CS 103 Unit 1a – CPP Syntax and Expressions

Review C++ Program Structure

```
// iostream allows access to 'cout'
#include <iostream>
using namespace std;


// Execution always starts at the main() function
int main()
{
  cout << "Hello world" << endl;
  return 0;
}
```

Starting to represent data

MODULE 1: TYPES (CONSTANTS AND VARIABLES)

Basic Output via `cout`

```
// iostream allows access to 'cout'
#include <iostream>
using namespace std;


// Execution always starts at the main() function
int main()
{
  cout << "some ";
  cout << "text" << endl;
  cout << 103  1889 << endl; // Bad...Fix me!
  return 0;
}
```

Printing Different Values & Types

- 'cout' requires appropriate use of the **insertion** operator << as a **separator** between consecutive values or different types of values
  - 'cout' does not add spaces between consecutive values; you must do so explicitly
- Generally good practice to give some descriptive text when outputting variables or computed numbers
  - Note: You may divide output over multiple 'cout' statements. Unless a newline is printed (with 'endl' or '\n'), the next output statement will resume where the last one left off

```
// iostream allows access to 'cout'
#include <iostream>
#include <string>
using namespace std;

int main()
{
  int x = 103;
  cout << x 1889 << endl; // Compile Error!
  cout << x << 1889 << endl; // Better, but no spaces
  cout << x << " " << 1889 << endl; // Best

  string msg = "minutes";
  cout << "There are " << 60*24*365 << " " << msg;
  cout << " in a year." << endl;
  return 0;
}
```

```
1031889
103 1889
There are 525600 minutes in a year.
```

The << operator has multiple (aka "overloaded") meanings. In C (and still in C++) it is used to shift bits in a variable to the left, but C++ also uses it for output. In that (output) context, it is NOT known as the shift operator but the "stream insertion" operator!

C++ Data Types

```
// Execution always starts at the main() function
int main()
{
  int a1 = -42;   // try assigning 4000000000 and see what happens
  unsigned int a2 = 4000000000; // try assigning -1 and see what happens
  double b1 = 3.14;
  float b2 = 1.5;  // double is PREFERRED over float.
  char c = 'a';
  char d = 97;
  bool e = true;
  string f = "abc";
  string g = "Fight On for ol' SC; We all Fight On to victory. Our Alma Mater dear, Looks up
to you, Fight On and win, For ol' SC, Fight On to victory, Fight On!";

  cout << "int: " << a1 << endl;
  cout << "unsigned int: " << a2 << endl;
  cout << "double: " << b1 << endl;
  cout << "float: " << b2 << endl;
  cout << "char " << c << " " << (int) c  << endl;
  cout << "char " << d << " " << (int) d  << endl;
  c = c+1;
  cout << "'a'+1: " << c << endl;
  cout << "bool " << d << endl;
  cout << "Boolean constants: " << true << " " << false << endl << endl;
  cout << "string: " << f << endl;
  cout << "string: " << g << endl;

  return 0;
}
```

C/C++ Data Types

- C/C++ **types** indicate how many bits (bytes) of **storage** (memory) are required and how to **interpret** the number being stored
- **Integer types**
  - **int, unsigned int,** and char (more explanation later)
- **Floating point types -** Very large 6.02E23 & very small numbers 6.626E-34 (i.e. an attempt to represent rational/real numbers)
  - float or double (in general, prefer double over float as it has a greater range of expressivity)
- **String/Text types**
  - char, char arrays, strings
- **Boolean type**
  - bool (true / false)

- Let's look at how to write constants (aka "literals") and declare variables of these types.

Constants (aka Literals)

- Integer: `496, 10005, -234`
- Double: `12.0, -16., 0.23, 6.02E23, 4e-2`
  - Both very large and very small numbers (i.e. fractions/decimals)
- Characters (char type): enclosed in **single quotes (')**
  - Printing characters: `'a', '5', 'B', '!'`
  - Non-printing special characters use "escape" sequences (i.e. preceded by a `\`): `'\n'` (newline/enter), `'\t'` (tab), `'\\'` (slash), `'\''` (apostrophe)
- C-Strings (Note: there is also a C++ string type…)
  - 0 or more characters between **double quotes (")**

    `"hi1\n", "12345", "b", "\tAns. is %d"`
  - Ends with a `'\0'`=0 (aka NULL character) added as the last byte/character to allow code to delimit the end of the string
- Boolean (C++ only): `true, false`
  - Physical representation: 0 = false, Non-zero (1, -5, 300) = true

C/C++ handling of single characters and strings is different than most other languages and a major source of confusion in C++.

0	104	'h'
1	105	'i'
2	49	'1'
3	10	'\n'
4	00	Null
5	17	
6	59	
7	120	
	…	

String Example (Memory Layout)

You're Just My Type

- Indicate which constants are matched with the correct type.

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

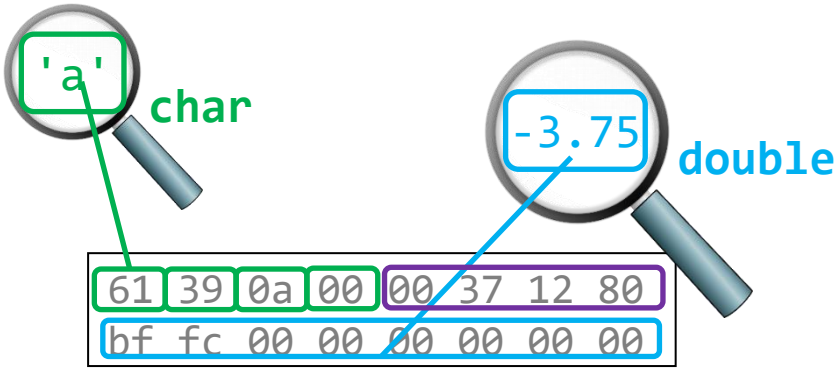
Solutions are provided at the end of the slide packet.

Motivation for Data Types

- How many data values are stored in the memory below (where does one value stop and another start) and what are their values?

0	1	2	3	4	5	6	7
0110 0001	0011 1001	0000 1010	0000 0000	0000 0000	0011 0111	0001 0010	1000 0000
1011 1111	1111 1100	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000
8	9	10	11	12	13	14	15

- C/C++ **types** indicate how many bits (bytes) of **storage** (memory) are required and how to **interpret** the number being stored



Memory contents
(using hexadecimal…see inset below)

As a human, would you rather write MANY 1s and 0s or a few digits? (i.e. 0110 0001 OR 61)
- Probably a few digits.

