09. if statements, Testing

CPSC 120: Introduction to Programming
Pratishtha Soni~ CSU Fullerton

Agenda

0. Sign-in sheet
1. Technical Q&A
2. if Statements
3. Make

1. Technical Q&A

Technical Q&A

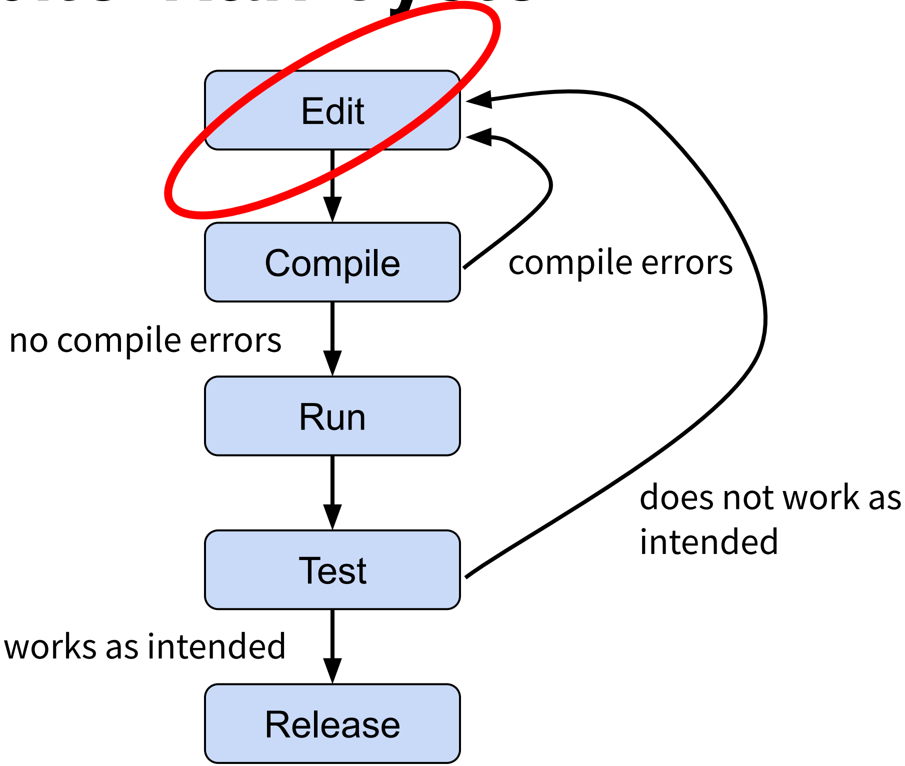
Let’s hear your noted questions about...

- This week’s Lab
- Linux
- Any other technical issues

Reminder: write these questions in your notebook during lab

2. if statements

The Edit-Compile-Run Cycle



@copyrights Kevin A Wortman

Control Flow

- **Flow** (of a program): order that statements are executed
- So far: **straight-line flow**
  - Always run `main()` top-to-bottom
- **Control flow**: syntax to manage flow
- **if statement**: control flow to choose between alternative statements
- **Loops** (future): control flow to repeat statements

Syntax: if statement

statement:

if (*condition-expr* **)** *true-statement*

else-clause(optional)

else-clause:

else *false-statement*

Semantics:

1. Evaluate *condition-expr* and convert result to `bool`
2. If result is true: execute *true-statement*
3. Otherwise, execute *false-statement* if it exists

Examples:

```
if (lives == 0)
  std::cout << “Game over”;
```

```
if (age >= 18)
  std::cout << “adult”;
else
  std::cout << “minor”;
```

Multiple Statements inside If

- Problem
  - often need to execute multiple statements in the true case
  - if syntax: *true-statement* is a single statement
  - (same for *false-statement*)

```
if (health == 0)
  std::cout << “You lost a life”;
if (health == 0)
  --lives;
```

repetitive

- Solution
  - **compound statement**: statement that contains multiple inner statements
  - counts as one statement for syntax purposes

Syntax: Compound Statement

statement:

{ *inner-statement*... **}**

Semantics:

