CSE 1325: Object-Oriented Programming

Lecture 03 – Chapter 04

Computation, Patterns, and UML Class Diagrams

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

This work is licensed under a Creative Commons Attribution 4.0 International License.

Lecture #2

Quick Review

- To search your local directory in addition to system library directories with #include, use "file.h" instead of <file.h>.
- **cout** sends a stream to STDOUT, while **cin** 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. HeLlOwOrLd e. **while**
- Define "primitive type". Type typically handled directly by the underlying hardware. List the 4 most commonly used primitive types in C++. **int, double, bool, char**
- **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. **False**
- 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. **True**

* A leading underscore is legal but not recommended

Lecture #2
Quick Review

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

Lecture #2
Quick Review

- Expand the UML acronym. **Unified Modeling Language**
- James Rumbaugh, Grady Booch, and Ivar Jacobson, the “three amigos”, are most famously known for what? **Compromising to define the UML**
- 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? **It provides an overview of system operation, and may be used as a hyperlinked index to UML diagrams**
- **Copyright** and **Trademark** intellectual property types are automatic (though registration is recommended). **Patents** are granted only via application.
- A **Protective** software license enforces share-alike rules for derivatives.
- The **Public Domain** and **Permissive** software license types allow source code to be reused in a proprietary product (though the latter sometimes requires attribution).

Today's Topics

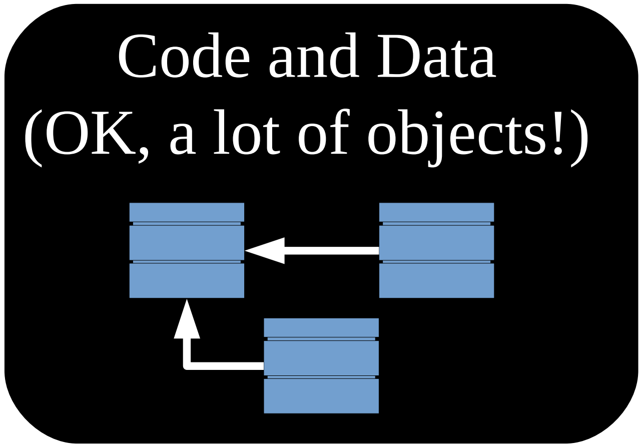
- **Iteration** (loops)
  - Make a breakfast sandwich each for 100 people
- **Expressions**
  - Gather 3 eggs plus 2 eggs per person
- **Decomposition** (functions)
  - To **make a sandwich**: cook eggs, mix sauce, assemble sandwich, and put on plate
  - To **cook eggs**: mix eggs, milk, spices and fry
  - To **fry**: place in pan over medium heat; stir
- **Selection** (conditionals)
  - If the eggs burn, reduce the stove temp
- **Data sequences** (vectors)
  - Line up plates of breakfast sandwiches on the table, each labeled with the name of the guest
- **Intro to UML Class Diagrams**
- **Intro to Software Design Patterns**
  - The Singleton Pattern
- **GitHub Basics**

Computation

(A Whirlwind Review of Programming)

Input Data
“gozinas”

- Keyboard
- Mouse / Touch
- Camera
- Other devices
- Files
- Network
- Other programs



- Math
- Logic
- Transforms

Algorithms!

Output Data
“gozoutas”

- Screens
- Files
- Printers
- Other devices
- Files
- Network
- Other programs

Computation

- Our job is to express computations
  - Correctly
  - Simply
  - Efficiently
- One tool is called Decomposition
  - Divide big computations into many little ones
- Another tool is Abstraction
  - Provide a higher-level concept that hides detail
- Organization of data is often the key to good code
  - Input/output formats
  - Protocols
  - Data structures

You don’t get good code just by writing a lot of statements

C++ Language Features

- Each feature exists to express a fundamental idea
  - **+** : addition
  - ***** : multiplication
  - **if (*expression*) *statement* else *statement* ;** selection
  - **while (*expression*) *statement* ;** iteration
  - **f(x);** function/operation
  - …
- We combine language features to create programs

Expressions

- An expression is “a sequence of operators and operands that specifies a computation”
  - Mathematical rules of precedence apply: 3+5*4 is 20
  - Parentheses are better! 3+(5*4) is more clearly 20
  - Break absurdly complex expressions up into separate expressions, or on separate lines
- Choose meaningful variable and method names
  - account_balance is better than ab
  - i and j are counters for historical reasons (why?)

```
// compute area:
int length = 20;      // the simplest expression: a literal (here, 20)
                     // (here used to initialize a variable)
int width = 40;
int area = length*width;  // a multiplication
int average = (length+width)/2;   // addition and division
```

Naming Conventions

- C++ has no standard naming convention. Every project has their own.
- Any convention will give you full credit, but inconsistent naming may result in deducted points. Pick a convention and stick with it.
- Here's what I typically use with C++:
  - Names are snake case, e.g., underscore word separation, for all names (my_name)
  - Objects, methods, and functions are NOT capitalized (graders.sort(), i, grade)
  - Bools are yes / no questions (database.is_running())
  - Classes and other types are capitalized (Coordinate_system)
  - Constants and preprocessor variables are shouted (NUM_WORDS, GLOBALS_H)
  - Opening brace is in-line with the scope identifier (while(true) {). Closing brace is in-line for a single line block (if (a == 5) {return 42;}) and on its own line otherwise.
  - Private / protected fields start with an underscore (_x, _counter).*
- Other options
  - Dr. Stroustrop's preferences http://www.stroustrup.com/bs_faq2.html
  - Gnu (gcc) conventions https://www.gnu.org/prep/standards/standards.html and https://gcc.gnu.org/codingconventions.html

* This is considered bad form by serious C++ professionals, but I'm not one of those :-)

Expressions

- Expressions are built from operators and operands
  - Operands specify the data for the operators to work with
  - Operators specify what is to be done
- Boolean type: **bool** (**true** and **false**)
  - Equality operators: **= =** (equal), **!=** (not equal)
  - Logical operators: **&&** (and), **||** (or), **!** (not)
  - Relational operators: **<** (less than), **>** (greater than), **<=**, **>=**
- Character type: **char** (e.g., **'a'**, **'7'**, and **'@'**)
- Integer types: **int (and short , long)** (e.g., 0, -117, 1024)
  - arithmetic operators: **+, -, *, /, %** (remainder)
- Floating-point types: **double (and float)** (e.g., 12.45 and 1.234e3)
  - arithmetic operators: **+, -, *, /**

Concise Operators

- For many binary operators, there are (roughly) equivalent more concise operators
  - For example

a += c	means		**a = a+c**
a *= scale	means		**a = a*scale**
++a	means		**a += 1**
		or	**a = a+1**

  - “Concise operators” are generally better to use (clearer, express an idea more directly)

Statements

- A statement is
  - an expression terminated with a semicolon, or
  - a declaration, or
  - a “control statement” that determines the flow of control

- For example
  - **a = b;**
  - **double d2 = 2.5;**
  - **if (x == 2) y = 4;**
  - **while (cin >> number) numbers.push_back(number);**
  - **int average = (length+width)/2;**
  - **return x;**

Selection

- Sometimes we must select between alternatives
- For example, suppose we want to identify the larger of two values. We can do this with an **if** statement

```
if (a<b)      // Note:  No semicolon here
   max = b;
else          // Note:  No semicolon here
   max = a;
```

- The syntax is

```
if (condition)
   statement-1      // if the condition is true, do statement-1
else
   statement-2      // if not, do statement-2
```

- An equivalent statement would be

```
max = (a<b) ? b : a;
```

Iteration (while loop)

- What it takes
  - A loop variable (control variable); here: **i**
  - Initialize the control variable; here: **int i = 0**
  - A termination criterion; here: if **i<100** is false, terminate
  - Increment the control variable; here: **++i**
  - Something to do for each iteration; here: **cout << …**

```
int i = 0;
while (i<100)  {
     cout << i << '\t' << square(i) << '\n';
     ++i ;    // increment i
}
```

Iteration (for loop)

- Another iteration form: the **for** loop
- You can collect all the control information in one place, at the top, where it's easy to see

```
for (int i = 0; I < 100; ++i) {
    cout << i << '\t' << square(i) << '\n';
}
```

becomes →

```
int i = 0;
if i < 100...
cout << i << ...
++i  // i is 1
if i < 100...
cout << i << ...
++i  // i is 2
if i < 100...
cout << i << ...
++i  // i is 3
if i < 100...
cout << i << ...
++i  // i is 4
if i < 100...
...
... repeat until...
...
++i  // i is 100
if i < 100...  // it's not
// exit loop
```

That is,

for (initialize**;** condition **;** increment **)**

controlled statement

Question: what is **square(i)**?

