

CSE 1325: Object-Oriented Programming

Lecture 02 – Chapter 03

Classes, Types, and Values

Intro to UML + TLDs

Mr. George F. Rice

george.rice@uta.edu

Based on material by Bjarne Stroustrup


www.stroustrup.com/Programming

ERB 402

Office Hours:

Tuesday Thursday 11 - 12

Or by appointment



Lecture #1

Quick Review

- Which tools and environments will be used for homework assignments and projects this semester? Ubuntu Linux, Gnu Compiler Collection (gcc) with the Data Display Debugger (ddd), Umbrello, and git
- The task of keeping a software system consisting of many versions and configurations well organized is called version control.
- Specify the git command for each of the following actions:
 - Initialize a local git repository git init
 - Add / update a file to the local repository git add
 - Commit added / updated files to the repository git commit -m “message”
 - Undelete a file by retrieving from the repository git checkout file
 - Comparing the differences between two file versions git diff file
- Does adding a file to a local git repository store the file? No, it only adds it to the “stage”. It will be copied to the repository on the next git commit command.
- List some options for backing up your class work. Duplicate your files to another directory, copy them to a flash or portable drive, or copy them to a cloud filesystem like github, dropbox, or owncloud.
 - Why is this important? Because All Technology Eventually Fails™.

Who was that Masked Makefile?

(Included in the Lecture 01 download)

info: **The default – typing “make” echoes the text literally (the @ tells make NOT to show the commands).**

```
@echo
@echo "Type 'make hello.cpp' to run the C++ version."
@echo "Type 'make hello.c' to run the C version."
@echo "Type 'make hello.py' to run the Python version."
@echo "Type 'make HelloWorld.java' to run the Java version."
@echo "      (You may need to run 'sudo apt-get install openjdk-9-jdk-headless'"
@echo "      to successfully run the Java version.)"
@echo
```

.PHONY tells make to ignore the listed files in the current directory and just follow the rules.

```
.PHONY: hello.cpp hello.c hello.py HelloWorld.java clean
```

hello.cpp: **First “real” rule! To “make hello.cpp”,**
 \$(CXX) -o hello hello.cpp **compile hello.cpp into an executable named hello,**
 ./hello **then run it.**

hello.c: **To “make hello.c”,**
 cc hello.c **compile hello.c into the default executable named a.out,**
 ./a.out **then run it.**

hello.py: **To “make hello.py”,**
 ./hello.py **just run it (Python self-compiles as needed).**

HelloWorld.java: **To “make HelloWorld.java”,**
 javac HelloWorld.java **compile HelloWorld.java to HelloWorld.class,**
 java HelloWorld **then run it using the “java” virtual machine.**

clean: **To “make clean”,**
 -rm -f *.o *~ *.class hello a.out **remove the listed files. 1st “-” means “ignore any errors”.**



Homework Questions?

- **Where are the ambiguities in homework #1?**
- All files must be submitted as a **single zip archive**
 - Check out the mandatory filename, directory structure, and file format requirements
- Submit early and often!
 - If you submit more than once, we'll simply grade the most recent one prior to the deadline *unless you email the grader to specify an earlier submission*
- **Always submit something for your homework**
 - Partial Credits Я Us!

Today's Topics

- **Input and (More) Output**
- **Operators and Relationals**
- **Names**
 - Legal vs illegal
 - Recommendations
- **Primitive Types** (built into the C++ compiler)
 - Integers, floating points, Booleans, etc.
- **Class types** (defined outside of the compiler)
 - Enumerations, structures, and full classes
- **Intro to Unified Modeling Language (UML)**
 - Top Level Diagram (TLD)
- **Intro to Intellectual Property**



A Closer Look at Hello World

`#include "file"` (as opposed to `#include <file>`) searches additional directories *first*, most commonly "."

Once found, the file's text is *literally* inserted in place of the `#include`, exactly as if it were typed there.

This is unrelated to Twitter.

The `cout` method (pronounced "see-out") sends characters* streamed to it to STDOUT. This is usually the console.

We return 0 to indicate success. Any other `int` is an error code indicating failure.

Namespace "std" is (more or less) a set of declarations useful in many programs

Stream operators like `<<` can be chained on the same line of code, like `1+2+3`

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello " << "World!" << '\n';
    return 0;
}
```

Single quotes surround single characters (in this case, a newline).
Double quotes surround multiple characters.

A semicolon (;) terminates statements, *but not directives* (which start with #)

* ASCII characters. And *sometimes* Unicode characters.
But C++ and Unicode go together like apples and airplanes.

Input in C++

The cin method (pronounced “see-in”) accepts characters streamed to it from STDIN – usually the keyboard – and converts them into the indicated variables.

Each word (separated by “whitespace” such as spaces or tabs) is handled *separately* in the stream.

```
#include <iostream>
using namespace std;
int main() {
    cout << "Enter two integers: ";
    int num1, num2;
    cin >> num1 >> num2;
    cout << "The sum is " << num1 + num2 << endl;
    cout << "The difference is " << num1 - num2 << endl;
    cout << "The product is " << num1 * num2 << endl;
    return 0;
}
```

Reserve memory for two “objects” that each contain an “int”. “num1” and “num2” are *variables*.

Read two “int” objects from the console, convert to ints, and store in the variables.

Recall the variable values, do math, and add the result to the cout stream.

“endl” is a macro that means “start a new line” or “\n”

```
riceg@pluto:~/dev/cpp/02$ ./a.out
Enter two integers: 5 3
The sum is 8
The difference is 2
The product is 15
```

What About newline?

- C++ also has a version of `getline()` that accepts a C++ string (as opposed to a C `char*`)

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

int main() {
    string s1;
    cout << "Enter your name: ";
    cin >> s1;
    cout << "Your name is " << s1 << endl;
}
```

```
ricegf@pluto:~/dev/cpp/201701/02$ g++ -o cin cin.cpp
ricegf@pluto:~/dev/cpp/201701/02$ ./cin
Enter your name: Professor Rice
Your name is Professor
ricegf@pluto:~/dev/cpp/201701/02$
```

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

int main() {
    string s1;
    cout << "Enter your name: ";
    getline(cin, s1);
    cout << "Your name is " << s1 << endl;
}
```

```
ricegf@pluto:~/dev/cpp/201701/02$ g++ -o getline getline.cpp
ricegf@pluto:~/dev/cpp/201701/02$ ./getline
Enter your name: Professor Rice
Your name is Professor Rice
ricegf@pluto:~/dev/cpp/201701/02$
```

Note that `cin >>` reads a whitespace-separated *word* while `getline` reads an entire `\n`-terminated *line*. Thus, `getline` consumes the `\n`, while `cin >>` does not.

The `-o` specifies the name of the executable to build.

Mixing cin and newline

- Mixing “cin >>” with newline requires care

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

int main() {
    string first, full;
    while(1) {
        cout << "Enter your first name: ";
        cin >> first;
        cout << "Enter your full name: ";
        getline(cin, full);
        cout << first << ", your full name is "
             << full << endl;
    }
}
```

Lines of code may be broken into several physical lines in the file.