- Execute *inner-statement*... in top-to-bottom order

Examples:

```
{
  std::cout << “Hi”;
  x = 0;
}

if (health == 0) {
  std::cout << “You lost a life”;
  --lives;
}
```

Best Practice: Always use Braces with if

- Style Guide: always use braces with if
- Without braces, it can be confusing what is inside the *true-statement* / *false-statement*
- Example:

```
if (health == 0)
    std::cout << “You lost a life”;
    --lives;
```

not inside if; always executes, even when health is not 0

- This caused the 2014 Apple SSL security bug
- Best practice: **always use braces around the statements controlled by an if statement**

bool Type

- George Boole: mathematician who studied true/false logic
- **Boolean**: related to true/false
    - Proper noun, so capitalized
- `bool`: data type for Boolean values
    - yes/no situations
- Only possible values: `true`, `false`
- Example:

```
bool succeeded{ false };
// ...
succeeded = true;
```

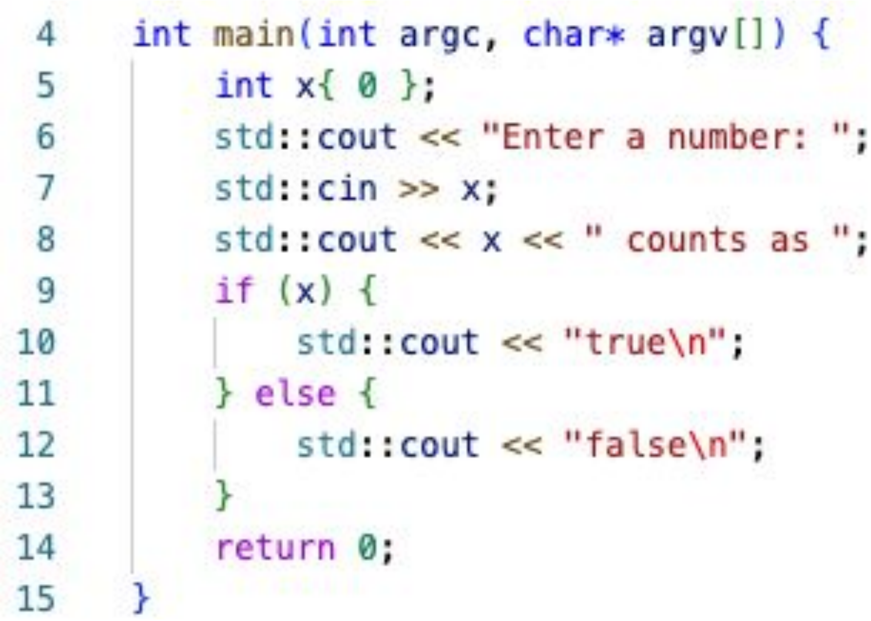
Conversion to `bool`

- Recall: in
  if **(** *condition-expr* **)**
  condition-expr is **converted to bool**
- Bool conversion is only available to **some data types**
  - Compile error

Conversion to `bool`

Data Type	`bool` Conversion Semantics
int	Non-zero is true, zero is false
double	Non-zero is true, zero is false
cin	good is `true`; failed is `false`
string	Not available

Example: if with Conversion to bool



```cpp
int main(int argc, char* argv[]) {
    int x{ 0 };
    std::cout << "Enter a number: ";
    std::cin >> x;
    std::cout << x << " counts as ";
    if (x) {
        std::cout << "true\n";
    } else {
        std::cout << "false\n";
    }
    return 0;
}
```

```
$ ./a.out
Enter a number: 3
3 counts as true
$ ./a.out
Enter a number: -1
-1 counts as true
$ ./a.out
Enter a number: 0
0 counts as false
```

Relational Operators

- Purpose of `if` is to make decisions
  - `bool` conversion rule is probably not what we want to decide
- **Relational operator**:
  - Binary operator
  - Compares two operands of the same type
  - Returns a `bool` value
  - Equal / not equal
  - Less / greater
- Inspired by math notation
- Limited to keyboard symbols
- ≠ becomes `!=`
- ≤ becomes `<=`

