University of Southern California

Viterbi School of Engineering

# EE355
# Software Design for Electrical Engineers

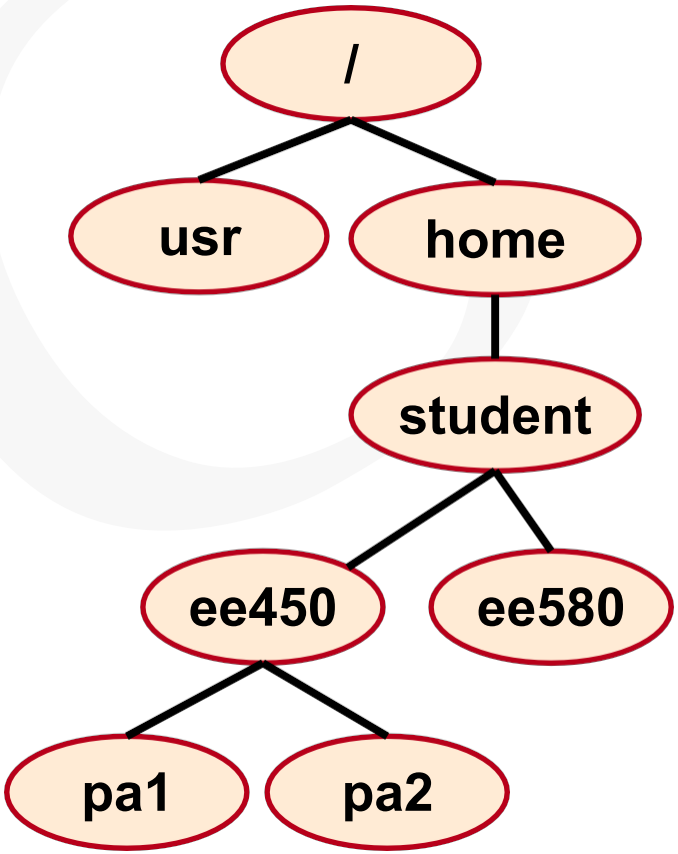
# EE599 – Special Topics:
# Software Design and Optimization

## C Review

# LINUX (MAC OS) FILE SYSTEM

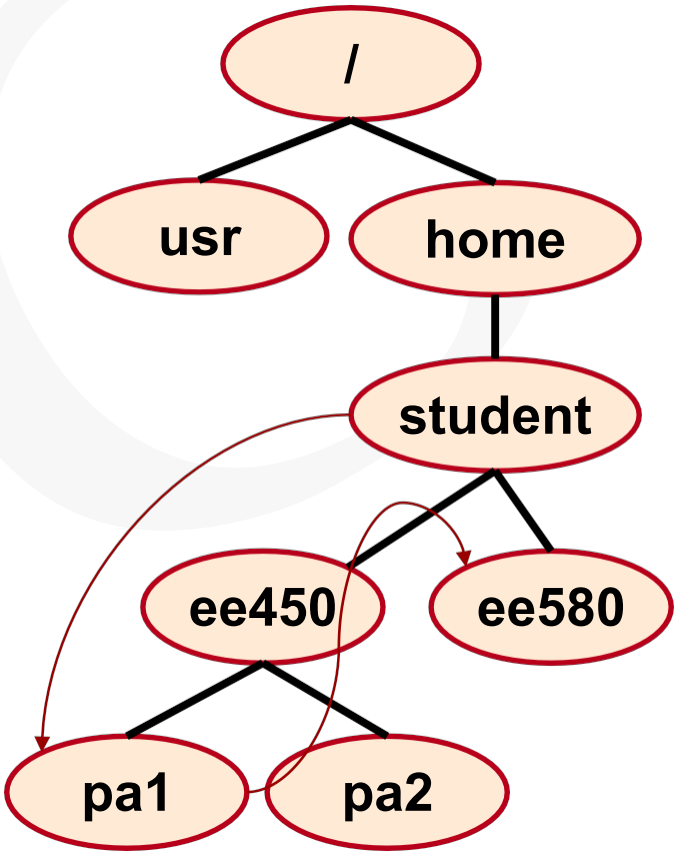
# File System Structure

- Hard drive starts at a folder names `/`
- Programs live in `/usr` or `/bin`
- Each user has a folder under `/home`
- When you start the terminal application you will start in `/home/student`



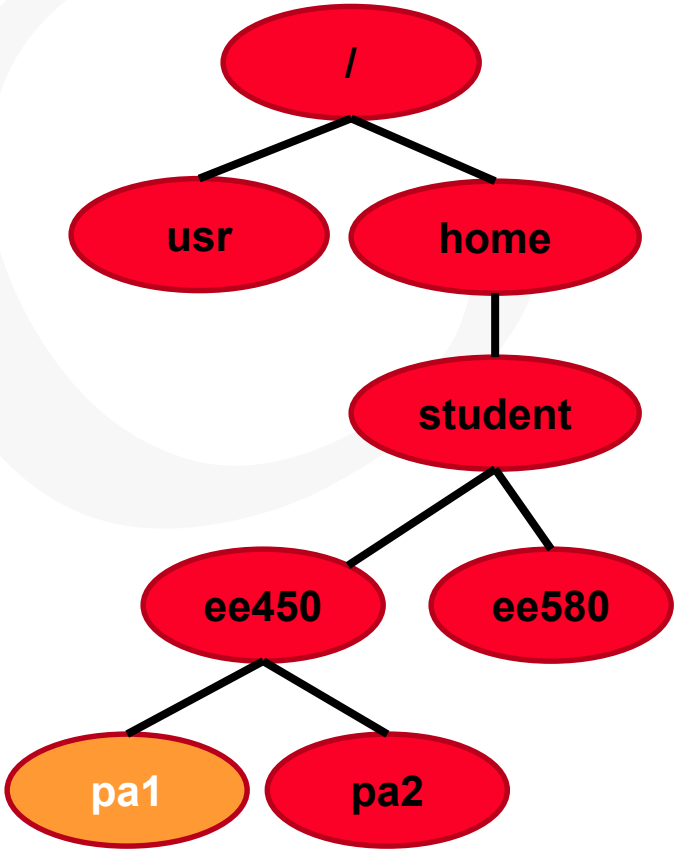
# Navigating the File System

- Use 'cd' to change directories
  - Remember . is a shortcut to the current path
  - .. is a shortcut to the parent folder
  - Paths can be relative or absolute
- Examples
  - Change to the pa1 folder
  - _____
  - Change from the pa1 folder to the ee580 folder
  - _____



# Copying

- Use 'cp' to copy files
  - You can use wildcarding (*)
- Examples
  - Copy all .cpp files from pa1 to ee301
  - `cd ee450/pa1`
  - _____
  - Copy all files from ee580 to current folder (pa1)
  - _____




Write your 'cp' command assuming you are here

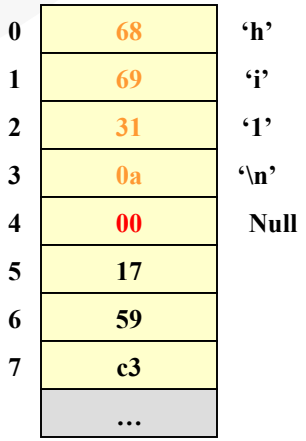
# DATA TYPES

# Constants

- Integer: 496, 10005, -234
- Double: 12.0, -16., 0.23, -2.5E-1, 4e-2
- Float: 12.0F // F = float vs. double
- Characters appear in single quotes
  - `'a', '5', 'B', '!', '\n', '\t', '\r', '\\', '\''`
- C-Strings
  - Multiple characters between double quotes

`"hi1\n", "12345\n", "b", "\tAns. is %d"`

  - Ends with a `'\0'`=NULL character added as the last byte/character
- Boolean (C++ only): true, false
  - Physical representation: 0 = false, (!= 0) = true




String Example

(Memory Layout)

# Just My Type

- Indicate which constants are matched with the correct type

| Constant | Type | Right / Wrong |
|---|---|---|
| 4.0 | int | |
| 5 | int | |
| 'a' | string | |
| "abc" | string | |
| 5. | double | |
| 5 | char | |
| "5.0" | double | |
| '5' | int | |

# Understanding ASCII and chars

```
char c = 'a';       // same as  char c = 97;
char c = 'a' + 1;   // c now contains 'b' = 98;
cout << a << endl; // I will see 'b' on the screen

char c = '1';       // c contains decimal 49, not 1
                    // i.e. '1' not equal to 1

c >= 'a' && c <= 'z'; // && means AND
                      // here we are checking if c
                      // contains a lower case letter
```

char c

97

ASCII printable characters

| | | | | | |
|---|---|---|---|---|---|
| 32 | space | 64 | @ | 96 | ` |
| 33 | ! | 65 | A | 97 | a |
| 34 | " | 66 | B | 98 | b |
| 35 | # | 67 | C | 99 | c |
| 36 | $ | 68 | D | 100 | d |
| 37 | % | 69 | E | 101 | e |
| 38 | & | 70 | F | 102 | f |
| 39 | ' | 71 | G | 103 | g |
| 40 | ( | 72 | H | 104 | h |
| 41 | ) | 73 | I | 105 | i |
| 42 | * | 74 | J | 106 | j |
| 43 | + | 75 | K | 107 | k |
| 44 | , | 76 | L | 108 | l |
| 45 | - | 77 | M | 109 | m |
| 46 | . | 78 | N | 110 | n |
| 47 | / | 79 | O | 111 | o |
| 48 | 0 | 80 | P | 112 | p |
| 49 | 1 | 81 | Q | 113 | q |
| 50 | 2 | 82 | R | 114 | r |
| 51 | 3 | 83 | S | 115 | s |
| 52 | 4 | 84 | T | 116 | t |
| 53 | 5 | 85 | U | 117 | u |
| 54 | 6 | 86 | V | 118 | v |
| 55 | 7 | 87 | W | 119 | w |
| 56 | 8 | 88 | X | 120 | x |
| 57 | 9 | 89 | Y | 121 | y |
| 58 | : | 90 | Z | 122 | z |
| 59 | ; | 91 | [ | 123 | { |
| 60 | < | 92 | \ | 124 | \| |
| 61 | = | 93 | ] | 125 | } |
| 62 | > | 94 | ^ | 126 | ~ |
| 63 | ? | 95 | _ | | |

# C/C++ Intrinsic Data Types

- In C/C++ variables can be of different types and sizes
  - Integer Types (signed by default…unsigned with leading keyword)

| C Type | Bytes | Bits | Signed Range | Unsigned Range |
|---|---|---|---|---|
| **[unsigned] char** | 1 | 8 | -128 to +127 | 0 to 255 |
| [unsigned] short[int] | 2 | 16 | -32768 to +32767 | 0 to 65535 |
| [unsigned] long [int] **[unsigned] int** | 4 | 32 | -2 billion to +2 billion | 0 to 4 billion |
| [unsigned] long long [int] | 8 | 64 | $-8*10^{18}$ to $+8*10^{18}$ | 0 to $16*10^{18}$ |

  - Floating Point Types (represent fractional or very large numbers)

| C Type | Bytes | Bits | Range |
|---|---|---|---|
| float | 4 | 32 | ±7 significant digits * $10^{+/-38}$ |
| **double** | 8 | 64 | ±16 significant digits * $10^{+/-308}$ |

# C/C++ Variables

- Variables have a:
  - type [`int`, `char`, `unsigned int`, `float`, `double`, etc.]
  - name/identifier that the programmer will use to reference the value in that memory location [e.g. `x`, `myVariable`, `cs101_variable_name`, etc.]
    - Lookup what legal characters can start and/or be part of a variable identifier/name (e.g. must start with A/a – Z/z or an underscore '_', etc.)
    - Use descriptive names(e.g. numStudents, doneFlag)
    - Avoid cryptic names ( myvar1, a_thing )
  - location [the address in memory where it is allocated]
  - Value
- Reminder: You must declare a variable before using it

Code

```
int quantity = 4;
double cost = 5.75;
cout << quantity*cost << endl;
```

Computer

# C/C++ Variables

- Values provided at run-time (by the user or some other data source) that are NOT known at compile-time
  - What values will change over the course of execution
  - Variable to store the current URL of your web-browser
- Values we want to save and re-use
  - Unless you assign a value into a variable that value is computed and then thrown away [ x + 5; is a valid statement, but will be computed and tossed...need to do something like x = x + 5; or y = x + 5;]
  - What values do we NOT want to compute over and over (i.e., compute it once & store it in a variable)
- Making code readable by giving values logical names
  - Compute area of a rectangle with height = $(a^2 + 4a + 5)$ and width = $(5i^3 - 3i + 8)$...
  - Could write $(a^2 + 4a + 5) * (5i^3 - 3i + 8)$
  - Might be more readable to put terms into separate variables 'h' and 'w' and then multiply result and place into a variable 'area' (area = h * w;)

# Arithmetic Operators

- Addition, subtraction, multiplication work as expected for both integer and floating point types
- Division works 'differently' for integer vs. doubles/floats
- Modulus is only defined for integers

| Operator | Name | Example |
|---|---|---|
| + | Addition | b + 5 |
| - | Subtraction | c - x |
| * | Multiplication | a * 3.1e-2 |
| / | Division<br>(Integer vs. Double division) | 10 / 3<br>(result will be 3) |
| % | Modulus (remainder)<br>[for integers only] | 17 % 5<br>(result will be 2) |
| ++ or -- | Increment (add 1) or<br>Decrement (subtract 1) | e++ (e = e+1)<br>i-- (i = i-1) |

# Precedence

- Order of operations/ evaluation of an expression
- Top Priority = highest (done first)
- Notice operations with the same level or precedence usually are evaluated left to right (explained at bottom)
- Evaluate:
  - `2*-4-3+5/2;`
- Tips:
  - Use parenthesis to add clarity
  - Add a space between literals (2 * -4) – 3 + (5 / 2)

## Operators (grouped by precedence)

| | |
|---|---|
| struct member operator | *name*`.`*member* |
| struct member through pointer | *pointer*`->`*member* |
| increment, decrement | `++, --` |
| plus, minus, logical not, bitwise not | `+, -, !, ~` |
| indirection via pointer, address of object | `*`*pointer*`, &`*name* |
| cast expression to type | `(`*type*`)` *expr* |
| size of an object | `sizeof` |
| multiply, divide, modulus (remainder) | `*, /, %` |
| add, subtract | `+, -` |
| left, right shift [bit ops] | `<<, >>` |
| relational comparisons | `>, >=, <, <=` |
| equality comparisons | `==, !=` |
| and [bit op] | `&` |
| exclusive or [bit op] | `^` |
| or (inclusive) [bit op] | `\|` |
| logical and | `&&` |
| logical or | `\|\|` |
| conditional expression | $expr_1$ `?` $expr_2$ `:` $expr_3$ |
| assignment operators | `+=, -=, *=, ...` |
| expression evaluation separator | `,` |

Unary operators, conditional expression and assignment operators group right to left; all others group left to right.