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

int main() {
    string first, full;
    while(1) {
        cout << "Enter your first name: ";
        cin >> first;
        cin.ignore();
        cout << "Enter your full name: ";
        getline(cin, full);
        cout << first << ", your full name is "
             << full << endl;
    }
}
```

ignore() is a method of the cin object.

```
ricegf@pluto:~/dev/cpp/201701/02$ g++ -o mixed_in1 mixed_in1.cpp
ricegf@pluto:~/dev/cpp/201701/02$ ./mixed_in1
Enter your first name: George
Enter your full name: George, your full name is
Enter your first name: ^C
ricegf@pluto:~/dev/cpp/201701/02$
```

```
ricegf@pluto:~/dev/cpp/201701/02$ g++ -o mixed_in2 mixed_in2.cpp
ricegf@pluto:~/dev/cpp/201701/02$ ./mixed_in2
Enter your first name: George
Enter your full name: George F. Rice
George, your full name is George F. Rice
Enter your first name: Bjarne
Enter your full name: Bjarne Stroustrup
Bjarne, your full name is Bjarne Stroustrup
Enter your first name: ^C
ricegf@pluto:~/dev/cpp/201701/02$
```

getline picks up the \n left by cin. We need to ignore() it!
<http://cplusplus.com/reference/istream/istream/ignore/>

Multi-Text and Integer Input in C++

Starting here, we will sometimes omit the standard `#include`, `main { }` and `return` verbiage to focus on new concepts, but *you must include them in your code!*
We also drop some build examples (although all code is test compiled).

```
cout << "What is your name and GPA? ";  
string first, last;  
double gpa;  
cin >> first >> last >> gpa;  
string name = first + ' ' + last;  
cout << "Hello " << name  
      << " (GPA " << gpa << ")!\n";
```

You can declare more than one variable of the same type on a line of code (but don't).

A variable of type "double" can hold any number, including one with a decimal point.

`cin` places whitespace-separated tokens into the variables specified – `first`, `last`, and `gpa`

"+" also concatenates (assembles together end to end) strings and characters.

```
ricegfp@pluto:~/dev/cpp/02$ ./a.out  
What is your name and GPA? George Rice 3.81  
Hello George Rice (GPA 3.81)!
```

Challenge

Improve your GPA by a half point. No, a full point!



Common Operators and Relationalals

- C++ uses standard notation for operators
 - + - * / (int, double) addition, subtraction, multiplication, and division
 - + (string) concatenates (- * and / are errors)
 - += -= *= /= (int, double) performs the operator on the target
 - += (string) concatenates the string to the end of the target (others are error)
- C++ uses standard notation for comparisons
 - == equal to
 - != not equal to
 - > greater than
 - >= greater than or equal to
 - < less than
 - <= less than or equal to

```
cout << "Enter a number? ";  
double x;  
cin >> x;  
cout << "x is " << x << endl  
      << "x squared is " << x*x << endl  
      << "twice x is " << x+x << endl  
      << "half x is " << x/2 << endl  
      << "√x is " << sqrt(x) << endl  
      << "x is 1.0 is " << (x==1.0) << endl;
```

```
ricegf@pluto:~/dev/cpp/02$ ./a.out  
Enter a number? 1.0  
x is 1  
x squared is 1  
twice x is 2  
half x is 0.5  
√x is 1  
x is 1.0 is 1
```

The Ternary Operator

- The C++ ternary (?) operator is an *in-line* if / else statement to select *data* rather than *statements*

```
- cout << (name == "Rice") ? "Prof" : "Student" << endl;
```

conditional If true If false

- Same as

```
if (name == "Rice") cout << "Prof" << endl;
else cout << "Student" << endl;
```

- Ternaries can be nested

```
- cout << (name == "Rice") ?
    ((class == "CSE1325") ? "Prof" : "Advisor")
    : "Student" << endl;
```

- These should only be used to *simply* select data within an expression
 - When in doubt, use if / else!

Names

- User-defined names:
 - Always start with a letter
 - May contain only letters, digits, and underscores
 - Do not duplicate C++ keywords
- Examples
 - Valid: `x`, `number_of_elements`, `Fourier_transform`, `z2`
 - Valid but not recommended: `_foo` (leading underscores are reserved for the system)
 - Invalid: `12x`, `timetomarket`, `main line` (no leading letter)
 - Invalid: `int`, `if`, `while`, `double` (already defined as keywords)

Names are case sensitive – `foo` and `Foo` are different names!

Primitive Types in C++

Type	Bytes	Min Value	Max Value
bool	1	false	true
char	1	0	255
unsigned char	1	0	255
signed char	1	-128	127
int	4	-2,147,483,648	2,147,483,647
unsigned int	4	0	4,294,967,295
signed int	4	-2,147,483,648	2,147,483,647
short int	2	-32,768	32,767
unsigned short int	2	0	65,535
signed short int	2	-32,768	32,767
long int	4	-2,147,483,648	2,147,483,647
signed long int	4	-2,147,483,648	2,147,483,647
unsigned long int	4	0	4,294,967,295
float	4	-3.4e +/- 38 (~7 digits)	3.4e +/- 38 (~7 digits)
double	8	-1.7e +/- 308 (~15 digits)	1.7e +/- 308 (~15 digits)
long double	8	-1.7e +/- 308 (~15 digits)	1.7e +/- 308 (~15 digits)
wchar_t	2 or 4		1 wide character

These values are typical, and are always implementation-dependent.

Primitive type – A data type that can typically be handled directly by the underlying hardware

Expertise
/eks-per-tyz/

def: Special skill or knowledge in a particular subject, eg. He has expertise in his field of molecular science.

Primitive Types in C++

Introspecting Your Compiler

Type	Bytes	Min Value	Max Value
bool	1	false	true
char	1	0	255
unsigned char	1	0	255
signed char	1	-128	127
int	4	-2,147,483,648	2,147,483,647
unsigned int	4	0	4,294,967,295
signed int	4	-2,147,483,648	2,147,483,647
short int	2	-32,768	32,767
unsigned short	2	0	65,535
signed short	2	-32,768	32,767
long int	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
signed long	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long int	8	0	18,446,744,073,709,551,615
float	4	-3.4e +/- 38 (~7 digits)	3.4e +/- 38 (~7 digits)
double	8	-1.7e +/- 308 (~15 digits)	1.7e +/- 308 (~15 digits)
long double	16	-3.4e +/- 4942 (~34 digits)	3.4e +/- 4942 (~34 digits)
wchar_t	2 or 4		1 wide character

