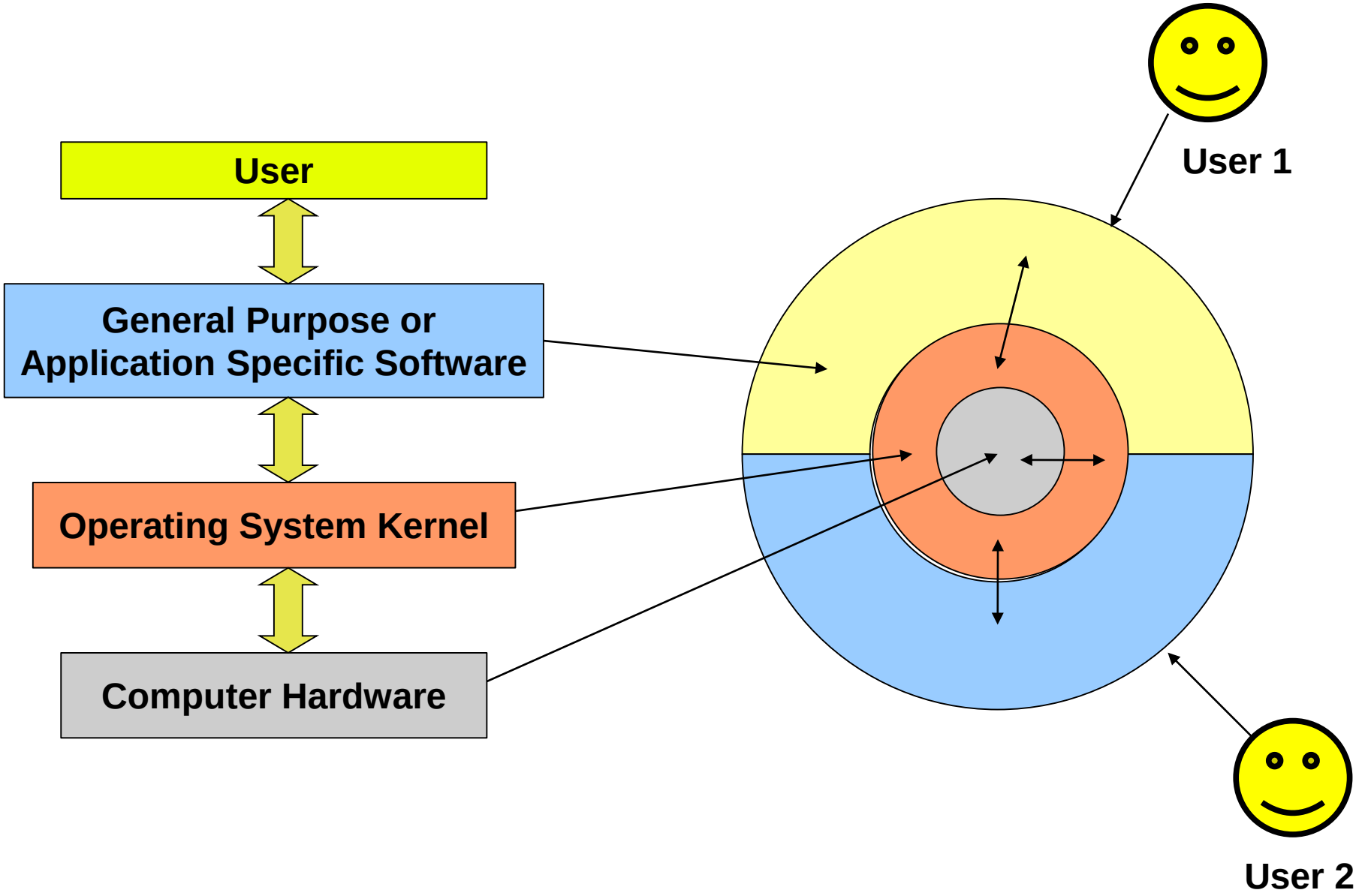
KYC - Know your compiler

Introduction to GCC

The Operating System



What is GCC?

