Introduction to Programming (CS 101)

Spring 2024

Lecture 18:

Preprocessing, Header files, Header guards

Instructor: Preethi Jyothi

Based on material developed by Prof. Abhiram Ranade and Prof. Manoj Prabhakaran

Recap (I): Self-referential node

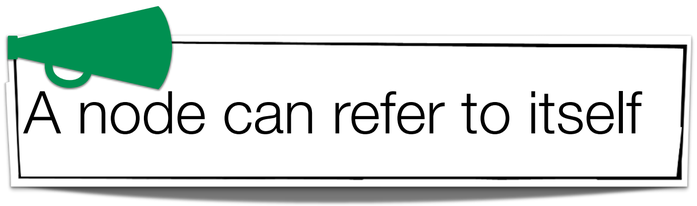
What is the output of this program?

```
#include <iostream>
using namespace std;

struct node {
  int val;
  node* next;
};

int main() {
  node* n1 = new node;
  n1->val = 11; n1->next = n1;
  cout << n1->val << " " << n1->next->val << endl;
  delete n1;
}
```

Compiler error

Undefined behaviour

11 11



Recap (II): new and delete Do you see any problems with this code?

```
struct node {
  int val;
  node* next;
};
```

```
node* traverse(node* head) {
  node* prev = nullptr;
  while(head) {
    prev = head;
    head = head->next;
  }
  return prev;
}
```

```
int main() {
  node* head = new node; head->val = 5;
  node* tail = new node; tail->val = 5; tail->next = nullptr; head->next = tail;
  head = traverse(head);
  delete head;
  delete tail;
}
```

`delete tail;` will try to delete a pointer that has already been deleted!

Recap (IIIA): Array of pointers

What is the output of the `cout` statement if this is run as: `./a.out hi there`?

```
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
  string result = "";
  for(int i = 1; i < argc; i++) {
    result += argv[i];
  }
  cout << result << endl;
}
```

OUTPUT: **hithere**

`argv[i]` refers to the null-terminated C-style string passed to the command line as the i^{th} argument

Recap (IIIB): Array of pointers

What is the output of the `cout` statement if this is run as `./a.out hi there`?

```
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
  string result = "";
  for(int i = 1; i < argc; i++) {
    result += *(argv[i]+i);
  }
  cout << result << endl;
}
```

OUTPUT: **ie**

`argv[i]+i` will access the address of the `i`th character within the `i`th command line argument

`vector` iterator

CS 101, 2025

`vector` iterator

- An iterator is designed to traverse through a `vector` or `array` and help provide access to each element

```
#include <iostream>
using namespace std;

int main() {
  vector<int> A = {1, 3, 5};
  vector<int>::iterator it;
  for(it = A.begin(); it != A.end(); it++)
    cout << *it << " ";
}
```

OUTPUT: 1 3 5

Building blocks of a program

CS 101, 2025

A program split across multiple files

- A program can be split across multiple files provided the following rules are satisfied:
  - Function declarations before definitions: If a function is being called by code in a file, then the function *must be declared* before any calls to it. (Note that a function definition is a declaration, but not vice-versa.)
  - Every function called must be defined exactly once in some file in the collection.

- Example:

```
#include <iostream>
using namespace std;
int lcm(int m, int n);

int main() {
  cout << lcm(36,24);
}
```

main.cpp

```
int gcd(int, int);

int lcm(int m, int n) {
  return m*n/gcd(m,n);
}
```

lcm.cpp

```
int gcd(int m, int n) {
  int tm, tn;
  while(tm % tn != 0) {
    tm = n;
    tn = m % n;
    m = tm; n = tn;
  }
  return n;
}
```

