# CSCI 135
# Control Flow (Selection)

# Basic Language Constructs

Recall: each statement in an imperative language updates the value of some variable.

⇒ Classes of Constructs:

- Declaration of variables: How does the variable map onto memory?
- Updating variable: How do we update the variable, and what do we update it with?
- I/O: How do we input and output data into the system?
- Control: How do we control which statement gets executed next?
- Modularity and Object Orientation: How do we organize the program to enable proper software engineering practices?
- Comments: Used to describe code; ignored by compiler, but code unmaintainable without good comments!
  C/C++: on line beginning with // or surrounded by /*, */

# Types of Control

- Selection: Execute a different instruction depending on the result of evaluating some [Boolean] condition.
  `if/else`, multiway `if`, `switch`, `?   :`
- Iteration: Repetitively execute some instruction until some condition is met,
  `for`, `while`, `do while`
- Goto: Execute some instruction (not necessarily the next one) next
  ☠ Absolutely, **NEVER** never never use this (See Dijkstra, "Go To Statement Considered Harmful"); leads to unmaintainable spaghetti code.
- Break: Leave block

# Conditions

A **condition** is a Boolean expression - *i.e.*, one that evaluates to true or false.

Most commonly either a relational operator or a logical combination of conditions:

- Relational (Comparison) Operators: `==` `!=` $<$ $>$ $<=$ $>=$ (semantics vary by type of operands)
- Logical operators: `!` `&&` $||$
  Note: conjuncts/disjuncts are evaluated left-to-right and only if needed (called *short-circuit evaluation*)

See Fig. 2.3 for precedence order (but better to use parentheses)

⚠Equality (==) is not the same as assignment (=).

⚠Recent versions of C++ support alternate syntaxes for logical operators (*e.g.*, "and"). Avoid these as they are rarely used (and not compatible with earlier C/C++ standards).

# Conditions - Examples

- `n <= 100 && n >= 0`
- `n < 101 && n >= 0`
- `grade > 'c' && grade <= 'f'`
- `y < x && y >= 0`

  ? Which points of the Cartesian plane does this cover?
- `!s.empty() && (s[0] == 'f' || s[0] == 'g')`

  ? Why does the order of conjuncts matter?

# Short-Circuit Evaluation

Almost all languages: left-to-right, completely
C/C++ Special Cases: `&&`, ||, `?`

- Evaluate the 2nd operand only if it can change the result of the expression; *i.e.*, *short-circuit* the evaluation. Ex:
  - Don't evaluate `Q` in `P && Q` if `P` evaluates to false
  - Don't evaluate `Q` in `P || Q` if `P` evaluates to true
  - Similarly for `?`

⇒ useful when evaluating the Q would lead to an error for the 'wrong' case of P
Ex: `!s.empty() && (s[0] ...)`

# Caveat: Conditions in C

A C condition actually evaluates to an integer (not a true-false bool). If the integer is 0, it is interpreted as false; otherwise it is interpreted as true.

Normally, you don't need to worry about this (phew!). BUT

❓ What would `if (n=5) S1 else S2` do if n is initially 2?

# Caveat: Conditions in C

A C condition actually evaluates to an integer (not a true-false bool). If the integer is 0, it is interpreted as false; otherwise it is interpreted as true.

Normally, you don't need to worry about this (phew!). BUT

? What would `if (n=5) S1 else S2` do if n is initially 2?

1. Evaluate the condition n=5, which is actually an assignment statement/expression that stores 5 in n and evaluates to the rval (5)
2. Since $5 \neq 0$, interpret it as true
3. Execute S1

⚠ Confusing = and == is a very common source of bugs!

# & vs &&

& *Bit-wise* and operator

&& *Logical* and operator, produces 1 iff both operands are true (non-zero), and 0 otherwise.