GCC is the GNU Compiler Collection

Provides Compilers for:

- C
- C++
- Objective C
- Fortran
- ADA
- Java

Basic steps in compilation

- Pre-Process directives like #include
- Compilation of the C Code to generate the assembly
- Assembly Code to object file generation
- Link the object code to generate the executable

Possible Outputs for C/C++ Programs

- Complete processing:
  - Executable files
  - Libraries
- Partial processing:
  - C/C++ Source files after preprocessing only
  - Assembly code corresponding to a source file
  - Object code corresponding to a C/C++ file

Step 1: Compiling a simple C Program

- **Syntax:** gcc <filename.c>

```
user@ws$ gcc HelloWorld.c
```

- Output: An executable called a.out
- To run: ./a.out

```
user@ws$ ./a.out
Hello World
user@ws$
```

Step 2: Compiling a simple C++ Program

- **Syntax:** g++ <filename.cpp>

```
user@ws$ g++ HelloWorld.cpp
```

- Output: An executable called a.out
- To run: ./a.out

```
user@ws$ ./a.out
Hello World
user@ws$
```

Step 3: Providing the executable name

- **Extra option: -o**
- **Syntax:** gcc <filename.c> -o <outputname>

```
user@ws$ gcc HelloWorld.c -o myapp
```

- Output: An executable called *outputname*
- To run: ./outputname

```
user@ws$ ./myapp
Hello World
user@ws$
```

Multi-file Programs

Why create multi-file programs?

- **Manageability**
- **Modularity**
- **Re-usability**
- **Abstraction**

General abstraction used in Multi-file Programs

- **Components:**
  - **Header files**
  - **Implementation Source files**
  - **Application source file (contains the main function)**

Header Files

Contents:

- **Pre-processor directives and macros**
- **Constant declarations**
- **Type declarations (enum, typedef, struct, union etc)**
- **Function prototype declarations**
- **Global variable declarations**
- **May also contain static function definitions**

Example: *HelloWorld.h*

```
#ifndef _HELLOWORLD_H_
#define _HELLOWORLD_H_

typedef unsigned int my_uint_t;
extern void printHelloWorld();
extern int iMyGlobalVar;
...
#endif
```

Implementation Source Files

Contents:

- **Function body for functions declared in corresponding header files**
- **Statically defined and inlined functions**
- **Global variable definitions**

Example: *HelloWorld.c*

```
#include <stdio.h>
#include “HelloWorld.h”

int iMyGlobalVar;
void printHelloWorld()
{
     iMyGlobalVar = 20;
     printf(“Hello World\n”);
     return;
}
```

Application Source File

Contents:

