Section 4
Program Building

1. Overview
2. Compilation
3. Linking
4. Makefiles

4.1 Overview

programmer

user

C code

Library code

And then ...
magic happens

executable

Overview (cont.)

- What is *program building*?
  - it's the translation of source code into machine code
    - *source code* is written in a high-level programming language
      - it **cannot** be executed directly by the CPU
      - examples: C, C++, Java
    - *machine code* is generated as low-level machine language
      - it **can** be executed by CPU
      - it cannot be understood by humans
  - program building is the creation of an *executable* from one or more source files

Overview (cont.)

- What is a *program executable*?
  - it's a file that contains machine code instructions
    - these instructions are OS and CPU dependent
    - you cannot compile on one platform and run on another
- Characteristics of an executable
  - it consists of code from multiple source files
    - your code, other people's code, library code
  - it must have exactly **one** `main()` function

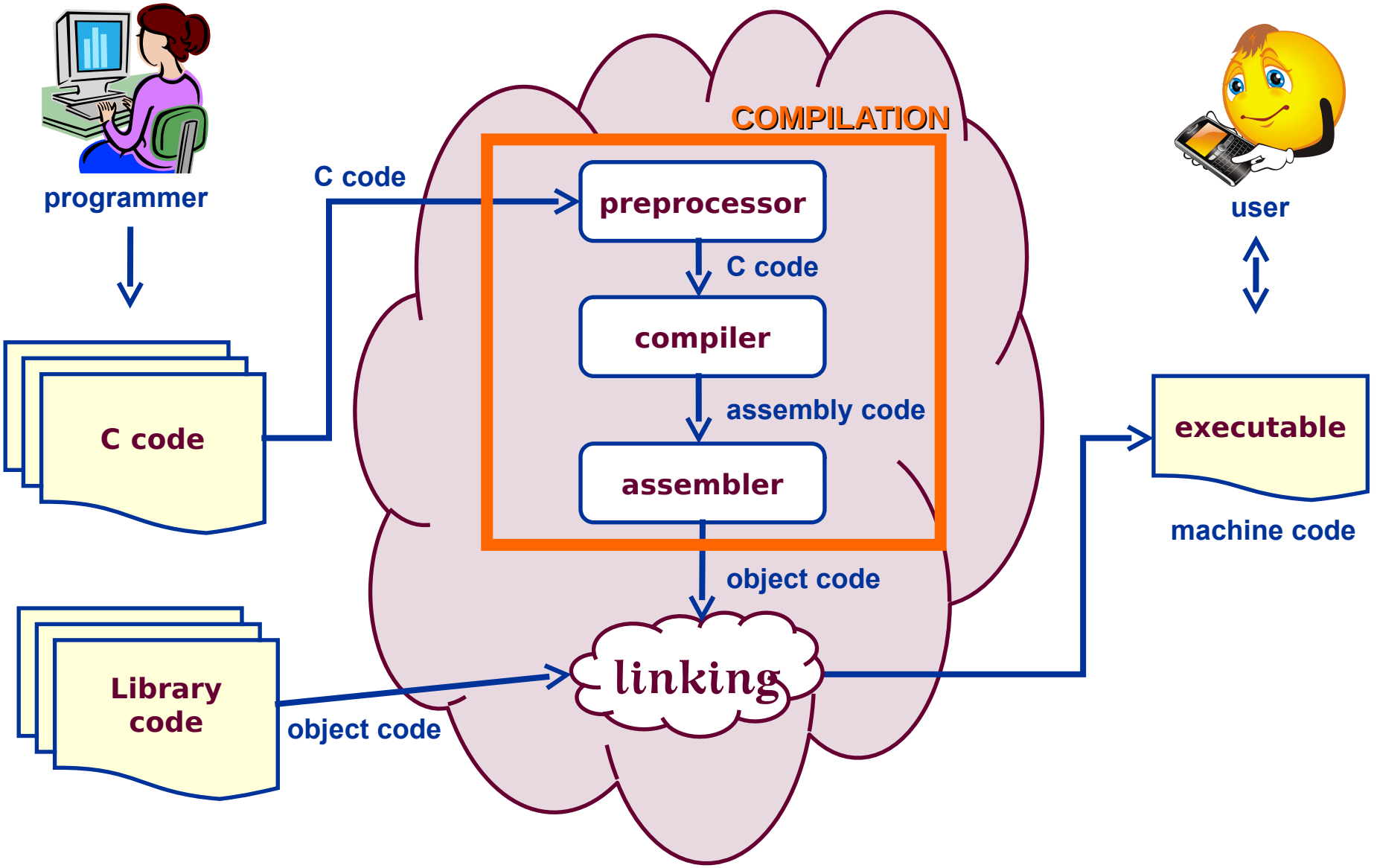
Overview (cont.)

