Section 1
Basics of C++ Development

1. Linux platform
2. Basic language features
3. Programming conventions
4. Class definitions
5. Constructors and destructors
6. Memory management

Section 1.1
Linux Platform

1. Overview
2. Shells
3. Tools
4. Program building

1.1.1 Overview

- Unix family of operating systems (OSs)
  - forefathers: Ken Thompson and Dennis Ritchie
  - open source (free!)
    - kernel can be modified
  - modular
  - allows broad access to OS functions

Overview (cont.)

- “root” or super-user can do **anything**
  - including deleting the entire file system!
  - can be extremely dangerous

- OS of choice for complex or scientific development

- Unix philosophy, summed up:
  - “Unix is simple. It just takes a genius to understand its simplicity” --Dennis Ritchie
  - “Unix was not designed to stop its users from doing stupid things, as that would also stop them from doing clever things” --Doug Gwyn
  - “Unix is user-friendly. It’s just choosy about who its friends are” --anonymous

Overview (cont.)

- Unix family of OSs includes:
  - Linux
  - Solaris
  - BSD
  - HP-UX
  - Mac OS X

- ... and many others over the years...
  - main differences involve hardware

1.1.2 Shells

- What is a shell?
  - it’s a program that provides direct access to the OS
    - it provides a command line
  - it allows users to run programs
  - it serves as a command line interpreter
  - users can run multiple shells, one in each window

Shells (cont.)

- Major Unix shells:
  - Bourne shell (*sh*)
  - Bourne-again shell (*bash*)
    - this is the default shell for Linux
  - C shell (*csh*)

- Differences between shells
  - command line shortcuts
  - environment variables

Shells (cont.)

- A shell provides flexibility
  - programs may be executed with command line arguments
    - these are options that are preceded by a dash -

- Common shell commands
  - navigating the file system
    - change directory: **`cd`**
    - list directory: **`ls`**
    - make directory: **`mkdir`**
  - manipulating files
    - view file: **`cat`**, **`more`**
    - search file: **`grep`**
  - help pages: **`man`**

