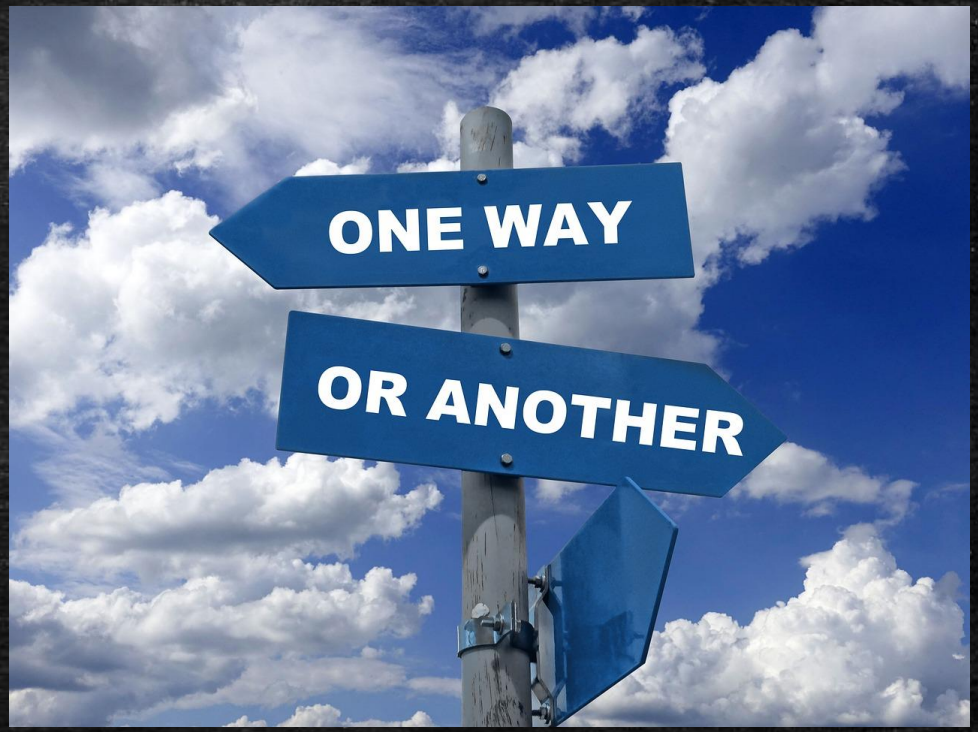
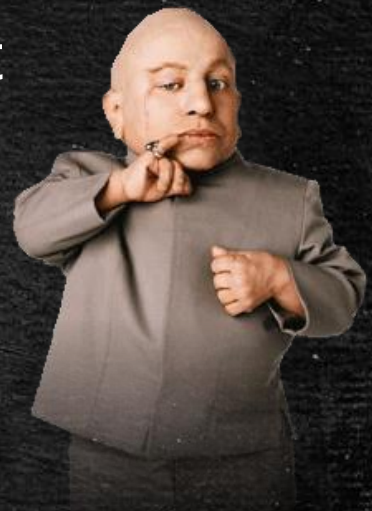
Functions & Decisions



CS 150 – C++ Programming I
Lecture 4

Writing Functions



- We are going move away from IPO programs and start moving towards writing functions
  - A named portion of code like *sqrt()*
  - Think of these as "mini" IPO programs
- Here are the major differences:
  - No user input. Input only through the parameter list
  - No visible output: output is returned from the function
- Can still have a *main()* function with regular input and output to test the function; normally no I/O in the function

Arguments & Parameters

- Most functions require extra information
  - *sqrt()* requires the number to operator on
- This is accomplished by using arguments and parameters
  - Arguments: values supplied when a function is called
  - Parameters: variables used in the function to store value
- Functions may be fruitful or void
  - Fruitful functions can be used as a value in expressions
  - void functions (AKA procedures) need a side-effect

Hands On: lastDigit

- Given an integer, return the last digit of the number
  - *lastDigit(3575) -> 5*
- Always start with a "first-draft" skeleton by asking:
  - 1. What kind of thing does the function return?
  - 2. What is the name of the function?
  - 3. Place input variable declaration in ( )
  - 4. Add braces to surround the function body.
  - 5. Create a variable for the result, and initialize
  - 6. Return the result
- Make sure this compiles and links, then complete

Decisions & the Structure Theorem

- What do we need for a programming language?
  - What features are required?
- You can write any program using three control structures[1]
  - Sequence : statements in order (1st, 2nd, 3rd, etc.)
    - That's what we've done with IPO programs (prompt, input, process…)
  - Selection : conditional execution (`if`, branching)
  - Iteration : conditional repetition (`while`, looping)
- Today, we're going to start with selection

[1] Bohm, C and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 336-371

Understanding C++ Conditions

- Decision making in C++ is based on these three foundations
  - 1. The built-in *bool* type: *true* & *false*
  - 2. Relational operators compare values (of any type) and produce a Boolean result
  - 3. Logical operators operate on Boolean expressions to combine or negate them
- Unfortunately, C++ truthiness is a little more "flexible" than you might expect, coming from Python or Java
  - As with numbers, C++ has implicit *bool* conversions
  - If the result is `0`, it is `false`; otherwise it is `true`!