# Assignment operator '='

- Syntax:

  variable = expression;

  (LHS) (RHS)

- LHS = Left Hand-Side, RHS = Right Hand Side
- Should be read: Place the value of *expression* into memory location of *variable*
  - z = x + 5 – (2*y);
- When variables appearing on RHS indicate the use of their associated value. Variables on LHS indicate location to place a value.
- Shorthand operators for updating a variable based on its current value:

  `+=, -=, *=, /=, &=, …`
  - `x += 5;` `(x = x+5)`
  - `y *= x;` `(y = y*x)`

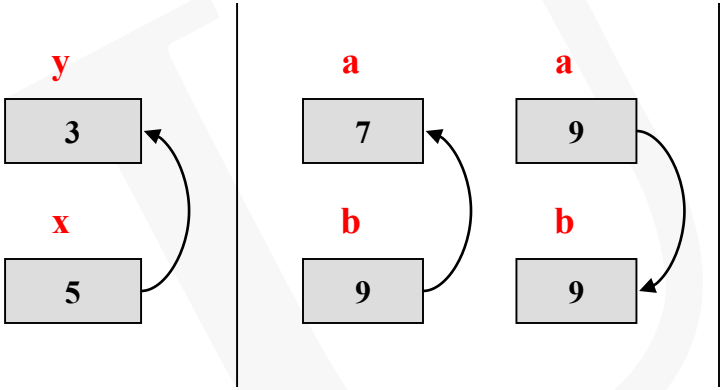
# More Assignments

- Assigning a variable makes a copy
- Challenge: Make a copy

```
int main()
{
  int x = 5, y = 3;
  x = y;   // copy y into x

  // now consider swapping
  // the value of 2 variables
  int a = 7, b = 9;
  a = b;
  b = a;


  return 0;
}
```



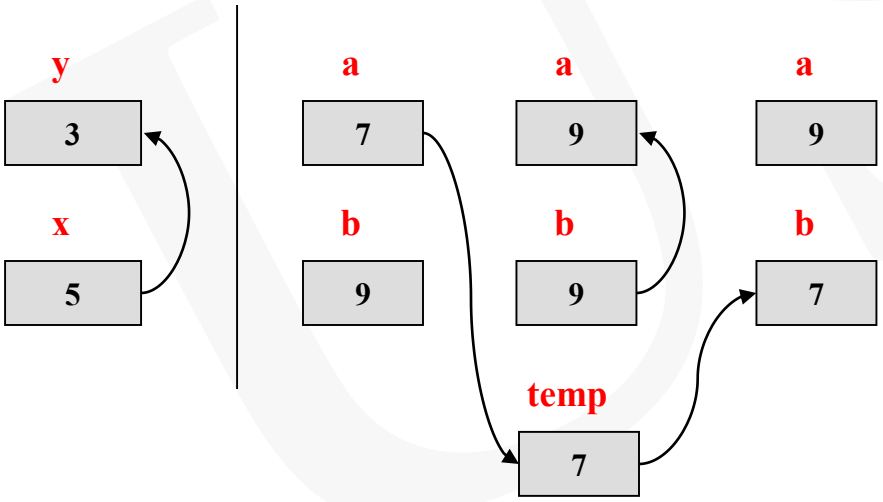
# More Assignments

- Assigning a variable makes a copy
- Challenge: Make a copy
  - Easiest method: Use a 3rd temporary variable to save one value and then replace that variable

```
int main()
{
  int x = 5, y = 3;
  x = y;    // copy y into x

  // let's try again
  int a = 7, b = 9, temp;
  temp = a;
  a = b;
  b = temp;


  return 0;
}
```



# Summary Examples 1

- maxplus
- 4swap

# Evaluate 5 + 3/2

- The answer is 6.5 ??

# Casting

- To achieve the correct answer for 5 + 3 / 2
- Could make everything a double
  - Write 5.0 + 3.0 / 2.0 [explicitly use doubles]
- Could use implicit casting (mixed expression)
  - Could just write 5 + 3.0 / 2
    - If operator is applied to mixed type inputs, less expressive type is automatically promoted to more expressive (int => double)
- Could use C or C++ syntax for explicit casting
  - 5 + (double) 3 / (double) 2 (C-Style cast)
  - 5 + static_cast<double>(3) / static_cast<double>(2) (C++-Style cast)
  - 5 + static_cast<double>(3) / 2 (can cast only one, rely on implicit cast of the other)
  - This looks like a lot of typing compared to just writing 5 + 3.0 / 2...but what if instead of constants we have variables
  - int x=5, y=3, z=2; x + y/z;
  - x + static_cast<double>(y) / z

# Optional: Exercise Review

- C++ Programming, 5th Ed., Ch. 2
  - Q6:
    - 25/3
    - 20-12/4*2
    - 33 % 7
    - 3 – 5 % 7
    - 18.0 / 4
    - 28 - 5 / 2.0
    - 17 + 5 % 2 - 3
- Assignment
  - int num1, num2, num3;
  - num1 = 3; num2 = 5;
  - num3 = 12 * ++num1 - 4
  - num3 = 12 * num1-- – 4
  - num1 * num2 = num3
  - num1++ = num2
- C++ Programming, 5th Ed. Ch. 4
  - Q3
  - Q5

# Pre- and Post-Increment

- ++ and -- operator and be used in an expression and cause the associated variable to "increment-by-1" or "decrement-by-1"
- Timing of the increment or decrement depends on whether pre- or post-increment (or decrement) is used
  - y = x++ + 5;  // Post-increment since ++ is used after the variable x
  - y = ++x + 5;  // Pre-increment since ++ is used before the variable x
- Meaning:
  - Pre: Update the variable before using it in the expression
  - Post: Use the old value of the variable in the expression then update it
- Examples [suppose int y; int x = 3; ]
  - y = x++ + 5; // Use old value of x and add 5, but add 1 to x after computing result
    // Result: y = 8, x = 4
  - y = ++x +5; // Increment x and use its new value when you add 5
    // [Result: y = 9, x = 4]
  - y = x-- + 5; // Use old value of x and add 5, but subtract 1 from x after
    // evaluating the expression [Result: y = 8, x = 2]

# Activity

- Consider the code below
  - `int x=5, y=7, z;`
  - `z = x++ + 3*y-- + 2*x++;`
- What is the value of x, y, and z after this code executes

# Summary Examples

- funccall
- Tacos
- Hello
- Quadratic
- Math

# I/O STREAMS

# C++ Streams

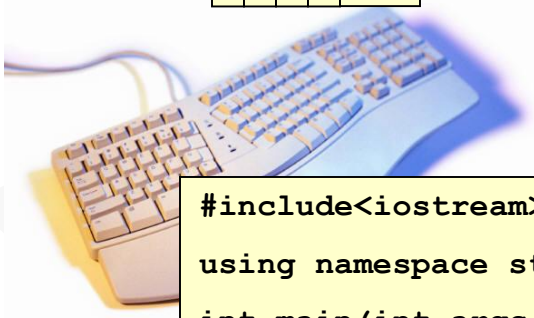
- What is a “stream”?
  - A sequence of characters or bytes (of potentially infinite length) used for input and output
- C++ has four major libraries we will use for streams:
  - <iostream>
  - <fstream>
  - <sstream>
  - <iomanip>
- C++ has two operators that are used with streams
  - Insertion Operator “<<”
  - Extraction Operator “>>”

# I/O Streams

- cin goes and gets data from the input stream (skipping over preceding whitespace then stopping at following whitespace)
- cout puts data into the output stream for display by the OS ('endl' = newline character + forces the OS to flush/display the contents immediately)

**input stream:**

| 7 | 5 | | y | ... |
|---|---|---|---|---|



```
#include<iostream>
using namespace std;
int main(int argc, char *argv[])
{
  int x;  cin >> x;
}
```

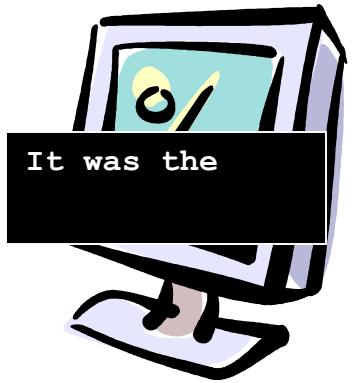
**input stream:**

| | y | ... |
|---|---|---|

```
#include<iostream>
using namespace std;
int main(int argc, char *argv[])
{
  cout << "It was the" << endl;
  cout << "4";
}
```

**output stream:**

| I | t | | w | a | s | | t | h | e | \n | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|




**output stream:**

| 4 |
|---|

# C++ I/O Manipulators

- The <iomanip> header file has a number of "manipulators" to modify how I/O behaves
  - Alignment: internal, left, right, setw, setfill
  - Numeric: setprecision, fixed, scientific, showpoint
  - http://www.cplusplus.com/reference/iostream/manipulators/
- Use these inline with your cout/cerr/cin statements
  - double pi = 3.1415;
  - cout << setprecision(2) << fixed << pi << endl;

# Formatted I/O with cout

- #include<iomanip>
- Tabular output
  - cout << setw(10) << 105 << setw(10) << 23 << endl;
- Floating point precision
  - cout << setprecision(2) << fixed << 98.5091 << endl;

```
       105        23
```

Field width of 10 characters with item printed right-justified

```
98.51
```

# C++ Console Input

- cin can be used to accept data from the user
  - int x;  // MUST DECLARE VARIABLE BEFORE USING W/ CIN
  - cout << “Enter a number: “;
  - cin >> x;
- What if the user does not enter a valid number?
  - Check cin.fail( ) to see if the read worked
  - Use cin.clear( ) & cin.ignore(...) on failure
- What if the user enters multiple values?
  - cin reads up until the first piece of whitespace
- The getline(...) method that will read an entire line (including whitespace) :
  - char x[80];
  - cin.getline(x, 80);

# Understanding cin

- User enters value "512" at 1st prompt, enters "123" at 2nd prompt

```
int x=0;

cout << "Enter X: ";

cin >> x;
```

X = 0 cin =

X = 0 cin = 5 1 2 \n

X = 512 cin = \n

cin.fail() is false

```
int y = 0;

cout << "Enter Y: ";

cin >> y;
```

Y = 0 cin = \n

Y = 0 cin = \n 1 2 3 \n

Y = 123 cin = \n

cin.fail() is false

# Understanding cin (cont.)



- User enters value "23 99" at 1st prompt, 2nd prompt skipped

*int x=0;*

X = 0 cin =

*cout << "Enter X: ";*

X = 0 cin = 2 3 ␣ 9 9 \n

*cin >> x;*

X = 23 cin = ␣ 9 9 \n

cin.fail() is false

*int y = 0;*

Y = 0 cin = ␣ 9 9 \n

*cout << "Enter Y: ";*

Y = 0 cin = ␣ 9 9 \n

*cin >> y;*

Y = 99 cin = \n

cin.fail() is false

# Understanding cin (cont.)



- User enters value "23abc" at 1st prompt, 2nd prompt fails

```
int x=0;

cout << "Enter X: ";

cin >> x;
```

X = 0 cin =

X = 0 cin = 2 3 a b c \n

X = 23 cin = a b c \n

cin.fail() is false

```
int y = 0;

cout << "Enter Y: ";

cin >> y;
```

Y = 0 cin = a b c \n

Y = 0 cin = a b c \n

Y = xx cin = a b c \n

cin.fail() is true

# Understanding cin (cont.)

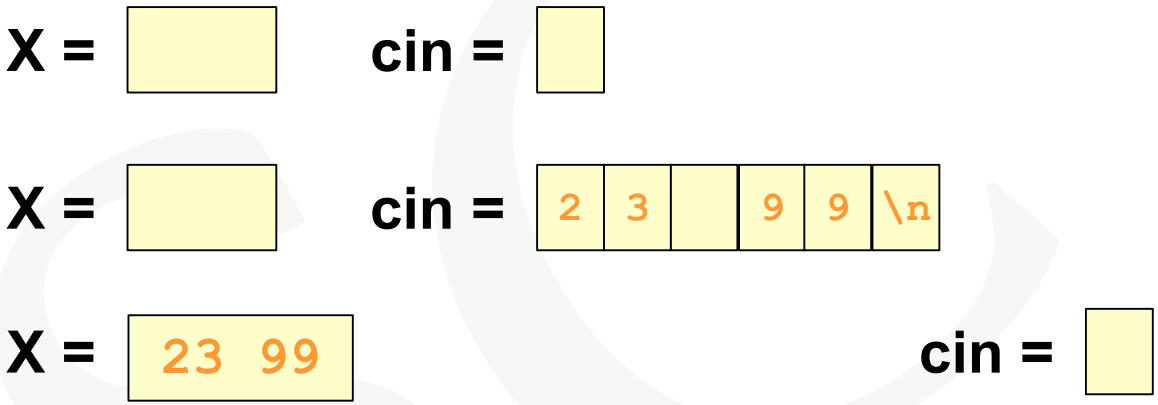


- User enters value "23 99" at 1st prompt, everything read as string

*char x[80];*

*cout << "Enter X: ";*

*cin.getline(x, 80);*




cin.fail() is
false

NOTE: \n character is discarded!