We (humans) and debuggers often show the contents of memory in base-16 (aka hexadecimal or just hex for short) rather than binary because it is less to write and easier to visually take-in. Everything is truly binary in the computer, but there is an easy and fast way to convert between binary and hex, so we show hex.

Limited Range of Data

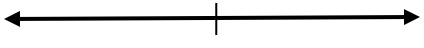
```
int main()
{
  int x = std::numeric_limits<int>::max();
  cout << "int max: " << x << " " << x + 1 << endl;
  x = std::numeric_limits<int>::min();
  cout << "int min: " << x << " " << x - 1 << endl;

  unsigned int y = std::numeric_limits<unsigned int>::max();
  cout << "unsigned int max: " << y << " " << y + 1 << endl;
  y = std::numeric_limits<unsigned int>::min();
  cout << "unsigned int min: " << y << " " << y - 1 << endl;

  char z = std::numeric_limits<char>::max();
  cout << "char max: " << z << " " << z + 1 << endl;
  z = std::numeric_limits<char>::min();
  cout << "char min: " << z << " " << z - 1 << endl;

  bool b = true;
  cout << "bool max: " << b << " " << b + 1 << endl;
  b = false;
  cout << "bool min: " << b << " " << b - 1 << endl;

  return 0;
}
```

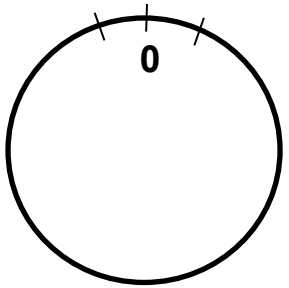
Finite Range of Numbers

- **Recall**: **EVERYTHING** in a computer is a **number**!
- **Key Idea**: In computers, numbers are FINITE because each memory cell has a fixed number of bits (digits)
- Scenario: A hotel has 3-digit room numbers.
  - How many rooms can the hotel have?
  - What if the hotel uses 4-digit room numbers?
  - Range for n-digit room numbers?
- What is 999+1?
  - 1000, obviously! Right!?
  - Well, if we limit ourselves to 3-digit numbers, then the answer is 000! We call this **overflow** and it is a common issue programmer's must account for.

3-digit Room Number

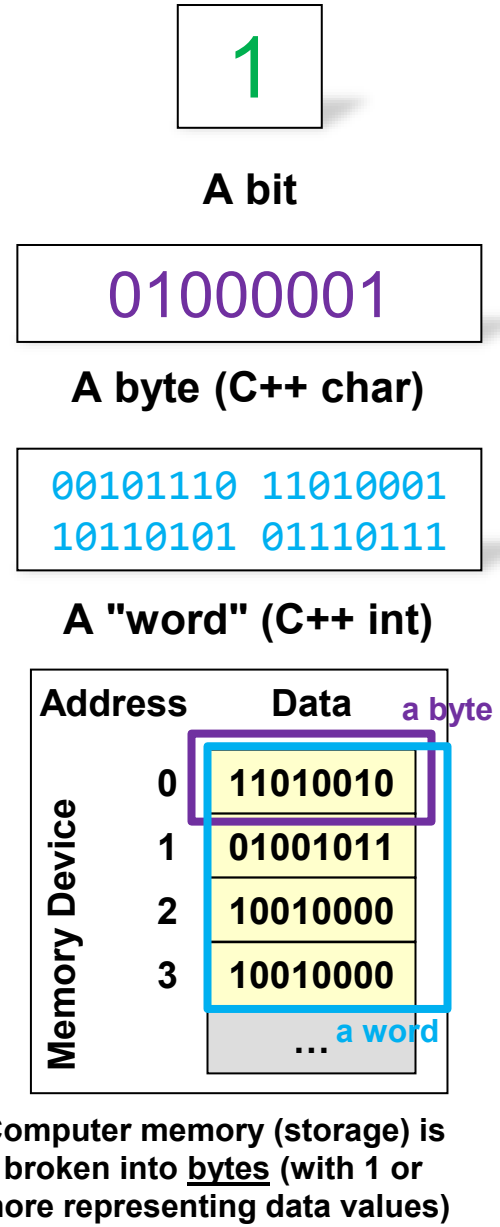
4-digit Room Number





Bits, Bytes, Words

- Computers store data as bits (binary digits) in units of memory with a fixed number of bits
- A single **bit** can only represent 1 and 0
- To represent more than just 2 values we need to use a combination / sequence of many bits
- Computer **hardware** (memory) defines common, easily accessible units of a fixed size:
  - A **byte** is defined as a group 8-bits
  - A **word** varies in size but is usually 32-bits (4 bytes)
- For n-bit numbers, the range of values we can represent is 0 to 2^n-1
  - For 8-bits, the range is 0 to 255.
  - For 32-bits, the range is 0 to 4,294,967,295



Computer memory (storage) is broken into bytes (with 1 or more representing data values)

C/C++ Integer Data Types

- Integer variable types
  - An unsigned (positive-only...including 0) number
  - A signed (positive or negative) number

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

***These are the three integer types we will use 99% of the time**

Summary: C/C++ Floating Point Types

- `float` and `double` types:
  - Allow decimal representation (e.g. 6.125) as well as very large integers (+6.023E23)

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

- Prefer `double` over `float`
  - Many compilers will upgrade floats to doubles anyhow
- Don't use floating-point if you don't need to
  - It suffers from rounding error
  - Some additional time overhead to perform arithmetic operations

Variables

```
int data = -1;  // global variable

void f1() {
  int x = 42;   // Does this work?
  cout << "X: " << x << endl;

  int y = -1;
  cout << "Y: " << y << endl;
  y = "abc";
  cout << "Y: " << y << endl;

  double z;
  cout << "Uninitialized z: " << z << endl;

  int a;
  a += 1;
  cout << a << endl;
}

int main() {
  cout << "Dummy call " << 41 << 42 << 43 << 44 << 45 << 46 << 47 << 48 << endl;
  f1();
  cout << a << " " << data << endl;
  return 0;
}
```

C/C++ Variables

- A variable is a reserved memory location that
  - **Stores a value that can be read (retrieved) or written (changed) as often as desired**
  - **Associates a descriptive name (e.g. x) the programmer will use with that memory location (aka address) and the value stored in that location**
- You must "**declare**" your variables before using/assigning to them
- If not initialized via assignment ('='), variables will NOT default to a value like 0, but will contain random data/garbage.
  - Good practice to initialize your variables