- Transforming C code into an executable requires two steps

1. compilation
   - transforms C code to *object code*
     - input: multiple C source code files
     - output: multiple object code files
   - there is a 1-to-1 correspondence between C files and object code files
   - **NOTE**: the use of the term *object* here is **not the same** as OO

2. linking
   - transforms object code into an executable
     - input: multiple object code files
     - output: one executable
   - linking is where *library code* gets linked in (added) to yours

4.2 Compilation



Preprocessing

- What does preprocessing do?
  - it interprets all the preprocessing directives of one source file
    - the preprocessing directives all begin with the hash symbol (**#**)
    - the preprocessing step performs *text substitutions*
      - it copies library header files, it replaces constants and aliases, etc.
    - it provides conditional compilation
  - input:
    - source code from one source file
  - output:
    - new source code with the substitutions incorporated
    - use the **`-E`** option to stop the compiler after preprocessing

Preprocessing Header Files

- What is a header file?
  - it's a file containing information needed by multiple source files
    - data type definitions
    - function prototypes
    - ... more on this later ...

  - it **never** contains function implementations or any statements!

- Characteristics
  - a header file is copied into a source file during preprocessing
    - use **angle brackets** to include a header file from a library
    - use **double quotes** to include a header file from the current directory

Compiling

- What does compiling do?
  - it translates source code into *assembly code*
    - it performs optimizations
    - it resolves internal function addresses
      - the functions that have implementations within the same source file
  - input:
    - preprocessed source code from one source file
  - output:
    - the corresponding assembly code
      - a human readable version of machine code
    - use the **`-S`** option to stop after compiling

- **`coding example <p1>`**

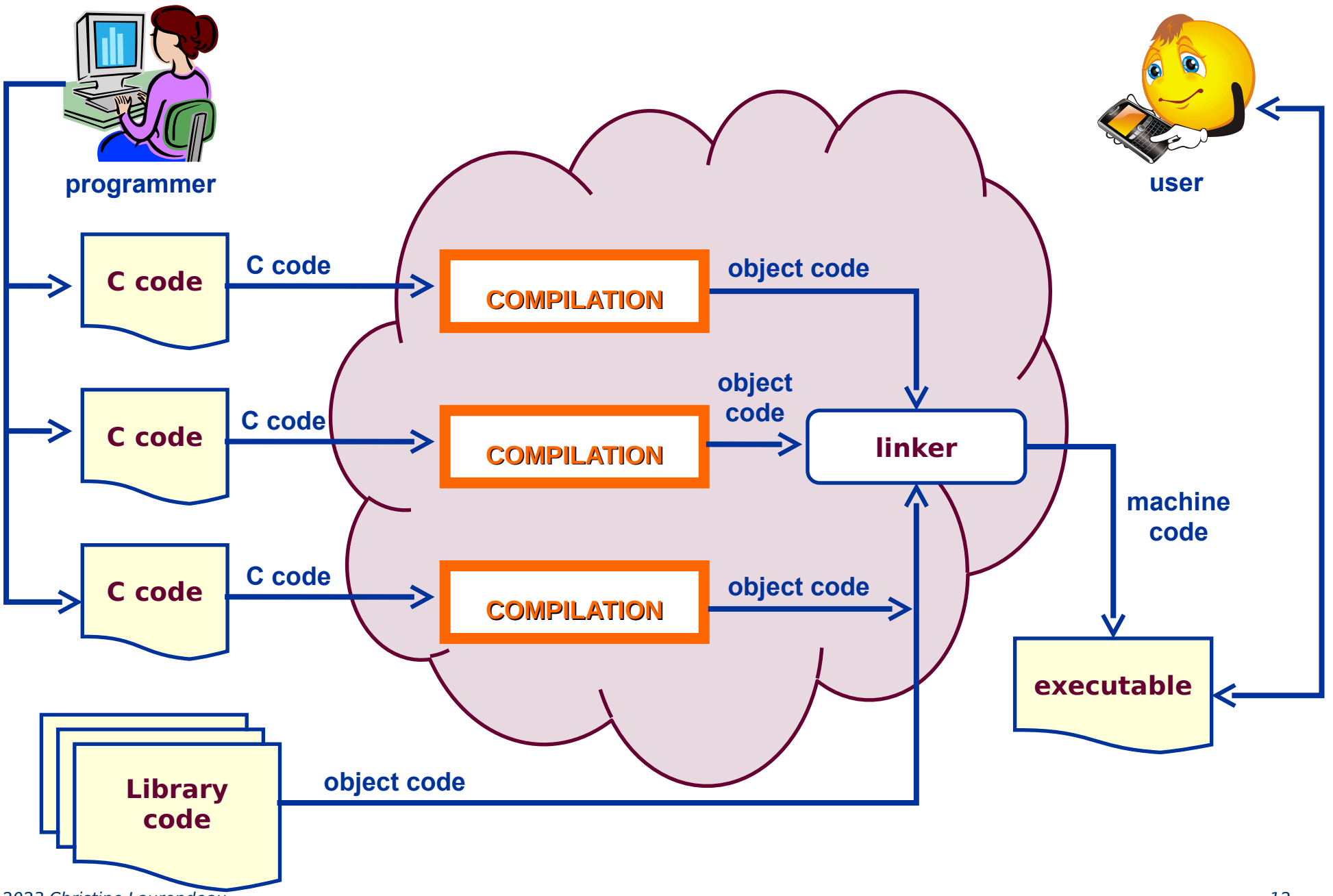
```
void sumIterative(int numElements,
                  int *intArray,
                  int *sum)
{
  int i;
  *sum = 0;

  for (i=0; i<numElements; ++i)
    *sum += intArray[i];
}
```

```
sumIterative:
.LFB4:
    ... start of proc stuff ...
    movq -40(%rbp), %rax
    movl $0, (%rax)
    movl $0, -4(%rbp)
    jmp .L14
.L15:
    movq -40(%rbp), %rax
    movl (%rax), %edx
    movl -4(%rbp), %eax
    cltq
    leaq 0(,%rax,4), %rcx
    movq -32(%rbp), %rax
    addq %rcx, %rax
    movl (%rax), %eax
    addl %eax, %edx
    movq -40(%rbp), %rax
    movl %edx, (%rax)
    addl $1, -4(%rbp)
.L14:
    movl -4(%rbp), %eax
    cmpl -20(%rbp), %eax
    jl   .L15
        ... end of proc stuff ...
```

