Multi-File Projects

Multiple Files

- Programs of any size are much more easily managed if the project is split into multiple files
- Typical C++ programs will have many files comprising the project
- The files are compiled separately, then linked together
- Each file typically contains one (or a small number of closely related) class definition(s)
- Class definitions are separated from method implementations (.h vs. .cpp)

Header Files

- Used to separate *interface* from *implementation*
- Can be system defined or user defined
- Header files should be named *`filename`*`.h`
- Used for
  - `Class declarations`
  - `Enumerations`
  - `Function prototypes`
  - `Constant definitions`

Header Files

- Header files provide the interface so that other classes can use the classes without knowing how the classes are implemented
- Typically header files are provided along with object file libraries to outside users (rather than source code)
- Rule of thumb – place the class definition in the header file (.h), place the implementation of the class methods in the source file (.cpp)
- The header file is `#include`'ed into the source file

Example – Class StringStack

Header File StringStack.h

```
class StringStack {

public:
  StringStack(int);
  StringStack(const StringStack &);
  ~StringStack();
  StringStack & operator =(const StringStack &);
  bool IsEmpty() const;
  void Clear();
  void Push(const string &);
  string Pop();

protected:
  void Init(const StringStack & other);
  void Free();
  void Grow();

  int capacity;
  string * stack;
  int top;
};
```

Example – Class StringStack

Source File StringStack.cpp

```
#include “StringStack.h”

StringStack :: StringStack( int initialCapacity) {
    capacity = initialCapacity;
    stack = new string[capacity];
    top = 0;
}

StringStack :: StringStack(const StringStack & other) {
    Init(other);
}

StringStack :: ~StringStack() {
    Free();
}
```

Example – Class StringStack

Source File StringStack.cpp (continued)

```
StringStack :: StringStack & operator =(const StringStack &
   other) {
    if (&other != this) {
      Free();
      Init(other);
    }
    return *this;
}

bool StringStack :: IsEmpty() const {
    return (top == 0);
}

void StringStack :: Clear() {
    while (top > 0) {
      stack[--top] = string();
    }
}
```

Example – Class StringStack

Source File StringStack.cpp (continued)

```
void StringStack :: Push(const string & value) {
    if (top == capacity) {
      Grow();
    }
    stack[top++] = value;
}

string StringStack :: Pop() {
    if (IsEmpty()) {
      throw CS240Exception("can't pop an empty stack");
    }
    else {
      string value = stack[--top];
      stack[top] = string();
      return value;
    }
}
```

Example – Class StringStack

Source File StringStack.cpp (continued)

```
void StringStack :: Init(const StringStack & other) {
    capacity = other.capacity;
    top = other.top;
    stack = new string[capacity];
    for (int i=0; i < top; ++i) {
      stack[i] = other.stack[i];
    }
}

void StringStack :: Free() {
    delete [] stack;
    stack = 0;
}
```

Example – Class StringStack

Source File StringStack.cpp (continued)

```
void StringStack :: Grow() {
    int newCapacity = capacity * 2;
    string * newStack = new string[newCapacity];
    for (int i=0; i < top; ++i) {
      newStack[i] = stack[i];
    }
    Free();
    capacity = newCapacity;
    stack = newStack;
}
```

Multiple Inclusions

- Sometimes a .cpp file ends up indirectly including the same header file multiple times (e.g., A.cpp includes A.h and B.h, but A.h and B.h both include C.h)
- Multiple inclusion of a file produces multiple definitions of the same symbol, which is an error
- Resolve this by using a #define in the header file and checking to see if it is defined
- Include this at the beginning of EACH header file

Using #define

Header file StringStack.h

```
#ifndef STRINGSTACK_H
#define STRINGSTACK_H

class StringStack {

      // define the class here
};

#endif
```