Relational Operators

Operator	**Semantics**	**Example** (x and y are same type)
`==`	Equal to	`x == y`
`!=`	Not equal to	`x != y`
`<`	Less than	`x < y`
`>`	Greater than	`x > y`
`<=`	Less than or equal to	`x <= y`
`>=`	Greater than or equal to	`x >= y`

Example: Relational Operator in if

```
if (player_1_score > player_2_score) {
  std::cout << "player 1 is winning\n";
} else {
  std::cout << "player 2 is winning\n";
}
```

Pitfall: `=` versus `==`

- = is **assignment** operator;

  `x = 3;`

  x *changes* to become 3
- == is equality comparison operator;

  `x == 3`

  produces `true` when x is 3, `false` otherwise, *leaving x unchanged*
- Easy mixup
  - Unfortunate!
  - `if (x = 3)` // should be ==
  - `x == 0;` // should be =

@copyrights Kevin A Wortman

Pitfall: = in if

Logic error: write = in `if` expression instead of ==

If statement on right:

1. **Assigns** (changes) `choice` to 1
2. `choice` is converted to bool
3. 1 is nonzero which **always** counts as `true`

So this **always** prints “you chose 1”, even if the user input something other than 1!

```
int choice{ 0 };
std::cin >> choice;
if (choice = 1)
  std::cout << “you chose 1”;

// if should be:

if (choice == 1)
  std::cout << “you chose 1”;
```

Pitfall: Stray Semicolon After if Expression

Review: if syntax:

if (*condition-expr* **)** *true-statement*

else-clause(optional)

```
if (x > 0)
  std::cout << “positive”;
```

Logic error:

if (*condition-expr* **) ;**

true-statement

stray semicolon

```
if (x > 0);
  std::cout << “positive”;
```

Pitfall: Stray Semicolon After if Expression

Logic error:

1. As usual, whitespace is ignored
2. The ; counts as the *true-statement* of the if
3. If *condition-expr* is true, execute ; (do nothing)
4. Then, always, execute cout << “positive”

```
if (x > 0);
  std::cout << “positive”; // always prints, regardless of x
```

@copyrights Kevin A Wortman

Pitfall: Unexpected Expression After Else

Compile error:

```
if (x < 0) {
  std::cout << “negative”;
} else (x >= 0) {
  std::cout << “non-negative”;
}
```

- Highlighted `(x >= 0)` is invalid syntax
- **Remember**: `else` means “otherwise” aka “in all other cases”
  - Doesn’t make sense to limit when `else` happens

Problem: Choose Between 3+ Alternatives

```
// do one thing, or nothing
if (count == 1) {
  std::cout << “once”;
}

// choose between two alternatives
if (count == 1) {
  std::cout << “once”;
} else {
  std::cout << “more than once”;
}
```

```
// choose between four alternatives
if (count == 1) {
  std::cout << “once”;
} else {
  if (count == 2) {
    std::cout << “twice”;
  } else {
    if (count == 3) {
      std::cout << “thrice”;
    } else {
      std::cout << count << “ times”;
    }
  }
}
```

works, but hard to read

Chaining If Statements

To decide between 3+ alternatives:

- chain together `if`s and `else`s
- Omit **{** between else and if
- Indent all the compound statements the same amount
- Still plain if syntax; nothing new

```
// choose between four alternatives
if (count == 1) {
  std::cout << “once”;
} else if (count == 2) {
  std::cout << “twice”;
} else if (count == 3) {
  std::cout << “thrice”;
} else {
  std::cout << count << “ times”;
}
```