```
#include <iostream>
using namespace std;
int main() {
    cout << "Size of bool : " << sizeof(bool) << endl;
    cout << "Size of char : " << sizeof(char) << endl;
    cout << "Size of unsigned char : " << sizeof(unsigned char) << endl;
    // Every other type goes here!
}
```

Primitive Types in C++

Declaring and Initializing Variables

Type	Bytes	Min Value	Max Value
bool	1	false	true
char	1	0	255
unsigned char			
signed char			
int			
unsigned int			
signed int			
short int			
unsigned short			
signed short			
long int			
signed long			
unsigned long			
float			
double			
long double			
wchar_t			

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

int main() {
    bool isHappy = true;
    char teach = '!';
    int ultimate_answer = 42;
    double earth_mass = 5.972e+27; // in grams
    string goodbye = "So long, folks!";
    cout << isHappy << ' '
         << teach << ' '
         << ultimate_answer << ' '
         << earth_mass << ' '
         << goodbye << endl;
}
```

```
ricegfp@pluto:~/dev/cpp/02$ g++ -w constants.cpp ; ./a.out
1 ! 42 5.972e+27 So long, folks!
ricegfp@pluto:~/dev/cpp/02$
```

These are the primitive types (along with string) that you will most often use by far!

Type Safety

- C++ compilers attempt to detect operations that violate restrictions that apply to a type
 - If detected during compile, an error is reported
 - If detected during runtime, an exception is thrown
 - If undetected, results in TO (That's Odd) problems

```
int main()
{
    int a = 20000;
    char c = a;
    int b = c;
    if (a != b)
        cout << "oops!: " << a << "!=" << b << '\n';
    else
        cout << "Wow! We have large characters\n";
}
```

a	20000	Range to 2,147,483,647
c	32	Range to 255
b	32	Range to 2,147,483,647

```
ricegf@pluto:~/dev/cpp/02$ ./a.out
oops!: 20000!=32
```

Note: 20,000 is 0x4E20
and 0x20 is 32

Types assign *some* semantic information to a variable – you must provide the rest!

Auto Typing

- “auto” is a valid type spec meaning “the type of the initializer” in C++ 11 as well as “the type of the return” in C++ 14.

```
#include <math.h>
auto answer() { return 42;}

int main() {
    auto x = 1;           // 1 is an int, so x is an int
    auto y = 'c';         // 'c' is a char, so y is a char
    auto d = 1.2;         // 1.2 is a double, so d is a double
    auto s = "Howdy";     // "Howdy" is a string literal of type const char[]
                        // so don't do that until you know what it means!
    auto sq = sqrt(2);    // sq is the right type for the result of sqrt(2)
    auto duh;             // error: no initializer for auto
}
```

```
ricegf@pluto:~/dev/cpp/201701/02$ g++ -std=c++11 test14.cpp
test14.cpp:2:13: error: 'answer' function uses 'auto' type specifier without trailing return type
    auto answer() { return 42;}
    ^
test14.cpp:2:13: note: deduced return type only available with -std=c++14 or -std=gnu++14
test14.cpp: In function 'int main()':
test14.cpp:11:8: error: declaration of 'auto duh' has no initializer
    auto duh;    // error: no initializer for auto
    ^
ricegf@pluto:~/dev/cpp/201701/02$ g++ -std=c++14 test14.cpp
test14.cpp: In function 'int main()':
test14.cpp:11:8: error: declaration of 'auto duh' has no initializer
    auto duh;    // error: no initializer for auto
    ^
ricegf@pluto:~/dev/cpp/201701/02$
```

-std=c++11

-std=c++14



Can We Make It Faster?

- Don't worry about "efficiency" unnecessarily
 - Most often, a programmer's time and the code's maintainability is more important than "efficiency"
 - Concentrate on correctness and simplicity of code
- C++ is derived from C, which is a systems programming language
 - C++'s built-in types map directly to computer main memory
 - A char is stored in a byte (8 bits – this predates Unicode and its 16 bits wide characters)
 - An int is stored in a word (typically 64 bits, except for micro-controllers)
 - A double fits in a floating-point register (typically 80 bits, which is why float is rarely used)
 - C++'s built-in operations map directly to machine instructions
 - An integer + is implemented by an integer add operation
 - An integer = is implemented by a simple copy operation
 - C++ provides direct access to most of the facilities provided by modern hardware
- C++ help users build safer, elegant, efficient new types and operations



Primitive Types vs Class Types

- Primitive types usually fit into a CPU register
 - bool, int, char, wchar_t and their variants fit an integer register
 - float and its variants fit a floating point register
- Class types are arbitrarily complex
 - May include complex data structures – linked lists, database records, control information for a USB port
 - May include code to manage the data structures – add items to and remove items from the list, validate the database record, or send and receive data via USB

In C++, we define new types *and the operations that manipulate them*
e.g., a complex number type with +, *, and << operators et. al.

Object-Oriented (\pm) Terminology

Class

A template, like a cookie cutter, used to create things ("objects").

Encapsulation

Bundling data and related code ("Methods") into a restricted container ("class")

Operator

A symbol that modifies an object, or generates a new "object" from other "objects".



Variable

Memory assigned to hold one or more things ("objects" or primitives).

Object

A thing created ("instantiated") from a "class". Sometimes also called an "**Instance**".