Ex:

```
x = 1; y=2;
z = x & y;     evaluates to 0
z = x && y;    evaluates to 1
```

(similarly for | vs ||)

# Selection: If/else

## if <cond> S1 else S2

(commonly called if/then, though C doesn't use keyword then)
Execute S1 if cond evaluates to true, and S2 otherwise, where S1/S2 are statements.
(`else S2` is optional)

Example:

```
if (hrs < 40)
  pay = rate * hrs;                    indent block
else {                                 start of compound statement
  pay = rate*40 + 1.5*rate*(hrs-40);
  pay = pay - ft_ded; // deduction for full timers
  };                                   end of compound statement
// December bonus for all
if (month==12) pay=pay*1.1;            no else case here
```

⚠Good practice (not practiced above): use {} even if block has only one statement

# Selection with Multiple Conditions

```
if (hrs < 10)
  {...}
else
  if (hrs < 15)     // [10,14] hours
    {...}
  else
    if (hrs < 20)   // [15,19] hours
      {...}
    else
      if (hrs < 30) // [20,29] hours
        {...}
      else          // >30 hours
        {...}
```

⇒ Indentation is a nightmare!

# Multiway If-Else

Better solution:

```
if (hrs < 10)
  {...}
else if (hrs < 15) // [10,14] hours
  {...}
else if (hrs < 20) // [15,19] hours
  {...}
else if (hrs < 30) // [20,29] hours
  {...}
else               // >30 hours
  {...}
```

- Not a new construct, just a different (better) indentation style

# Selection with Many Cases

- A statement for controlling multiple ($>2$) branches
- Condition must be based on some expression that evaluates to an integral value (all types of int, char, some others)
- Can do the same with if statements, but `switch` may be more convenient and readable
- Especially useful for 'menus' (one case for each menu option)

# Selection: Switch

```
switch (expr) {
  case val_1:
    stmt_1               executed if expr evaluates to val1
    break;               exit [entire] switch statement
  case val_2:
    stmt_2
    break;
  ...
  case val_n
    stmt_n
    break;
  default:               executed if none of above cases applied
    default_stmt
  }
```

- Expr must be integral
- Without break, control goes through the next case (common error to omit break, but sometimes useful)

# Switch Example

```
char testGrade;
switch(testGrade) {
  case 'a':
    cout << "Congratulations!";
    break;            prevents next message from being printed
  case 'b':
    cout << "Just␣a␣little␣more␣work␣needed";
    break;
  case 'c':
    cout << "Need␣to␣work␣harder!";
    break;
  case 'd':           no break; falls through to next case
  case 'f':
    cout << "Not␣good";
    break;
  default:            don't forget the 'error' case!
    cout << "Error:␣unrecognized␣grade";
  };
cout << endl;         after switch; done in all cases
```

# Selection: Conditional

Shorthand for [typically] 1-line if-else

```
if (m > n)
  max = m;
else
  max = n;
```

≡

```
max = (m > n) ? m : n;
```

- Also called *ternary* operator
- Good for quick one-liner, but can be misused:

  ❓ What does this do? `(a<b) ? ((b<c) ? b : ((a<c) ? c : a) ) : ((a<c) ? a : ((b<c) ? c : b))`

  ❗ median of a,b,c
- Avoid mis-/over- use, but be prepared to see it in code

# Exercises

1. Write **conditions** for each of the following, making sure it works for all cases:
    1. `name` (a string, possibly empty) starts with one of {a,b,c,d,e}
    2. `name` starts with one of {a,b,c,d,A,B,C,D}
    3. `name` starts and ends with with one of {a,A}
    4. `n` (an int) is positive and even
    5. `n` is at most $2^{30}$ (so it can be doubled without overflow, assuming 32-bit ints)
    6. The point (x,y) in the Cartesian plane is outside a circle of radius r (`x,y,r` are doubles).
       Hint: what is the equation of a circle?
2. Write a **driver** program to test the above conditions.
   *i.e.*, input a name (or n), evaluate the condition, and output the result of the condition.
3. Modify the driver to iterate so that you can run multiple **test vectors**.