# CONTROL STRUCTURES

# Comparison/Logical Operators

- If, while, for structures require a condition to be evaluated resulting in a True or False determination.
- In C...
  - 0 means False
  - Non-Zero means True
- Example 1

```
int x = 100;
while(x)
   { x--;}
```

- Example 2

```
char flagged = 0;
flagged = evaluate_quality_of_item(); //returns 1 if
   bad
if( flagged )
   { throw_out_item(); }
```

- Usually conditions results from comparisons

  ==, !=, >, <, >=, <=

# Logical AND, OR, NOT

- Often want to combine several conditions to make a decision
- Logical AND => expr_a && expr_b
- Logical OR => expr_a || expr_b
- Logical NOT => ! expr_a
- Precedence (order of ops.) => ! then && then ||
  - !x || y && !z
  - (!x || (y && !z))
- Eat sandwich if has neither tomato nor lettuce
  - if ( !tomtato && !lettuce) { eat_sandwich(); }
  - if ( !(tomato || lettuce) ) { eat_sandwich(); }

| A | B | AND |
|---|---|---|
| False | False | False |
| False | True | False |
| True | False | False |
| True | True | True |

| A | B | OR |
|---|---|---|
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | True |

| A | NOT |
|---|---|
| False | True |
| True | False |

# Exercise

- Which of the following is NOT a condition to check if the integer x is in the range [-1 to 5]
  - x >= -1 && x <= 5
  - -1 <= x <= 5
  - ! ( x < -1 || x > 5)
  - x > -2 && x < 6
- Consider ( (!x) || (y && (!z) ) )
  If x=100, y= -3, z=0 then this expression is...
  - true
  - false

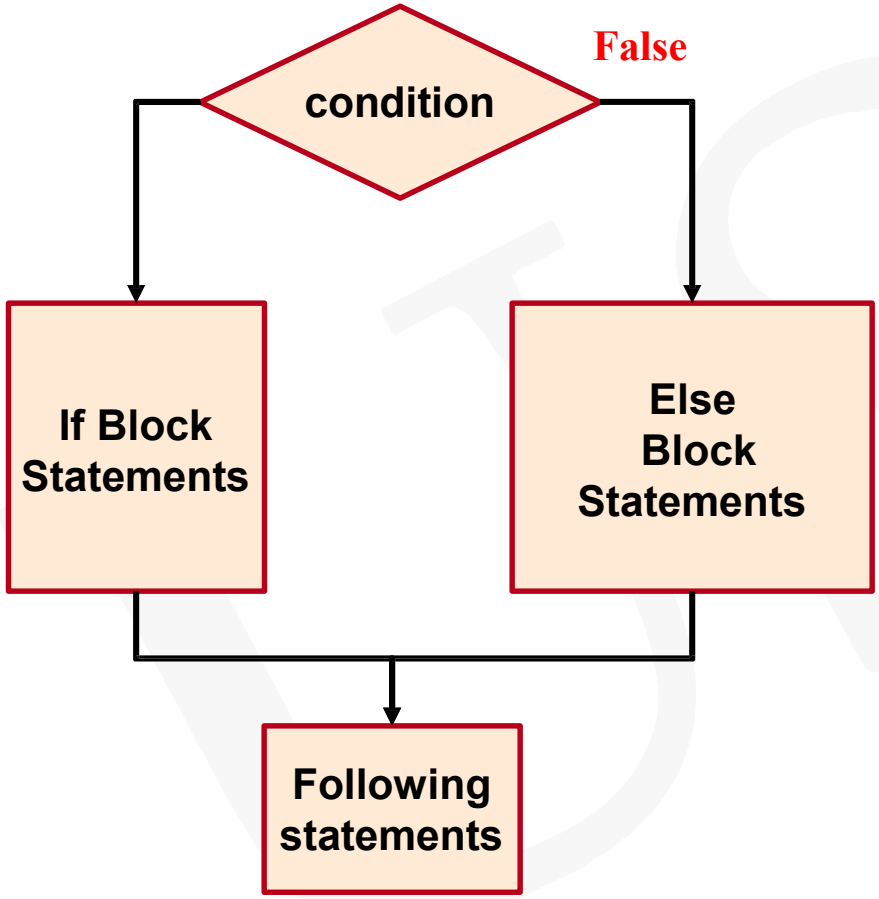
# If...Else If...Else

- Use to execute only certain portions of code
- Else If is *optional*
  - Can have any number of else if statements
- Else is *optional*
- { ... } indicate code associated with the if, else if, else block

```
if (condition1)
{
  // executed if condition1 is True
}
else if (condition2)
{
  // executed if condition2 is True
  //  but condition1 was False
}
else if (condition3)
{
  // executed if condition3 is True
  //  but condition1 and condition2
  //  were False
}
else
{
  // executed if neither condition
  // above is True
}
```

# Flow Chart



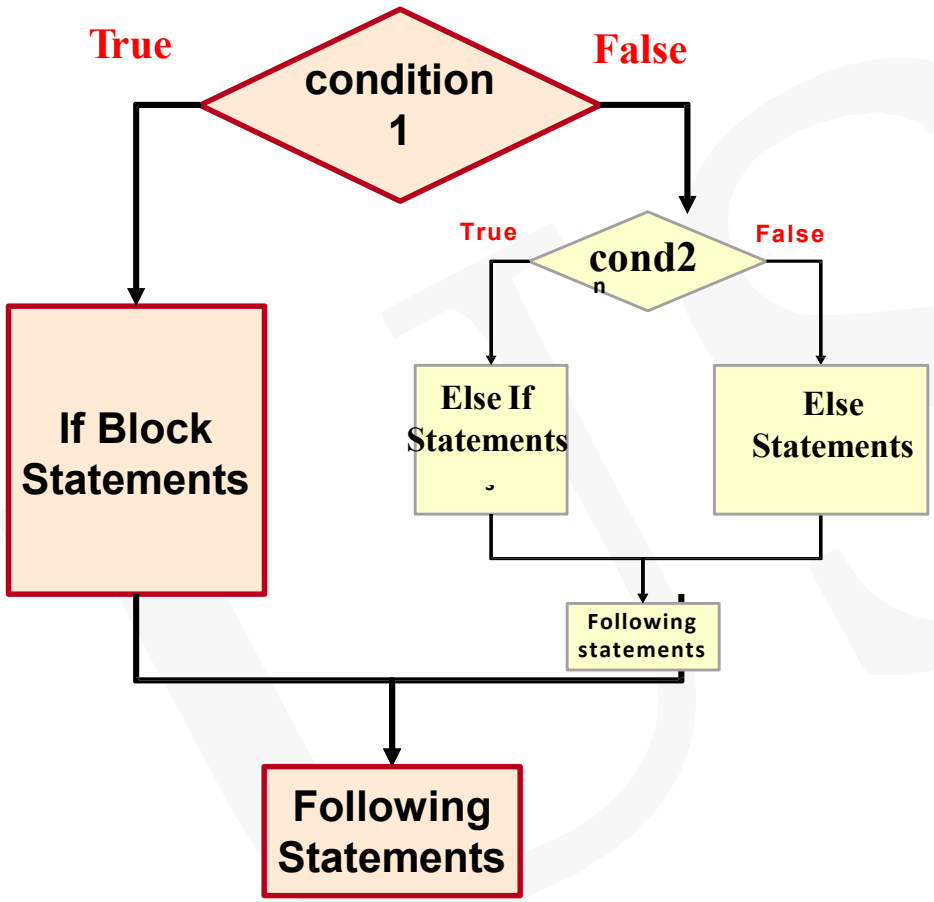



```
if (condition1)
{
   // executed if condition1 is True
}
else
{
   // executed if neither condition
   // above is True
}

// following statements
```

# Flow Chart (cont.)






```
if (condition1)
{
  // executed if condition1 is True
}
else if (condition2)
{
  // executed if condition2 is True
  //  but condition1 was False
}
else
{
  // executed if neither condition
  // above is True
}
// following statements
```

# Exercises

- Discount
- Weekday
- N-th

# The Right Style

- Is there a difference between the following two code snippets
- Both are equivalent but the bottom is preferred because it makes clear to other programmers that only one or the other case will execute

```
int x;
cin >> x;

if( x >= 0 ) { cout << "Positive"; }
if( x < 0  ) { cout << "Negative"; }
```

```
int x;
cin >> x;

if( x >= 0 ) { cout << "Positive"; }
else { cout << "Negative"; }
```

# Find the bug

- What's the problem in this code...

```
// What's the problem below
int x;
cin >> x;
if (x = 1)
  { cout << "X is 1" << endl;}
else
   { cout << "X is not 1" << endl; }
```

# Switch

- Again used to execute only certain blocks of code
- Best used to select an action when an expression could be 1 of a set of values
- { ... } around entire set of cases and not individual case
- Computer will execute code until a break statement is encountered
  - Allows multiple cases to be combined
- Default statement is like an else statement

```
switch(expr) // expr must eval to an int
{
 case 0:
   // code executed when expr == 0
   break;
 case 1:
   // code executed when expr == 1
   break;
 case 2:
 case 3:
 case 4:
   // code executed when expr is
   // 2, 3, or 4
   break;
 default:
   // code executed when no other
   // case is executed
   break;

}
```

# Switch (cont.)

- What if a break is forgotten?
  - All code underneath will be executed until another break is encountered

```
switch(expr) // expr must eval to an int
{
 case 0:
   // code executed when expr == 0
   break;
 case 1:
   // code executed when expr == 1
   // what if break was commented
   // break;
 case 2:
 case 3:
 case 4:
   // code executed when expr is
   // 3, 4 or 5
   break;
 default:
   // code executed when no other
   // case is executed
   break;

}
```

Performing repetitive operations

# LOOPS

# Need for Repetition

- We often want to repeat a task but do so in a concise way
  - Print out all numbers 1-100
  - Keep taking turns until a game is over
    - Imagine the game of 'war'...it never ends!!
- We could achieve these without loops, but...

```
#include <iostream>
using namespace std;

int main()
{
  cout << 1 << endl;
  cout << 2 << endl;
  ...
  cout << 100 << endl;
  return 0;
}
```

```
#include <iostream>
using namespace std;

int main()
{
  bool gameOver;
  gameOver = take_turn();
  if( ! gameOver ){
    gameOver = take_turn();
    if( ! gameOver ) {
        ...
    {
  }
}
```

# while Loop

- While
  - Cond is evaluated first
  - Body only executed if cond. is true (maybe 0 times)
- Do..while
  - Body is executed at least once
  - Cond is evaluated
  - Body is repeated if cond is true
- Rule of thumb: Use when number of iterations is determined by a condition updating inside the loop body

```
// While Type 1:
while(condition)
{
  // code to be repeated
  // (should update condition)
}



// While Type 2:
do {
  // code to be repeated
  // (should update condition)
} while(condition);
```

# while Loop (cont.)

- One way to think of a while loop is as a repeating 'if' statement
- When you describe a problem/solution you use the words 'until some condition is true' that is the same as saying *'while* some condition is *not* true'

```
// guessing game
bool guessedCorrect = false;

if( ! guessedCorrect )
{
   guessedCorrect = guessAgain();
} // want to repeat if cond. check again
```

**An if-statement will only execute once**

```
// guessing game
bool guessedCorrect = false;

while( ! guessedCorrect )
{
   guessedCorrect = guessAgain();
}
```

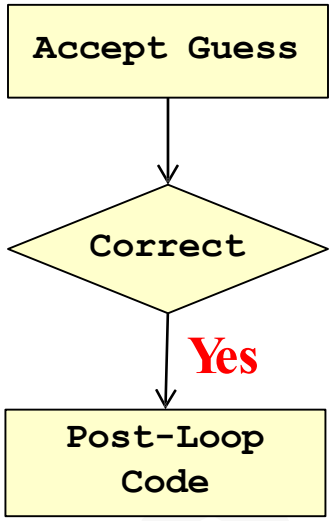
**A 'while' loop acts as a repeating 'if' statement**

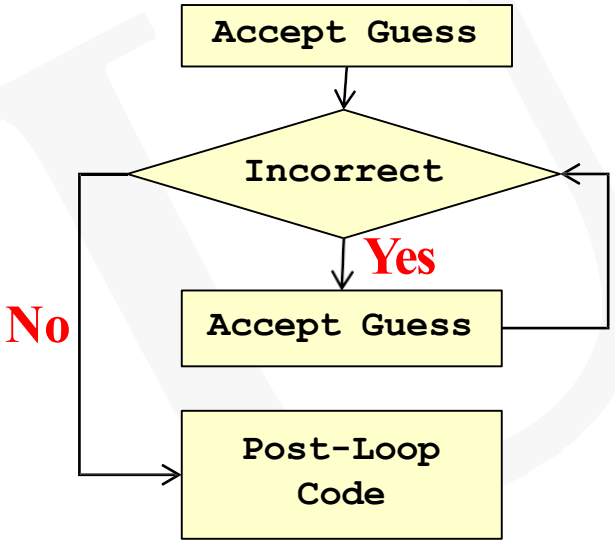
# While Loop Exercise

- Exercises
  - countodd

# Finding the 'while' Structure





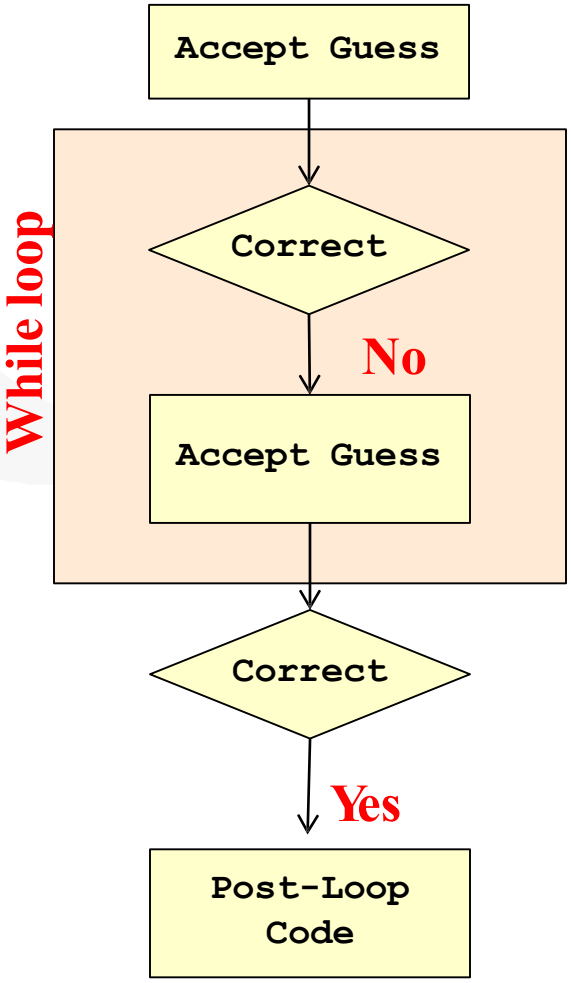

Here we check at the end to see if we should repeat…perfect for a do..while loop

```
do
{ body }
while (cond)
```

But a while loop checks at the beginning of the loop, so we must accept one guess before starting:

```
while( cond )
{ body }
```

"Unroll" (show more iterations of) the loop and re-organize



# for Loop

- Init stmt executed first
- Cond is evaluated next
- Body only executed if cond. is true
- Update stmt executed
- Cond is re-evaluated and execution continues until it is false
- Multiple statements can be in the init and update statements
- Rule of thumb: Use when number of iterations is independent of loop body

```
for(init stmt; cond; update stmt)
{
  // body of loop
}

// Outputs 0 5 10 15 … 95
for(i=0; i < 100; i++){
  if(i % 5 == 0){
    cout << i << " is a multiple of 5";
    cout << endl;
  }
}

for(i=0; i < 20; i++){
  cout << 5*i << " is a multiple of 5";
  cout << endl;
}

// compound init and update stmts.
for(i=0, j=0; i < 20; i++,j+=5){
  cout << j << " is a multiple of 5";
  cout << endl;
}
```

# for Loop (cont.)

- Convert the following for loops to equivalent while loop structures

```
for(init stmt; cond; update stmt)
{
  // body of loop
}


// Equivalent while structure
```

# Loop Practice



- **Write a for loop to compute the first 10 terms of the Liebniz approximation of π/4:**
  - **π/4 = 1/1 – 1/3 + 1/5 – 1/7 + 1/9 …**
  - **Tip: write a table of the loop counter variable vs. desired value and then derive the general formula**
  - **Exercise: liebnizapprox**

| Counter (i) | Desired | Pattern | Counter (i) | Desired | Pattern |
|---|---|---|---|---|---|
| 0 | +1/1 | for(i=0;i<10;i++)<br>Fraction:<br><br><br>+/- => | 1 | +1/1 | for(i=1; i<=19; i+=2)<br>Fraction:<br><br><br>+/- => |
| 1 | -1/3 | | 3 | -1/3 | |
| 2 | +1/5 | | 5 | +1/5 | |
| … | … | | … | … | |
| 9 | -1/19 | | 19 | -1/19 | |