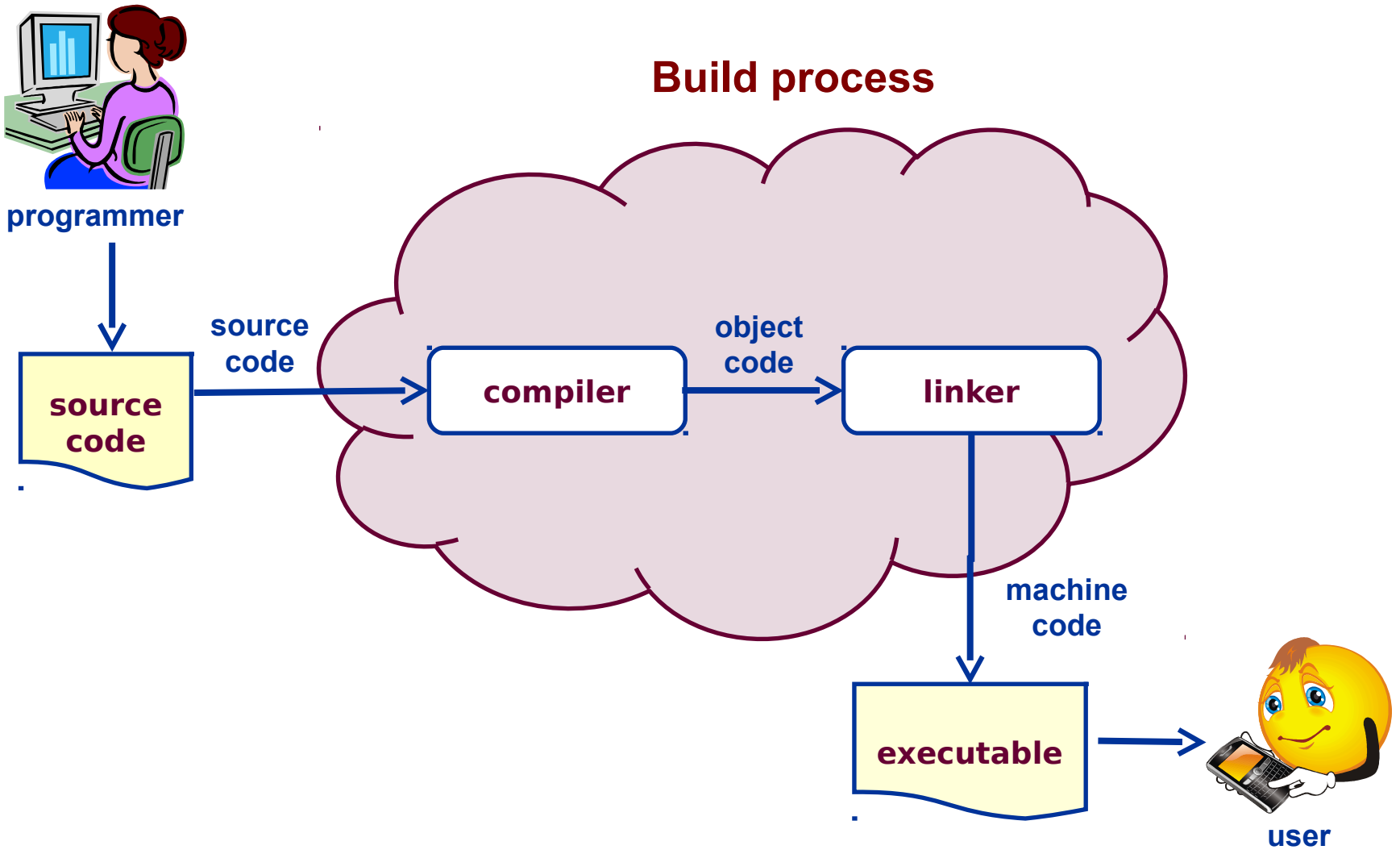
1.1.3 Tools

- Text editor
  - pick one and get good at it
  - options: Atom, vi/vim, emacs, gedit, many others
  - required for editing:
    - your program
    - your build files (Makefile)

Tools (cont.)

- Compiler
  - GNU C++ compiler: `g++`
    - default standard is C++98
    - it's **mandatory** in this course
  - it has many command line options
    - `-o` specifies a name for the *output file* (usually the executable)
    - `-c` creates the *object code*
    - we'll see lots of examples in this course
- Code **must** be compiled on the same platform where it executes
  - example: code that is compiled in Windows will **not** run in Linux

1.1.4 Program Building



Program Building (cont.)

- What is program building?
  - it is 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 a low-level machine language
      - it can be executed by the CPU
      - it cannot be understood by humans
  - it is the creation of an *executable* from one or more source files

Program Building (cont.)