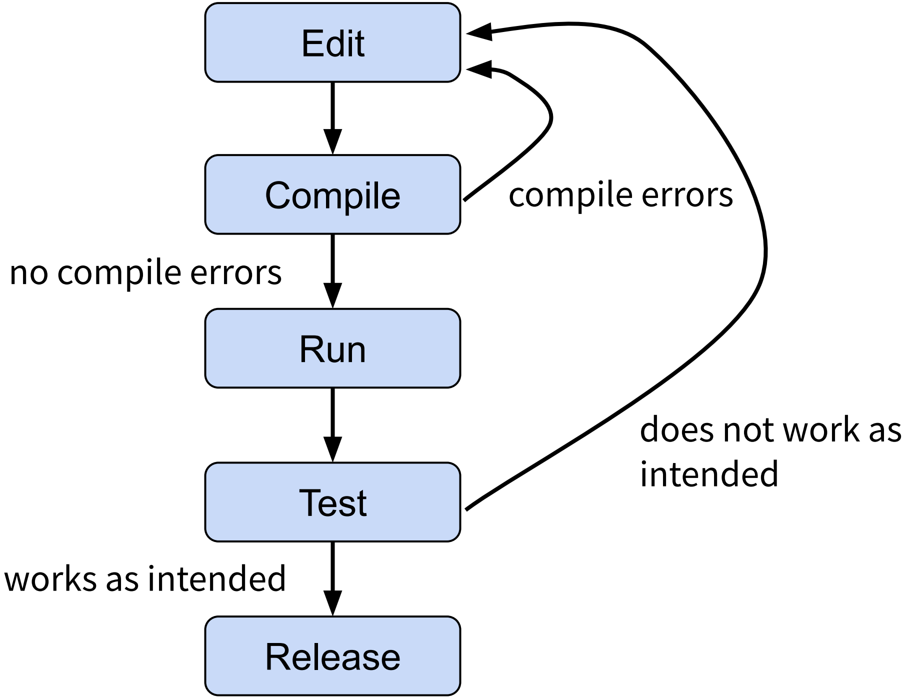
3. Make

Makefiles in Labs

- Labs 3 on involve **makefiles**
- Makes your life easier

Understand the Problem: Typing Shell Commands

- Edit-Compile-Run-Test-Release cycle
- **Shell commands** for
  - Edit: `$ code`
  - Compile: `$ clang++`
  - Run: `$ ./`*`programname`*
  - Test: run program or unit test program
  - Release: git add, commit, push
- Lots of **commands**
- Lots of **details**



Review: Ideal Division of Labor

- **Business Logic:** the human meaning of algorithm data
- Programs
  - **Cannot** understand business logic or design algorithms
  - Can perform tedious, repetitive work flawlessly, quickly, cheaply
- Humans
  - **Can** understand business logic and design algorithms
  - Busy-work is tedious, error-prone, expensive
- Division of Labor Best Practice
  - Humans think about business logic and algorithms
  - Computer programs do repetitive work

Solution: Build Tool

- **Build tool:** program that automates running development commands
- Humans configure a build tool to automatically compile-run-test
- Build tool runs individual commands
- Humans only have to run the build tool
  - Simpler
  - Easier

Make

- Make: build tool built in to Ubuntu
- Old, simple, widely used
- Has its own syntax for automating commands
- **Makefile:** source file for make
- Lab prompt has Makefiles
- You should understand
  - What they do
  - How to use them

Makefile Syntax

```
makefile:

rule...

rule:

target : dependency...(optional)

	command...
```

Must indent with **Tab** key

Semantics:

- *target* is a filename or name of a goal
- *dependency* is a filename or target that contributes to the target
- *command...* are the shell commands to create the file / achieve the goal

Makefile example

```
units: units.cc
     clang++ units.cc -o units

units_unittest: units_functions.h units_functions.cc units_unittest.cc
     clang++ units_unittest.cc unites_functions.cc -o units_unittest

test: units_unittest
     ./units_unittest

clean:
     rm -f units units_unittest
```

Review: Pattern: Shell Command

```
$ COMMAND [ARGUMENT...]
```

- Cues that this is a shell command
  - Dollar sign
  - Fixed-width font
- You type everything **after the $**, then press Enter key
- ALL-CAPS are fill-in-the blank
- [BRACKETS] means optional
- ELLIPSIS... means you may repeat