# Loop Practice (cont.)



- **Write a for loop to compute the first 10 terms of the Liebniz approximation of π/4:**
  - **π/4 = 1/1 – 1/3 + 1/5 – 1/7 + 1/9 …**
  - **Tip: write a table of the loop counter variable vs. desired value & then derive the general formula**
  - **Exercise: liebnizapprox**

| Counter (i) | Desired | Pattern | Counter (i) | Desired | Pattern |
|---|---|---|---|---|---|
| 0 | +1/1 | for(i=0; i <10; i++)<br>Fraction:<br>1/(2*i+1)<br><br>+/- =><br>pow(-1,i)<br>if(i is odd)<br>neg. | 1 | +1/1 | for(i=1; i <=19; i+=2)<br>Fraction:<br>1/i<br><br>+/- =><br>if(i%4==3)<br>neg. |
| 1 | -1/3 | | 3 | -1/3 | |
| 2 | +1/5 | | 5 | +1/5 | |
| … | … | | … | … | |
| 9 | -1/19 | | 19 | -1/19 | |

# Loop Practice

- Write for loops to compute the first 10 terms of the following approximations:
  - $e^x$: $1 + x + x^2/2! + x^3/3! + x^4/4!$ ...
    - Assume 1 is the 1st term and assume functions
      - fact(int n) // returns n!
      - pow(double x, double n) // returns $x^n$
  - Wallis:
    - $\pi/2 = 2/1 * 2/3 * 4/3 * 4/5 * 6/5 * 6/7 * 8/7$ ...
      - Hint: You can generate multiple terms per loop iteration...
    - Exercises: wallisapprox

# The Loops That Keep On Giving

- There's a problem with the loop below
- We all write "infinite" loops at one time or another
- Infinite loops never quit
- When you do write such a program, just type "Ctrl-C" at the terminal to halt the program

```
#include <iostream>
using namespace std;
int main()
{ int val;
  bool again = true;
  while(again = true){
    cout << "Enter an int or -1 to quit";
    cin >> val;
    if( val == -1 ) {
       again = false;
    }
  }
  return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
  int i=0;
  while( i < 10 ) {
     cout << i << endl;
     i + 1;
  }
  return 0;
}
```

http://blog.codinghorror.com/rubber-duck-problem-solving/

# break and continue

- **break**
  - Ends the current loop [not if statement] immediately and continues execution after its last statement
- **continue**
  - Begins the next iteration of the nearest loop (performing the update statements if it is a for loop)
  - Can usually be accomplished with some kind of if..else structure
  - Can be useful when many nested if statements...

```
bool done = 0;
while ( !done ) {
  cout << "Enter guess: " << endl;
  cin >> guess;
  if( guess < 0 )
    break;
  }
  // ... Process guess

}

// Guess an int >= 0
while( !done ) {
  cin >> guess;
  if(guess < 0){
    continue;
  }
  // Can only be here if guess >= 0
}
// Equivalent w/o using continue
while( !done ) {
  cin >> guess;
  if(guess >= 0){
    // Process
  }
}
```

# break and continue (cont.)

- break and continue apply only to the inner most loop (not all loops being nested)
  - Break ends the current (inner-most) loop immediately
  - Continue starts next iteration of inner-most loop immediately
- Consider problem of checking if a '!' exists anywhere in some lines of text
  - Use a while loop to iterate through each line
  - Use a for loop to iterate through each character on a particular line
  - Once we find first '!' we can stop

```
bool flag = false;
while( more_lines == true ){
  // get line of text from user
  length = get_line_length(...);

  for(j=0; j < length; j++){
   if(text[j] == '!'){
     flag = true;
     break; // only quits the for loop
   }
  }
}

bool flag = false;
while( more_lines == true && ! flag ){
  // get line of text from user
  length = get_line_length(...);

  for(j=0; j < length; j++){
   if(text[j] == '!'){
     flag = true;
     break; // only quits the for loop
   }
  }
}
```

# Activity

- Fix the compiler errors in a sample program
- Copy the sample program into your account

  lec1_visual_errors.cpp
- Edit the file

  $ gedit  lec1_visual_errors.cpp &
- Compile the file

  $ g++ -g -o lec1_visual_errors lec1_visual_errors.cpp
- Attempt to fix the errors and iterate

# Single Statement Bodies

- An if, while, or for construct with a single statement body does not require { ... }
- Another if, while, or for counts as a single statement

```
if (x == 5)
  y += 2;
else
  y -= 3;


for(i = 0; i < 5; i++)
  sum += i;


while(sum > 0)
  sum = sum/2;


for(i = 1 ; i <= 5; i++)
  if(i % 2 == 0)
     j++;
```

# More Exercises

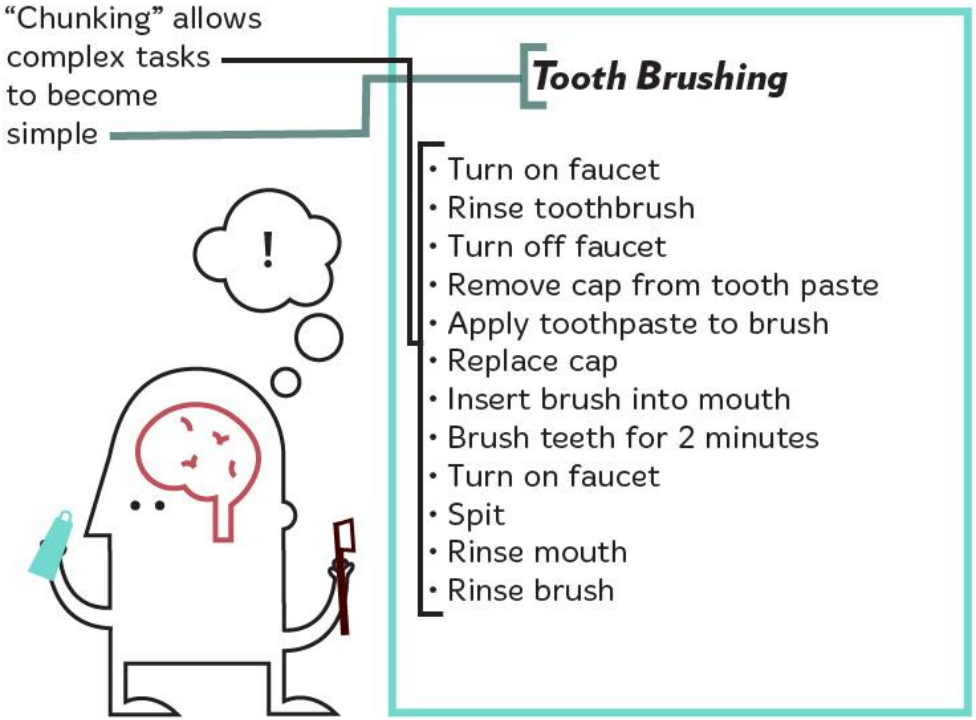
- Determine if a user-supplied positive integer > 1 is prime or not
  - How do we determine if a number is a factor of another?
  - What numbers could be factors?
  - How soon can we determine a number is not-prime?
- Reverse the digits of an integer*
  - User enters 123 => Output 321
  - User enters -5293 => -3925
  - Exercise: Revdigits



*Taken from D.S. Malik, C++ Programming, 6th Ed.

# 20-second Timeout: Chunking

- How to remember all the little details (all the parts of a for loop, where do you need semicolons, etc.)
- As you practice these concepts they will start to "chunk" together where you can just hear "for loop" and will immediately know the syntax and meaning
- Chunking occurs where something more abstract takes the place of many smaller pieces




https://designbyben.wordpress.com/tag/chunking/

# NESTED LOOPS

# Nested Loops

- Inner loops execute fully (go through every iteration before the next iteration of the outer loop starts)

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  for(int i=0; i < 2; i++){

    for(int j=0; j < 3; j++){
      // Do something based
      // on i and j
      cout << i << " " << j;
      cout << endl;

    }
  }
  return 0;
}
```

**Output:**

```
0 0
0 1
0 2
1 0
1 1
1 2
```

# Nested Loops (cont.)

- Write a program using nested loops to print a multiplication table of 1..12
- Tip: Decide what abstract "thing" your iterating through and "read" the for loop as "for each "thing" ...
  - For each "row" ...
    - For each column...
      print the product

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 1 | 2 | 3 |
| 2 | 2 | 4 | 6 |
| 3 | 3 | 6 | 9 |

```
#include <iostream>

using namespace std;

int main()
{
  for(int r=1; r <= 12; r++){
    for(int c=1; c <= 12; c++){
       cout << r*c;
    }
  }
  return 0;
}
```

**This code will print some not so nice output:**

**1234567891011122468101214161 8202224…**

# Nested Loops (cont.)

- Tip: Decide what abstract "thing" your iterating through and "read" the for loop as "for each "thing" ...
  - For each "row" ...
    - For each column...
      print the product followed by a space
    - Print a newline

```
#include <iostream>

using namespace std;

int main()
{
  for(int r=1; r <= 12; r++){
    for(int c=1; c <= 12; c++){
       cout << " " << r*c;
    }
    cout << endl;
  }
  return 0;
}
```

This code will still print some not so nice output:

1 2 3 4 5 6 7 8 9 10 11 12
2 4 6 8 10 12 14 16 18 20 22 24

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 1 | 2 | 3 |
| 2 | 2 | 4 | 6 |
| 3 | 3 | 6 | 9 |

# Nested Loops (cont.)

- Tip: Decide what abstract "thing" your iterating through and "read" the for loop as "for each "thing" ...
  - For each "row" ...
    - For each column... print the product

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  for(int r=1; r <= 12; r++){
    for(int c=1; c <= 12; c++){
       cout << setw(4) << r*c;
    }
    cout << endl;
  }
  return 0;
}
```

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 1 | 2 | 3 |
| 2 | 2 | 4 | 6 |
| 3 | 3 | 6 | 9 |

# Nested Loop Practice

- 5PerLine series (some mistakes in each one)
  - Exercises:
    - 5perlineA
    - 5perlineB
    - 5perlineC
  - Each exercise wants you to print out the integers from 100 to 200, five per line, as in:

```
100 101 102 103 104
105 106 107 108 109
 ...
195 196 197 198 199
200
```

# Common Loop Tasks

- Aggregation / Reduction
  - Sum or combine information from many pieces to a single value
  - E.g. Sum first 10 positive integers
  - Declare aggregation variable and initialize it outside the loop and update it in each iteration
- Search for occurrence
  - Find a particular occurrence of some value or determine it does not exist
  - Declare a variable to save the desired occurrence or status, then on each iteration check for what you are looking for, and set the variable if you find it and break the loop

```
// aggregation example
int sum = 0;
for(int i=1; i <= 10; i++){
  sum += i;
}


// search for first int divisible
// by 2 and 3
int div2_3 = 0;
for(int i=1; i < 100; i++){
  if( i%2 == 0 && i%3 == 0){
    div2_3 = i;
    break;
  }
}
if( div2_3 != 0 ){
  // we have found such an int
}
```

# C LIBRARIES & RAND()

# Preprocessor & Directives

- Somewhat unique to C/C++
- Compiler will scan through C code looking for directives (e.g. #include, #define, anything else that starts with '#' )
- Performs textual changes, substitutions, insertions, etc.
- #include <filename> or #include "filename"
  - Inserts the entire contents of "filename" into the given C text file
- #define *find_pattern replace_pattern*
  - Replaces any occurrence of *find_pattern* with *replace_pattern*
  - #define PI 3.14159

Now in your code:

```
x = PI;
```

is replaced by the preprocessor with

```
x = 3.14159;
```

# #include Directive

- Common usage: To include “header files” that allow us to access functions defined in a separate file or library
- For pure C compilers, we include a C header file with its filename: #include <stdlib.h>
- For C++ compilers, we include a C header file without the .h extension and prepend a ‘c’: #include <cstdlib>

| C | Description | C++ | Description |
|---|---|---|---|
| **stdio.h cstdio** | Old C Input/Output/File access | **iostream** | I/O and File streams |
| **stdlib.h cstdlib** | rand(), Memory allocation, etc. | **fstream** | File I/O |
| **string.h cstring** | C-string library functions that operate on character arrays | **string** | C++ string class that defines the ‘string’ object |
| **math.h cmath** | Math functions: sin(), pow(), etc. | **vector** | Array-like container class |

# rand() and RAND_MAX

- (Pseudo)random number generation in C is accomplished with the rand() function declared/prototyped in cstdlib
- rand() returns an integer between 0 and RAND_MAX
  - RAND_MAX is an integer constant defined in <cstdlib>
- How could you generate a flip of a coin [i.e. 0 or 1 w/ equal prob.]?

```
int r;
r = rand();
 if(r < RAND_MAX/2){ cout << "Heads"; }
```

- How could you generate a decimal with uniform probability of being between [0,1]

```
double r;
r = staic_cast<double>(rand()) / RAND_MAX;
```

# Seeding Random # Generator

- Re-running a program that calls rand() will generate the same sequence of random numbers (i.e. each run will be exactly the same)
- If we want each execution of the program to be different then we need to seed the RNG with a different value
- srand(int seed) is a function in <cstdlib> to seed the RNG with the value of seed
  - Unless seed changes from execution to execution, we'll still have the same problem
- Solution: Seed it with the day and time [returned by the time() function defined in ctime]

```
srand( time(0) );  // only do this once at the start of the program


int r = rand();    // now call rand() as many times as you want
int r2 = rand();   // another random number
// sequence of random #'s will be different for each execution of
program
```

Only call srand() ONCE at the start of the program, not each time you want to call rand()!!!

# PRACTICE

# Exercise

- Guessing game
  - Number guessing game [0-19]...indicate higher or lower until they guess correctly or stop after 5 unsuccessful guesses

```
#include <iostream>
#include <cstdlib>
#include <ctime>

int main()
{
  srand(time(0));
  int secretNum = rand() % 20;
  // Now create a game that
  //  lets the user try to guess
  //  the random number in up to
  //  5 guesses

}
```

Ch. 5, PE 16 of D.S. Malik, C++ Programming, 6th Ed.