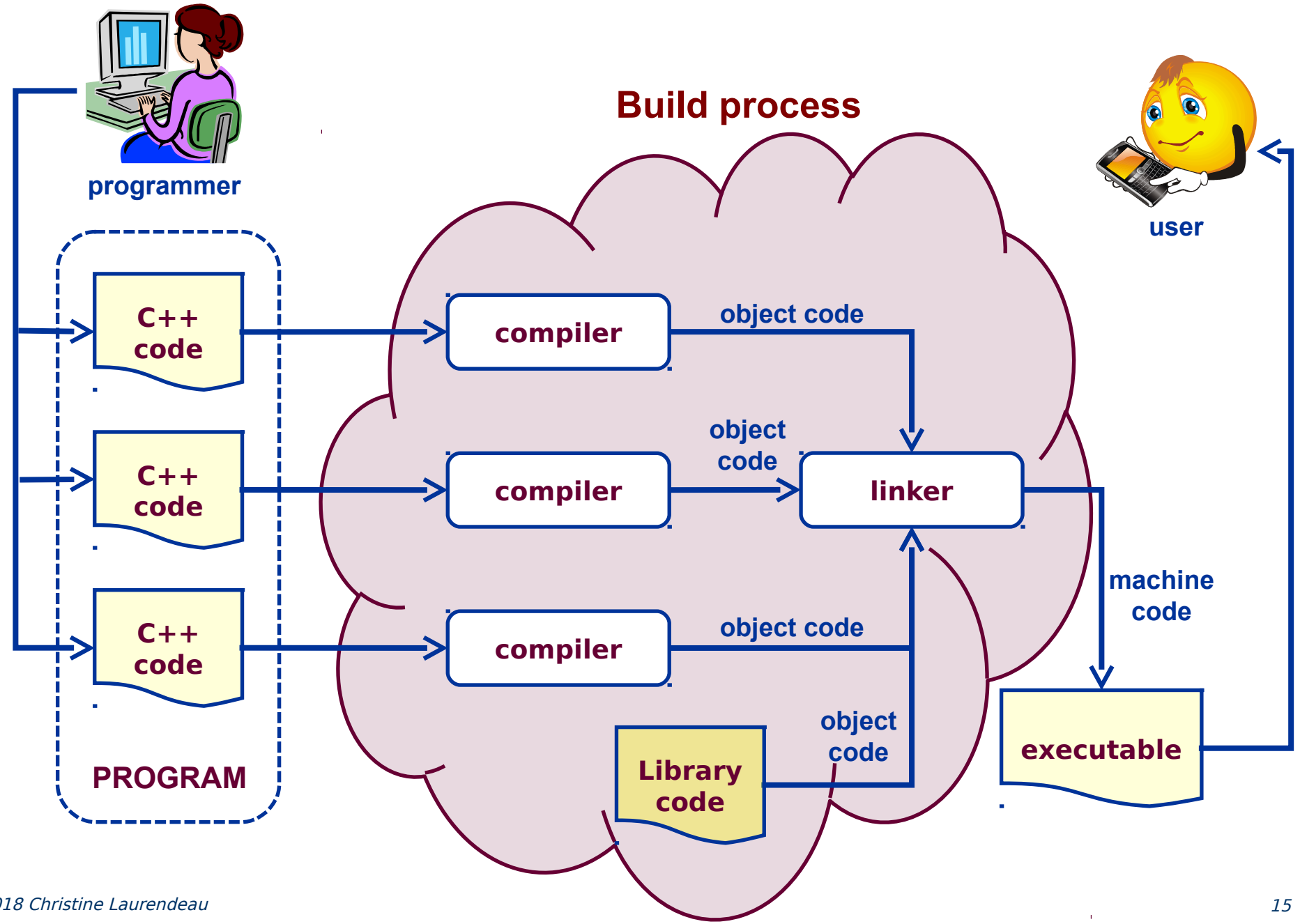
- What is a program executable?
  - a file that contains machine code instructions
    - these instructions are OS and CPU dependent
    - you cannot compile on one platform and run on another

- 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

Program Building (cont.)

- Transforming C++ code into an executable in Linux
  - compilation
    - transforms C++ code to object code
      - input: multiple C++ source code files
      - output: multiple object code files
    - 1-to-1 correspondence between C++ files and object code files
  - linking
    - transforms object code into an executable
      - input: multiple object code files
      - output: one executable
    - linking is where library code (as object code) gets added to yours

Program Building (cont.)



Program Building (cont.)

- Compiling and linking **one** source file:
  - given one C++ source file: **`hello.cc`**
  - to compile and link:
    - **`g++ -o hello hello.cc`** -- this creates the executable **`hello`**

- Compiling and linking **multiple** source files manually:
  - given two C++ source files: **`file1.cc`** and **`file2.cc`**
  - to compile:
    - **`g++ -c file1.cc`** -- this creates object file **`file1.o`**
    - **`g++ -c file2.cc`** -- this creates object file **`file2.o`**
  - to link:
    - **`g++ -o run file1.o file2.o`** -- this creates the executable **`run`**

Program Building (cont.)

- Using Makefiles:
  - it’s an easy way to compile and link multiple source files

- What is a Makefile?
  - it’s a text file
  - it’s a tool to organize compiling and linking commands
  - it manages dependencies between source and header files
    - only recompiles source files that have changed