CSCI 262
Data Structures

2 - Review

Welcome Back

What you learned in CSCI 261 (or equivalent):

- Variables
- Types
- Arrays
- Expressions
- Conditionals
- Branches & Loops
- Functions
- Recursion
- Classes & Objects
- Streams
- Vectors
- Strings

You remember all of this, right? ☺

Hello, Let's Review

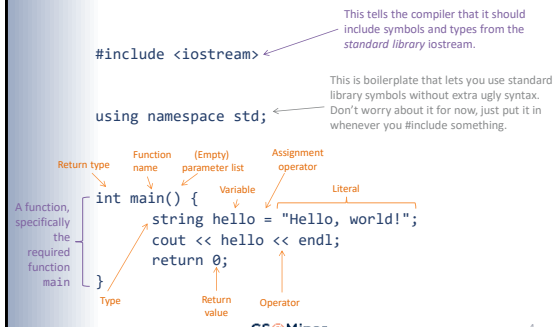
Here's a simple C++ program:

```
#include <iostream>

using namespace std;

int main() {
    string hello = "Hello, world!";
    cout << hello << endl;
    return 0;
}
```


Hello, Let's Review



How to Review

- Remaining slides:
  - Mostly review – not exhaustive, though!
  - Depending on your previous exposure, maybe some new material
- Your responsibility:
  - Go through all the slides that follow
  - Note any questions on old *or* new concepts
  - Try to learn concept from textbook
  - Ask instructor if you still have questions!

Starts with "fun"!

FUNDAMENTALS

Variables

Declaration:

```
int x;
```

Use in expressions:

```
x + 10
```

Set via assignment operator:

```
x = 4;
```

Declare and initialize:

```
int x = 42;
```


Types

- Basic types
  - Integer types:
    - `int` : 42, -99, 103482039
    - `unsigned`: like `int`, but non-negative values only
    - `char` : 'k'
  - Floating point types:
    - `double` : 3.14159, 4.5e3, -0.0001
  - Boolean type:
    - `bool` : true, false
- Pointers
- Arrays
- Class/struct types

Expressions

Working definition: anything with a *value* is an expression:

- Variables
  - `x`
- Indexed array variables
  - `arr[10]`
- Literals
  - `42`
  - `"Hello"`
  - `true`
- Function calls returning a value
  - `sqrt(17)`
- Arithmetic/logical expressions using *operators* (next page)

Operators

Operators are like functions, but expressed in a more "mathematical" format:

The addition operator. It is a *binary infix* operator, i.e., it acts on the two *operands* on either side.

x + 4

The two operands must be expressions. Here, one is a variable, and one is a literal.

The logical negation operator. It is a *unary prefix* operator.

!a

Operators & Expressions

- Arithmetic expressions
  - `4 + 7 / 3.0` Mixed type expressions allowed due to numeric *type conversions*
  - `(x * sqrt(2) + 1) % y`

Q. What is the value 5 / 2 in C++?

- Logical expressions:
  - `count == 0` `// true if count = 0`
  - `a || b && !c` `// a or b and not c`

Which operators act first? Use parentheses or know *precedence* rules.

Expressions and Types

Anything with a value also has a type!

- Literal types are inferred from their formats:

```
42          -> int
"Hello"     -> char[] (not string – more soon)
true        -> bool
```

- Variables/indexed array variables get the type of the variable:

```
int x;
          x          -> int
```

- Function definitions specify return type

```
double sqrt(double n) { … }

          sqrt(42)   -> double
```

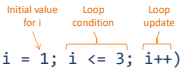
- Operator expressions: type depends on operator **and** operands

```
int x = 1;
          x + 17     -> int
          x + 17.0   -> double

string a = "Hello", b = "world";
          a + b      -> string
```

Loops

What if we want to print "Hello, world!" **three** times?




```
…
for (int i = 1; i <= 3; i++) {
    cout << i << " Hello, world!" << endl;
}
…
```

Output:

```
1 Hello, world!
2 Hello, world!
3 Hello, world!
```


Another Loop

```
…
int i = 3;

while (i > 0) {
    cout << i << " Hello, world!" << endl;
    i--;
}
…
```

Output:

```
3 Hello, world!
2 Hello, world!
1 Hello, world!
```

Also should know use of:
break
continue

Conditionals

```
if (true-false-expression) {
    true-block
}
else {
    false-block
}
```


Hello, if?