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

int main()
{ // Sample variable declarations
  char w = 'A';
  int x; // Random: 0?, -12? 1758554321?

  x = 0;
  x = x + 3;

  ...
}
```

`char w = 'A';`
A single-byte variable. The name `w` is associated the the memory location 100

`int x;`
A four-byte variable

A picture of computer memory (aka RAM)

100	01000001
101	01001011
102	10010000
103	11110100
104	01101000
105	11010001
106	01101000
107	11010001
...	...
	00001011

Variables are actually allocated in RAM when the program is run

Difference: C required that variables be declared at the beginning of a function before any operations.
C++ relaxes this and allows declarations anywhere in the code.

Scope

- "Scope" of a variable refers to the
  - **Visibility** (who can access it) and
  - **Lifetime** of a variable (how long is the memory reserved
- For now, there are 2 scopes we will learn
  - **Global:** Variables are declared ***outside*** of any function and are visible to ***all*** the code/functions in the program
  - **Local:** Variables are declared ***inside*** of a function and are ***only*** visible in that function and ***die*** when the function ends

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

// Global Variable
int x=1;

int add_x(int input)
{
  // y and z NOT visible (in scope) here
  // but x is since it is global
  return (input + x);
}  // input dies here

int main()
{
  // y and z are "local" variables
  int y, z=5; // y is garbage, z is five

  z = add_x(z);
  y += z;    // BAD!!  Why?
  cout << x << " " << y << endl;
  return 0;
} // y and z die here
```

Summary: C/C++ Variable Types

- A **type** indicates how many bits / bytes of **storage** (memory) are required and how to **interpret** the number being stored
- **Integer types**
  - Are signed (numbers can be positive or negative) by default, or unsigned (positive-only...including 0)
  - A character (more on this later)
- **Floating point types**: Very large 6.02E23 & very small numbers 6.626E-34)
  - A float or double
- **String/Text types**
  - A single char (1 character)
  - character arrays (C-Strings) / string (C++ string type)
- **Boolean type**
  - bool (true / false)

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

int main()
{
  int          a = -1;
  unsigned int b = 2;
  char c = 'A';  // 'A'=65

  float  d1 = 1.5;
  double d2 = 3.14;

  char e[6] = "Hello";
  string f = "Goodbye";

  bool g = true;

  // ...

}
```

Variable

Constant

Variable

Constant

Summary: Common Variable Types

- Variables are declared by listing their type and providing a name
- They can be given an initial value using the '=' operator

```
// iostream allows access to 'cout'
#include <iostream>
using namespace std;

// Execution always starts at the main() function
int main()
{
  int w = -400;
  double x = 3.7;
  char y = 'a';
  bool z = false;
  cout << w << " " << x << " ";
  cout << y << " " << z << endl;
  return 0;
}
```

C Type	Usage	Bytes	Bits	Range
char	Text character Small integer value	1	8	ASCII characters -128 to +127
bool	True/False value	1	8	true / false
int **unsigned int**	Integer values	4	32	-2 billion to +2 billion 0 to +4 billion
double	Rational/real values	8	64	±16 significant digits * $10^{+/-308}$
string	Arbitrary text	1 or more	-	-

Assignment operator (=)

- Syntax:

```
variable = expression;
```

(LHS) ⟸ **(RHS)**

  - **LHS = Left Hand-Side, RHS = Right Hand Side**

```
int x = 0;
x = x + 3;
```

new-value of x (3)

current-value of x (0)

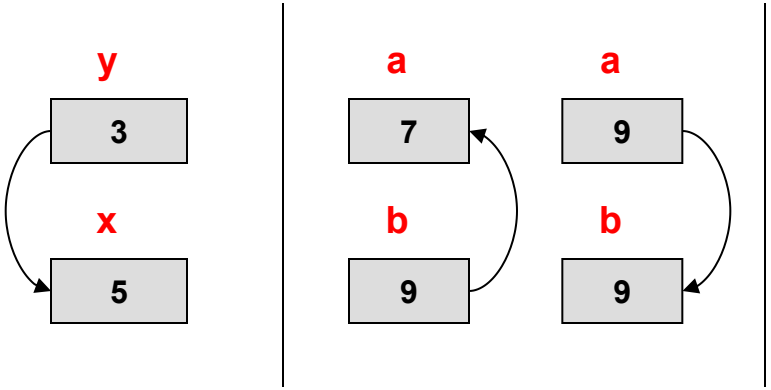
- Should be read: Place the value of *expression* into memory location of *variable*
  - `z = x + y - (2*z);`
  - If variable is on both sides, we use the old/current value of the variable on the RHS

Evaluate **everything** on the right hand side (RHS) before considering the left-hand side (LHS)

- **Note**: Without assignment values are computed and then forgotten
  - `x + 5;` // will take x's value add 5 but NOT update x (just throws the result away)
  - `x = x + 5;` // will actually updated x (i.e. requires an assignment)
- **Shorthand assignment** operators exist for updating a variable based on its current value: `+=, -=, *=, /=, &=, …`
  - `x += 5;` (x = x+5)
  - `y *= x;` (y = y*x)

Assignment Means Copy

- Assigning a variable makes a copy
  - It leaves the source variable unchanged
- Challenge: Swap the value of 2 variables

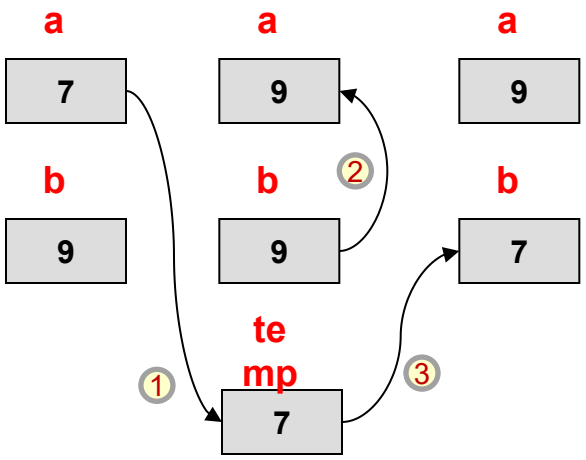



```
int main()
{
  int x = 5, y = 3;
  x = y;    // copy y into x
            // y still has 3
  return 0;
}
```