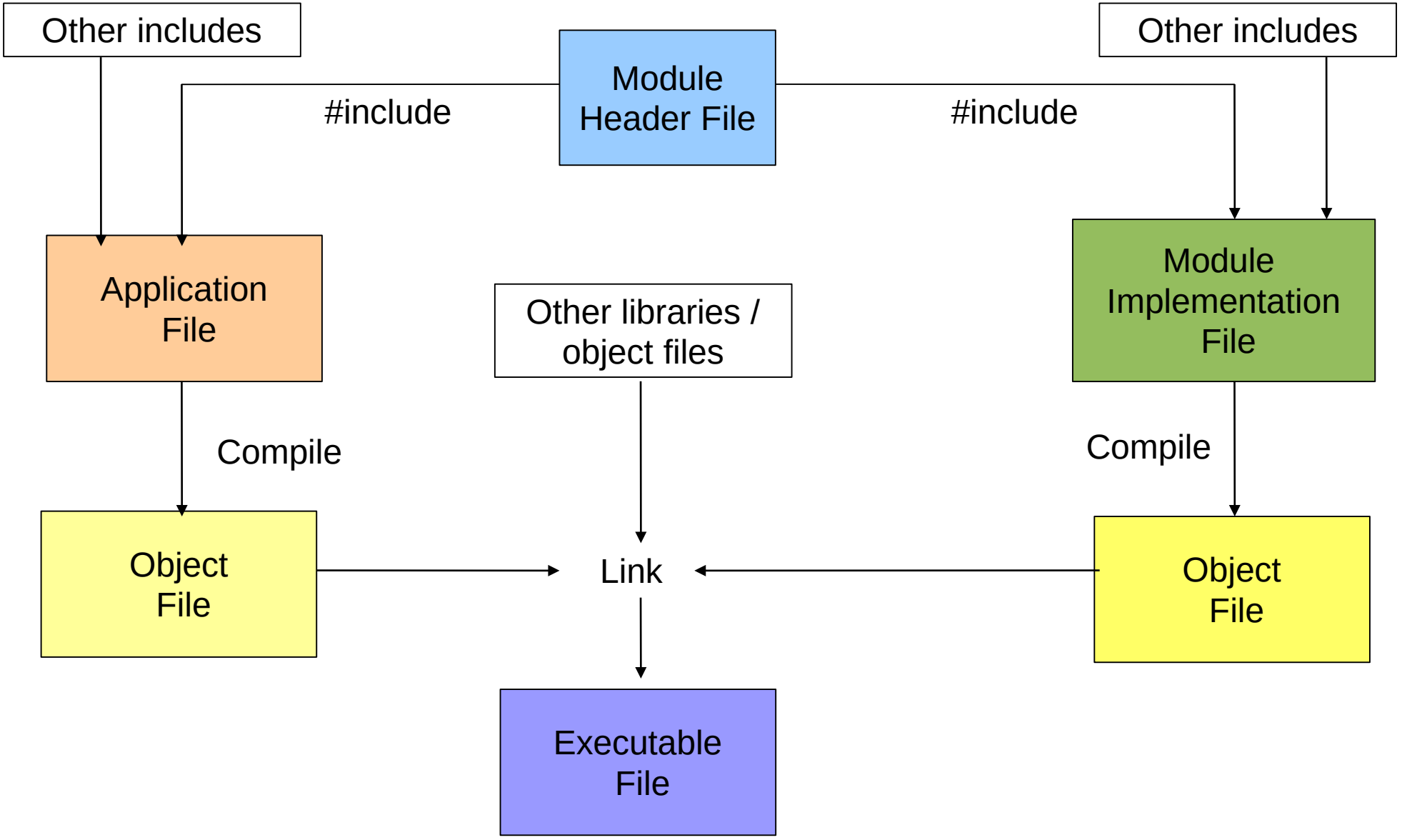
- **Function body for the main function**
- **Acts as client for the different modules**

Example: *app.c*

```
#include <stdio.h>
#include “HelloWorld.h”

int main()
{
     iMyGlobalVar = 10;
   printf(“%d\n”, iMyGlobalVar);
     printHelloWorld();
   printf(“%d\n”, iMyGlobalVar);
     return 0;
}
```

Correlation between components



Step 4: Compiling a simple multi-file program

- **Syntax:** gcc <file1.c> <file2.c> ... -o filename
- **Example:**

```
user@ws$ gcc HelloWorld.c app.c -o my_app
user@sw$ ./my_app
10
Hello World
20
user@ws$
```

Step 5: Multi-step compilation and linking

- **Steps:**
  - **Compile source files to object files**
  - **Link multiple object files into executables**

- **Compilation to object files**
  - **Special Option: -c**
  - **Syntax: gcc -c <filename(s).c>**
- **Link multiple files into the final executables**
  - **Syntax: gcc <filename1.o> <filename2.o> [-o output]**

Step 5: Continued

- **Example:**

```
user@ws$ gcc -c HelloWorld.c
user@ws$ gcc -c app.c
user@ws$ gcc HelloWorld.o app.o -o my_app
user@sw$ ./my_app
10
Hello World
20
user@ws$
```