Let's modify Hello to respond to an input:

```
…
char answer;
cout << "Say (H)ello or (G)oodbye?" << endl;
cin >> answer;

if (answer == 'H') {
    cout << "Hello, world!" << endl;
} else {
    cout << "Goodbye, world!" << endl;
}
…
```

What happens if the user enters "h" instead of "H"?

Arrays

```
…
int numbers[3];

numbers[1] = 14;
numbers[2] = -3;
numbers[3] = 7093;
…
```

Oops! What's wrong here?

Arrays

```
…
int numbers[3];

numbers[0] = 14;
numbers[1] = -3;
numbers[2] = 7093;
…
```

Let's print out the numbers in the array.
What about in reverse order?

Loops on Arrays

```
…
int numbers[] = {14, -3, 7093};

for (int i = 0; i < 3; i++) {
    cout << numbers[i] << endl;
}

for (int i = 2; i >= 0; i--) {
    cout << numbers[i] << endl;
}
```

Array initialization – only when array is declared!

Even more "fun"!

FUNCTIONS

Functions

We've seen one function:

```
int main() { … }
```

Here's another:

return type | name | parameter list

```
int print_it(string msg) {
    cout << msg << endl;
    return msg.length();
}
```

body (statements)

Hello Functions!

A silly program.

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

int print_it(string);

int main() {
    int n;
    double nroot;

    n = print_it("Hello, world!");
    nroot = sqrt(n);
    cout << "The square root of the number of characters printed is ";
    cout << nroot << endl;
    return 0;
}

int print_it(string msg) {
    cout << msg << endl;
    return msg.length();
}
```

Note, we have to declare a function **before** we use it. Here we used a *function prototype* declaration; alternately we could have just defined the entire function here.

Recursion

Functions can call themselves.

```
void print_n_times(string s, int n) {
    if (n == 0) return;
    cout << s << endl;
    print_n_times(s, n – 1);
}
```

Base case. Very important!

Recursive call. Note that the parameter n moves towards the base case condition.

Function Overloading

- C++ allows multiple functions of the same name:

```
void print_it(int x) {
    cout << "an integer: " << x << endl;
}

void print_it(string s) {
    cout << "a string: " << s << endl;
}
```

- What to call based on the parameter list
  - So parameter lists *must* be different for each overload
  - Can get confusing when mixed with type promotion:

```
print_it(3.1415);   // what does this do?
```

Default Parameters

Alternative when one overload is just a specialized version of another:

```
// prints n times, or just once if n omitted
void print_n_times(string s, int n = 1) {
    for (int j = 0; j < n; j++) {
        cout << s << endl;
    }
}
```

With the above, we can do:

```
print_n_times("Hello", 10);        // prints Hello 10 times
```

or

```
print_n_times("Goodbye");          // prints Goodbye once
```

Rules:

- Cannot omit earlier parameters, supply later ones
- Cannot overload if parameter list is interpretable as call to function with default params omitted, e.g., cannot also define

```
void print_n_times(string s) { … }
```


Pass by Value or Reference

What does this program print?

```
void set_to_zero(int x) {
    x = 0;
}

int main() {
    int n = 42;
    set_to_zero(n);
    cout << n << endl;
}
```

Answer: 42
Parameter passed by *value*

Passing Parameters by Reference

```
void set_to_zero(int &x) {
    x = 0;
}

int main() {
    int n = 42;
    set_to_zero(n);
    cout << n << endl;
}
```

This prints: 0

The Stack

- Holds "stack frames" aka "activation records"
- Each function call results in a new stack frame
- Each stack frame contains memory for:
  - Local variables declared in the function
  - Arguments passed into function
  - Return address for function
- When the function is exited, all of this memory is returned to the stack automatically.

Function Call Example

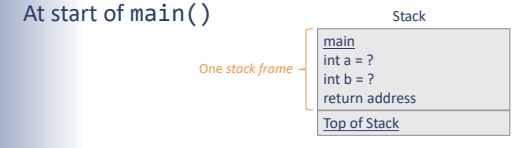
```
void quotient(double num, double den) {
    double q = num / den;
    cout << num << '/' << den << " is " << q << endl;
}

void print_quotients(int x, int y) {
    quotient(x, y);
    quotient(y, x);
}

int main() {
    int a, b;
    cout << "Please enter 2 non-zero integers: ";
    cin >> a >> b;
    print_quotients(a, b);
    return 0;
}
```


Example

At start of `main()`



By custom, the function call stack is visualized as growing *down*.