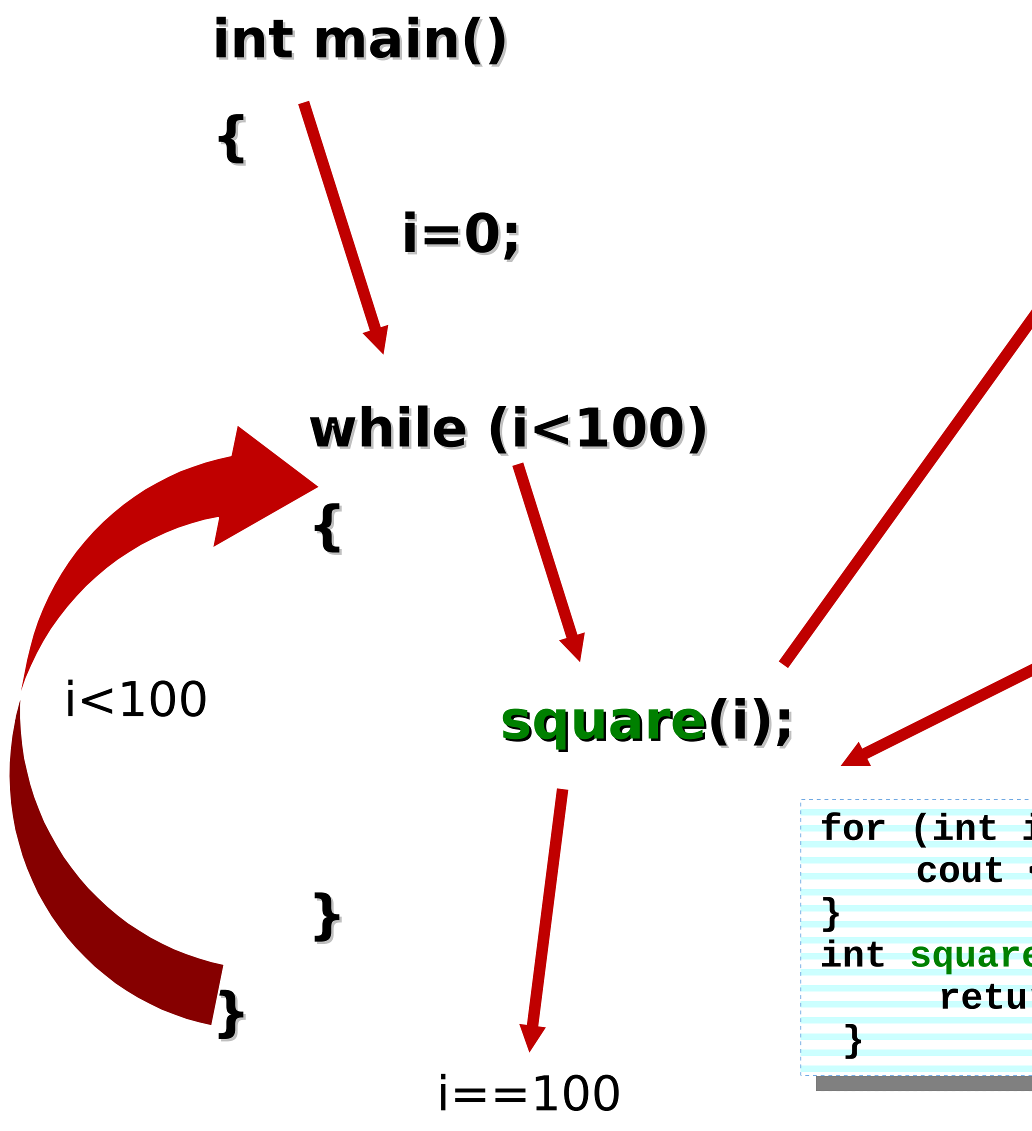
Functions

- **square(i) is a *function call*** *(or method call)*
  - A call of the function **square()**

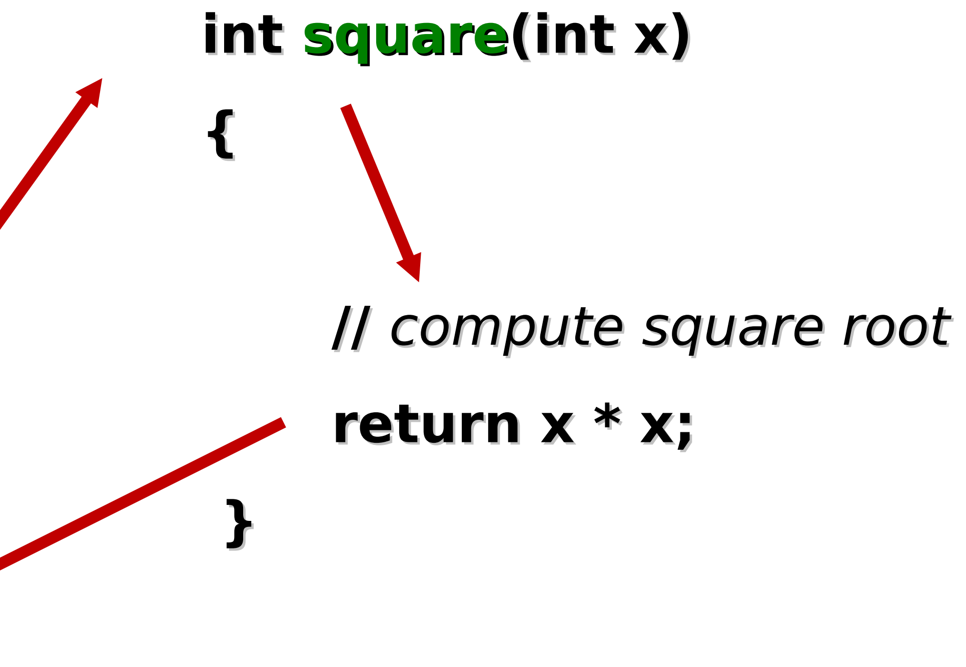
```
int square(int x) {
      return x*x;
}
```

  - We define a function when we want to separate a computation because it
    - is logically separate
    - makes the program text clearer (by naming the computation)
    - is useful in more than one place in our program
    - eases testing, distribution of labor, and maintenance

Control Flow – Another View



```
int main()
{
        i=0;

    while (i<100)
    {

              square(i);


    }
}
```



```
int square(int x)
{

      // compute square root
      return x * x;
 }
```

```
for (int i = 0; i<100; ++i) {
    cout << i << '\t' << square(i) << '\n';
}
int square(int x) {
     return x*x;
 }
```

Another Example Function

- Earlier we looked at code to find the larger of two values. Here is a function that compares the two values and returns the larger value.

```
int max(int a, int b) {  // this function takes 2 parameters
    if (a<b)
        return b;
    else
        return a;
}
```

```
int x = max(7, 9);        // x becomes 9
int y = max(19, -27);     // y becomes 19
int z = max(20, 20);      // z becomes 20
```

- A shorter version

```
int max(int a, int b) {return (a<b) ? b : a;}
```

Methods

- A method is a function within a class scope, which (unlike a function) has access to its private variables

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

class Complex_number {
  public:
    Complex_number(double re, double im) : _re{re}, _im{im} { }
    string to_string() {
      return std::to_string(_re) + "+" +
             std::to_string(_im) + 'i';
    }
  private:
    double _re, _im;
};
```

- A method* can only be called on an *instance* of the class

```
int main() {
  Complex_number c{3,5};
  cout << c.to_string() << endl;
}
```

* Technically a *non-static* method. We'll get to this distinction later today.

Constructors

- **A constructor is <u>not</u> a method,** but is invoked when a variable of the type is declared to *construct* it

We usually start with the *public* declarations first, including the constructors, as this is the *interface* to the class.

Save the private declarations for the end.

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

class Complex_number {
  public:
    Complex_number(double re, double im) : _re{re}, _im{im} { }
    string to_string() {
      return std::to_string(_re) + "+" +
             std::to_string(_im) + 'i';
    }
  private:
    double _re, _im;
};
```

The constructor has the same name as the class and *no return type*. It is otherwise very similar to a method. If you don't specify any constructors, a default will be provided.

When defining an instance of the class, its constructor is invoked like calling a function but using curly braces.*

```
int main() {
  Complex_number c{3,5};
  cout << c.to_string() << endl;
}
```

If no curly braces are provided, the default constructor with no parameters is automatically invoked if available, otherwise an error is thrown.

* Parentheses often work, too, for historical reasons. But don't use parentheses.