# FUNCTIONS

# Program Decomposition

- C is a procedural language
  - Main unit of code organization, problem decomposition, and abstraction is the “function” or “procedure”
  - Function or procedure is a unit of code that
    - Can be called from other locations in the program
    - Can be passed variable inputs (a.k.a. arguments or parameters)
    - Can return a value to the code that called it
- C++ is considered an “object-oriented” language (really just adds objected-oriented constructs to C)
  - Main unit of organization, problem decomposition, and abstraction is an object (collection of functions & associated data)

# Think functions

- A mathematical function takes in inputs (a.k.a. parameters/arguments) and produce (a.k.a. return) *a* value
  - Some functions are well known:
    - sin(x), add(a, b)
  - Others can be user defined
    - Let $f(x, y) = x^2 - y^2$
  - And can even be factored into or composed from smaller functions
    - Let g(x,y) = x+y and h(x,y) = x-y
    - Then f(x,y) = mul(g(x,y), h(x,y))

# Use It or Lose It

- A function returns a value, but the computer then must either
  - use it immediately as part of an expression
    - x + sin(x) / pow(x,2)
  - save it in a variable (otherwise it will just throw it away)
    - y = sin(x)

# Use Functions

- If you are writing similar code try to generalize it
  - Dumb example: rather than solve specific polynomials like $5x^2 + 3x - 10$ write a function where we can solve $ax^2 + bx + c$ and pass in a, b, and c
- If you find yourself repeating code (cut/paste) pull it out to a separate function

# Function call statements

- C++ predefines a variety of functions for you. Here are a few of them:
  - sqrt(x): returns the square root of x (in <cmath>)
  - pow(x, y): returns xy, or x to the power y (in <cmath>)
  - sin(x): returns the sine of x if x is in radians (in <cmath>)
  - abs(x): returns the absolute value of x (in <cstdlib>)
  - max(x, y): returns the maximum of x and y (in <algorithm>)
  - min(x, y): returns the maximum of x and y (in <algorithm>)
- You call these by writing them similarly to how you would use a function in mathematics:

```
#include <iostream>
#include <cmath>
#include <algorithm>
using namespace std;

int main(int argc, char *argv[])
{
  // can call functions
  //  in an assignment
  double res = cos(0);

  // can call functions in an
  //  expression
  res =  sqrt(2) + 2.3 << endl;

  // can call them as part of a
  //  output statement
  cout << max(34, 56) << endl;

  return 0;
}
```

# Exercise

- To decompose a program into functions, try listing the verbs or tasks that are performed to solve the problem
- Model a game of poker or doing laundry as a series of tasks/procedures…
  - Poker
    - Shuffle, deal, bet, flop, bet, turn, bet, river, bet, show
  - Laundry
    - Put_clothes_in_washer, add_soap, pay_money, start_wash, transfer_to_dryer, pay_money, dry, fold

# Functions aka Methods aka Procedures

- Also called *procedures* or *methods*
- Collection of code that performs a task
  - Has a name to identify it (e.g. 'avg')
  - Takes in 0 or more inputs (a.k.a. parameters or arguments)
  - Performs computation
    - Start and end of code belonging to the function are indicated by curly braces { ... }
    - Sequence of statements
  - Returns at most a single value (i.e., 0 or 1 value)
  - Return value is substituted for the function call

# Anatomy of a function

- Return type (any valid C type)
  - void, int, double, char, etc.
  - void means return nothing
- Function name
  - Any valid identifier
- Input arguments inside ()
  - Act like a locally declared variable
- Code
  - In {...}
- Non-void functions must have 1 or more return statements
  - First 'return' executed immediately quits function

```
void printMenu()
{
  cout << "Welcome to ABC 2.0:" << endl;
  cout << "====================" << endl;
  cout << "  Enter an option:" << endl;
  cout << "    1.) Start" << endl;
  cout << "    2.) Continue" << endl;
  cout << "    3.) End\n" << endl;
}

bool only_2_3_factors(int num)
{
   while(num % 2 == 0){
     ...
   }
   ...
   if(num==1)
     return 1;

   return 0;

}

double triangle_area(double b, double h)
{
   double area;
   area = 0.5 * b * h;
   return area;
}
```

# Execution of a Function

- Statements in a function are executed sequentially by default
- Defined once, called over and over
- Functions can 'call' other functions
  - Goes and executes that collection of code then returns to continue the current function
- Compute max of two integers

  Each 'call' causes the program to pause the current function, go to the called function and execute its code with the given arguments then return to where the calling function left off,
- Return value is substituted in place of the function call

```
#include <iostream>
using namespace std;

int max(int a, int b)
{
  if(a > b)
    return a;
  else
    return b;
}

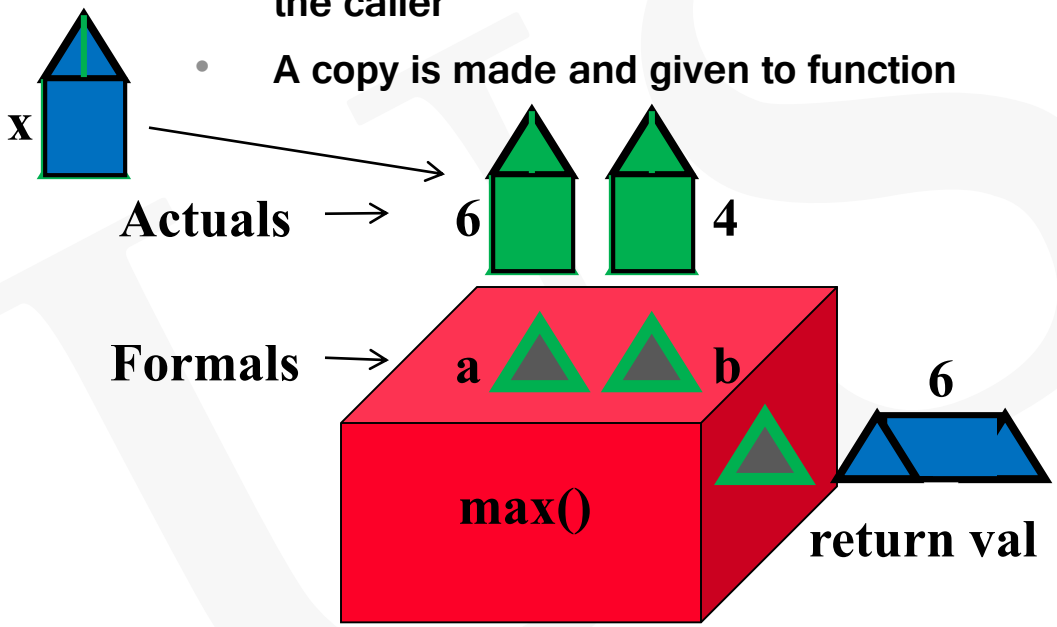

int main(int argc, char *argv[])
{
   int x=6, z;
   z = max(x,4);
   cout << "Max is " << z << endl;
   z = max(125, 199);
   cout << "Max is " << z << endl;
   return 0;
}
```

6

199

# Execution of a Function (cont.)

- Formal parameters, a and b
  - Type of data they expect
  - Names that will be used internal to the function to refer to the values
- Actual parameters
  - Actual values input to the function code by the caller
  - A copy is made and given to function




Each type is a "different" shape (int = triangle, double = square, char = circle). Only a value of that type can "fit" as a parameter..

```
#include <iostream>
using namespace std;

int max(int a, int b)
{
  if(a > b)
    return a;
  else
    return b;
}

int main(int argc, char *argv[])
{
   int x=6, z;
   z = max(x, 4);
   cout << "Max is " << z << endl;
   z = max(125, 199);
   cout << "Max is " << z << endl;
   return 0;
}
```

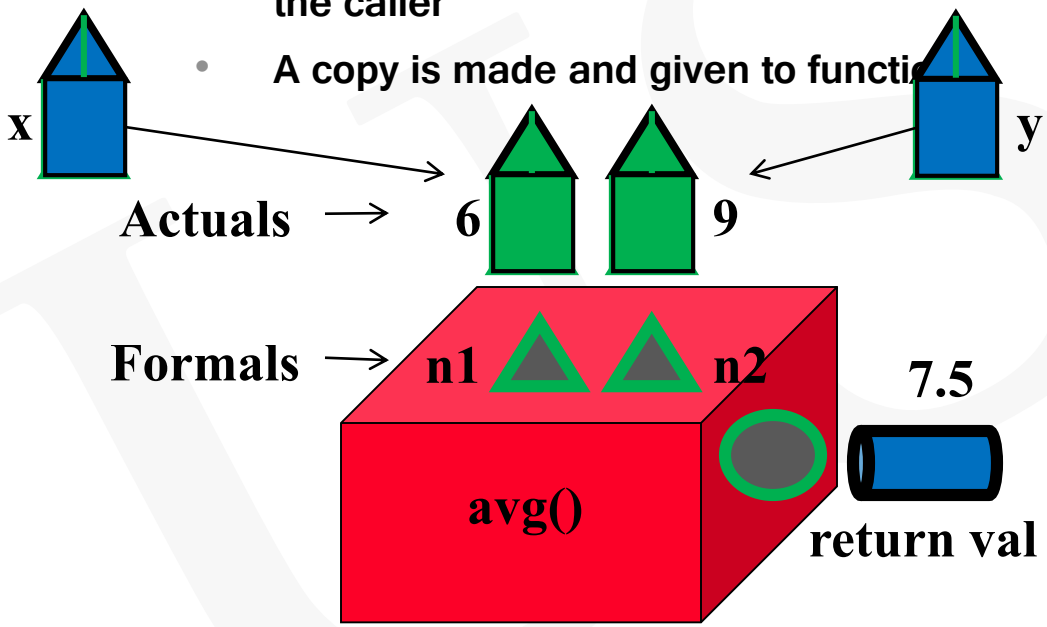
Formals

Actuals

Actuals

# Execution of a Function (cont.)

- Formal parameters, n1 and n2
  - Type of data they expect
  - Names that will be used internal to the function to refer to the values
- Actual parameters
  - Actual values input to the function code by the caller
  - A copy is made and given to functi




Each type is a "different" shape (int = triangle, double = square, char = circle). Only a value of that type can "fit" as a parameter..

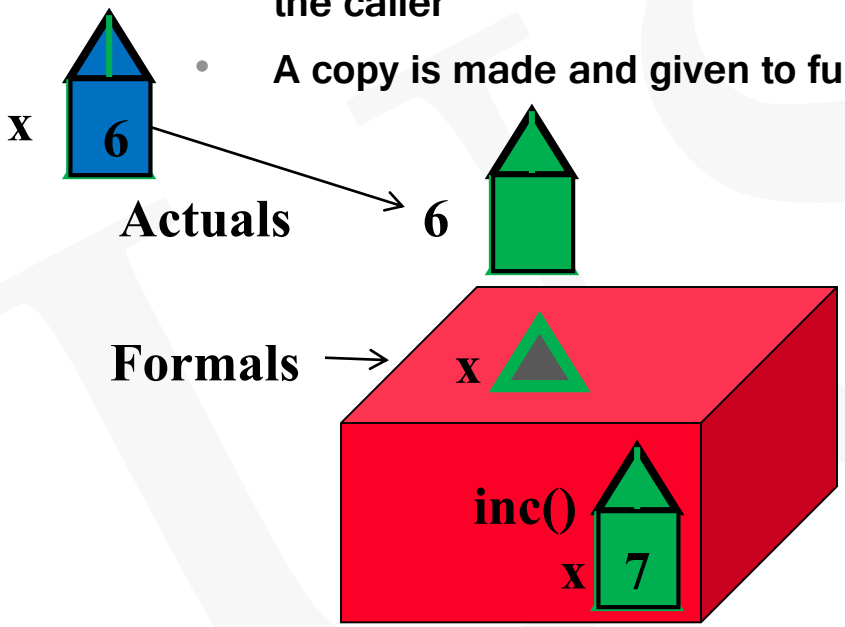
```
#include <iostream>
using namespace std;

double avg(int n1, int n2)
{
  double sum = n1 + n2;
  return sum/2.0;
}

int main(int argc, char *argv[])
{
   int x=6, y = 9;  double z;
   z = avg(x,y);
   cout << "AVG is " << z << endl;
   z = avg(x, 2);
   cout << "AVG is " << z << endl;
   return 0;
}
```

# Execution of a Function (cont.)

- Formal parameters, n1 and n2
  - Type of data they expect
  - Names that will be used internal to the function to refer to the values
- Actual parameters
  - Actual values input to the function code by the caller
  - A copy is made and given to function




Each type is a "different" shape (int = triangle, double = square, char = circle). Only a value of that type can "fit" as a parameter..

```
#include <iostream>
using namespace std;

void inc(int x)
{
  x = x+1;
}

int main(int argc, char *argv[])
{
   int x=6;
   inc(x);
   cout << "X is " << x << endl;
   return 0;
}
```

# Execution of a Function (cont.)

- Statements in a function are executed sequentially by default
- Defined once in the code, called/executed over and over at run-time
- Functions can 'call' other functions
  - Goes and executes that collection of code then returns to continue the current function
- Compute factorial
  Defined as n! = n*(n-1)*(n-2)*...*2*1

  Function declaration/prototype

  ```
  int fact(int);
  ```

  Each 'call' causes the computer to go to that function and execute the code there then return to where the calling function left off
- Functions can take any number of arguments/parameters
- Functions can return a single value
  - The return value "replaces" the call in an expression

```
#include <stdio.h>
int fact(int);

int main(int argc, char *argv[])
{
   int x=6,z;
   z = fact(4);
   cout << "Z = " << z << endl;
   z = fact(x);
   cout << "Z = " << z << endl;
   return 0;
}

int fact(int n)
{
  int answer;
  // use n to compute n!
  // putting the result in 'answer'
  return answer;
}
```

# Formals vs. Actuals

- A function defines the input arguments/parameters it needs
  - Data values that can be different each time the function is executed
- The function defines names by which it will refer to these input values (known as 'Formals')
  - Formals act as local variables inside the function
- The code that calls the function will provide the actual data values it wants the function to operate on (known as 'Actuals')
- Call 1 => b=9.5,h=4.0
- Call 2 => b=10.0,h=8.0
- Call 3 => b=5.0, h=3.5

```
double triangle_area(double b, double h)
{
  return 0.5 * b * h;
}


int main()
{
  double x,y,m,n;
  double area1,area2,area3;
  m = 9.5; n = 4.0;
  x = 12.0; y = 7.0;
  area1 = triangle_area(m,n);
  area2 = triangle_area(x-2,y+1);
  area3 = triangle_area(5.0,3.5);
  return 0;
}
```

# Function Prototypes

- The compiler ('g++') needs to "know" about a function before it can handle a call to that function
- The compiler will scan a file from top to bottom
- If it encounters a call to a function before the actual function code it will complain...[Compile error]
- ...Unless a prototype ("declaration") for the function is defined earlier
- A prototype only needs to include data types for the parameters and not their names (ends with a ';')
  - Prototype is used to check that you are calling it with the correct syntax (i.e., parameter data types & return type)
    (like a menu @ a restaurant)

X

```
int main()
{
  double area1,area2,area3;
  area3 = triangle_area(5.0,3.5);
}


double triangle_area(double b, double h)
{
   return 0.5 * b * h;
}
```

Compiler encounters a call to triangle_area() before it has seen its definition (Error!)

```
double triangle_area(double, double);

int main()
{
  double area1,area2,area3;
  area3 = triangle_area(5.0,3.5);
}


double triangle_area(double b, double h)
{
   return 0.5 * b * h;
}
```

Compiler sees a prototype and can check the syntax of any following call and expects the definition later.