Step 6: Including files from other directories

- **Special Option: -I<directory_name>**
- **Syntax:**

  gcc -I<directory1> -I<directory2> <filename(s).c>
- **Example:**

```
user@ws$ cd HelloWorld/src
user@ws$ gcc -c -I../../HelloWorld/include HelloWorld.c
user@ws$ cd ../../app/src
user@ws$ gcc -c -I../../HelloWorld/include main.c
user@ws$ cd ../../
user@ws$ gcc HelloWorld/src/HelloWorld.o app/src/main.o -o
my_app
user@ws$ ./my_app
```

Object Libraries

- Libraries contain pre-compiled object codes
- Are of 2 types:
  - Statically Linked: Object codes are linked into and placed inside the executable during compilation.
    - Name format: lib<name>.a
  - Dynamically Linked: Object code is loaded dynamically into memory at runtime.
    - Name format: lib<name>.so

Statically Linked Libraries

- Consists of a set of routines which are copied into a target application
- An archive of object files
- Object code corresponding to the required functions are copied directly into the executable
- Library format is dependent on linkers
- Increases executable size

Dynamically Linked Libraries

- Contains position independent code for different functions
- Executable code is loaded by the *loader at runtime*
- *The symbol table in the library contains blank addresses which are filled up later by the loader*
- *Increases reuse*
- *Decreases program size and execution time*

Step 7: Linking with external libraries

- Static linking: Link to lib*name.a*
  - Special option: *-static and -l*
  - Syntax: gcc -static <filename(s)> -l*name*

```
user@ws$ gcc -static math_test.c -lm
```

- Dynamic linking: Link to lib*name*.so
  - Special option: *-l*
  - Syntax: gcc <filename(s)> -l*name*

```
user@ws$ gcc math_test.c -lm
```

Step 8: Linking to a library at non-standard path

- **Special option:** -L<path>
- **Syntax:**

gcc <filename> -l<name> -L<path>

```
user@ws$ gcc math_test.c -lm -L/usr/lib
```

Step 9: Building Static libraries

Required external tools: *ar, ranlib*

1. *Create object files for the source codes*
2. *User ar to create the archives with names of the form: lib<libraryname>.a*
3. *Use ranlib to generate the index within the library*

Step 9: Continued

- **Example**

```
user@ws$ gcc -c HelloWorld.c
user@ws$ ar rcs libHW.a HelloWorld.o
user@ws$ ranlib libHW.a
user@ws$ gcc app.c -lHW -L. -o my_app
user@ws$ ./my_app
10
Hello World
20
user@ws$
```

Step 10: Building Dynamically linked libraries

- Requirements: Object code needs to be position independent
- Steps:
  1. Compile sources in a Position Independent manner
     - Option: -fPIC
  2. Combine objects to create shared library:
     - Option:

       -shared -W1,-soname,lib<name>.so.<version>

Step 10: Continued

- **Example**

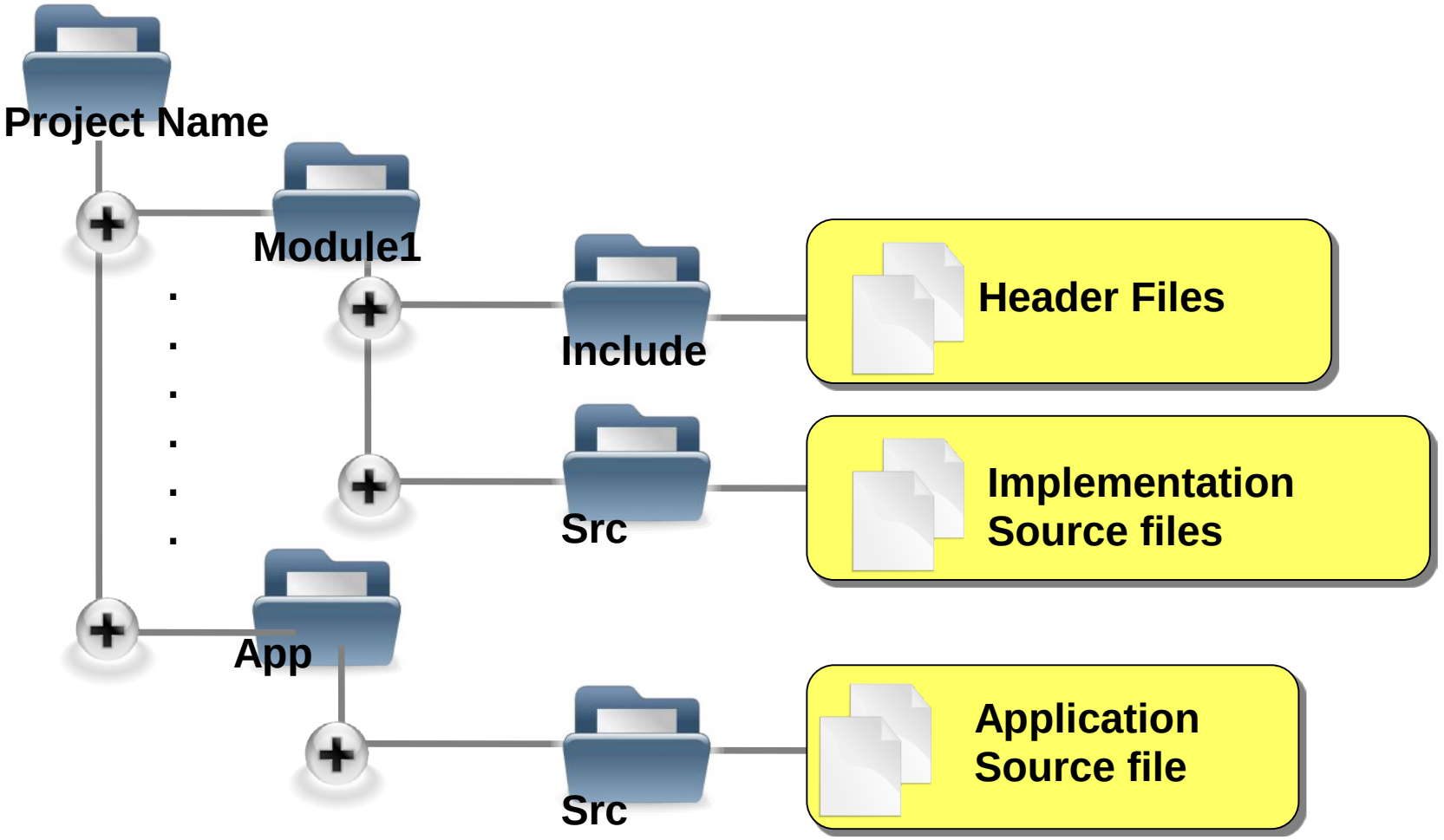
```
user@ws$ gcc -c -fPIC HelloWorld.c
user@ws$ gcc -shared -W1,-soname,libHW.so.1 -o libHW.so
HelloWorld.o
user@ws$ gcc app.c -lHW -L. -o my_app
user@ws$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
user@ws$ ./my_app
Hello World
20
user@ws$
```

- **LD_LIBRARY_PATH**
  - Set to the directory containing the .so file

Some more details about linking

- If -static is specified, linking is always static
  - if lib<name>.a is not found gcc errors out
- Otherwise
  - If lib<name>.so is found linking is dynamic
  - otherwise linking is static
- In case of dynamic linking, lib<name>.so should be placed in a standard location, otherwise LD_LIBRARY_PATH needs to be set

Recommended organization of Multi-file Projects



Compiler Warnings

- Special option: -Wall
- Syntax : gcc -Wall ....

```
user@ws$ gcc warning.c
user@ws$ gcc warning.c -Wall
warning.c: In function ‘main’:
warning.c:8: warning: suggest explicit braces to avoid
ambiguous ‘else’
user@ws$
```

Wrapping up

Questions?