```
int main()
{
  int a = 7, b = 9;

  // now consider swapping
  // the value of 2 variables
  a = b;
  b = a;

  return 0;
}
```

More Assignments

- Assigning a variable makes a **copy**
  - It leaves the source variable unchanged
- Example: Swap the value of 2 variables
  - Easiest method: Use a 3rd temporary variable to save one value and then replace that variable
- Challenge: 4swap exercise



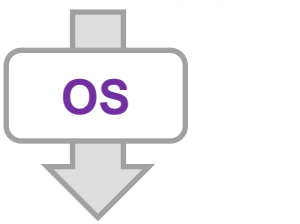

```
int main()
{
 int a = 7, b = 9, temp;

 // let's try again
 temp = a;
 a = b;
 b = temp;

 cout << a << " " << b << endl;
 return 0;
}
```

Inputting and outputting data

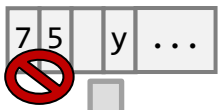
MODULE 2:
C++ I/O (INPUT/OUTPUT)

I/O Streams

- C++ and the OS use the notion of **streams** to temporarily store (aka buffer) data to be input or output and then uses the `cin` and `cout` objects (from the `<iostream>` library) to access those streams
- `cin` **extracts** data from the input stream [stdin] (skipping over preceding whitespace then stopping at following whitespace)
- `cout` **inserts** data into the output stream [stdout] for display by the OS

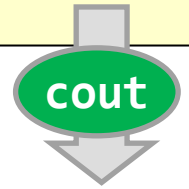


input stream
memory (aka stdin):

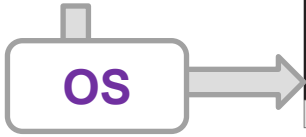



```
#include<iostream>
int main()
{
  int x;
  std::cin >> x;
  return 0;
}
```

```
#include<iostream>
int main()
{
  std::cout << "It was the" << std::endl;
  std::cout << "best of times.";
  return 0;
}
```



output stream
memory (aka stdout):

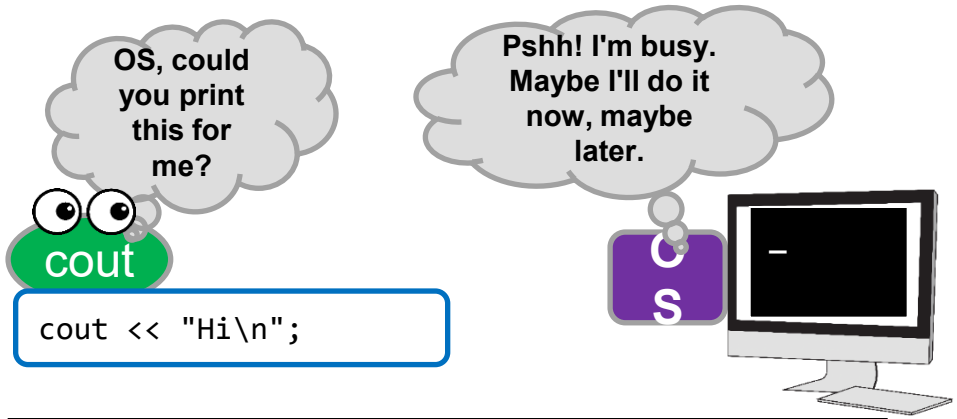


endl and Flushing

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

void task_that_may_crash(bool x);

int main() {
    task_that_may_crash(false);
    cout << "A\n";

    task_that_may_crash(true);
    cout << "B\n";

    return 0;
}
```

Newlines, endl, and Flushing

- To move the cursor to the next line we need to print a new line, '\n' (char)
- cout only inserts the characters to the output stream, but the OS must then copy them to the screen.
- The OS may choose to delay and not print immediately causing strange issues (see bottom)
- endl = '\n' + a flush of the output stream




```
int main() {
  task_that_might_crash();  // Doesn't crash
  cout << "Got Here 1";
  task_that_might_crash();  // Does crash!
  cout << "Got Here 2";
  return 0;
}
```

```
<Segmentation fault>
```

```
int main() {
  task_that_might_crash();  // Doesn't crash
  cout << "Got Here 1" << endl;
  task_that_might_crash();  // Does crash!
  cout << "Got Here 2" << endl;
  return 0;
}
```

```
Got Here 1
<Segmentation fault>
```

Use descriptive messages and endls when debugging.

Input with cin

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

int main() {
    string name;
    int age;

    cout << "Enter your first name and age: " << endl;
    // now get the input
    cin >> name >> age;

    if( age >= 18 ) {
        cout << name << ", you are allowed to vote in the next election." << endl;
    }
    else {
        cout << name << ", wait " << 18-age << " more years to vote." << endl;
    }

    return 0;
}
```

Coming From Java

- If you come from Java, cin works most similarly to the Scanner class with its nextInt(), nextLine(), nextFloat()

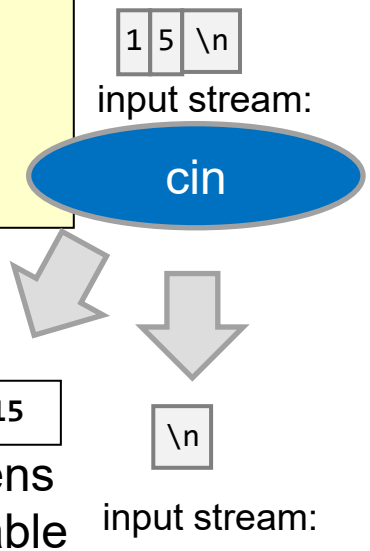
Input with cin

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

int main() {
    char w;
    int x;
    double y;
    string z;

    cout << "Enter a char, int, double, and string: " << endl;
    cin >> w >> x >> y >> z;
    cout << w << " " << x << " " << y << " " << z << endl;

    return 0;
}
```

Keyboard Input

- In C++, the '`cin`' object is in charge of receiving input from the keyboard
- Keyboard input is captured and stored by the OS (in an "input stream") until `cin` is called upon to "extract" info into a variable
- '`cin`' converts text input to desired format (e.g. integer, double, etc.)