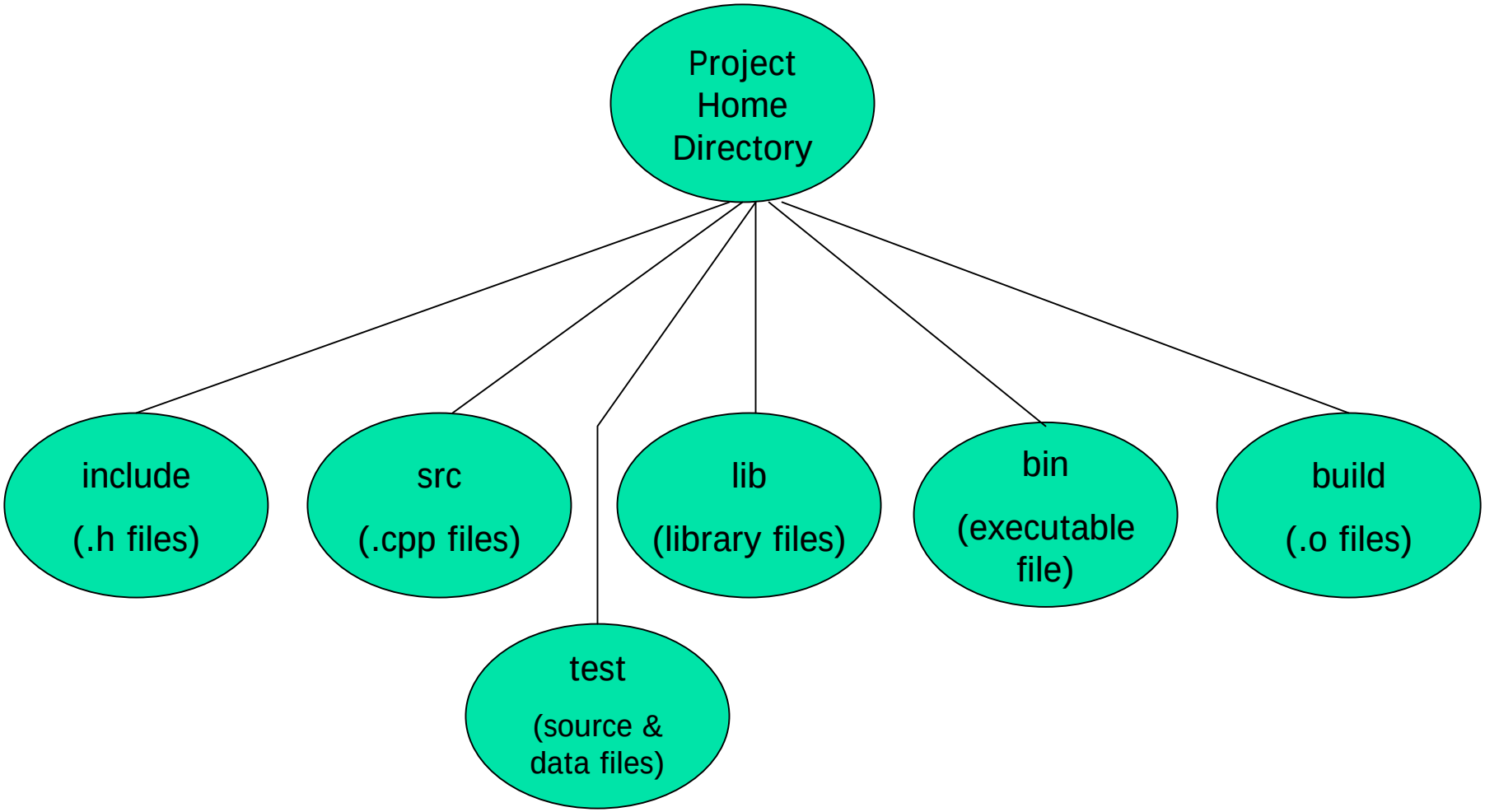
Managing the Files

- Since there will be a lot of different files in your project, keeping track of the files becomes more difficult
- Take advantage of the UNIX file system to assist you
- Create sub directories for
  - Source (.cpp) files
  - Header (.h) files
  - Libraries
  - The executable file
  - The object files
  - Test source and data files

Managing the Files

- So, for a student with home directory ~henry, with subdirectory cs240, with a separate directory for the web crawler:
  - The .cpp files would be in
    `~henry/cs240/webcrawler/src`
  - The .h files would be in
    `~henry/cs240/webcrawler/include`
  - The executable file would be in
    `~henry/cs240/webcrawler/bin`
  - The object files would be in
    `~henry/cs240/webcrawler/build`
  - The user created libraries would be in
    `~henry/cs240/webcrawler/lib`
  - The test source and data files would be in
    `~henry/cs240/webcrawler/test`

Project Directory Structure



Locating Header Files

- If .h files are not in the same directory as the .cpp files that include them, you need to tell the compiler where to find the .h files

```
cd ~/cs240/crawler/src
g++ -I../include -I../cs240utils/include
 -o crawler WebCrawler.cpp
```

Linking

- When compiling a single stand-alone .cpp file, the program is compiled and linked automatically
- When compiling multiple files, each .cpp file needs to be compiled separately, then after all compilation is done, the object code and libraries are linked together
- E.g. a project consisting of person.h, person.cpp, schedule.h, schedule.cpp, main.cpp

Linking Example

- Compile person.cpp (which `#include`'s `person.h`)

```
g++ -c person.cpp
```

- Compile schedule.cpp (which `#include`'s `schedule.h`)

```
g++ -c schedule.cpp
```

- Compile main.cpp

```
g++ -c main.cpp
```

- Link them all together

```
g++ -o scheduler person.o schedule.o main.o
```

Libraries

- C++ provides several libraries for your use
- Frequently used functions are placed in libraries so that they can be reused
  - i.e., math routines, string routines, I/O, etc.
- The user can also create libraries
- Generally used to combine several object files together into a central location
- The user can then link the library into his/her code and use the functions in the library

Creating Static Libraries

- The command to create statically linked libraries in `c++` is `ar`
- Syntax:

```
ar option archive members
```

option can be:

`r`	insert members into archive
`d`	delete members from archive
`t`	display the contents of the archive

UNIX ar Example

Create a library of the object files from the CS240 utilities:

```
ar r ../lib/libcs240utils.a FileInputStream.o
   HTTPInputStream.o ObjectCountBase.o
   StringUtil.o URLConnection.o
```

Linking Static Libraries

- To make calls to the functions in a library, the library must be linked into the executable
- This can be done in two ways

```
g++ -o ../bin/webcrawler Main.o Parser.o CreateHtml.o
  Index.o ../lib/libcs240utils.a
```

OR

```
g++ -o ../bin/webcrawler -L../lib Main.o Parser.o
  CreateHtml.o Index.o -lcs240utils

# g++ automatically prepends "lib" and appends ".a" to
# the name of the library
```

Compiler Options

`-c`	Compile the `C++` file into an object file but do not link
`-l`*`library`*	Link code from `library` into the executable
`-I`*`dir`*	Append `dir` to the list of directories to search for include files
`-L`*`dir`*	Append `dir` to the list of directories to search for libraries
`-o`*`file`*	Place the executable from the compilation into `file`
`-D`*`macro`*	Define `macro` to be `1`
`-D`*`macro=defn`*	Define `macro`, set it equal to `defn`
`-g`	Produce debugging information