Object-Oriented Terminology

The Boring List Version

- **Encapsulation** – bundling data and code into a restricted container
- **Class** – a template encapsulating data and code that manipulates it
- **Method** – a function that manipulates data in a class
- **Instance** – an encapsulated bundle of data and code
- **Object** – an instance of a class containing a set of encapsulated data and associated methods
- **Variable** – a block of memory associated with a symbolic name that contains an object or a primitive data value
 - Variables within a class are sometimes called “fields”
- **Operator** – a short string representing a mathematical, logical, or machine control action



These are the actual definitions you'll need to know for the first exam.



Writing Simple C++ Classes

- A class consists of data and (optionally) code
 - The data is the information managed
 - The code (called methods) manipulates the data
- C++ provides 3 basic types of classes
 - **Enum**: Just a list of words assigned an integer each
 - **Struct**: Just a list of data declarations with no supporting code*
 - **Class**: Both data and supporting code

* In its usual usage. In practice, a C++ struct is *almost* identical to a C++ class. We'll cover the details soon.

Defining an Enum

- An enumeration consists of
 - The keyword “enum” and an (optional) name
 - The first enumerated value with (optional) starting integer
 - The remaining enumerated values

```
enum Color {green, yellow, red};

int main() {
    cout << "Green is " << green <<
        ", yellow is " << yellow <<
        ", and red is " << red << "." << endl;
    Color stop = red;
}
```

```
ricegf@pluto:~/dev/cpp/02$ ./a.out
Green is 0, yellow is 1, and red is 2.
```


Defining a Struct

- An struct consists of
 - The keyword “struct” and an (optional) name
 - A set of type and variable name declarations

```
enum Color {green, yellow, red};
```

```
struct Traffic_light {  
    bool operating = true;  
    Color light = red;  
};
```

```
int main() {  
    Traffic_light Cooper_and_UTA;  
    cout << "Traffic light is " << (Cooper_and_UTA.operating ? "working" : "off") << endl;  
    cout << "Traffic light is " << (Cooper_and_UTA.light == red ? "red" :  
                                     (Cooper_and_UTA.light == yellow ? "yellow" :  
                                     "green")) << endl;  
  
    Cooper_and_UTA.light = green;  
    cout << "Traffic light is " << (Cooper_and_UTA.light == red ? "red" :  
                                     (Cooper_and_UTA.light == yellow ? "yellow" :  
                                     "green")) << endl;  
}
```

```
riceg@pluto:~/dev/cpp/02$ ./a.out  
Traffic light is working  
Traffic light is red  
Traffic light is green
```



Visibility

- The definitions in an enum and struct are all public by default – visible to code outside the definition
 - This raises the “global variable” problem, to wit: “WHO CHANGED MY VARIABLE?!?”
- Once we add code to manage this data, we may want to make (some of) our data *private* – visible only to code included in our class (methods)
 - Who changed my variable? “Round up the usual suspects” - the *methods*

Defining a Class

- A class consists of
 - The keyword “class” and an (optional) name
 - A set of type and variable name declarations
 - A set of constructors and methods

```
enum Color {green, yellow, red};  
class Traffic_light {  
    bool operating = false;  
    Color light = red;  
public:  
    bool is_operating() {return operating;}  
    void turn_on() {operating = true;}  
    void turn_off() {operating = false;}  
    Color this_color() {return light;}  
    Color next_color() {  
        light = (light == green) ? yellow :  
                (light == yellow) ? red : green;  
        return light;  
    }  
    string string_color() {  
        return (light == green) ? "green" :  
                (light == yellow) ? "yellow" : "red";  
    }  
};
```

```
int main() {  
    Traffic_light Cooper_and_UTA;  
    Cooper_and_UTA.turn_on();  
    for (int i = 0; i < 10; ++i) {  
        Cooper_and_UTA.next_color();  
        cout << "Traffic light is " <<  
            Cooper_and_UTA.string_color()  
            << endl;  
    }  
}
```

```
ricegfp@pluto:~/dev/cpp/02$ ./a.out  
Traffic light is green  
Traffic light is yellow  
Traffic light is red  
Traffic light is green  
Traffic light is yellow  
Traffic light is red  
Traffic light is green  
Traffic light is yellow  
Traffic light is red  
Traffic light is green
```



A Class is a *Type*

- Some types are *primitive* – they represent data that is directly manipulated in the hardware
 - int and its variants, e.g., short and long
 - double and its cousin float
 - char
 - bool
- The C++ Standard Library and Standard Template Library (STL) define other types as classes* (or sometimes enums)
 - string is a class from the standard library
 - vector is a class from the STL
- The classes you write are just as much a type as the above!

* A struct is just another name for a class – they are (almost) identical in C++. We'll consistently use “class” instead of “struct” in CSE1325

Using a Class

- Your code defines how your class type is used

```
enum Color {green, yellow, red};
class Traffic_light {
    bool operating = false;
    Color light = red;
public:
    bool is_operating() {return operating;}
    void turn_on() {operating = true;}
    void turn_off() {operating = false;}
    Color this_color() {return light;}
    Color next_color() {
        light = (light == green) ? yellow :
                (light == yellow) ? red : green;
        return light;
    }
    string string_color() {
        return (light == green) ? "green" :
                (light == yellow) ? "yellow" : "red";
    }
};
```

```
int main() {
    Traffic_light Cooper_and_UTA;
    Cooper_and_UTA.turn_on();
    for (int i = 0; i < 10; ++i) {
        Cooper_and_UTA.next_color();
        cout << "Traffic light is " <<
            Cooper_and_UTA.string_color()
            << endl;
    }
}
```

```
riceg@pluto:~/dev/cpp/02$ ./a.out
Traffic light is green
Traffic light is yellow
Traffic light is red
Traffic light is green
Traffic light is yellow
Traffic light is red
Traffic light is green
Traffic light is yellow
Traffic light is red
Traffic light is green
```



Compiling Multiple Files

- Most C++ compilers run in 3 phases
 - Phase 1 – Preprocessing
 - Modify the source based on preprocessor directives, e.g., “#include <iostream>”
 - Phase 2 – Compilation
 - Translates your source code text into machine instructions, leaving placeholders for references to source code definitions in other files
 - Phase 3 – Linking
 - Replaces each placeholder with the actual address of definitions in other (compiled) files, producing an executable
- With gcc, all phases are run by the “g++” tool