make Command

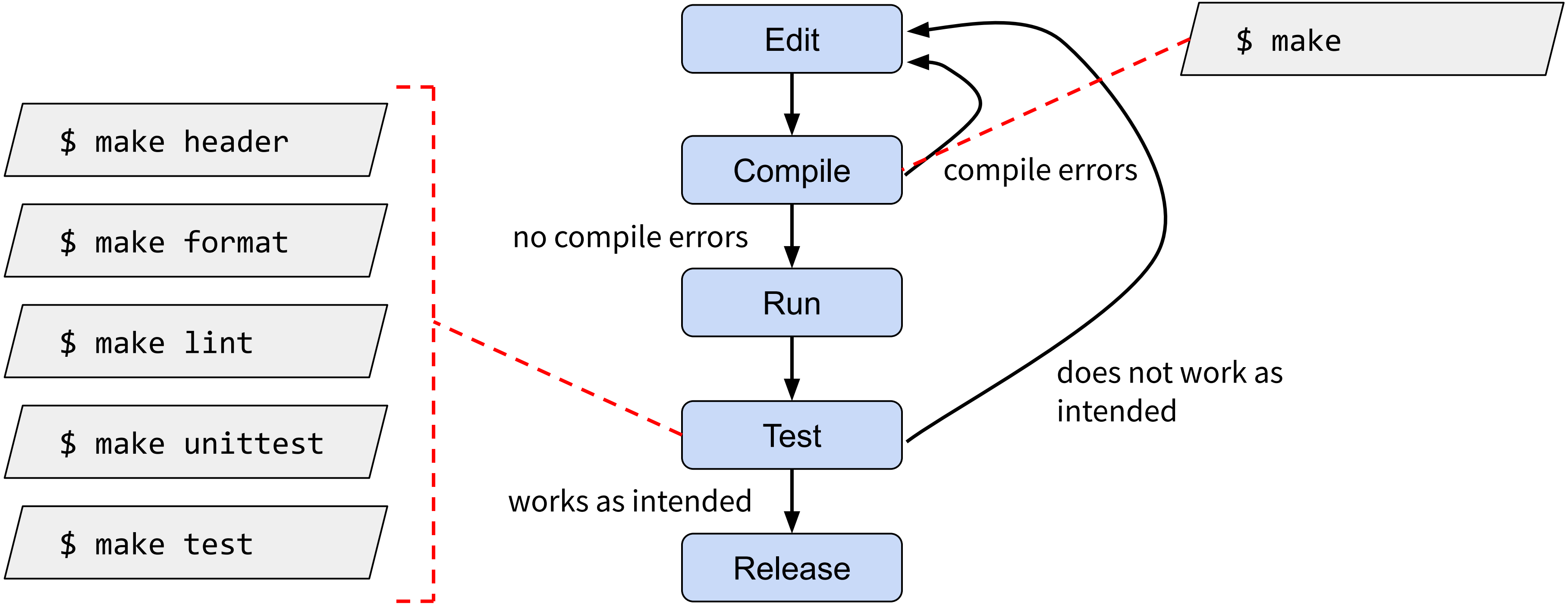
```
$ make [TARGET]
```

- Build TARGET
  - Must be one of the named targets in Makefile
- Automatically runs all necessary commands
  - Builds dependencies
  - And their dependencies... (**recursive**)
- Default TARGET is the first one in the Makefile

make Targets in Labs

Target	Command	Purpose
all	`$ make all`	compile all programs
clean	`$ make clean`	delete files created by make (except programs)
spotless	`$ make spotless`	clean, and also delete programs
header	`$ make header`	test header
format	`$ make format`	test formatting
lint	`$ make lint`	test linting
unittest	`$ make unittest`	test unit tests
test	`$ make test`	test system tests
(default)	`$ make`	same as $ make all

Edit-Compile-Run with Make



```
$ make header
$ make format
$ make lint
$ make unittest
$ make test
```

```
$ make
```

Edit

Compile

Run

Test

Release

compile errors

no compile errors

does not work as intended

works as intended

Make only Does What's Necessary

- **Optimization:** avoid unnecessary work
- Make optimizes the build by skipping commands that are unnecessary
- Looks at file modification dates
- Skips targets that are still up-to-date
- `$ make all` may run fast, no problem

```
units: units.cc
    clang++ units.cc -o units

units_unittest: units_functions.h units_functions.cc units_unittest.cc
    clang++ units_unittest.cc unites_functions.cc -o units_unittest
```