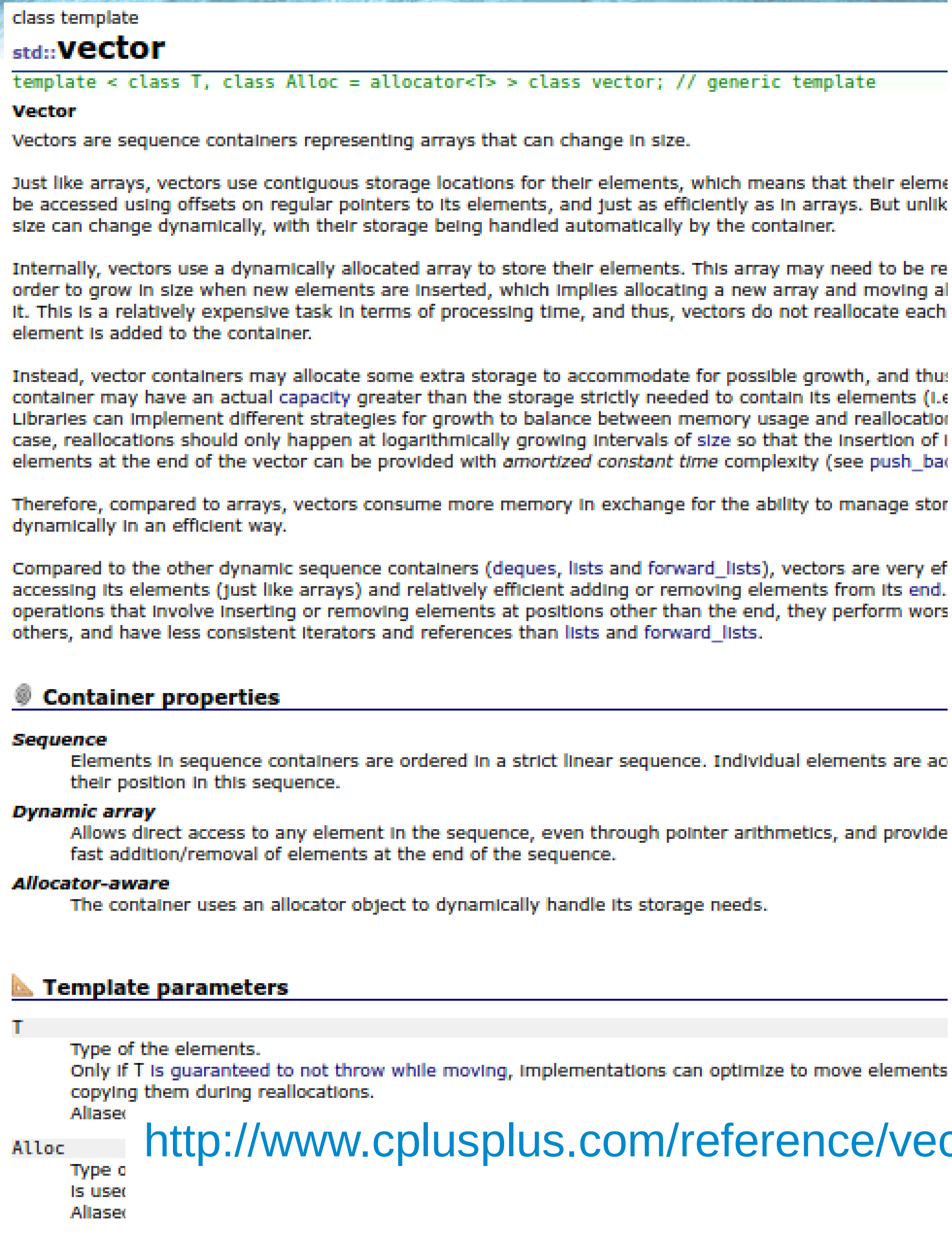
Vectors

- To do just about anything of interest, we need a collection of data to work on. We can store this data in a **vector**.
  - The vector class is defined in <vector>, which you must #include.
  - The type of data managed by a vector must be specified when a vector object is created, and it can manage only that type*
  - The vector class includes many methods:
    - Append data to the vector: push_back().
    - Determine the number of elements appended to the vector: size()
    - Find the beginning and end of the vector: begin(), end()
    - Accessing any arbitrary element: operator[ ]
    - And many, many more!

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

* Vector is a “template”. We’ll discuss how to write your own templates (much) later.

Vector Docs



class template

std::vector

```
template < class T, class Alloc = allocator<T> > class vector; // generic template
```

Vector

Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their eleme
be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlik
size can change dynamically, with their storage being handled automatically by the container.

Internally, vectors use a dynamically allocated array to store their elements. This array may need to be re
order to grow in size when new elements are inserted, which implies allocating a new array and moving al
it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each
element is added to the container.

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus
container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e
Libraries can implement different strategies for growth to balance between memory usage and reallocatio
case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of i
elements at the end of the vector can be provided with *amortized constant time* complexity (see push_bac

Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage stor
dynamically in an efficient way.

Compared to the other dynamic sequence containers (deques, lists and forward_lists), vectors are very ef
accessing its elements (just like arrays) and relatively efficient adding or removing elements from its end.
operations that involve inserting or removing elements at positions other than the end, they perform wors
others, and have less consistent iterators and references than lists and forward_lists.

Container properties

Sequence
Elements in sequence containers are ordered in a strict linear sequence. Individual elements are ac
their position in this sequence.

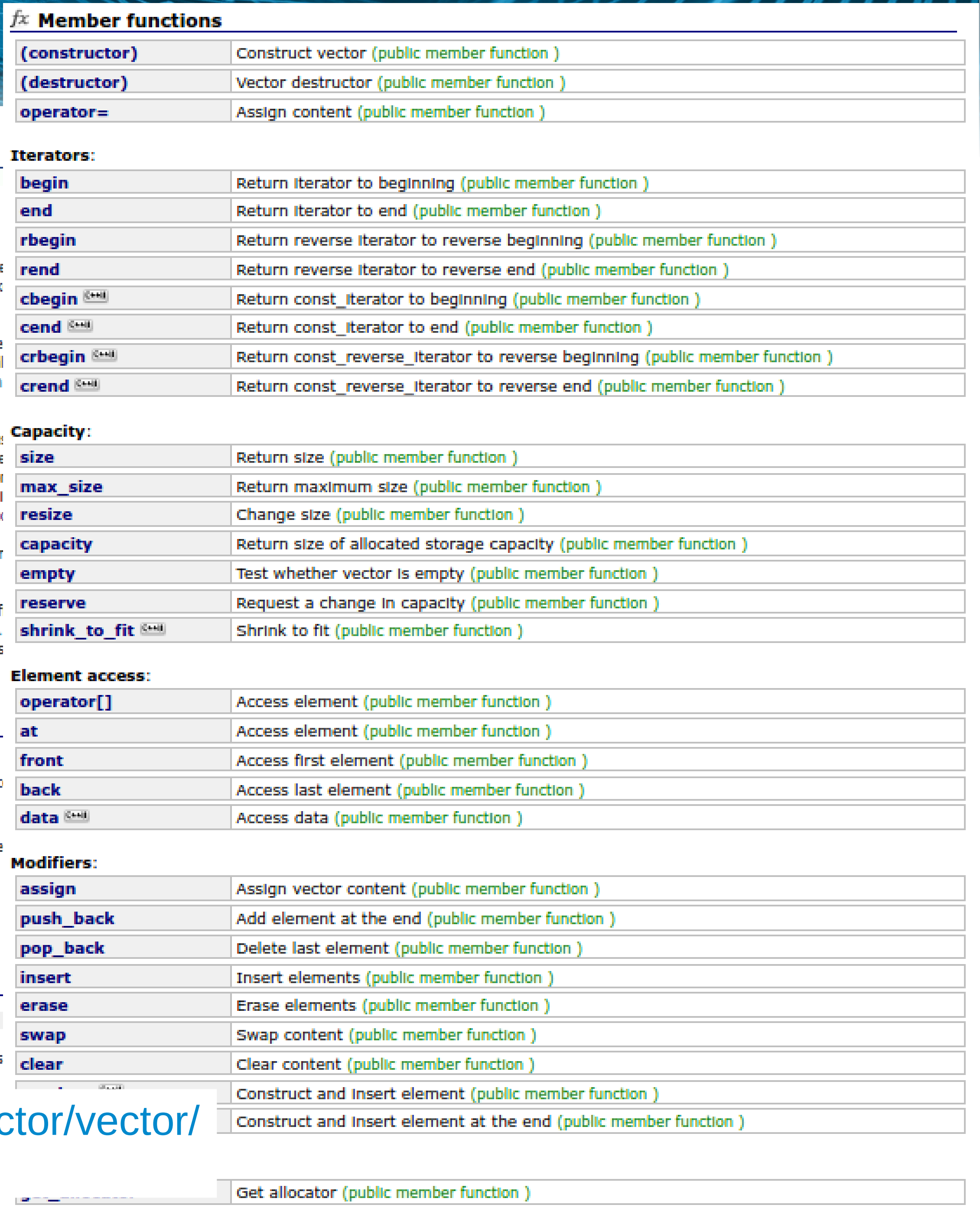
Dynamic array
Allows direct access to any element in the sequence, even through pointer arithmetics, and provide
fast addition/removal of elements at the end of the sequence.

Allocator-aware
The container uses an allocator object to dynamically handle its storage needs.