Separate Compilation

- If class `Traffic_light` is in `traffic_light.cpp` and `main` is in `main.cpp`, they can be separately compiled
 - The “-c” flag for `g++` performs the first 2 phases (preprocessing and compilation) only
 - Without the “-c” flag, `g++` will also perform the 3rd phase (linking)
 - So we can compile each file separately, then link

```
ricegf@pluto:~/dev/cpp/201801/02/traffic$ ls
main.cpp  traffic_light.cpp
ricegf@pluto:~/dev/cpp/201801/02/traffic$ g++ -c -std=c++14 main.cpp
ricegf@pluto:~/dev/cpp/201801/02/traffic$ ls
main.cpp  main.o  traffic_light.cpp
ricegf@pluto:~/dev/cpp/201801/02/traffic$ g++ -c -std=c++14 traffic_light.cpp
ricegf@pluto:~/dev/cpp/201801/02/traffic$ ls
main.cpp  main.o  traffic_light.cpp  traffic_light.o
ricegf@pluto:~/dev/cpp/201801/02/traffic$ g++ -std=c++14 main.o traffic_light.o
ricegf@pluto:~/dev/cpp/201801/02/traffic$ ls
a.out  main.cpp  main.o  traffic_light.cpp  traffic_light.o
ricegf@pluto:~/dev/cpp/201801/02/traffic$
```



Rebuilding Multi-File Programs

- A source file must be recompiled if it changes
 - The compiler won't catch this problem – it will happily link the .o file that you generated *before* you made changes to your .cpp
 - Automation is essential, e.g., we could automatically recompile all .cpp files with a “bash script”
- Simply:
 - Put the compile commands into a text file
 - Tell the computer it's OK to execute that text file
 - Type “./” and the filename to execute the file
 - “./” means “in the current working directory (cwd)”

Creating a “run” bash Script

```
ricegf@pluto:~/dev/cpp/201708/02/traffic$ ls
main.cpp run traffic_light.cpp
ricegf@pluto:~/dev/cpp/201708/02/traffic$ cat run
g++ -std=c++11 -c traffic_light.cpp
g++ -std=c++11 -o main main.cpp
./main

ricegf@pluto:~/dev/cpp/201708/02/traffic$ chmod a+x run
ricegf@pluto:~/dev/cpp/201708/02/traffic$ ./run
Traffic light is green
Traffic light is yellow
Traffic light is red
Traffic light is green
Traffic light is yellow
Traffic light is red
Traffic light is green
Traffic light is yellow
Traffic light is red
Traffic light is green
ricegf@pluto:~/dev/cpp/201708/02/traffic$
ricegf@pluto:~/dev/cpp/201708/02/traffic$ ls
main main.cpp run traffic_light.cpp traffic_light.o
ricegf@pluto:~/dev/cpp/201708/02/traffic$
ricegf@pluto:~/dev/cpp/201708/02/traffic$
```

“cat” concatenates (prints to STDOUT) the file contents

For the g++ compiler:

- c compiles to a .o but doesn't link

- o specifies the executable name

“./main” executes main from cwd

“chmod **a+x** run” marks run as **e**xecutable by **a**ll users

“./run” executes run from cwd

“ls” lists the files in the cwd – those in green have been marked as executable

traffic_light.o is the compiled version of traffic_light.cpp

* Yes, I *know* traffic_light.cpp should have a traffic_light.h, and that's the file we should #include in main.cpp. Baby steps, Bob, *baby steps*!



Efficient Multi-File Programs

- This works fine for small projects
- For large projects, recompiling every file can take a long time
 - So we only compile *changed* files
 - But as the number of files grows, it becomes much harder to remember which files changed since the last compile
- Happily, the operating system remembers when every file was last changed
 - So we could modify our script to check the “last modified” date of *every source file* and calculate the fewest command lines needed
 - How hard can it be? (ahem)

A Smarter Run Script

(tl;dr – don't read this!)

- The “date -r” prints the last modified date of a file
 - Appending “+%Y%m%d%H%M%S” formats the date as year, month, date, hour, minute, second (no spaces)
 - e.g., 20170831151358 (Aug 31, 2017 at 3:13:58 pm)
- “if” compares the time and then builds if needed
 - “touch” resets the last modified time to now
 - “\$(cmd)” replaces itself with the output of “cmd”

```
if [ $(date -r traffic_light.cpp +%Y%m%d%H%M%S) >
    $(date -r traffic_light.o   +%Y%m%d%H%M%S) ] ; then
    g++ -std=c++11 -c traffic_light.cpp # Compile the .cpp to a .o
    touch main.cpp                     # Ensure we also relink
fi
if [ $(date -r main.cpp +%Y%m%d%H%M%S) > $(date -r main +%Y%m%d%H%M%S) ] ; then
    g++ -std=c++11 -o main main.cpp    # Compile and link main.cpp
fi
./main                                # Run the program
```



Multi-File Programs

using make

- That's pretty complicated – isn't there an easier way to do the same thing?
- Yes - “make”!
 - “make” is the standard tool for building and installing C and C++ programs
 - “make” relies on a “Makefile”, a text file that contains a set of rules describing how to build a program
- Other options exist – but every serious C++ programmer should know how to use “make”
 - Traditionally, Linux source code can be built and installed using “./configure ; make ; make install”

Super-simple Makefiles

<http://www.cprogramming.com/tutorial/makefiles.html>

- The “Makefile” is the key to using “make”
 - This will become very important with bigger builds

```
# Super-simple Makefile example
foo:
    echo "Making foo"
```

Annotations:
← This is a comment (points to the first line)
← This is a “target” - “make foo” invokes this rule (points to the second line)
← These can be ANY bash commands (points to the third line)

This must be a TAB – spaces or other whitespace won’t work!

Typing “make foo” will invoke the rule, which will execute the “echo” bash command.

Simply typing “make” will invoke the first rule in the Makefile, so in this case, “make” is the same as “make foo”.