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

int main()
{
  int dozens;

  cout << "Enter number of dozen: " << endl;
  cin  >> dozens;

  cout << 12 * dozens << " eggs" << endl;
  return 0;
}
```



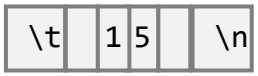
The >> operator also has multiple (aka "overloaded") meanings. In C (and still in C++) it is used to shift bits in a variable to the right, but C++ also uses it for input. In that (input) context, it is known not as the shift operator but the "stream extraction" operator!

Dealing With Whitespace

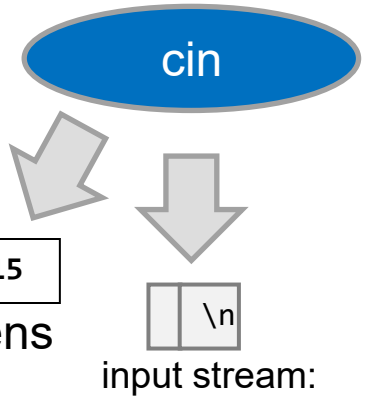
- **Whitespace *(def.):***
  - Characters that represent horizontal or vertical blank space. Examples: newline (`'\n'`), TAB (`'\t'`), spacebar (`' '`)
- cin sequentially discards **leading** whitespace characters until it hits a non-whitespace.
- cin then checks the characters can be converted to the appropriate variable type and keeps scanning for more
- cin will STOP at the first **trailing** whitespace (or on a character unable to be converted to the desired type) and await the next cin command

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

int main()
{
  int dozens;

  cout << "Enter number of dozen: "
       << endl;
  cin  >> dozens;

  cout << dozens << " dozen "
       << " is " << 12*dozens
       << "items." << endl;

  return 0;
}
```

Suppose at the prompt the user types:



input stream:



Main Take-aways:

cin SKIPS leading whitespace

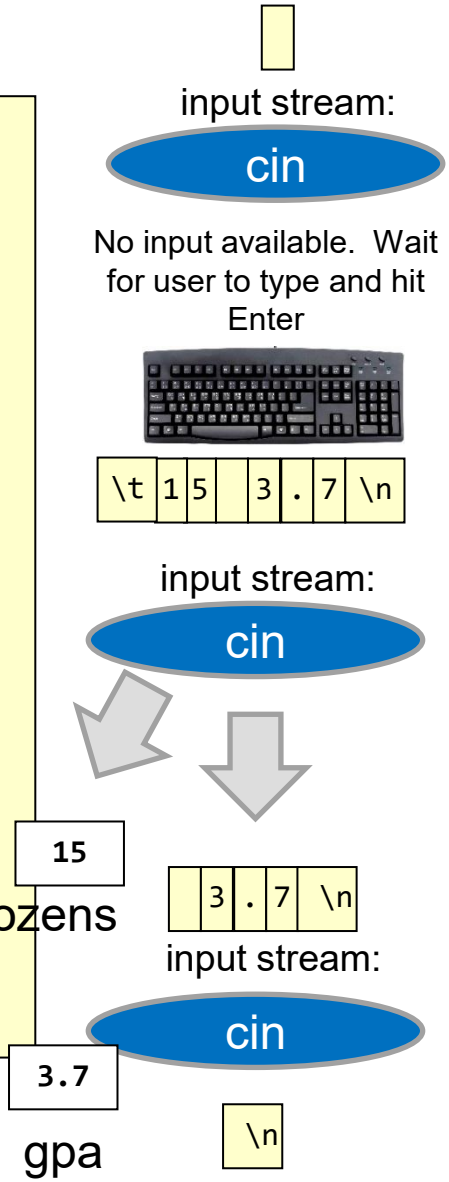
cin STOPS on the first trailing whitespace

Space ≠ Whitespace (Whitespace = ' ', '\t', '\n',

Timing of Execution

- When execution reaches a 'cin' statement, it will either:
  - **Wait** for input if nothing is available in the input stream
    - OS will capture what is typed until the next 'Enter' key is hit
    - User can type as little or much as desired until Enter (\n)
  - **Immediately extract** from the input stream if some text is available and convert it to the desired type of data

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

int main()
{
  int dozens;

  cout << "Enter number of dozen: "
       << endl;
  cin  >> dozens; // input stream empty
                  // so wait for input

  cout << 12*dozens << " eggs" << endl;

  double gpa;
  cout << "What is your gpa?" << endl;
  cin  >> gpa; // input stream has text
               // so do not wait…
               // just use next text

  cout << "GPA = " << gpa << endl;
  return 0;
}
```



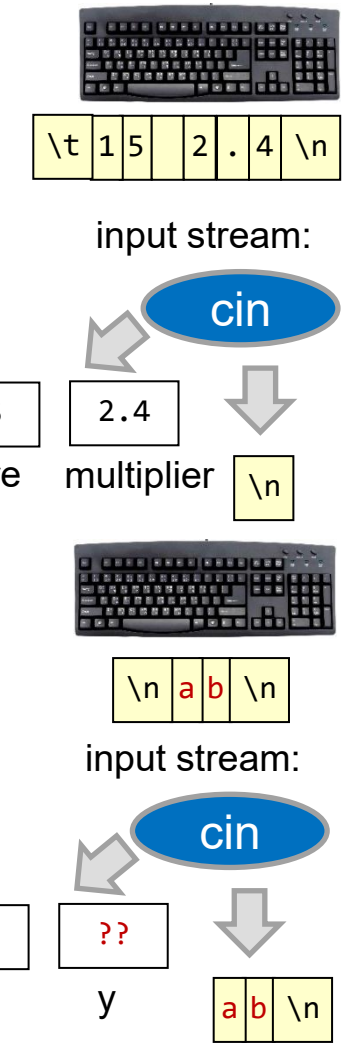
USC Viterbi School of Engineering

Multiple Inputs and Unexpected Inputs

- Use the '>>' operator to separate any number of variables you want to read in
- If non-whitespace characters are encountered, cin simply stops and leaves the variable values unchanged
  - It does not discard the unexpected characters so they will likely cause another error on the next read, too.
  - More on error handling and input validation in a few weeks

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

int main()
{
  int score;
  double multiplier;
  cin >> score >> multiplier;

  cout << "Your new score is "
       << score * multiplier << endl;


  int x, y;
  cin >> x >> y;
  cout << x << " " << y << endl;
  return 0;

}
```

```
15 2.4
Your new score is 36
ab
0 21999
```



Question

- What do you think would happen if the user typed a double when an integer was expected?
- What happens if you type numeric digits when a string is expected?