```
int main() {
    int a, b;
    cout << "Please enter 2 non-zero integers: ";
    cin >> a >> b;
    print_quotients(a, b);
    return 0;
}
```

Example

After getting input:

```
> Please enter 2 non-zero
integers: 7 2
```

```
int main() {
    int a, b;
    cout << "Please enter 2 non-zero integers: ";
    cin >> a >> b;
    print_quotients(a, b);
    return 0;
}
```

Stack

main
int a = 7
int b = 2
return address

Top of Stack

CS@Mines

Example

At beginning of call to `print_quotients`:

```
> Please enter 2 non-zero
integers: 7 2
```

```
void print_quotients(int x, int y) {
    quotient(x, y);
    quotient(y, x);
}
```

Stack

main
int a = 7
int b = 2
return address

print_quotients
int x = 7
int y = 2
return address

Top of Stack

CS@Mines

Example

At beginning of first call to `quotient`:

```
> Please enter 2 non-zero
integers: 7 2
```

```
void quotient(double num, double den) {
    double q = num / den;
    cout << num << '/' << den << " is " << q << endl;
}
```

Stack

main
int a = 7
int b = 2
return address

print_quotients
int x = 7
int y = 2
return address

quotient
double num = 7
double den = 2
double q = ?
return address

Top of Stack

CS@Mines

Example

At end of call to quotient:

```
> Please enter 2 non-zero
integers: 7 2
> 7/2 is 3.5
```

```
void quotient(double num, double den) {
    double q = num / den;
    cout << num << '/' << den << " is " << q << endl;
}
```

Stack

main
int a = 7
int b = 2
return address

print_quotients
int x = 7
int y = 2
return address

quotient
double num = 7
double den = 2
double q = 3.5
return address

Top of Stack

CS@Mines

Example

After return from call to `quotient`:

```
> Please enter 2 non-zero
integers: 7 2
> 7/2 is 3.5
```

```
void print_quotients(int x, int y) {
    quotient(x, y);
    quotient(y, x);
}
```

Stack

main
int a = 7
int b = 2
return address

print_quotients
int x = 7
int y = 2
return address

Top of Stack

CS@Mines

Example

At beginning of second call to `quotient`:

```
> Please enter 2 non-zero
integers: 7 2
> 7/2 is 3.5
```

```
void quotient(double num, double den) {
    double q = num / den;
    cout << num << '/' << den << " is " << q << endl;
}
```

Stack

main
int a = 7
int b = 2
return address

print_quotients
int x = 7
int y = 2
return address

quotient
double num = 2
double den = 7
double q = ?
return address

Top of Stack

CS@Mines

Example

At end of second call to `quotient`:

```
> Please enter 2 non-zero
integers: 7 2
> 7/2 is 3.5
> 2/7 is 0.285714
```

```
void quotient(double num, double den) {
    double q = num / den;
    cout << num << '/' << den << " is " << q << endl;
}
```

Stack

main
int a = 7
int b = 2
return address
print_quotients
int x = 7
int y = 2
return address
quotient
double num = 2
double den = 7
double q = 0.285714
return address
Top of Stack

Example

After return from second call to `quotient`:

```
> Please enter 2 non-zero
integers: 7 2
> 7/2 is 3.5
> 2/7 is 0.285714
```

```
void print_quotients(int x, int y) {
    quotient(x, y);
    quotient(y, x);
}
```

Stack

main
int a = 7
int b = 2
return address
print_quotients
int x = 7
int y = 2
return address
Top of Stack

Example

After call to `print_quotients`:

```
> Please enter 2 non-zero
integers: 7 2
> 7/2 is 3.5
> 2/7 is 0.285714
```

Stack

main
int a = 7
int b = 2
return address
Top of Stack

CLASSES AND OBJECTS

Objects

C++ is an *object-oriented* (OO) language.

What's an *object*?

A working definition:

An object is a package of data with associated behavior.

More specifically, we say that an object has *properties* (fields, attributes, data, state), and that it has associated *methods* (functions).

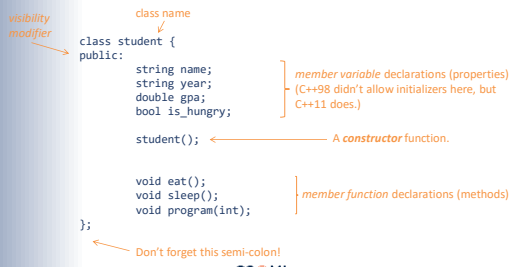
Classes

- Objects also have *type*. Objects of the same type:
  - Have a common set of properties and methods
  - Used in a similar manner to primitive types.
- Types are (usually) modeled by *classes*. Classes formally define the properties and methods.
- Essentially, defining classes is a way to add new types to C++.

(Classes do some other neat things, too, but we'll get to that later.)

Classes in C++

Classes are created via a `class` declaration:



Defining Member Functions

The declaration only gave the member function *signatures* (prototypes); we still have to write the functions themselves:

Scope resolution operator defines what class the method belongs to.

```
void student::eat() {
    is_hungry = false;
}

void student::program(int assignment) {
    if (grade(this, assignment) == 'A') gpa++;
}
```

Etc.

`this` is a special keyword that references the object itself. More on `this` later.

Using Objects in C++

- Objects can be created just like chars, ints, etc.:
  ```
  student s;
  ```
- Properties are referenced by the "`.`" operator:
  ```
  s.name = "April";
  s.gpa = 4.0;
  double d = s.gpa;
  ```
- Methods are invoked on objects also using "`.`":
  ```
  s.sleep();
  ```


Some Notes on Visibility

- Many philosophies around visibility
  - "All data should be private"
  - Partly a matter of style
- Rule of thumb:
  - If it is specific to the *implementation*, it is private
  - Else, it is public
- Not all OO languages have visibility modifiers. (But they all have commenting systems!)

Input/Output

STREAMS

Streams

- Console I/O: #include <iostream>
  ```
  cin >> some_var;
  cout << expression << endl;
  string s;
  getline(cin, s);  // must #include <string>
  ```
- File I/O: #include <fstream>
  ```
  ifstream fin("words.txt");
  fin >> some_var;
  getline(fin, s);

  ofstream fout("output.txt");
  fout << expression << endl;
  ```
- We'll also learn about stringstream objects (later).

VECTORS

Arrays and Vectors

Arrays:

```
int foo[10];
for (int j = 0; j < 10; j++)
      foo[j] = j;
```

Vectors:

```
#include <vector>
…

vector<int> foo(10);
for (int j = 0; j < 10; j++)
      foo[j] = j;   // ≈ foo.at(j) = j
```

Declares that this vector will hold int values.

Gives an initial size to the vector (optional).

Do More with Vectors

E.g. you can append to a vector – it automatically resizes:

```
vector<int> foo;
for (int j = 0; j < 10; j++) {
      foo.push_back(j);
}
```

Initially size 0 without optional argument.

foo contains:

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

And so much more: see Help page of course website for C++ documentation websites.

Something New-ish

C++ 11 added a new type of `for` loop:

```
vector<int> numbers = {14, -3, 7093};

for (int x: numbers) {
   cout << x << endl;
}
```

Note vector *initializer list* – can be used *almost* like a literal in certain contexts.

This denotes that x is a variable of type int which will take on each value in numbers in turn.

STRINGS

About Strings

In C/C++, the literal `"Hello"` is called a **string**.
It is of type `char[]` (a char array).

Confusingly, C++ defines a new type, **string**.

A **string** is *mostly* interchangeable with a **string** (which in C++ is called a "C-string").
But, you can do more with **string** objects:

```
#include <string>
…
string foo = "Hello"; // note assignment of string to string
string bar = "World"; // actually implicit constructor call
string hello = foo + ", " + bar + "!";
if (foo == bar) { … } // test for equality works with string
```

More About Strings

Know/learn the `string` interface!

- See Help page of course website for C++ documentation websites
- Some string *methods* you should know:

```
length    operator[]
size      operator+
find      operator+=
substr    relational operators
```


Up Next

- Please finish reviewing chapters 1 – 6, 7.7, 8, 9.1 – 9.9, and 9.11 in your textbook
- Friday, January 11:
  - Lab 1 – Compile
  - APT 1 assigned
  - Reading: Chapter 7.1 and optionally Appendix F
- TBA (tentative: Thursday at 6pm, Sunday at 3pm)
  - (Optional) Transitioning from Java to C++ sessions
- Monday, January 14
  - Pointers
  - Reading: 14.1 – 14.2
  - **Lab 1 due**