# Overloading: A Function's Signature

- What makes up a signature (uniqueness) of a function
  - name
  - number and type of arguments
- No two functions are allowed to have the same signature; the following 6 functions are unique and allowable...
  - int f1(int), int f1(double), int f1(int, double)
  - int f1(int, char), double f1(), void f1(char)
- Return type does not make a function unique
  - int f1() and double f1() are not unique and thus not allowable
- Two functions with the same name are said to be "overloaded"
  - int max(int, int); double max(double, double);

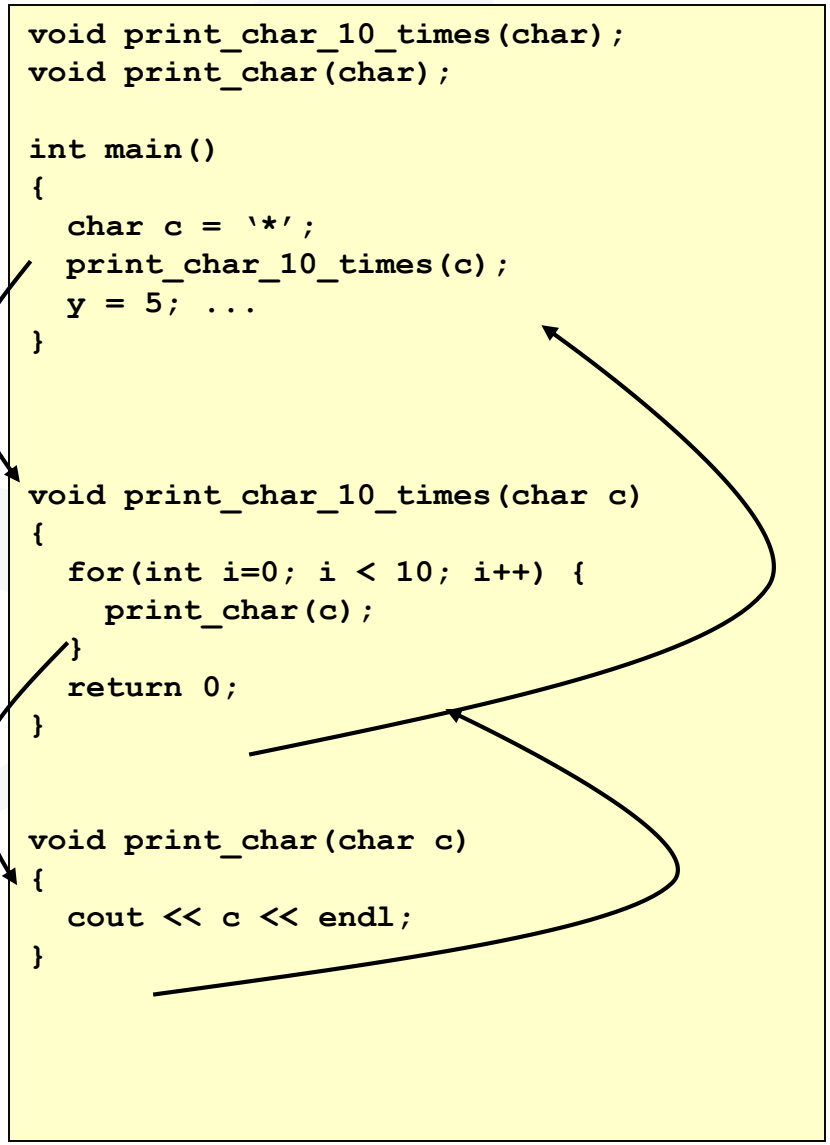
# Exercises

- Remove Factors
  - remove_factor
- Draw an ASCII square on the screen
  - draw_square
- Practice overloading a function
  - overload

# FUNCTION CALL SEQUENCING

# Function Call Sequencing

- Functions can call other functions and so on...
- When a function is called the calling function is suspended (frozen) along with all its data and control jumps to the start of the called function
- When the called function returns execution resumes in the calling function
- Each function has its own set of variables and "scope"
  - Scope refers to the visibility/accessibility of a variable from the current place of execution



```
void print_char_10_times(char);
void print_char(char);

int main()
{
  char c = '*';
  print_char_10_times(c);
  y = 5; ...
}

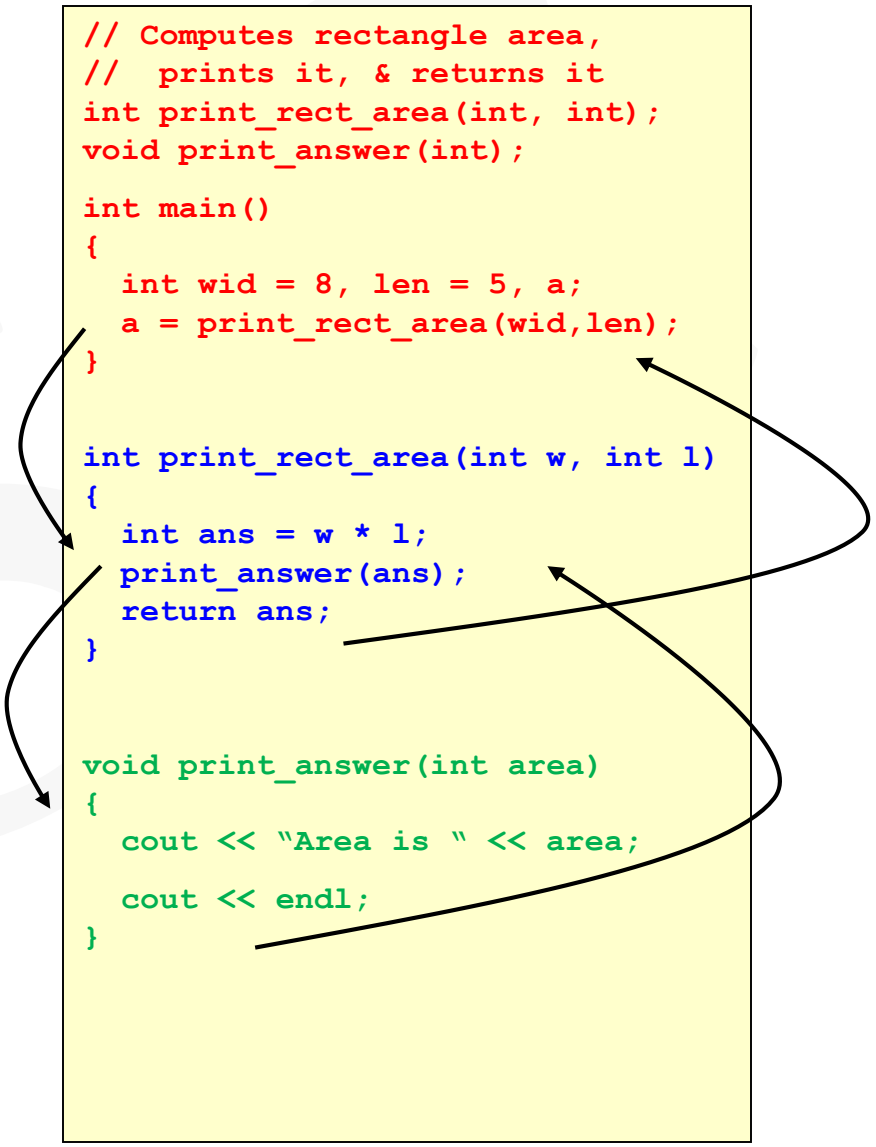


void print_char_10_times(char c)
{
  for(int i=0; i < 10; i++) {
    print_char(c);
  }
  return 0;
}


void print_char(char c)
{
  cout << c << endl;
}
```

# More Function Call Sequencing

- As one function calls another, they execute in a last-in, first-out fashion (i.e. the last one called is the first one to finish & return)
  - Just like in the cafeteria the last plate put on the top of the stack is the first one to be pulled off (always access the top item)
- How does the computer actually track where to return to when a function completes



```
// Computes rectangle area,
//  prints it, & returns it
int print_rect_area(int, int);
void print_answer(int);

int main()
{
  int wid = 8, len = 5, a;
  a = print_rect_area(wid,len);
}

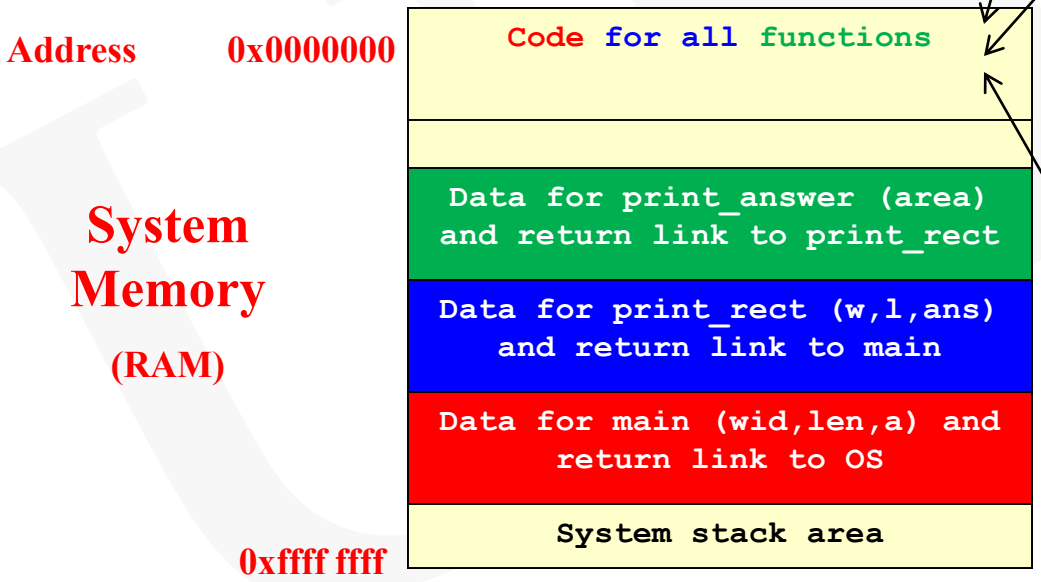

int print_rect_area(int w, int l)
{
  int ans = w * l;
  print_answer(ans);
  return ans;
}


void print_answer(int area)
{
  cout << "Area is " << area;

  cout << endl;
}
```

# More Function Call Sequencing (cont.)

- Computer maintains a "stack" of function data and info in memory (i.e. RAM)
  - Each time a function is called, the computer allocates memory for that function on the top of the stack and a link for where to return
  - When a function returns that memory is de-allocated and control is returned to the function now on top




```
// Computes rectangle area,
//  prints it, & returns it
int print_rect_area(int, int);
void print_answer(int);

int main()
{
  int wid = 8, len = 5, a;
  a = print_rect_area(wid,len);
}


int print_rect_area(int w, int l)
{
  int ans = w * l;
  print_answer(ans);
  return ans;
}


void print_answer(int area)
{
  cout << "Area is " << area;

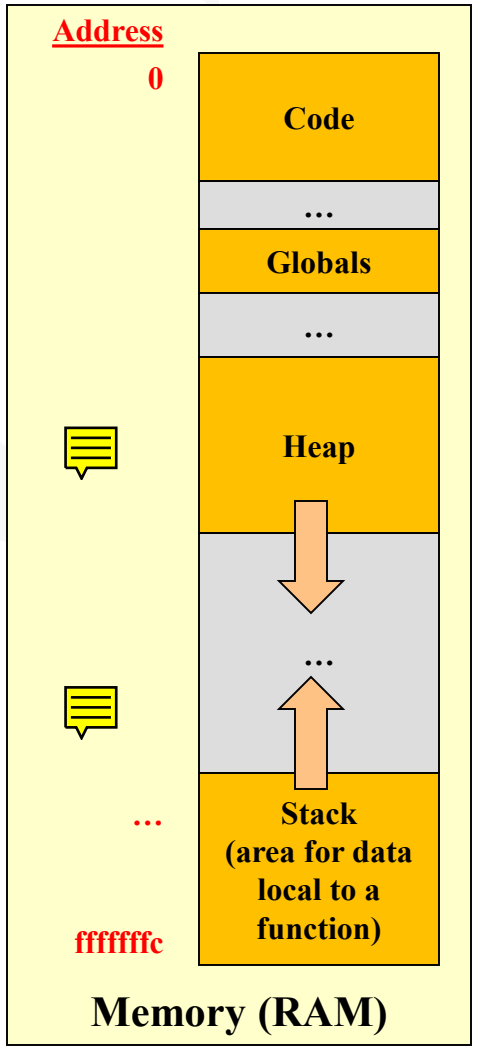
  cout << endl;
}
```

# Nested Call Practice

- Find characters in a string then use that function to find how many vowels are in a string
  - Exercises: vowels

# Recall: Memory Organization

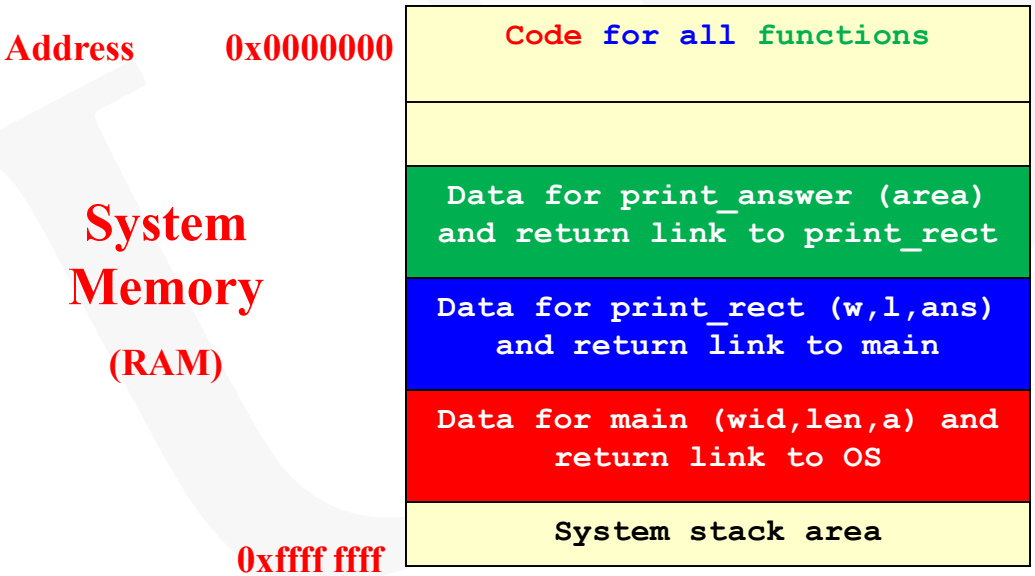
- 32-bit address range (0x0 – 0xffffffff)
- Code usually sits at low addresses
- Global variables/data somewhere after code
- Heap: Area of memory that can be allocated and de-allocated during program execution (i.e., dynamically at run-time) based on the needs of the program
- System stack (memory for each function instance that is alive)
  - Local variables
  - Return link (where to return)
  - etc.
- Heap grows downward, stack grows upward…
  - In rare cases of large memory usage, they could collide and cause your program to fail or generate an exception/error