Invoking a rule that’s not in the Makefile results in the error message shown.

```
ricegf@pluto:~/dev/cpp/201708/02$ cat Makefile
# Super-simple Makefile example
foo:
    echo "Making foo"

ricegf@pluto:~/dev/cpp/201708/02$ make foo
echo "Making foo"
Making foo
ricegf@pluto:~/dev/cpp/201708/02$ make
echo "Making foo"
Making foo
ricegf@pluto:~/dev/cpp/201708/02$ make my_day
make: *** No rule to make target 'my_day'.  Stop.
ricegf@pluto:~/dev/cpp/201708/02$
```

Super-simple Makefiles

<http://www.cprogramming.com/tutorial/makefiles.html>

- Rules can invoke other rules

```
# Super-simple Makefile example ← This is a comment
foo: bar ← "foo" depends on (will invoke) "bar"
    echo "Making foo"
bar: ← We can also "make bar" directly
    echo "Making bar"
```

Note: A Makefile is NOT a conventional program with a list of steps to follow. It is a set of rules that "make" will follow based on your command.

```
ricegf@pluto:~/dev/cpp/201708/02$ cat Makefile
# Super-simple Makefile example
foo: bar
    echo "Making foo"
bar:
    echo "Making bar"

ricegf@pluto:~/dev/cpp/201708/02$ make bar
echo "Making bar"
Making bar
ricegf@pluto:~/dev/cpp/201708/02$ make foo
echo "Making bar"
Making bar
echo "Making foo"
Making foo
ricegf@pluto:~/dev/cpp/201708/02$
```


Smart Makefiles

<http://www.cprogramming.com/tutorial/makefiles.html>

- Rules can invoke other rules based on a comparison of “last modified” time stamps

```
# More interesting Makefile example
foo: foo.cpp bar.o ← “foo” is invoked if EITHER foo.cpp is newer than foo
    @echo "Making foo"      OR if bar.o is invoked (or both)
    @touch foo              # "create" a new foo
bar.o: bar.cpp ← “bar.o” is invoked if bar.cpp is newer than bar.o
    @echo "Making bar.o"
    @touch bar.o            # "create" a new bar.o
```

Study this example until you **UNDERSTAND** it!

- WHY did touching bar.cpp cause both bar.o and foo to be made?
- WHY did only touching foo.cpp cause only foo to be made?

```
ricegf@pluto:~/dev/cpp/201708/02/maker/makedemo$ cat Makefile
```

```
# More interesting Makefile example
foo: foo.cpp bar.o
    @echo "Making foo"
    @touch foo          # "create" a new foo
bar.o: bar.cpp
    @echo "Making bar.o"
    @touch bar.o        # "create" a new bar.o
```

```
ricegf@pluto:~/dev/cpp/201708/02/maker/makedemo$ ls
bar.cpp bar.o foo foo.cpp Makefile
ricegf@pluto:~/dev/cpp/201708/02/maker/makedemo$ make
make: `foo' is up to date.
```

```
ricegf@pluto:~/dev/cpp/201708/02/maker/makedemo$ touch bar.cpp
ricegf@pluto:~/dev/cpp/201708/02/maker/makedemo$ make
Making bar.o
Making foo
ricegf@pluto:~/dev/cpp/201708/02/maker/makedemo$ make
make: `foo' is up to date.
ricegf@pluto:~/dev/cpp/201708/02/maker/makedemo$ touch foo.cpp
ricegf@pluto:~/dev/cpp/201708/02/maker/makedemo$ make
Making foo
ricegf@pluto:~/dev/cpp/201708/02/maker/makedemo$
```

A More Realistic Makefile

<http://www.cprogramming.com/tutorial/makefiles.html>

- You'll see this pattern repeated all semester
 - If you don't understand it, ASK!

```
# Super-simple Makefile for Traffic_light
main: main.o Traffic_light.o ← This LINKS the application if anything changed
    g++ -std-c++11 -o main main.o Traffic_light.o
main.o: main.cpp ← This COMPILES main.cpp into main.o, if needed
    g++ -std-c++11 -c main.cpp
Traffic_light.o: Traffic_light.cpp ← Ditto for the Traffic_light class
    g++ -std-c++11 -c Traffic_light.cpp
clean: ← This forces everything to be rebuilt (why?)
    -rm -f *.o main
    ↗ The leading "-" means "ignore any errors"
```

In summary, a Makefile rule format is:

rule: dependencies

command(s)

#must start with a tab, NOT spaces, and may be any bash command

- Typing "make rule" (e.g., "make main") at the bash prompt invokes the specified rule IF any of its dependencies have changed
- Typing "make" invokes the *first* rule by default (above, is same as "make main").

* I know! Compiling Traffic_light.cpp separately doesn't actually do anything here, since it gets recompiled by inclusion in main.cpp anyway. But Makefiles can be hard to grasp, so we're starting as early as possible. You'll thank me later.

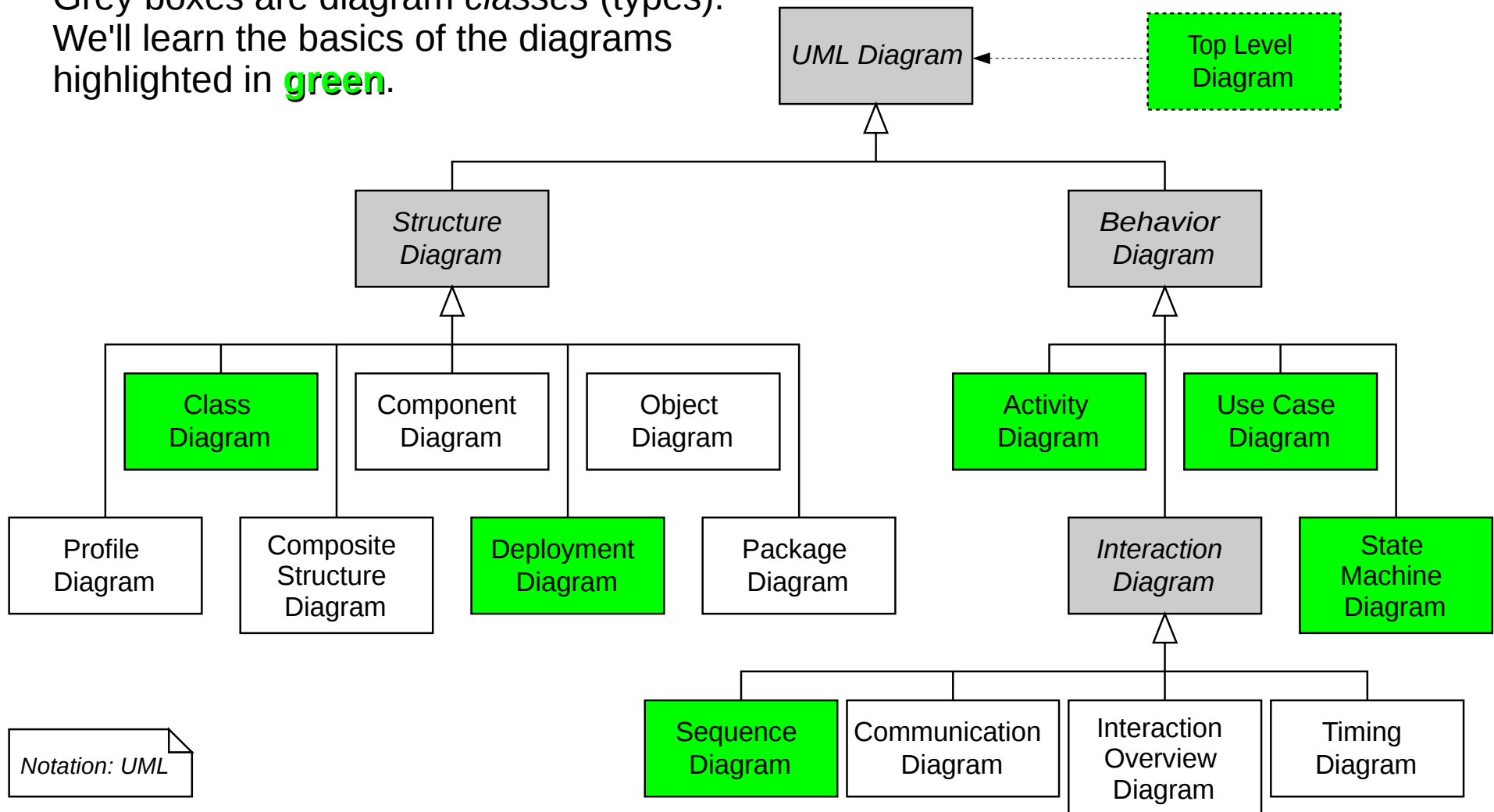
Unified Modeling Language (UML)