Template parameters

T
Type of the elements.
Only if T is guaranteed to not throw while moving, implementations can optimize to move elements
copying them during reallocations.
Aliasec

Alloc
Type c
is usec
Aliasec



Member functions

(constructor)	Construct vector (public member function)
(destructor)	Vector destructor (public member function)
operator=	Assign content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)

Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)
shrink_to_fit C++11	Shrink to fit (public member function)

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data C++11	Access data (public member function)

Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
[illegible]	Construct and insert element (public member function)
[illegible]	Construct and insert element at the end (public member function)
[illegible]	Get allocator (public member function)

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

Vector Example

- Here's an example that collects doubles from STDIN and stores them in a vector
  - Note that we needn't know in advance how many doubles we'll need to store – one major advantage of vectors over C's arrays

```
// read some temperatures into a vector:

int main()
{
    vector<double> temps;       // declare a vector of type double to store
                                // temperatures – like 62.4

    double temp;                // a variable for a single temperature value

    while (cin>>temp)           // cin reads a value and stores it in temp
        temps.push_back(temp);  // store the value of temp in the vector

     // … do something with the temperatures
}

// cin>>temp  will return true until we reach the end of file or encounter
// something that isn't a double: like the word "end".
// We'll cover how to learn more about the state of the input stream soon.
```

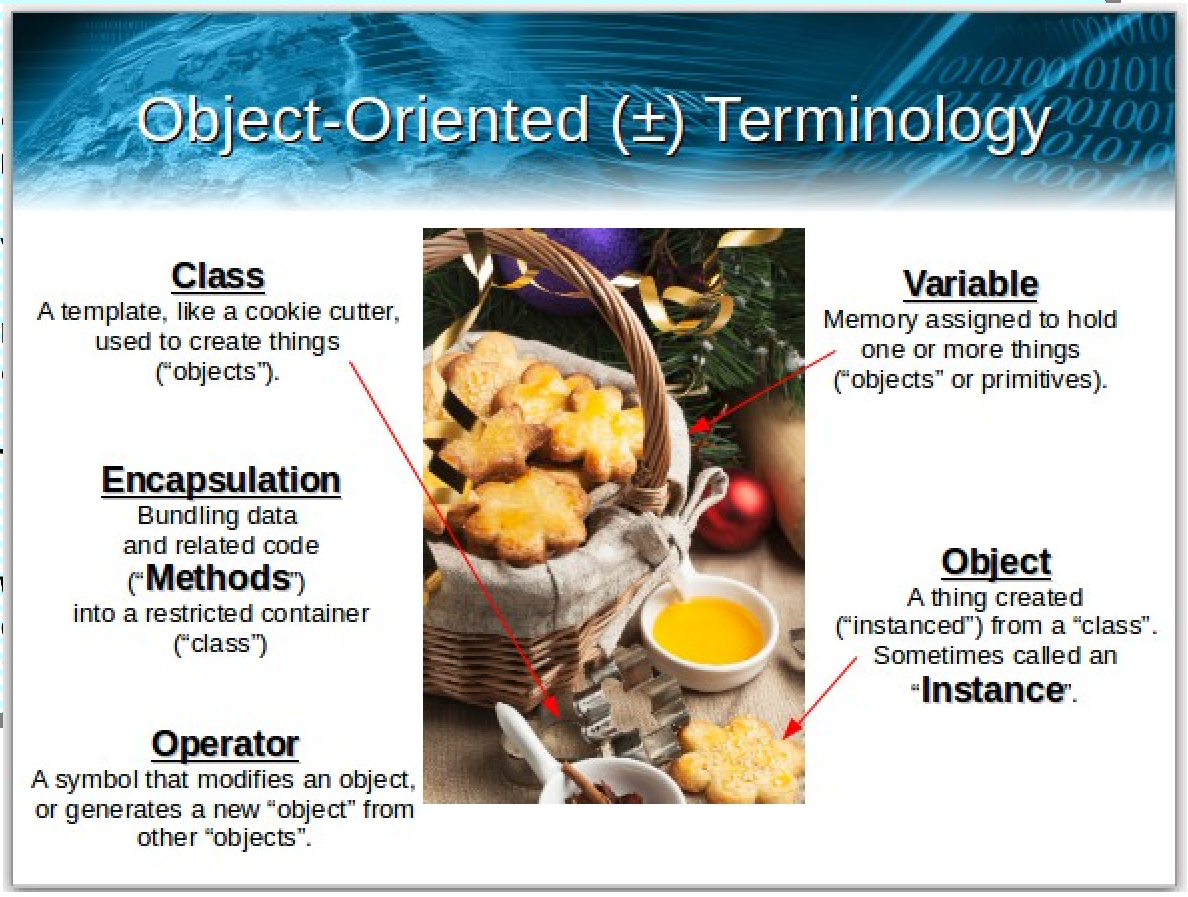
Vocabulary Review

- vector is a *template* class, which means we must specify a type (inside the < >) when we instance an object of it.

class　　　object

```
// read some temperatures into a vector:

int main()
{
    vector<double> temps;          // de
                                   // te

    double temp;                   // a 

    while (cin>>temp)              // ci
        temps.push_back(temp);     // st

     // … do something with the temper
}

// cin>>temp  will return true until 
// something that isn't a double: lik
// We'll cover how to learn more abou
```

method



Object-Oriented (±) 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".

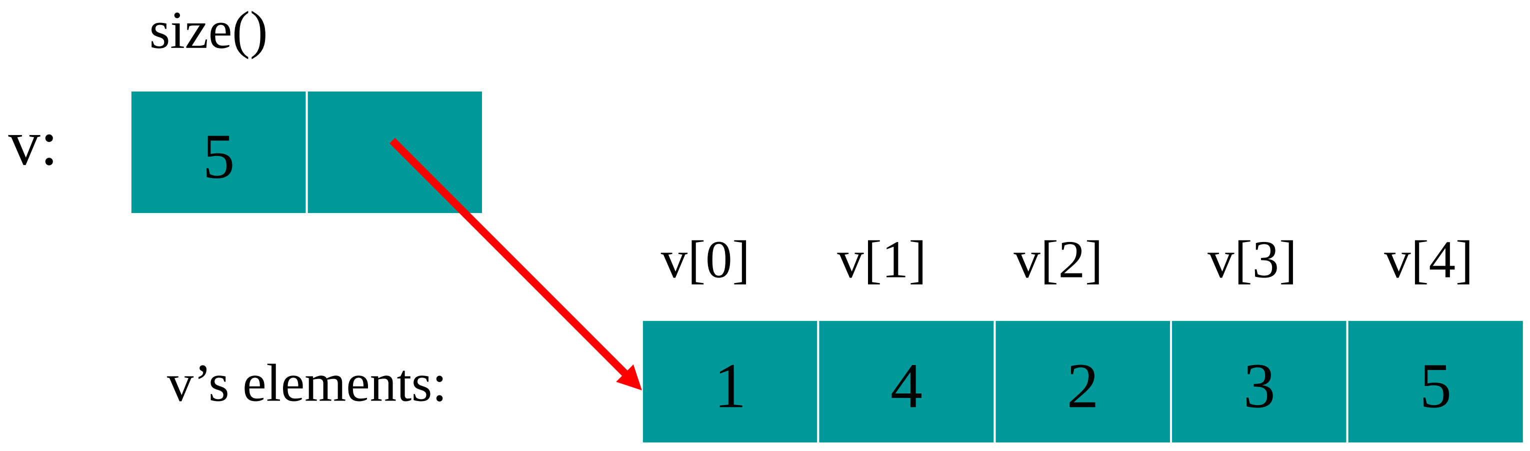
Variable
Memory assigned to hold one or more things ("objects" or primitives).

Object
A thing created ("instanced") from a "class". Sometimes called an "**Instance**".

Vector Mental Model

- Vector is the most useful standard library data type
  - A **vector<T>** object holds a sequence of values of type **T.** We call the vector class a “template”, because it can be instanced for any type.
  - Think of a vector this way

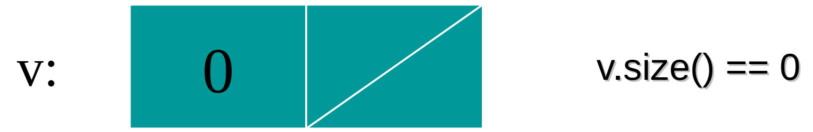
    A vector object named **v** of type int contains 5 elements: {1, 4, 2, 3, 5}:



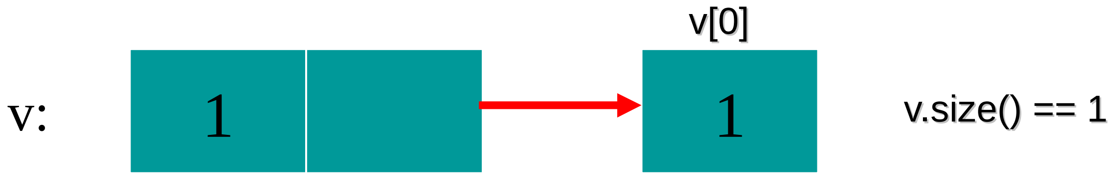
We can add elements to and remove elements from our vector as needed!