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

int main()
{
  int x;
  cin >> x;    // User types 1.5 42

  double y, z;
  cin >> y >> z;

  string s;
  cin >> s;    // User types 103.25

  cout << "x = " << x << endl;
  cout << "y,z= " << y << " " << z << endl;
  cout << "s = " << s << endl;
  return 0;

}
```

```
1.5 42
103.25
x =
y,z=
s =
```

Performing computation

MODULE 3: EXPRESSIONS

Expressions and Operators

```
int main() {
    int x = 7, y = 5, z = 12;

    cout << "Modulo:" << endl;
    cout << "7 % 5 = " << x % y << endl;
    cout << "7 % 12 = " << x % z << endl;

    cout << "Integer division:" << endl;
    cout << "7 / 5 = " << x / y << endl;
    cout << "7 / 5.0 = " << x / 5.0 << endl;

    cout << "Update Assignment operator:" << endl;
    x += 1;
    cout << "x = " << x << endl;
    x /= 2;
    cout << "x = " << x << endl;

    cout << "Pre/post increment/decrement:" << endl;
    cout << "z++ : " << z++ << endl;
    cout << "z-- : " << z-- << endl;
    cout << "++z : " << ++z << endl;
    cout << "--z : " << --z << endl;
    return 0;
}
```

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	2 + 5
-	Subtraction	41 - 32
*	Multiplication	4.23 * 3.1e-2
/	Division (**Integer** vs. **Double** division)	10 / 3 (=3) 10.0 / 3 (=3.3333)
%	Modulus (remainder) [for integers only]	17 % 5 (result will be 2)

```
17 % 10 = __
 4 %  7 = __
```

Division

- Computers perform division differently based on the type of values used as inputs
- **Integer** Division:
  - When dividing two integral values, the result will also be an integer (any remainder/fraction will be dropped)
  - 10 / 4 = 2 52 / 10 = 5 6 / 7 = 0
- **Floating-point** (Double) & Mixed Division
  - 10.0 / 4.0 = 2.5 52.0 / 10 = 5.2 6 / 7.0 = 0.8571
  - Note: If one input is a double, the other will be promoted temporarily (aka ***implicitly casted***) to compute the result as a double

Precedence

- Order of operations/ evaluation of an expression
- Higher level (level 16 in table) done first
- Notice operations with the same level or precedence usually are evaluated left to right)
- Evaluate:
  - `2*-4-3+5/2;`
- Tips:
  - Use parenthesis to add clarity
  - Add a space between literals
    `(2 * -4) - 3 + (5 / 2)`

Level	Operators	Description	Associativity
16	::	Scope Resolution	-
15	() [] -> . ++ -- static_cast, dynamic_cast etc	Function Call Array Subscript Member Selectors Postfix Increment/Decrement Type Conversion	Left to Right
14	++ -- + - ! ~ (type) * & sizeof new, delete	Prefix Increment / Decrement Unary plus / minus Logical negation / bitwise complement C-style typecasting Dereferencing Address of Find size in bytes Dynamic Memory Allocation / Deallocation	Right to Left
13	* / %	Multiplication Division Modulo	Left to Right
12	+ -	Addition / Subtraction	Left to Right
11	>> <<	Bitwise Right Shift Bitwise Left Shift	Left to Right
10	< <= > >=	Relational Less Than / Less than Equal To Relational Greater / Greater than Equal To	Left to Right
9	== !=	Equality Inequality	Left to Right
8	&	Bitwise AND	Left to Right
7	^	Bitwise XOR	Left to Right
6	|	Bitwise OR	Left to Right
5	&&	Logical AND	Left to Right
4	||	Logical OR	Left to Right
3	?:	Conditional Operator	Right to Left
2	= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operators	Right to Left
1	,	Comma Operator	Left to Right

https://discuss.codechef.com/upfiles/CPP.PNG

Exercise Review

- Evaluate the following:

```
25 / 3
17 + 5 % 2 - 3
28 - 5 / 2.0
```

Exercises from: D.S. Malik, C++ Programming, 5th Ed., Ch. 2, Q6.

Casting (C and C++ Style)

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

int main() {
  int x = 3, y = 2;
  double z = x/y;
  cout << "z = " << z << endl;

  // What will this print?
  int a = 42;
  cout << "int cast to string: " << static_cast<string>(a) << endl;

  // What will this print?
  double b = 3.14;
  cout << "double cast to string: " << static_cast<string>(b) << endl;

  string s1 = "-3";
  cout << "string cast to int: " << static_cast<int>(s1) << endl;

  string s2 = "4.5";
  cout << "string cast to double: " << static_cast<double>(s2) << endl;

  return 0;
}
```

Casting Motivation

- ***Def***. casting: *Temporarily converting the type of a data value*
- What is the result of 5 + 3/2 ?
  - To achieve the correct answer for 5 + 3 / 2 we 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 implicitly cast (promoted) to the more expressive (int is promoted to double)
- But what if instead of constants we have variables
  - ```
    int x=5, y=3, z=2;
    x + y/z;   // Won't work & you can't write y.0
    ```
- We can perform an **explicit** cast using either the C or C++ syntax
  - ```
    x + (double) y  /  z;                  // C   style casting method
    ```
  - ```
    x + static_cast<double>(y)  /  z ; // C++ style casting method
    ```
- BE CAREFUL!! This won't yield the 6.5 answer you expect.
  - ```
    x + static_cast<double>(y/z); // Why not?
    ```

Common Casting Errors

- Only changes the type **temporarily** for the sake of the expression (not a permanent type change)

- Casting only really works on numeric types and NOT strings
  - Different than many other languages like Python
  - When converting to/from a string, do NOT use casting, but functions from the string library (to_string(), stoi(), stod(), etc.)

```
def f1():
  e = "123"
  f = int(e)
  c = str(42)
```

```
class Main {
 public static f1(){
   int e = Integer.parseInt("42");
   String c = Integer.toString(123);
} }
```

```
#include <iostream>
#include <string>
using namespace std;
int main() {

 double a = 3.6;
 int b = static_cast<int>(a) / 2;
    // Works! b = 1 (casts 3.6 to 3)

 int c = 123;
 string d = static_cast<string>(c);
    // Error! Doesn't compile.
 string d = to_string(c);
    // Works! But only since C++11

 string e = "42";
 int f = static_cast<int>(e);
   // Error! Doesn't compile.
 int f = stoi(e);  // string-to-int
   // Works! But only since C++11
   // use stod() for string-to-double
 return 0;
}
```