Some C++ Boolean Pitfalls

- Numbers and *Boolean* values are implicitly convertible
  - Number to a `bool`: `0`->`false`, otherwise `true`
  - Assign `bool` to a number: `false`->`0`, `true`->`1`
  - Input or `output`? Default? `0` and `1`
    - Use `boolalpha` manipulator to use `true`/`false` like Java
- Embedded assignment: *`if (area = 0) …`*
  - Assigns `0` to `area`, Boolean value is `false`
  - If you assign a non-zero, then the condition is `true`
  - Java and C# protect you from this error (sometimes)

Quest

```
int a = 3;
if (a = 4)
    cout << "a is 4; weird!" << endl;
else
    cout << "a NOT 4" << endl;
```

- What prints?
  - A. `a is 4; weird!`
  - B. `a NOT 4`
  - C. Syntax error
  - D. Neither one

More Conditional Pitfalls

- Impossible Condition: if you use `&&`, make sure it is possible for all conditions to be `true` (simultaneously)
  - `if (age < 13 && age > 65) // impossible`
- Unavoidable Condition: if you use `||`, make sure it is possible for all conditions to be `false` (simultaneously)
  - `if (age > 13 || age < 65) // unavoidable`
- Implicit conversion: always use complete relational expressions in conditions. Not a syntax error.
  - `if (age == 12 || 13 || 14)  // always true`

Question

```
string grade = "C";
if (grade == "A" || "A+" || "A-")
    cout << "Got an A" << endl;
else if (grade == "B" || "B+" || "B-")
    cout << "Got a B" << endl;
else if (grade == "C" || "C+" || "C-")
    cout << "Got a C" << endl;
```

- What prints? (Assume all includes, etc.)
  - A. `Got an A`
  - B. `Got a B`
  - C. `Got a C`
  - D. Syntax error

Real Number Comparison Pitfall

- Real-numbers: syntactically comparable, but . . .
  - Naïve solutions lead to errors

```
double root = sqrt(2);
if (root * root == 2.0) cout << "OK" << endl;
else cout << "Back to Math 30" << endl;
```

- No perfect solution: in general
  - Decide what's "close enough" (epsilon)

```
const double EPSILON = 1.0e-14
```

  - Is the absolute value of difference less than epsilon?

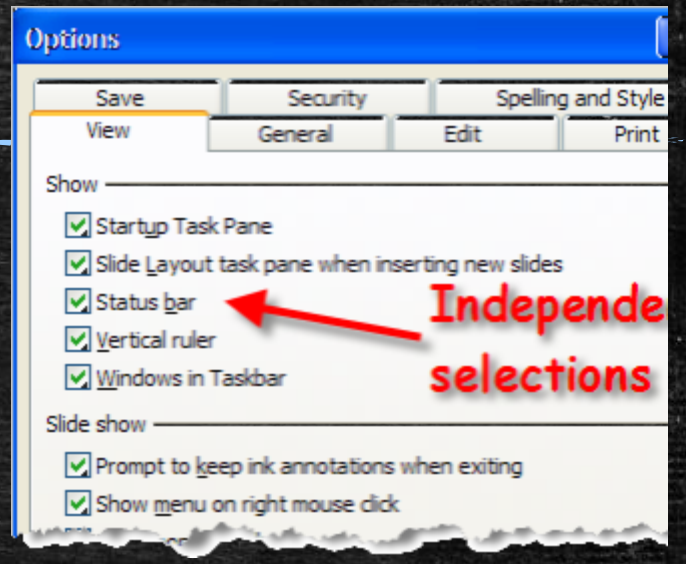
Reasons to Use Selection

- Selection is used to control the flow of data:
  - Select a particular value or kind of value from a data flow (all debits or credits, for instance)
  - Selectively update a counter or accumulator (counting vowels and consonants)
  - Route data to the correct part of a program in response to user commands (ie menus)
  - Error check data; make sure it falls within boundaries or correct it
- Selection produces information: IPO

Six Selection Structures

- One-way: independent - `if ...`
- Two-way: either-or actions - `if ... else ...`
- Sequential: dependent - `if ... else if ... else ...`
- Nested: leveled - `if if ... else ... else if ... else ...`
- Labeled: integral - `switch(selector) ... case ?:`
- Expression: `val = test? true : false`
- #1: produce the *correct* output
- #2: understandable, *maintainable* code
- #3: use the correct *semantics*
- #4: write *efficient* code

Guarded & Alternative Actions



- Are decisions and output truly independent?
  - If so, you must use independent *if*s

```
if (startup.Checked())…
if (layout.Checked())…
```

  - Called the guarded action idiom
  - No other cases should ever use independent *if*s
- Either/Or decisions or a 2-way branch
  - Also known as the alternative action idiom
  - Always use *if-else* (covers all possibilities)
- Exercise: *DoubleSum* (which type of decision is correct)?

Multi-way Interdependent Tests

- Examine this. What grade do you get if your percent is .65?
  - Test order is significant; what do you get now?
  - Not the correct way to write interdependent tests

```
string result;
double percent = . . .
if (percent > 1.0)  result = "Out of range";
if (percent <= 1.0) result = "A";
if (percent < .90)  result = "B";
if (percent < .70)  result = "D";
if (percent < .80)  result = "C";
if (percent < .55)  result = "F";
if (percent < 0.0)  result = "Out of range";
```

Sequential *if* statements

- Used when only one condition can be true
  - AKA as "ladder-style" *if-else-if* statements

```
if (n == 1) result = "one";
else if (n == 2) result = "two";
else if (n == 3) result = "three";
//. . . and so on
else result = "four billion, two hundred and
```

  - Correct, understandable, order matters (more care)
  - Semantically correct - says there will be one output
  - Efficient - once answer is found, no more checking
- Exercise: *Four Seasons*