- The UML is the standard visual modeling language used to describe, specify, design, and document the structure and behavior of software systems, particularly OO
 - UML can **formally** specify a system so that code can be generated
 - UML can **informally** specify a system to enhance team communication
 - UML can **casually** represent a system to enhance your understanding during implementation and maintenance
- We'll cover just enough UML this semester to help with homework planning and project team communication!



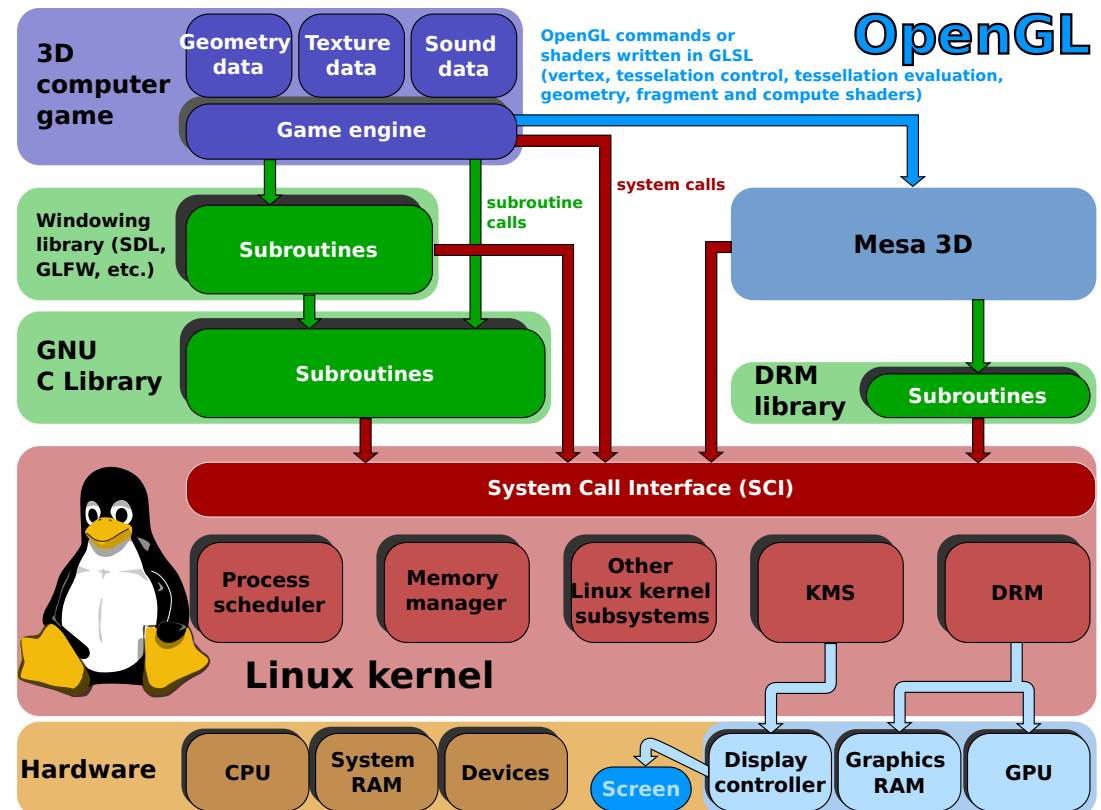
UML Includes a LOT of Diagrams

Grey boxes are diagram *classes* (types).
We'll learn the basics of the diagrams
highlighted in **green**.



Top Level Diagram (TLD) is NOT a UML Diagram

- But it's a useful concept anyway
 - No rules or constraints, just “PowerPoint Engineering” at its finest! 😊
 - May be used as an alternate index to UML diagrams or code repositories such as GitHub



“Intellectual Property”

- “To promote the Progress of Science and useful Arts, by securing for limited Times to Authors and Inventors the exclusive Right to their respective Writings and Discoveries” – US Constitution, Article 1, Section 8, ¶ 8
- Three primary types of interest to CSE
 - **Trademark** – symbol or name established by use as representing a company or product
 - **Patent** – exclusive right to make, use, or sell an invention
 - **Copyright** – exclusive right to print, publish, perform, execute, or record a creative work or its derivatives, and to authorize others to do the same

The materials available in this lecture are for informational purposes only and not for the purpose of providing legal advice. You should contact a competent attorney to obtain specific advice with respect to any particular issue or problem.