Understanding ASCII and chars

- A char is just an integer type that
  - Is only 1 byte (limited range 0 to 255 or -128 to +127)
  - cout uses the **type**, char or int, to infer if we want the ASCII character or integer
- We can perform arithmetic/comparison operations on ASCII chars since they are converted to integers

```
char c = 'a';       // same as  char c = 97;
cout << c << endl; // prints 'a'
c = 97;
cout << c << endl; // prints 'a'
int x = c;
cout << x << endl; // prints 97

char d = 'a' + 1;  // d now contains 98 (ASCII 'b')
cout << d << endl; // prints 'b' on the screen
if(c >= 'a' && c <= 'z') { }    // && means AND
    // better than if(c >= 97 && c <= 122)
c = '1';  d = 1;   // c stores decimal 49, d stores 1
cout << c << " " << d << endl;
                   // only prints: "1 ", not "1 1"
```

char c: 97

int x: 97

char d: 98

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	_		

Pre-/Post- Increment/Decrement

- Increment and decrement operators: ++ and -- (add and subtract 1)
  - If ++ comes **before** a variable it is call pre-increment; if **after**, it is called post-increment
  - `x++; // If x was 2 it will be updated to 3 (x = x + 1)`
  - `++x; // Same as above (no difference when not in a larger expression)`
  - `x--; // If x was 2 it will be updated to 1 (x = x – 1)`
  - `--x; // Same as above (no difference when not in a larger expression)`
- Difference between pre- and post- is only evident when used in a larger expression
- Meaning:
  - Pre: Update (inc./dec.) the variable before using it in the expression
  - Post: Use the old value of the variable in the expression then update (inc./dec.) it
- Examples [suppose we start each example with: `int y; int x = 3;`]
  - `y = x++ + 5; // Post-inc.; Use x=3 in expr. then inc. [y=8, x=4]`
  - `y = ++x + 5; // Pre-inc.; Inc. x=4 first, then use in expr. [y=9, x=4]`
  - `y = x-- + 5; // Post-dec.; Use x=3 in expr. then dec. [y=8, x=2]`
  - `y = --x + 5; // Pre-dec.; Dec. x=2 first, then use in expr. [y=7, x=2]`

Library Functions

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

// Execution always starts at the main() function
int main()
{
  cout << "cos(pi) = " << cos(M_PI) << endl;
  cout << "2 to the 3rd power = " << pow(2,3) << endl;
  cout << "sqrt(100) = " << sqrt(100) << endl;
  cout << "|-3| = " << abs(-3) << endl;
  cout << "Smaller of -5 and -2 = " << min(-5, -2) << endl;
  cout << "\"103\" converted to an int and added to 1 = "
       << atoi("103")+1 << endl;
  cout << "\"1.25\" converted to a double and added to 2.5 = "
       << atof("1.25")+2.5 << endl;
  return 0;
}
```

Math & Other Library Functions

- 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 x^y, or x to the power y (in <cmath>)
  - `sin(x)/cos(x)/tan(s)`: 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)` and `min(x,y)`: returns the maximum/minimum of x and y (in <algorithm>)
- You call these by writing them similarly to how you would use a function in mathematics [using parentheses for the inputs (aka) arguments]
- Result is replaced into bigger expression
- Must #include the correct library
  - #includes tell the compiler about the various pre-defined functions that your program may choose to call

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

int main()
{
  // can call functions
  //  in an assignment
  double res = cos(0); // res = 1.0

  // can call functions in an
  //  expression
  res =  sqrt(2) / 2; // res = 1.414/2

  cout << max(34, 56) << endl;
  // outputs 56

  return 0;
}
```

http://www.cplusplus.com/reference/cmath/

#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**	C Input/Output/File access (printf, fopen, snprintf, etc.)	**iostream**	I/O and File streams (cin, cout, cerr)
stdlib.h **cstdlib**	rand(), Memory allocation, etc.	**fstream**	File I/O (ifstream, ofstream)
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

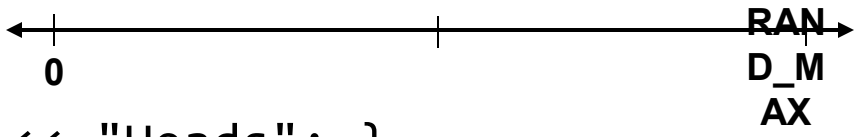
Random Number Generator

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

int main() {
   // srand(42);
   // srand(time(0));
   cout << "RAND_MAX is" << RAND_MAX << endl;
   cout << rand() << endl;
   cout << rand() << endl;
   cout << rand() << endl;
   cout << rand() << endl;
}
```

rand() and RAND_MAX

- (Pseudo)random number generation [(P)RNG] 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 value that simulates the flip of a coin [i.e. 2 outcomes: 0 or 1 with equal probability]?

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



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

```
double r;
r = static_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()!!!

Approximate rand() function:

```
val = ((val * 1103515245) + 12345) % RAND_MAX;
```

SOLUTIONS

You're Just My Type

- Indicate which constants are matched with the correct type.

Constant	Type	Right / Wrong
4.0	int	double (.0)
5	int	int
'a'	string	char
"abc"	string	C-string
5.	double	float/double (. = non-integer)
5	char	Int…but if you store 5 in a char variable it'd be okay (char = some number that fits in 8-bits/1-byte
"5.0"	double	C-string
'5'	int	char

Question

- What do you think would happen if the user typed a double when an integer was expected?
  - `cin` will stop on the decimal point (`'.'`)
- What happens if you type numeric digits when a string is expected?
  - Numeric digits can be part of a string, so it simply reads all characters through the first whitespace.