Assembling

- What does assembling do?
  - it translates assembly code into *object code*
  - input:
    - assembly code from one source file
  - output:
    - the corresponding object code
    - use the `-c` option to stop after assembling
      - this is **mandatory** if using multiple source files

4.3 Linking



Linking (cont.)

- What does linking do?
  - it combines code from multiple object files into one executable
  - it resolves external function addresses
    - the functions with implementations in different object files
    - the library functions
  - input:
    - object code from multiple object files, including library object code
  - output:
    - one executable file
- **coding example <p2>**

Linking (cont.)

- Why do we separate compiling and linking?
  - #1 reason: it saves on total program building time
    - compilation is slow
    - linking is fast
    - very large systems try to avoid recompiling at all costs
  - #2 reason: it minimizes the amount of recompilation
    - only source files that have *changed* need to be recompiled!
    - we can use existing object files for the rest of the code
    - re-link new object files with unchanged ones

Linking in Libraries

- What is a library?
  - it's a collection of related functions
  - it may be written by us or by other programmers

- To use a library:
  - #1: **include** the library header file, using a preprocessor directive
  - #2: **link in** the library object file
    - this step is **NOT** optional!

- C standard library: **`libc.a`**
  - this library is always linked in *by default*

Linking in Libraries (cont.)

- Different types of linking
  - static linking
    - the library object code is copied into the executable
    - this increases the size of the executable
    - it results in faster execution time
  - dynamic linking
    - this is the default setting
    - the library object code is loaded at runtime, as needed
    - it results in a smaller executable, but slower execution time

4.4 Makefiles

- What is a *Makefile*?
  - it's a text file
    - it's just called **`Makefile`**, and it doesn't have a file extension
  - it's a tool to organize compiling and linking commands
  - it helps us manage the dependencies between files
    - dependencies between:
      - header files and source files
      - source files and object files
    - it only recompiles source files that have changed since the last **`make`**

Makefiles (cont.)

- Characteristics
  - a Makefile uses a special syntax
  - it is invoked from the shell using the **`make`** command
  - it is composed of three parts:
    - targets
    - dependencies
    - commands
- **`coding example <p3>`**

Makefiles (cont.)

- Why use a Makefile?
  - it keeps track of dependencies and what needs to be re-built
    - it compares the timestamps:
      - between header and source files
      - between source and object files
      - between object and executable files
    - if the header file is newer, the source file gets recompiled
    - if the source file is newer than the object file, it gets recompiled
    - if the object file is newer than executable, the code is re-linked
  - it decreases the number of commands for the programmer
    - with a Makefile, one **`make`** command
    - without a Makefile:
      - one compilation command for each source file
      - one linking command

Makefiles (cont.)

- Makefile macros
  - these are similar to variables
  - they can be used to
    - specify compilation options
    - define groups of files
    - examples: group of object files, **`gcc`** command and options

- Common housekeeping Makefile targets
  - **`all`**
    - this is used to compile all executables
  - **`clean`**
    - this is used to remove all intermediate files