Trademark

- Trademarks avoid customer confusion
 - “Microsoft Windows” is a specific operating system
 - It is illegal to sell another OS product by that name
- Established simply by use, but better to register
 - Public notice that it's your trademark
 - Stronger court case against infringement
- Lasts as long as it is defended
 - Xerox used to be a trademark for copying papers

Patent

- Patents protect new or improved processes, machines, manufactured articles, and states of matter if they are:
 - Novel and non-obvious
 - Not yet published or in general use
 - Not previously patented
- Patents must be filed with the USPTO, and generally last for 20 years from date of application
 - “Patent pending” gives notice of a patent application

<http://www.uspto.gov/patents-getting-started/general-information-concerning-patents>

Copyright

- Copyright protects creative works
 - Copyright is automatic on creating the new work
 - Registering the copyright offers better protection
- Copyright* lasts the shorter of:
 - 70 years after the death of the last surviving author
 - 95 years after publication
 - 120 years after creation
- CSE who work for corporations usually sign a “work for hire” contract, assigning copyright to the employer
 - Even for non-work related software you write
 - You can – and probably should – negotiate limited exclusions

“for limited Times”?

<http://www.uspto.gov/learning-and-resources/ip-policy/copyright/copyright-basics>

* Works created prior to 1978 follow different rules

Types of Software Licenses

- If no license is specified, then “all rights reserved”
 - **Public Domain** - all ownership is disclaimed (SQLite)
 - **Permissive** (MIT, BSD, Apache) permits use, copying, distribution, and (usually with attribution) derivatives, even proprietary derivatives (BSD Unix, Apache)
 - **Protective** (GPL 2, 3, Lesser GPL, EPL) permits use, copying, distribution, and derivatives *with share-alike rules* (Linux, Gimp)
 - GPL 3 also includes important patent clauses
 - **Shareware** permits (sometimes limited) use, copying, and (usually) distribution (Irfanview, early WinZip releases)
 - **Proprietary** permits (often restricted) use (Windows, Photoshop)
 - **Trade Secret** typically restricts use to the copyright holder

Software License Significance

- Know the license that applies to software you use and its requirements
 - If you have questions, ask an expert
 - If you can't follow them, don't use that software
- Select and Document the license that applies to the code you author
 - If a “work for hire”, ask your manager
 - If an independent work, chose wisely
- Always follow copyright law – it's your job



git's GUI

- The git-all package in Ubuntu includes a native GUI
 - It's a blessing... and a curse
 - Launch using “git gui” and learn on your own, if you like
- Most C++ IDEs also support git, e.g.,
 - Code::Blocks offers the GitBlocks plug-in
 - Eclipse includes the Egit plug-in by default
 - Visual Studio 2013.1 and later support git directly
- git GUIs are sometimes helpful, particularly with more advanced git features (e.g., branch / merge)
 - But you need to know the command line interface
 - So for now, use the command line (almost) exclusively



What We Learned Today

- (Review) C++ names, operators, and types
- Creating our own types with enum and class
 - (and struct, but anything struct can do, class can do better ☺)
- Three phases of (most) C++ compilers
 - Preprocessing, compiling, and linking
- Building multiple files independently
 - Preprocessing and compiling each, then link together
 - Using a bash script or (better) Makefile for this
- The (non-)UML Top Level Diagram (TLD)
 - It's an index to your diagrams, and PowerPoint engineering
- Intellectual property basics
 - Trademarks, patents, and copyrights
 - Licenses, and how to choose 'em and use 'em

Quick Review

- To search your local directory in addition to system library directories with `#include`, use `__file.h__` instead of `__file.h__`.
- `____` sends a stream to `STDOUT`, while `_____` reads a stream from `STDIN`.
- Which of these is **not** a valid C++ operator or comparator?
a. `!=` b. `+=` c. `)(` d. `<>` e. `-` f. `/=` g. `==`
- Which of these is **not** a valid C++ name?
a. `$_` b. `profit_2016` c. `_secret_` d. `HeLIOWOrLd` e. `while`
- Define “primitive type”.
List the 4 most commonly used primitive types in C++.
- **True or False:** In C++, a variable may hold any type as long as the operators and comparators applied to it by the code are defined.
- C++ allows the programmer to define their own types using which mechanism(s)?
a. Type extensions b. Classes c. C plug-ins d. Enums e. Structs
- **True or False:** The integer assigned to each enum value can be explicitly defined by the programmer or left to the compiler.



Quick Review

- The bundling of data and code into a restricted container is called _____.
- A template that encapsulates data and the code that manipulates it is called a(n) _____.
- A function that manipulates data in a class is called a(n) _____.
- An instance of a class containing a set of encapsulated data and methods is called an _____.
- By default, are the definitions inside each of these class types public or private?
Enum Struct Class
- Why should most non-static variables in a class be private?
- “make” follows the rules in a file in the current working directory named _____. Traditionally, the first character is an (1) m (2) M (3) _.
- Why is a Makefile superior to scripts for building C++ programs?
- Which of the following usually builds and installs a program from source?
(1) make install (2) setup.exe (3) ./configure ; make ; make install



Quick Review

- Expand the UML acronym.
- James Rumbaugh, Grady Booch, and Ivar Jacobson, the “three amigos”, are most famously known for what?
- Which types (classes) of diagrams comprise the UML?
a. Structure b. Top Level c. Behavior d. Interaction e. Data Flow
- Although not a UML diagram, why is a Top Level Diagram (TLD) often included with UML models?
- _____ and _____ intellectual property types are automatic (though registration is recommended). _____ are granted only via application.
- A _____ software license enforces share-alike rules for derivatives.
- The _____ and _____ software license types allow source code to be reused in a proprietary product (though the latter sometimes requires attribution).

Homework #1 Questions?

- **Build and Run “Hello World”** *using your own name*
 - **Bonus:** Ask the user for a name and use that
 - **Extreme Bonus:** Determine current user's name *without asking* and use that
- Deliver code and screen shots to Blackboard by **Thursday, 25 January at 8 am**
- **Details are provided on Blackboard**

(HINT: *These are the easiest points you'll earn all semester.* Don't blow it!)

You learn to program by programming!

NOW is the time to start. A week before the first homework will be too late!



For Next Class

- (Optional) Read Chapter 3 in Stroustrup
 - Do the Drills!
- Code is attached to these slides in Blackboard
 - Review and explore as much as possible
- Skim Chapter 4 in Stroustrup for next lecture – we'll cover
 - Computation
 - UML Class Diagram
 - Singleton Pattern

You learn to program by programming!

NOW is the time to start. A week before the first homework will be too late!