# LOCAL VARIABLES & SCOPE

# Local Variables

- Any variable declared inside a function is called a "local" variable
- It lives in the stack area for that function
- It dies when the function returns




```
// Computes rectangle area,
//  prints it, & returns it
int print_rect_area(int, int);
void print_answer(int);

int main()
{
  int wid = 8, len = 5, a;
  a = print_rect_area(wid,len);
}


int print_rect_area(int w, int l)
{
  int ans = w * l;
  print_answer(ans);
  return ans;
}


void print_answer(int area)
{
  cout << "Area is " << area;

  cout << endl;
}
```

# Scope

- Global variables live as long as the program is running
- Variables declared in a block { ... } are 'local' to that block
  - { ... } of a function
  - { ... } of a loop, if statement, etc.
  - Die/deallocated when the program reaches the end of the block...don't try to access them intentionally or unintentionally after they are 'out of scope'/deallocated
  - Actual parameters act as local variables and die when the function ends
- When variables share the same name the closest declaration will be used by default

# Scope Example

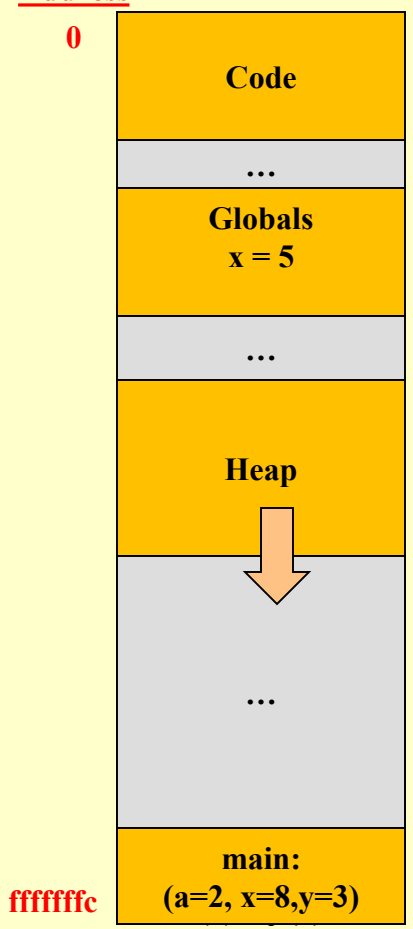
- Globals live as long as the program is running
- Variables declared in a block { ... } live as long as the block has not completed
  - { ... } of a function
  - { ... } of a loop, if statement, etc.
- When variables share the same name the closest declaration will be used by default

```
#include <iostream>
using namespace std;

int x = 5;

int main()
{
  int a, x = 8, y = 3;
  cout << "x = " << x << endl;
  for(int i=0; i < 10; i++){
    int j = 1;
    j = 2*i + 1;
    a += j;
  }
  a = doit(y);
  cout << "a=" << a ;
  cout << "y=" << y << endl;
  cout << "glob. x" << ::x << endl;
}


int doit(int x)
{
   x--;
   return x;
}
```



# PASS BY VALUE

# Pass-by-Value

- Passing an argument to a function makes a copy of the argument
- It is like e-mailing an attached document
  - You still have the original on your PC
  - The recipient has a copy which he can modify but it will not be reflected in your version
- Communication is essentially one-way
  - Caller communicates arguments to callee, but callee cannot communicate back because he is working on copies...
  - The only communication back to the caller is via a return value

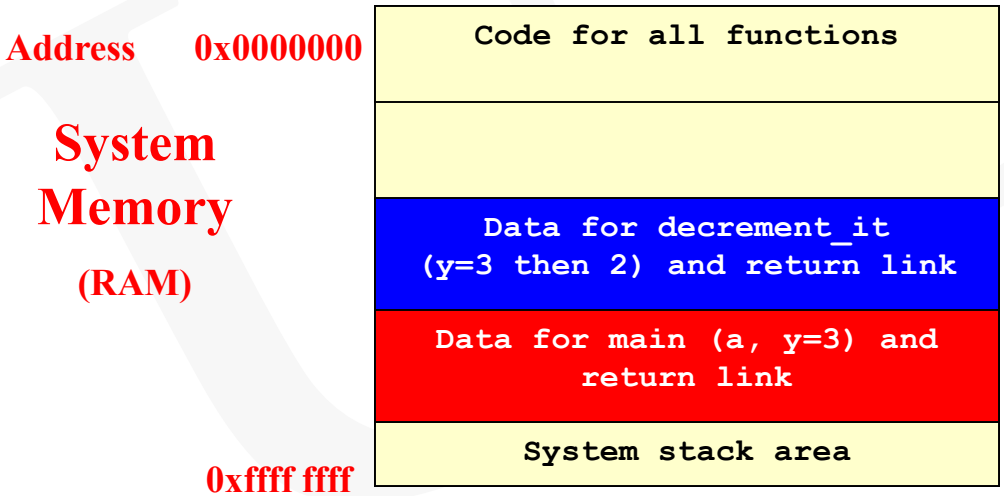
# Pass by Value (cont.)

- Notice that actual arguments are different memory locations/variables than the formal arguments
- When arguments are passed a copy of the actual argument value (e.g. 3) is placed in the formal parameter (x)
- The value of y cannot be changed by any other function (remember it is local)

```
void decrement_it(int);

int main()
{
  int a, y = 3;
  decrement_it(y);
  cout << "y = " << y << endl;
  return 0;
}

void decrement_it(int y)
{
   y--;
}
```



# Practice

- reset_bad
- Guessing Game
  - Return to the number guessing game from our previous lecture
  - Factor (re-organize) your code to use the following function and also write its implementation:

bool checkGuess(int guess, int secretNum);

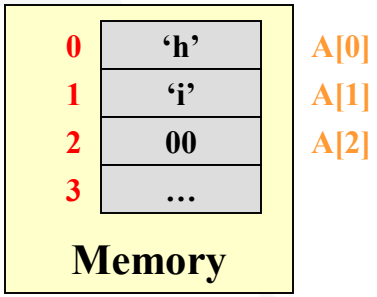
  - It should take the current guess and the secret number and return 'true' if they match, 'false' otherwise, printing status to the user as well.

# ARRAYS

# Arrays

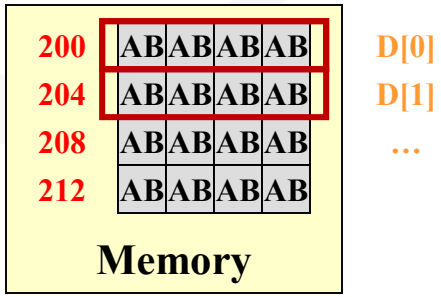
- Informal Def: Collection of variables of the same typed accessed by index/position
- Formal Def: A statically-sized, contiguously allocated collection of homogenous data elements
- Collection of homogenous data elements
  - Multiple variables of the same data type
- Contiguously allocated in memory
  - One right after the next
- Statically-sized
  - Size of the collection must be a constant and can't be changed after initial declaration/allocation
- Collection is referred to with one name
- Individual elements referred to by an offset/index from the start of the array [in C, first element is at index 0]

char A[3] = "hi";

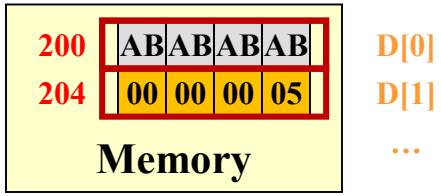



char c = A[0]; // 'h'

int D[20];




D[1] = 5;

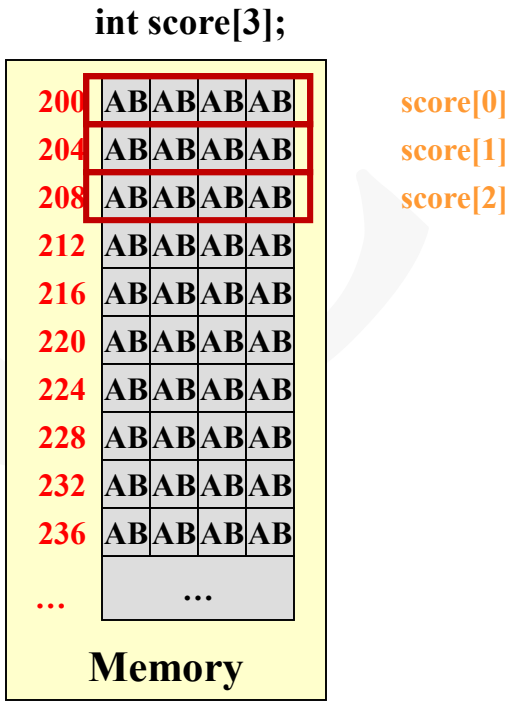


# Arrays As Collections

- If I have several players's scores to track in a game I could declare a separate variable to track each one's score:
  - `int player1 = N; int player2 = N; int player3 = N; …`
  - SAD!!
- Better idea: Use an array where the index to the desired element can be a variable:
  - ```
    for(i=0; i < N; i++){
      player[i] = N;
    ```
- Can still refer to individual items if desired: player[2]
    ```

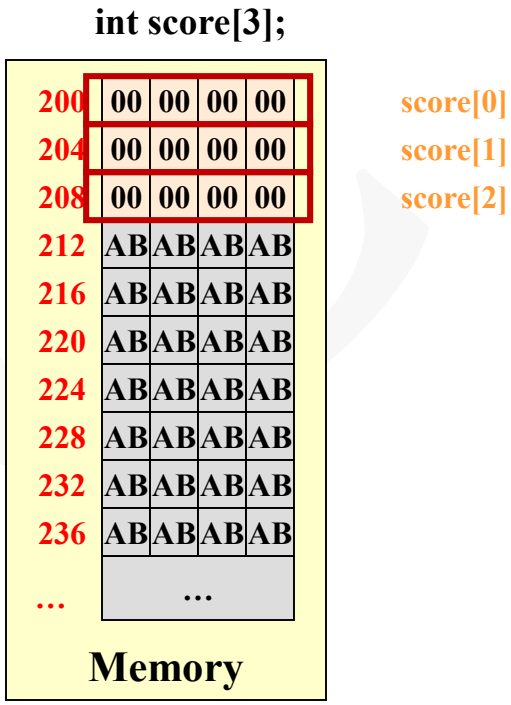
# Arrays

- Track the score of 3 people
- Homogenous data set (i.e., integer scores) for multiple people...perfect for an array
  - int score[3];



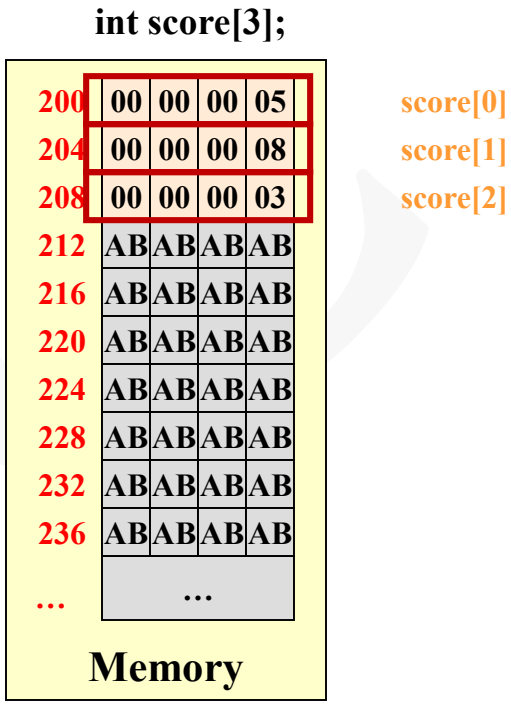
# Arrays (cont.)

- Track the score of 3 people
- Homogenous data set (i.e. integer scores) for multiple people...perfect for an array
  - int score[3];
- Must initialize elements of an array
  - for(int i=0; i < 3; i++)
    score[i] = 0;



# Arrays (cont.)

- Track the score of 3 people
- Homogenous data set (i.e. integer scores) for multiple people...perfect for an array
  - int score[3];
- Must initialize elements of an array
  - for(int i=0; i < 3; i++)
    score[i] = 0;
- Can access each persons amount and perform ops on that value
  - score[0] = 5;
    score[1] = 8;
    score[2] = score[1] - score[0]



# ARRAY ODDS AND ENDS

# Static Size/Allocation

- For now, arrays must be declared as fixed size (i.e., a constant known at compile time)
  - Good:
    - `int x[10];`
    - `#define MAX_ELEMENTS  100`
      `int x[MAX_ELEMENTS];`
    - `const int MAX_ELEMENTS = 100;`
      `int x[MAX_ELEMENTS];`
  - Bad:
    - `int mysize;`
      `cin >> mysize;`
      `int x[mysize];`
    - `int mysize = 10;`
      `int x[mysize];`


**Compiler must be able to figure out how much memory to allocate at compile-time**

# Initializing Arrays

- Integers or floating point types can be initialized by placing a comma separated list of values in curly braces {...}
  - int data[5] = {4,3,9,6,14};
  - char vals[8] = {64,33,18,4,91,76,55,21};
- If accompanied w/ initialization list, size doesn't have to be indicated (empty [ ])
  - double stuff[] = {3.5, 14.22, 9.57}; // = stuff[3]
- However the list must be of constants, not variables:
  - BAD: double z = 3.5; double stuff[] = {z, z, z};

Understanding array addressing and indexing

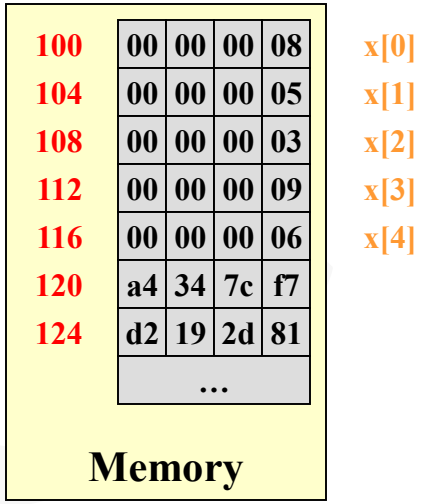
# ACCESSING DATA IN AN ARRAY

# Exercise

- Consider a train of box cars
  - The initial car starts at point A on the number line
  - Each car is 5 meters long
- Write an expression of where the i-th car is located (at what meter does it start?)
- Suppose a set of integers start at memory address A, write an expression for where the i-th integer starts?
- Suppose a set of doubles start at memory address A, write an expression for where the i-th double starts?

# More on Accessing Elements

- Assume a 5-element int array
  - int x[5] = {8,5,3,9,6};
- When you access x[2], the CPU calculates where that item is in memory by taking the start address of x (i.e. 100) and adding the product of the index, 2, times the size of the data type (i.e. int = 4 bytes)
  - x[2] => int. @ address 100 + 2*4 = 108
  - x[3] => int. @ address 100 + 3*4 = 112
  - x[i] @ start address of array + i * (size of array type)
- C does not stop you from attempting to access an element beyond the end of the array
  - x[6] => int. @ address 100 + 6*4 = 124 (Garbage!!)




Compiler must be able to figure out how much memory to allocate at compile-time

Fun Fact 1: If you use the name of an array w/o square brackets it will evaluate to the starting address in memory of the array (i.e. address of 0th entry)
Fun Fact 2: Fun Fact 1 usually appears as one of the first few questions on the midterm.

# Fact or Fiction and Other Questions

- Array indexing starts at 1
- Arrays store values of different types
- "hello" is really just a character array (char [ ])
- If the array 'char str[50]' starts at address 100, then str[5] is located at what address?
- Given the array above, what does 'str' evaluate to when written in code
- If the array int data[40] starts at address 200, where is data[30] located?
- Where is data[42] located?

# Array Exercises

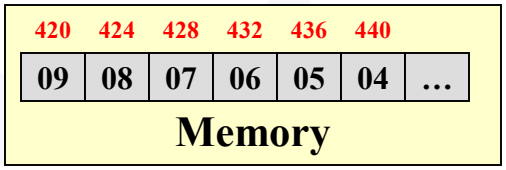
- pow2
- echo
- arrayprint
- arraybad

Passing arrays to other functions

# ARRAYS AS ARGUMENTS

# Passing Arrays as Arguments

- In function declaration / prototype for the *formal* parameter use
  - "type []" or "type *" to indicate an array is being passed
- When calling the function, simply provide the name of the array as the *actual* argument
  - In C/C++ using an array name without any index evaluates to the starting address of the array
- C does NOT implicitly keep track of the size of the array
  - Thus either need to have the function only accept arrays of a certain size
  - Or need to pass the size (length) of the array as another argument




