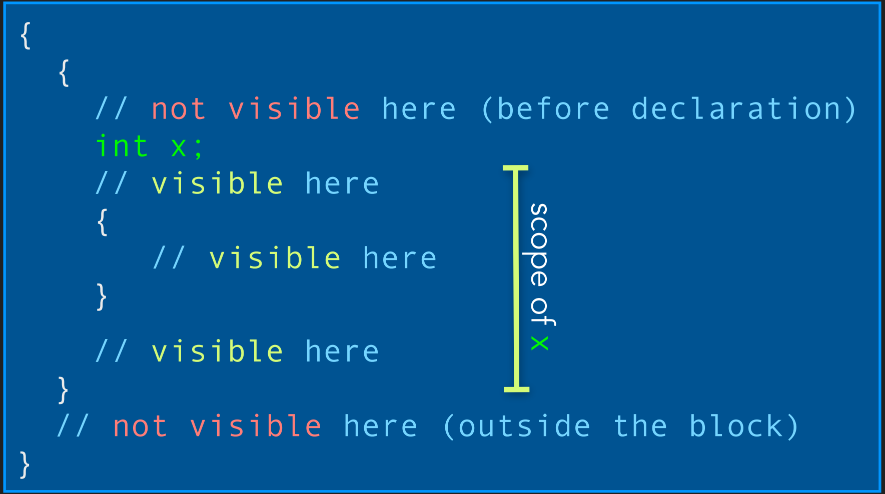
```
void f(int x) {
  ...
}
```

```
{
  ...
}
```

```
for(int x=0;;) {
  ...
}
```

```
while(condition) {
  ...
}
```

```
if(condition) {
  ...
}
```



```
{
  {
    // not visible here (before declaration)
    int x;
    // visible here
    {
      // visible here
    }

    // visible here
  }
  // not visible here (outside the block)
}
```

- A few different kinds of blocks (more later):
  - A function’s body (including parameter declarations)
  - A block of statements enclosed in braces
  - A for loop (including declarations in the initialisation)
  - A while or do-while statement (condition can have declarations)
  - If-Else statement (condition can have declarations; visible in both if & else parts)

Scope of Variables

Demo

```
int g; // a global variable. remains visible till the end of the file
...
void f(int x) { // x is visible inside the body of the function
  int y; // visible from here till the end of the function
  for(int g=x; g<3; g–) { // a new local g! visible till
    ...                   // the end of the for statement.
  } // now this g goes out of scope. global g visible again.
  {  // start of a new scope
    g = x + 1;  // this refers to the global g
    float g; // this is a different g! global g not visible.
  }  // now this g goes out of scope. global g visible again.
  g++;  // global g
}       // here x, y go out of scope.
```

Namespaces

- Standard library contains useful functions (swap, max, min, distance, begin, end, sort, move, …), data types (string, vector, list, …) and global variables (cout, cin, …), many with common names
- But this can be problematic, especially due to function overloading!
- Suppose you write a function `to_string` as follows:

```
#include <simplecpp>
string to_string(short x) { return x==0 ? "zero" : "non-zero"; }
int main() {
    short a = 1; int b = 1;
    cout << to_string(a) << " vs. " << to_string(b) << endl;
}
```

invokes our to_string

invokes to_string from the standard library!

```
non-zero vs. 1
```

Namespaces

- To keep entities (functions, types, variables) in a library separate from ours
  - `to_string` vs. `std::to_string`
  - <simplecpp> has a statement `using namespace std;` which made all the entities in std namespace available without the qualifier `std::`
  - We shall instead use the standard header <iostream>

Risky!

```
#include <iostream>
std::string to_string(short x) { return x==0 ? "zero" : "non-zero"; }
int main() {
    short a = 1; int b = 1;
    std::cout << to_string(a) << " vs. " << to_string(b) << std::endl;
}
```

Invokes our `to_string`, with b cast into a short.
Only `std::to_string` invokes the one from the library.

Example

Demo

numbers.h

```cpp
namespace num {
int GCD(int, int);
int LCM(int, int);
bool coprimes(int,int);
bool covers(int w, int x);
bool PFE(int w, int x);
int reduce(int w, int x);
}
```

numbers.cpp

```cpp
#include "numbers.h"
#include <cmath>

int num::LCM(int a, int b) {
     return std::abs(a*b)/GCD(a,b); // GCD is num::GCD
}
bool num::coprimes(int a, int b) {
     return GCD(a,b) == 1;
}
...
```

prog.cpp

```cpp
#include <iostream>
#include "numbers.h"

using std::cout; using std::cin; using std::endl;

int main() {
  cout << "Enter 2 positive numbers: ";
  int a, b; cin >> a >> b;
  if (a<=0 || b<=0) return -1;
  cout << (num::PFE(a,b) ? "":"Not ") << "PFE" << endl;
  cout << "GCD(a,b) = " << num::GCD(a,b) << endl;
}
```

```
$ g++ -c prog.cpp       # this produces prog.o

$ g++ -c numbers.cpp    # this produces numbers.o

$ g++ prog.o numbers.o  # this produces a.out
```