Vector Mental Model

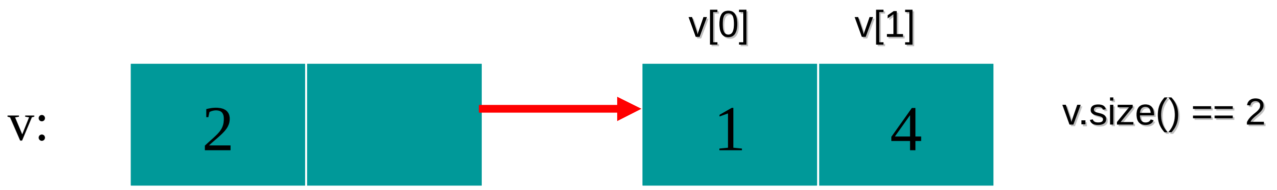
vector<int> v; // start off empty



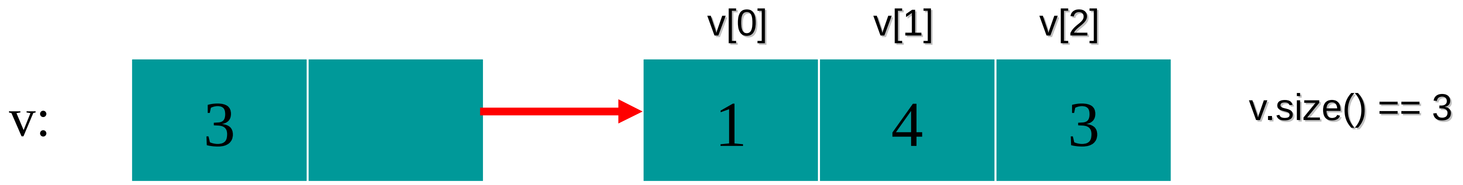
v.push_back(1); // add an element with the value **1**



v.push_back(4); // add an element with the value **4** at end ("the back")

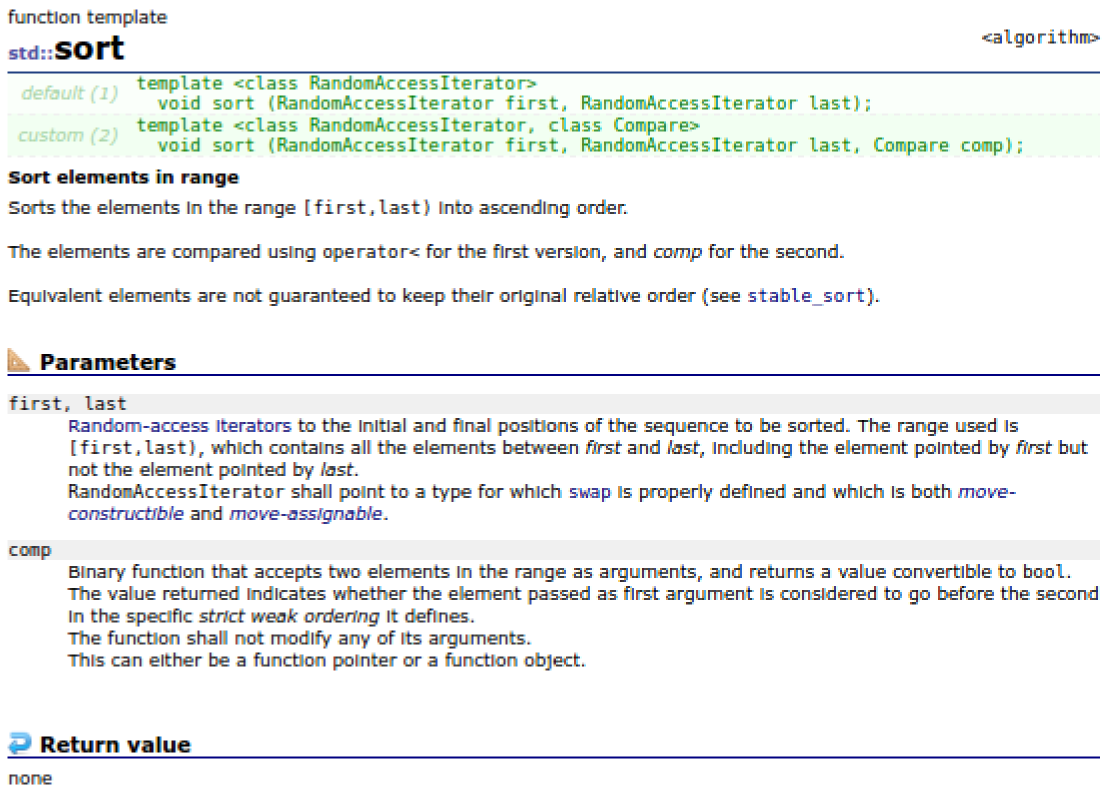


v.push_back(3); // add an element with the value **3** at end ("the back")



Many Library Functions Manipulate Vectors*

- Take “sort” for instance
  - Reorders the vector in place from smallest to largest



function template
<algorithm>

std::sort

default (1)	`template <class RandomAccessIterator>`
	`  void sort (RandomAccessIterator first, RandomAccessIterator last);`

custom (2)	`template <class RandomAccessIterator, class Compare>`
	`  void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);`

Sort elements in range

Sorts the elements in the range `[first,last)` into ascending order.

The elements are compared using `operator<` for the first version, and *comp* for the second.

Equivalent elements are not guaranteed to keep their original relative order (see `stable_sort`).

Parameters

`first, last`
Random-access iterators to the initial and final positions of the sequence to be sorted. The range used is `[first,last)`, which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.
`RandomAccessIterator` shall point to a type for which `swap` is properly defined and which is both *move-constructible* and *move-assignable*.

`comp`
Binary function that accepts two elements in the range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific *strict weak ordering* it defines.
The function shall not modify any of its arguments.
This can either be a function pointer or a function object.

Return value

none

http://www.cplusplus.com/reference/algorithm/sort/

* Well, technically “containers”, of which vector is the most commonly used.

Vectors

- Sort requires that we specify the first and last vector element to sort. That's where the begin() and end() methods come in.

```
// compute mean (average) and median temperatures:
#include <iostream>  // for cin and cout
#include <vector>    // for vector
#include <algorithm> // for sort
using namespace std;

int main()
{
    vector<double> temps;  // temperatures in Fahrenheit, e.g. 64.6
    double temp;

    while (cin>>temp)
        temps.push_back(temp); // read and put into vector

    double sum = 0;
    for (int i = 0; i<temps.size(); ++i) sum += temps[i];  // sums temperatures

    cout << "Mean temperature: " << sum/temps.size() << '\n';

    sort(temps.begin(), temps.end());
    cout << "Median temperature: " << temps[temps.size()/2] << '\n';
}
```

Some More Useful Vector Methods

- size() – number of elements
- resize(n[,val]) – change number of elements to n, pad with val
- empty() – returns true if size() == 0
- clear() – empties the vector
- [n] – access element n*
- push_back(val) – adds val to end of vector
- pop_back() – deletes the last val in vector

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

* Yep, that's a method. You would define it for your own classes as "operator[]". More to follow!

A New “For” Variant

- Initializing a vector directly with a list
  - **vector<int> v = { 1, 2, 3, 5, 8, 13 };**
- Often we want to look at each element of a vector in turn:

```
double sum = 0;
for (int i = 0; i<temps.size(); ++i) sum += temps[i];  // sums temperatures
```

- A better choice is the For Each loop:

```
double sum = 0;
for (int t : temps) sum += t;  // sums temperatures
```

Note: The For Each loop is making a copy of each element of the vector.
This may be unacceptably expensive for vectors of large objects.
We'll discuss a way to reference vector elements in place soon,
until then the first variant of for is preferred for vectors of large objects.

Combining Language Features

- You can write many new programs by combining language features, built-in types, (primitives) and user-defined types (classes) in new and interesting ways.
  - So far, we have
    - Variables and literals of types **bool, char, int, double**
    - **vector, push_back(), [ ]** (subscripting)
    - **!=, ==, =, +, -, +=, <, &&, ||, !**
    - **max( ), sort( ), cin>>, cout<<**
    - **if, for, while**
  - You can write a lot of different programs with these language features! Let's try to use them in a slightly different way…

Example – Word List

```
/*
    read a bunch of strings into a vector of strings, sort
    them into lexicographical order (alphabetical order),
    and print the strings from the vector to see what we have.
*/

// “boilerplate” left out

     vector<string> words;
     for (string s; cin>>s && s != "quit“; )          // && means AND
          words.push_back(s);

     sort(words.begin(), words.end());               // sort the words we read

     for (string s : words)
          cout << s << endl;
```

In bash, this is equivalent to `cat words | sort`.

Example – Unique Word List

```
/*
    read a bunch of strings into a vector of strings, sort
    them into lexicographical order (alphabetical order),
    eliminate any duplicate (non-unique) strings,
    and print the strings from the vector to see what we have.
*/

// “boilerplate” left out

int main() {
    vector<string> words;

    cout << "Enter words, followed by EOF (e.g., Control-d):" << endl;
    for (string s; cin>>s && s != "quit"; )  // && means AND
        words.push_back(s);

    sort(words.begin(), words.end());        // sort the words we read
    words = uniq(words);                     // eliminate non-unique words

    for (string s : words)
        cout << s << '\n';
}
```

In bash, this is equivalent to `cat words | sort | uniq`.