gcd.cpp

```
g++ main.cpp lcm.cpp gcd.cpp
```

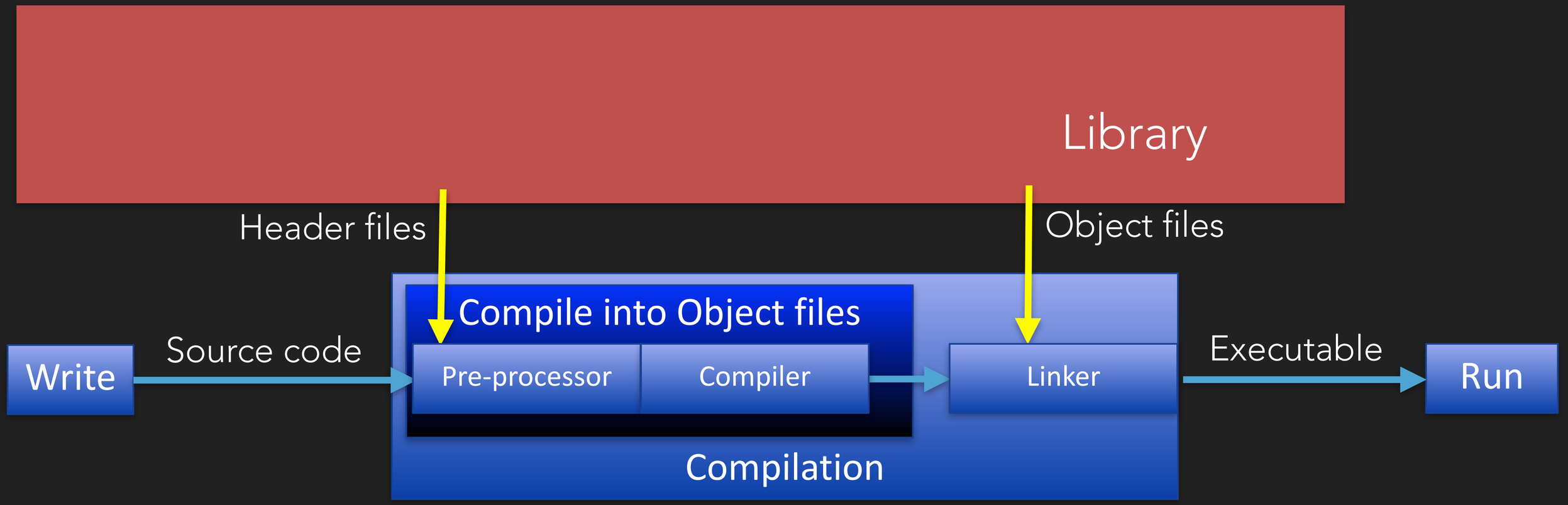
Separately compile each file

- Instead of compiling all files together to produce an executable, one can also compile each file separately using `g++ -c main.cpp`
  - This produces an *object module* `main.o`
  - Object modules are not executables

- We can form the executable `./a.out` from the object modules using:
  `g++ main.o lcm.o gcd.o`
  - The executable is produced via a *linker* that links the object modules together

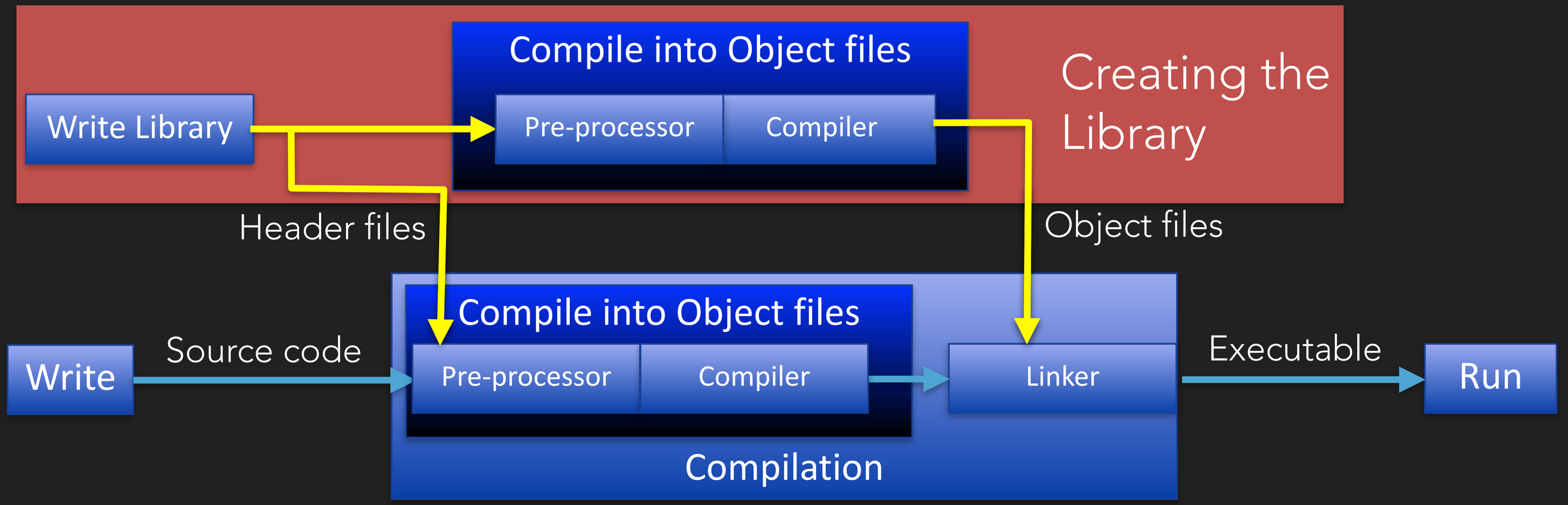
- One can mix `.cpp` files and `.o` files as arguments to `g++`. Example:
  `g++ main.cpp lcm.o gcd.o`

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

Preprocessing and Headers

CS 101, 2025

Pre-Processor Directives

- `#include "numbers.h"` is a *pre-processor directive*
  - Here, `numbers.h` is a *header file*

- The `#include` directive causes the contents of the header file to be placed at the position where the directive appears

- `#include` and other pre-processor directives 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

numbers.h

```
int GCD(int, int);
int LCM(int, int);
```

main.cpp

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

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

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


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


int GCD(int, int);
int LCM(int, int);


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

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

inc.h

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

main.cpp

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

- 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

- `#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");
```

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
```

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
```

Optional: 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
```

main.cpp

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

Exercise: Explain this output

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