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

int main()
{
  int x;
  cin >> x;    // User types 1.5 42

  double y, z;
  cin >> y >> z;

  string s;
  cin >> s;    // User types 103.25

  cout << "x = " << x << endl;
  cout << "y,z= " << y << " " << z << endl;
  cout << "s = " << s << endl;
  return 0;

}
```

```
1.5 42
103.25
x = 1
y,z= 0.5 42
s = 103.25
```

DIGGING DEEPER

Parts of 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, numStudents, is_high_enough`, etc.]
    - Identifiers must start with [A-Z, a-z, or an underscore '_'] and can then contain any alphanumeric character [0-9, A-Z, a-z, _] (but no punctuation other than underscores)
    - 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

What's in a name?

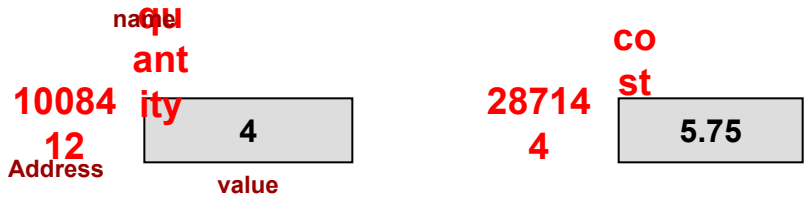
To give descriptive names we often need to use more than 1 word/term. But we can't use spaces in our identifier names. Thus, most programmers use either camel-case or snake-case to write compound names

Camel case: Capitalize the first letter of each word (with the possible exception of the first word)
numStudents, isHighEnough

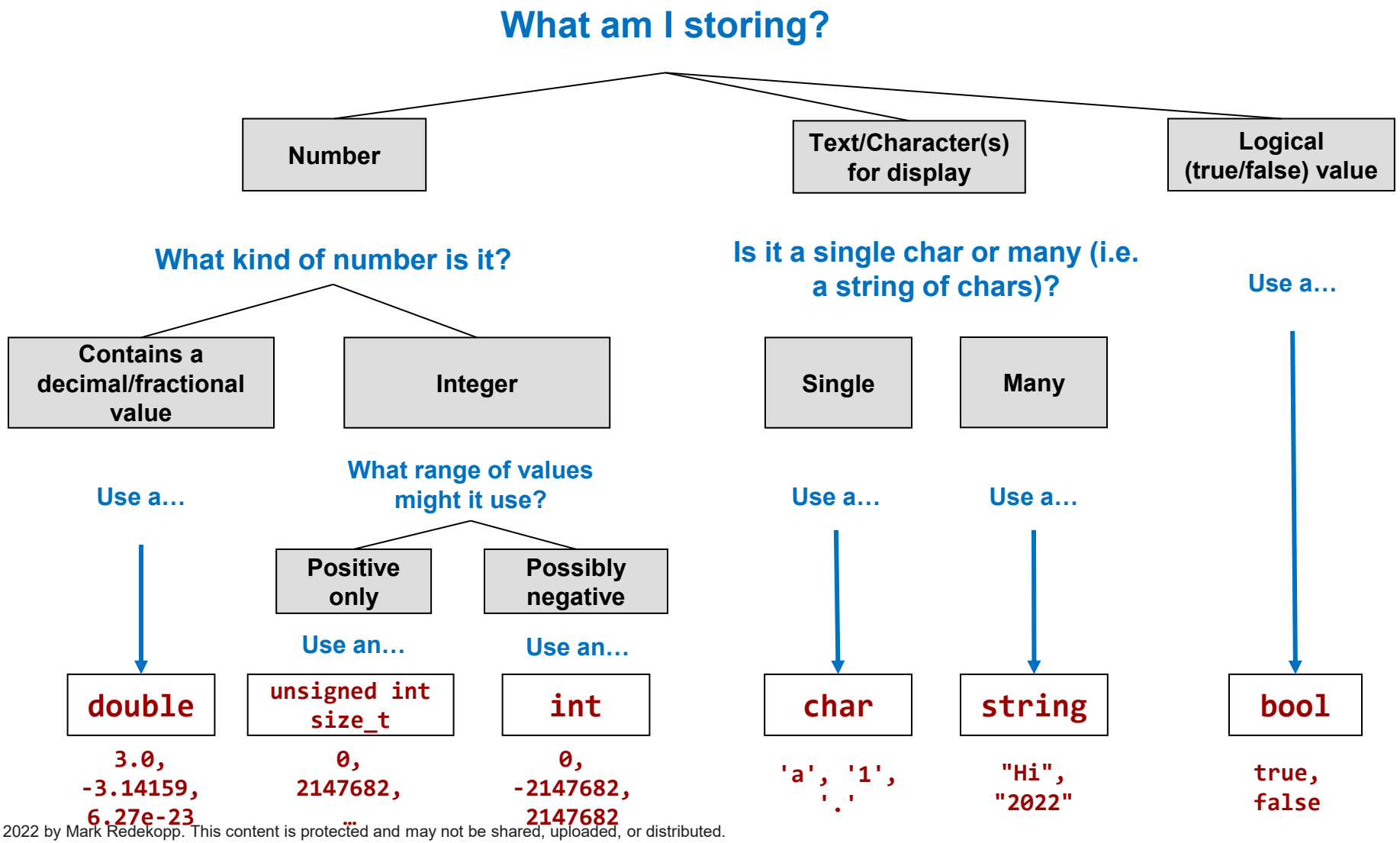
Snake case: Separate each word with an underscore '_'
num_students, is_high_enough

Code

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



Choosing Your Type



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;
    ```

I Do Declare

- Again, (unlike Python) you must do a one-time declaration of a variable before using it
- If not initialized via assignment ('='), variables will NOT default to a value like 0, but will contain random data/garbage.
  - Good practice to initialize your variables
- C++ is a **strongly-typed** language; Python is too, but C++ has more restrictions
  - You cannot *change what type* of value the variable stores); this is because in C++ a variable name corresponds to a reserved, fixed-size memory location

```
#include <iostream>
using namespace std;
int main() {
  x = 5;      // Error: x assigned before
              // it is declared
  int x;      // uninitialized variables
              // will have a (random) garbage
              // value until we initialize it
  x = x+1;    // BAD. X is uninitialized still
  x = 1;      // Initialize x's value to 1

  double y = 3.14;

  x = "pi is"; // Error: x declared as int
               // cannot be assigned a string
  cout << x << " " << y << endl;
  return 0;
}
```

C++ is "strongly-typed" and requires variables to be declared before being used.

```
def main():
    x = 5        # x stores an integer
    y = 3.14
    x = "pi is" # x changes to store a string
    print(x, y)
```

Python does not require explicitly declaring and typing a variable

`int x;`

104	01101000
105	11010001
106	01101000
107	11010001

C/C++ Integer Data Types

- Integer variable types
  - An unsigned (positive-only...including 0) number
  - A signed (positive or negative) number

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

***These are the three integer types we will use 99% of the time**

Summary: C/C++ Floating Point Types

- `float` and `double` types:
  - Allow decimal representation (e.g. 6.125) as well as very large integers (+6.023E23)

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

- Prefer `double` over `float`
  - Many compilers will upgrade floats to doubles anyhow
- Don't use floating-point if you don't need to
  - It suffers from rounding error
  - Some additional time overhead to perform arithmetic operations