Example – Unique Word List

```
/*
    Remove any non-unique strings from a sorted vector
    by creating a new vector “news” with only unique strings
*/

vector<string> uniq(vector<string> original) {
    vector<string> news = { };         // default in case of empty original vector
    if (original.size() > 0) {         // note style { }
        news.push_back(original[0]);
        for (int i=1; i<original.size(); ++i)  // note: not a range-for
             if(original[i-1] != original[i])
                 news.push_back(original[i]);
    }
    return news;
}
```

Algorithm

- We just used a simple *algorithm*
- An algorithm is (from Google search)
  - "a logical arithmetical or computational procedure that, if correctly applied, ensures the solution of a problem." – *Harper Collins*
  - "a set of rules for solving a problem in a finite number of steps, as for finding the greatest common divisor." – *Random House*
  - "a detailed sequence of actions to perform or accomplish some task. Named after an Iranian mathematician, Al-Khawarizmi. Technically, an algorithm must reach a result after a finite number of steps, …The term is also used loosely for any sequence of actions (which may or may not terminate)." – *Webster's*

We eliminated the duplicates by first sorting the vector (so that duplicates are adjacent), and then copying only strings that differ from their predecessor into another vector.

Algorithm – A procedure for solving a specific problem, expressed in terms of an ordered set of actions to execute

Ideal

- Basic language features and libraries should be usable in essentially arbitrary combinations
  - We are not too far from that ideal
  - If a combination of features and types make sense, it will probably work
    - The compiler helps by rejecting *some* absurdities, but accepts some that are quite surprising

Creating our Own Types

- C++ gives us useful pre-defined types
  - int, double, bool, char
- C++ defines additional useful types in libraries
  - string, vector
  - These store data and also provide methods
- C++ allows us to define our own types - *classes*
  - We declare the data that will be stored, and whether that data is visible to the program or not
  - We declare the methods that manipulate that data

A Class Encapsulates Data and Methods

- Consider an RC rotary wing drone aircraft
  - What data might we *really* want to know?
    - Drone on or off – Boolean
    - Camera on or off – Boolean
    - # rotors – const (we hope!), integer
    - Rotor speed – of each rotor
    - Altitude – off the ground? Above sea level?
  - What methods might the drone perform?
    - power_on() and power_off()
    - set_rotor_speed(rotor)
    - get_num_rotors(), get_altitude(), get_video()...

These are called "getters"

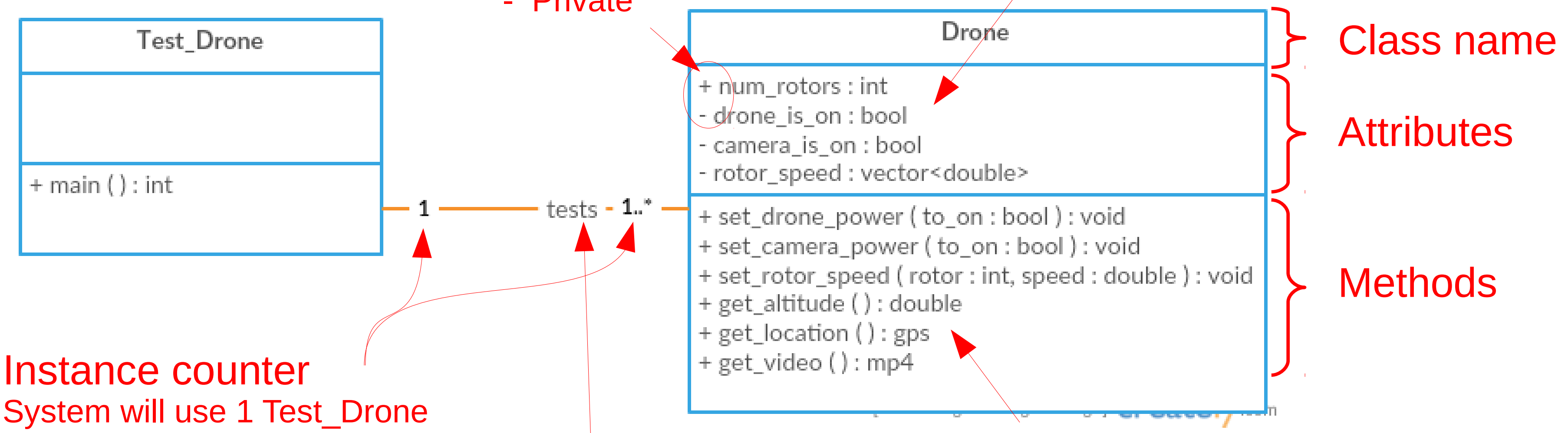
Image by Doodybutch - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=47682806

UML Can Represent Our Drone Type

Test_Drone is technically not a class (main is by C++requirement a function), so we're fudging just a little here.

Visibility
+ Public
- Private

Type


Test_Drone
+ main ( ) : int
1
tests
1..*
Drone
+ num_rotors : int
- drone_is_on : bool
- camera_is_on : bool
- rotor_speed : vector<double>
+ set_drone_power (to_on : bool) : void
+ set_camera_power (to_on : bool) : void
+ set_rotor_speed (rotor : int, speed : double) : void
+ get_altitude ( ) : double
+ get_location ( ) : gps
+ get_video ( ) : mp4
Class name
Attributes
Methods

Instance counter
System will use 1 Test_Drone and 1 to infinite Drones

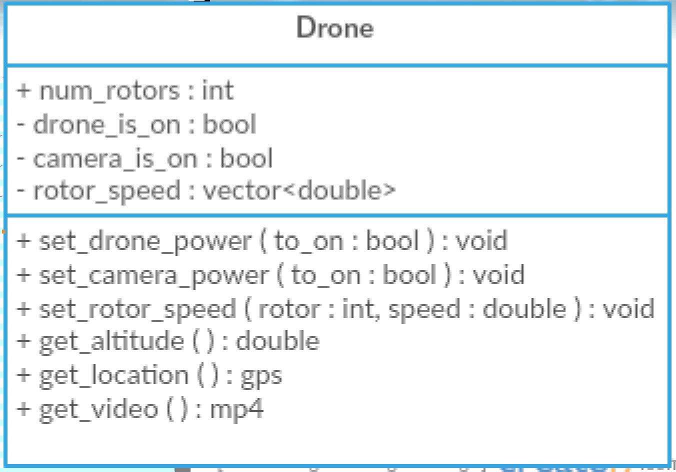
Named association
“Test_Drone tests Drone” or
“1 Test_Drone tests 1 to many Drones”

Return type
(void is sometimes omitted)

The class diagram is your “battle map” for implementation
Simply (ahem) write the classes, attributes, and methods in C++ as shown

Writing a Class in C++ Based on the UML Spec

```
class gps { }; // hardware dependent
class mp4 { }; // hardware dependent

class Drone {
  private:
    bool drone_is_on = false;
    bool camera_is_on = false;
    vector<double> rotor_speed;
  public:
    static const int num_rotors = 4;
    void set_drone_power(bool to_on) {drone_is_on = to_on;}
    void set_camera_power(bool to_on) {camera_is_on = to_on;}
    void set_rotor_speed(int rotor, double speed) {
      rotor_speed[rotor] = speed;
    }
    double get_altitude( ) { }  // hardware dependent
    gps get_location( ) { } // hardware dependent
    mp4 get_video( ) { } // hardware dependent
};
```



Drone

+ num_rotors : int
- drone_is_on : bool
- camera_is_on : bool
- rotor_speed : vector<double>