```
void add_1_to_array_of_10(int []);
void add_1_to_array(int *, int);
int main(int argc, char *argv[])
{
  int data[10] = {9,8,7,6,5,4,3,2,1,0};
  add_1_to_array_of_10(data);
  cout << "data[0]" << data[0] << endl;
  add_1_to_array(data,10);
  cout << "data[9]" << data[9] << endl;
  return 0;
}

void add_1_to_array_of_10(int my_array[])
{
  int i=0;
  for(i=0; i < 10; i++){
    my_array[i]++;
  }
}
void add_1_to_array(int *my_array, int size)
{
  int i=0;
  for(i=0; i < size; i++){
    my_array[i]++;
  }
}
```

420

420

420

# Passing Arrays as Arguments (cont.)

- In function declaration / prototype for the *formal* parameter use *type [ ]*
- When calling the function, simply provide the name of the array as the *actual* argument
- Scalar values (int, double, char, etc.) are "passed-by-value"
  - Copy is made and passed
- Arrays are "passed-by-reference"
  - We are NOT making a copy of the entire array (that would require too much memory and work) but passing a reference to the actual array (i.e. an address of the array)
  - Thus any changes made to the array data in the called function will be seen when control is returned to the calling function

```
void f1(int []);

int main(int argc, char *argv[])
{
  int data[10] = {10,11,12,13,14,
                  15,16,17,18,19};
  cout << "Loc. 0=" << data[0] << endl;
  cout << "Loc. 9=" << data[9] << endl;

  f1(data);

  cout << "Loc. 0=" << data[0] << endl;
  cout << "Loc. 9=" << data[9] << endl;
  return 0;

}

void f1(int my_array[])
{
  int i=0;
  for(i=0; i < 10; i++){
    my_array[i]++;
  }
}
```

420

420

**Output:**

Loc. 0=10
Loc. 9=19
Loc. 0=11
Loc. 9=20

# Exercises

- mergearrays
- distinct

# COMMON ARRAY DESIGN PATTERNS

# Design Pattern: Search

- A design pattern is a common recurrence of an approach
- Search: Find one item in an array/list/set of items
- Pattern:
  - Loop over each item likely using an incrementing index
  - For each item, use a conditional to check if it matches the search criteria
  - If it does match, take action (i.e. save index, add value to some answer, etc.) and possibly break, else, do nothing, just go on to next

```
// search 'data' array of size 'len' for 'target' value
bool search(int data[], int len, int target)
{ bool found = false;
  for(int i=0; i < len; i++){
    if(data[i] == target){
       found = true;
       break;
  }  }
  return found;
}
```

# Design Pattern: Search (cont.)

- What's not a search :
  - Indicating the search failed if a single element doesn't match
  - Consider data = {4, 7, 9} and target = 7
  - 4 won't match and set found=false and stop too soon

```
// search 'data' array of size 'len' for 'target' value
bool search(int data[], int len, int target)
{ bool found = false;
  for(int i=0; i < len; i++){
    if(data[i] == target)
       return true;
    else
       return false;
  }
}
```

# Design Pattern: Search (cont.)

- What's not a search :
  - Indicating the search failed if a single element doesn't match
  - Consider data = {4, 7, 9} and target = 7
  - 4 won't match and set found=false and stop too soon
  - 7 will match and set found = true, but only for a second...
  - 9 won't match and set found = false...forgetting that 7 was found

```
// search 'data' array of size 'len' for 'target' value
bool search(int data[], int len, int target)
{ bool found = false;
  for(int i=0; i < len; i++){
    if(data[i] == target)
       found = true;
    else
       found = false;
  }
  return found;
}
```

# Design Pattern: Search (cont.)

- What's not a search :
  - Declaring your result variable inside the for loop
  - Bool found only lives in the current scope (i.e., the 'if' statement and will not be visible afterwards when you need it

```
// search 'data' array of size 'len' for 'target' value
for(int i=0; i < len; i++){
  if(data[i] == target)
     bool found = true;
     break;
}  } // found is deallocated here..too early!
// check found for result of search
```

# Design Pattern: Reduction (cont.)

- Reduction: Combine all items in an array/list/set to produce one value (i.e., sum, check if all meet a certain criteria, etc.)
- Pattern:
  - Declare a variable to hold the reduction
  - Loop over each item likely using an incrementing index
  - For each item, combine it appropriately with your reduction variable

```
// sums 'data' array of size 'len'
int sum = 0;
for(int i=0; i < len; i++){
  sum = sum + data[i];  // sum += data[i]
}
// use sum
```

# Design Pattern: Reduction (cont.)

- Reduction: Combine all items in an array/list/set to produce one value (i.e., sum, check if all meet a certain criteria, etc.)
- Pattern:
  - Declare a variable to hold the reduction
  - Loop over each item likely using an incrementing index
  - For each item, combine it appropriately with your reduction variable

```
// checks if all elements are positive
bool allPos = true;
for(int i=0; i < len; i++){
  allPos = allPos && (data[i] > 0);
}
```

Character Arrays

# C STRINGS

# C Strings

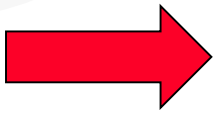
- Character arrays (i.e., C strings)
  - Enclosed in double quotes " "
  - Strings of text are simply arrays of chars
  - Can be initialized with a normal C string (in double quotes)
  - C strings have one-byte (char) per character
  - cout “knows” that if a char array is provided as an argument it will print the 0th character and keep printing characters until a ‘\0’ is encountered

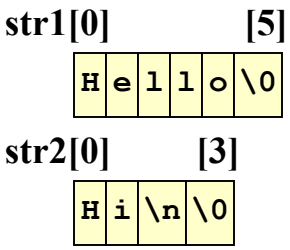
```
#include<iostream>

using namespace std;

int main()
{
  char str1[6] = "Hello";
  char str2[] = "Hi\n";
  cout << str1 << str2;

}
```





# Exercises

- Write a function to compute the length of a C-string
  - Exercises: strlen
- Write a function to copy a C-string from a source character array into a destination character array
  - Exercise: strcpy

# Example: C String Functions

- Write a function to determine the length (number of characters) in a C string
- Write a function to copy the characters in a source string to a destination character array
- Copy the template to your account
  - string_funcs.cpp
- Edit and test your program and complete the functions:
  - int strlen(char str[])
  - strcpy(char dst[], char src[])
- Compile and test your functions
  - main() is complete and will call your functions to test them

# MORE ARRAY APPLICATIONS AND EXERCISES

# Arrays as Look-Up Tables

- Use the value of one array as the index of another
- Suppose you are given some integers as data [in the range of 0 to 5]
- Suppose computing squares of integers was difficult (no built-in function for it)
- Could compute them yourself, record answer in another array and use data to “look-up” the square

```
// the data
int data[8] = {3, 2, 0, 5, 1, 4, 5, 3};

// The LUT
int squares[6] = {0,1,4,9,16,25};
```

```
// the data
int data[8] = {3, 2, 0, 5, 1, 4, 5, 3};

// The LUT
int squares[6] = {0,1,4,9,16,25};

for(int i=0; i < 8; i++){
  int x = data[i]
  int x_sq = squares[x];
  cout << i << ",” << x_sq << endl;
}
```

```
// the data
int data[8] = {3, 2, 0, 5, 1, 4, 5, 3};

// The LUT
int squares[6] = {0,1,4,9,16,25};

for(int i=0; i < 8; i++){
  int x_sq = squares[data[i]];
  cout << i << ",” << sq[i] << endl;
}
```

# Activity



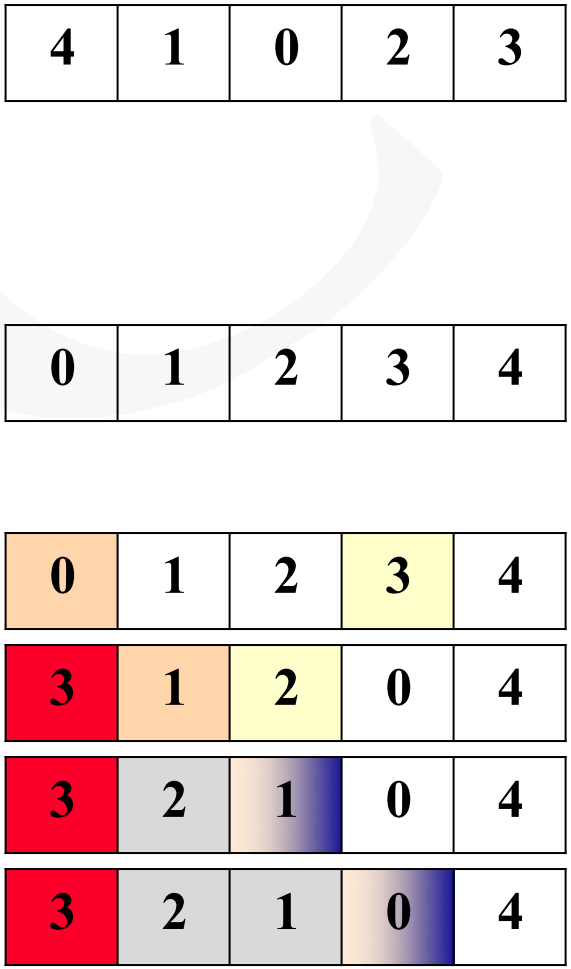
- **Extend your knowledge**
  - **Modify cipher program to accept a whole line of text (including spaces)**
  - **Encrypt either upper or lower case but ignore spaces (i.e., leave them in)**

# Example

- Using an array as a Look-Up Table
  - cipher.cpp
  - Let's create a cipher code to encrypt text
  - `abcdefghijklmnopqrstuvwxyz => ghijklmaefnzyqbcdrstuopvwx`
  - char orig_string[] = "helloworld";
  - char new_string[11];
  - After encryption:
    - new_string = "akzzbpbrzj"
  - Define another array
    - char cipher[27] = "ghijklmaefnzyqbcdrstuopvwx";
    - How could we use the original character to index ("look-up" a value in) the cipher array

# Permute/Shuffle Algorithm

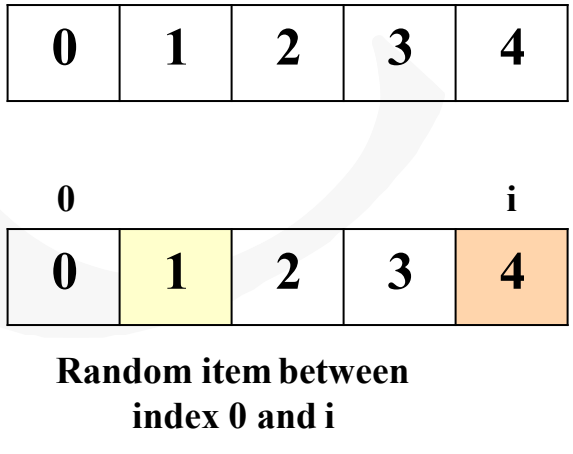
- Problem
  - Generate a random permutation of the first N integers
- Approaches
  - Generate the contents of the array in random order being sure not to duplicate
  - Start with an array with each element then "shuffle"
- A (good, fast) Solution
  - Start with an array in ascending order
  - Pick item at index 0 and swap it with item at a random index [0 to N-1]
  - Pick item at index 1 and swap it with item at a random index [1 to N-1]
  - Pick item at index 2 and swap it with item at a random index [2 to N-1]



# Permute Implementation



- If we start at index i=0,1,2,3 and count up…
  - We need to generate a random number, r, between i and N-1
  - Swap item at i with item at r
- Easier to generate a random number between 0 and k




- If we start at index i=N-1,N-2,… and count down
  - We need to generate a random number, r, between 0 and i
  - Swap item at i with item at r

# 8-Tile Puzzle Exercise



- puzzle8.cpp
- 8-Tile Puzzle
  - 1 blank spot, 8 numbered tiles
  - Arrange numbered tiles in order
  - Use an array to store 9 tile values, 1-8 and 0 as "blank" value
  - What functions would you decompose this program into?
  - Write isSolved() or printBoard()

| | | |
|---|---|---|
| 1 | 3 | 7 |
| 2 | 8 | 6 |
| 5 | 4 | |

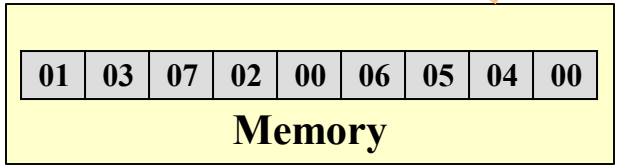
Game Board

Memory representation of the board

| | | |
|---|---|---|
| | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Solved Board

Tiles[9]



# Activity



- Fix the compiler errors in a sample program
- Copy the sample program into your account into examples directory

```
$ mkdir examples
$ cd examples
```

- Edit the file

```
$ gedit lec1_visualerrors.cpp &
```

- Compile the file

```
$ g++ -g -o lec1_visualerrors lec1_visualerrors.cpp
```

- Attempt to fix the errors and iterate