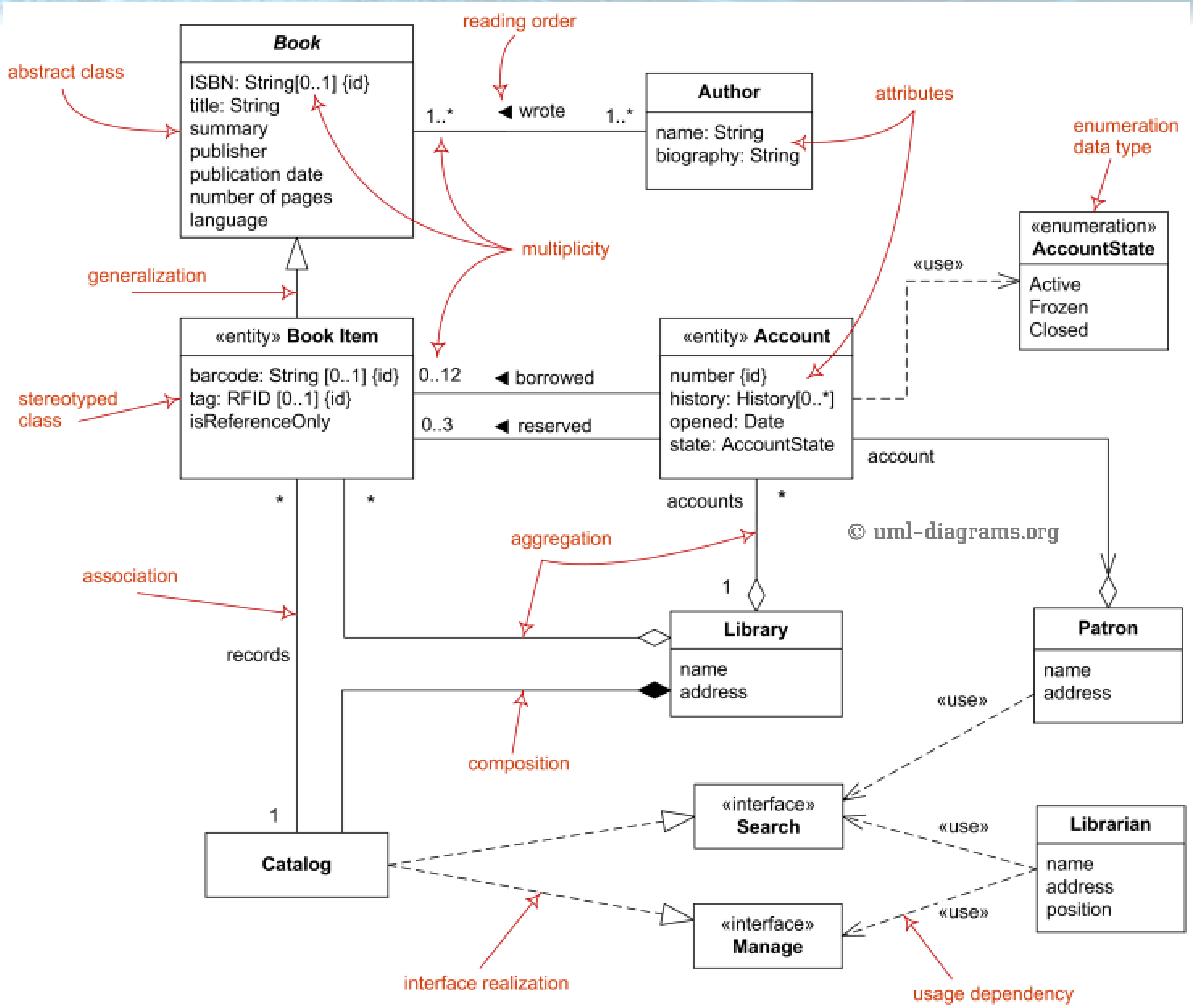
+ set_drone_power (to_on : bool) : void
+ set_camera_power (to_on : bool) : void
+ set_rotor_speed (rotor : int, speed : double) : void
+ get_altitude ( ) : double
+ get_location ( ) : gps
+ get_video ( ) : mp4

...and so on

Note: Use the -c option to compile a class file with no main( )

- "Static" means that this variable is stored with the class, not the object, and can be accessed without instancing the class, e.g., Drone::num_rotors.
  - Methods can be static, too!
  - Constructors are by their nature always static.
- "Const" means that the value stored in this variable cannot be changed

More Elements of a UML Class Diagram



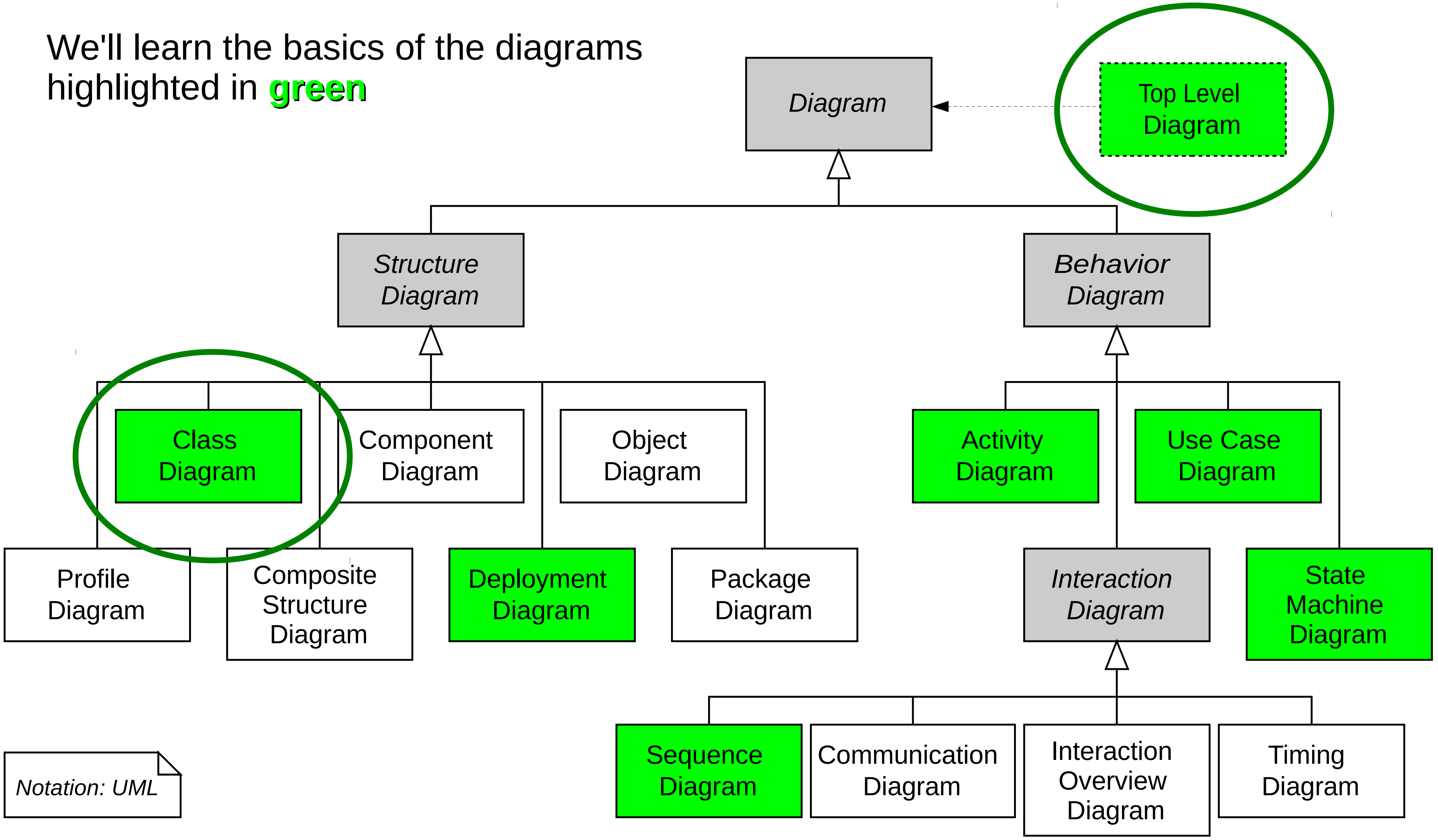
http://www.uml-diagrams.org/class-diagrams-overview.htm

This is for future reference. We haven't covered a LOT of this terminology yet, so don't sweat it!

Used with written permission.

The Diagrams (Thus Far) in Context

We'll learn the basics of the diagrams highlighted in **green**



Original source: Wikipedia, Public Domain SVG

Patterns

Expertise
/eks-per-tyz/
def: Special skill or knowledge in a particular subject, eg. He has expertise in his field of molecular sci

- A **<u>software design pattern</u>** is a general reusable algorithm solving a common problem
  - <u>Design Patterns: Elements of Reusable Object-Oriented Software</u> by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the “Gang of Four”, or GoF) is the canonical reference
- Patterns are **<u>discovered</u>**, not invented
- Patterns are documented (typically in UML) and validated by the software development community at large - **<u>meritocracy</u>**

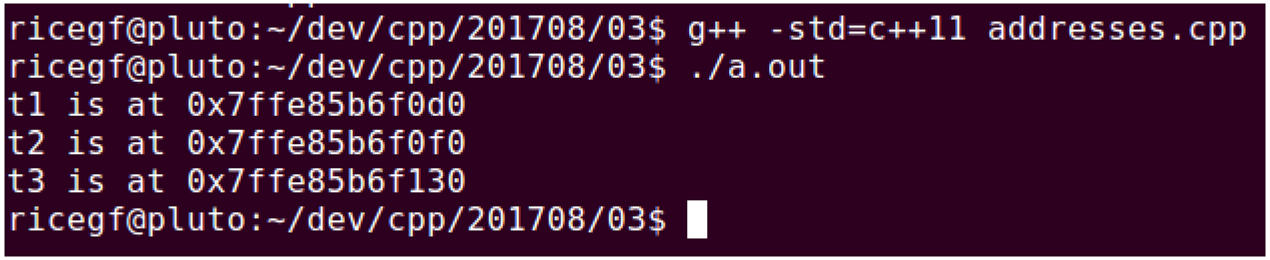
Patterns

- Example: The **<u>Singleton</u>** pattern restricts a class to instancing a single object, typically to coordinate actions across a system, e.g.,
  - A quarterback in American football
  - An orchestra conductor
  - A General in the military

Looking for Unique Objects

Class Instances are Unique by Default

```
#include <iostream>

using namespace std;

class Tester {
    string message;
  public:
    Tester(string text = "Default") : message{text} { }
    string to_string() {return message;}
};

int main() {
  Tester t1;
  Tester t2;
  Tester t3{"3 ftw!"};

  cout << "t1 is at " << &t1 << endl;  // Output the address in memory of this object
  cout << "t2 is at " << &t2 << endl;
  cout << "t3 is at " << &t3 << endl;
}
```

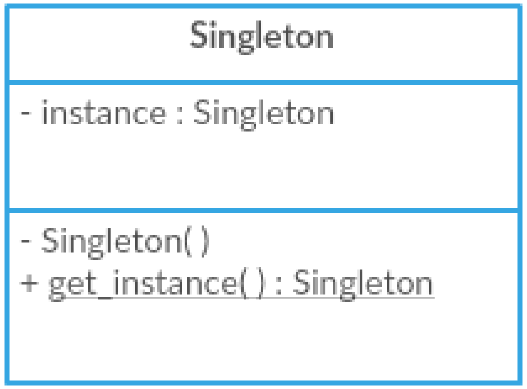


```
ricegf@pluto:~/dev/cpp/201708/03$ g++ -std=c++11 addresses.cpp
ricegf@pluto:~/dev/cpp/201708/03$ ./a.out
t1 is at 0x7ffe85b6f0d0
t2 is at 0x7ffe85b6f0f0
t3 is at 0x7ffe85b6f130
ricegf@pluto:~/dev/cpp/201708/03$
```

Different addresses in memory means different objects

Designing a Singleton Class

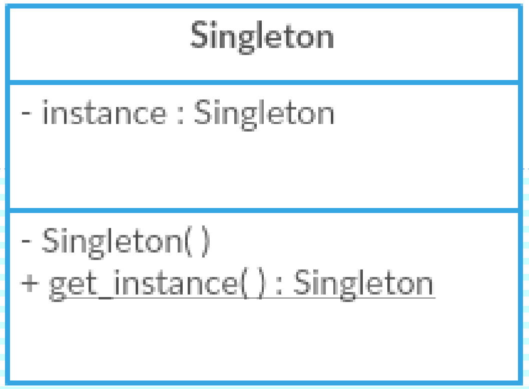
- Here's Singleton in UML
  - The Singleton instance keeps a private reference to itself
  - The constructor (which creates new Singleton instances) is also private – *only a Singleton class can create an instance of itself*
  - A static getInstance() method, when called, simply:
    - Returns the existing instance if it exists
    - Creates the instance otherwise and saves it to return for this and all future calls



The Singleton Pattern

(Slightly Simplified)

A call to Singleton.getInstance() will return a reference to the only instance of Singleton in the system – every single time!



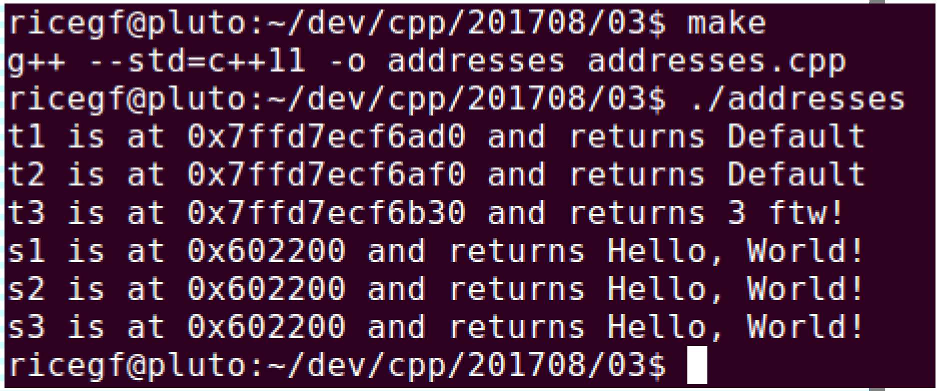
```
 class Singleton {
   public:
     static Singleton& getInstance() {
       static Singleton  instance;
       return instance;
     }
     Singleton(Singleton const&)      = delete; // Don't Implement
     void operator=(Singleton const&) = delete; // Don't implement
   private:
        Singleton() {};
}
```

Don't sweat the ampersands, which are kinda sorta pointers, or the deletes, which are a feature (ahem) of C++. We'll get to them soon!

Looking for Unique Objects

Class Instances are Unique by Default

```
// Insert after class Tester
class Singleton {
    string message;
    Singleton(string text = "Default") : message{text} { }
  public:
    static Singleton& get_instance(string text = "Default") {
      static Singleton instance{text};
      return instance;
    }
    string to_string() {return message;}
    Singleton(Singleton const&)      = delete;
    void operator=(Singleton const&) = delete;
};
```



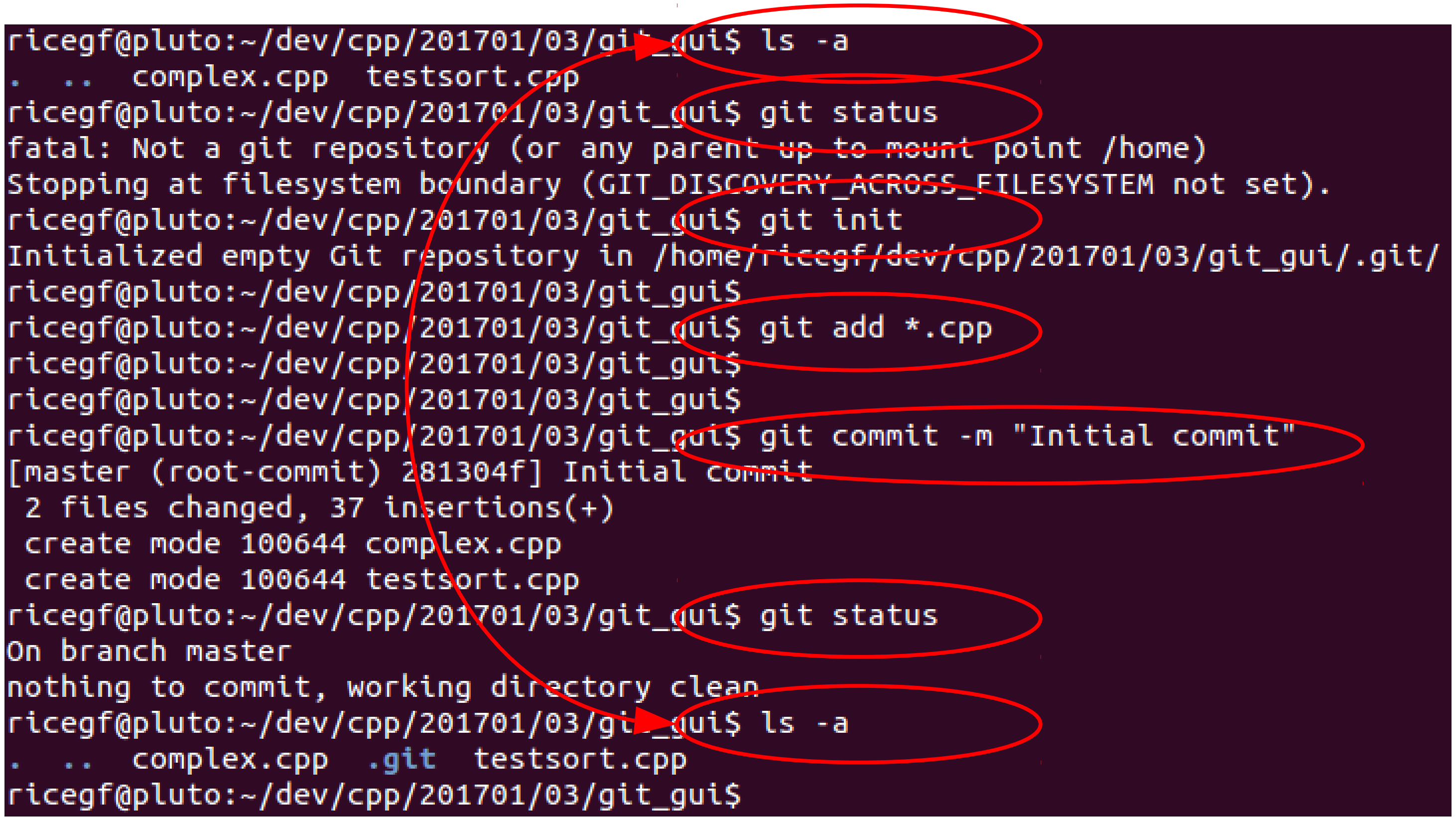
```
ricegf@pluto:~/dev/cpp/201708/03$ make
g++ --std=c++11 -o addresses addresses.cpp
ricegf@pluto:~/dev/cpp/201708/03$ ./addresses
t1 is at 0x7ffd7ecf6ad0 and returns Default
t2 is at 0x7ffd7ecf6af0 and returns Default
t3 is at 0x7ffd7ecf6b30 and returns 3 ftw!
s1 is at 0x602200 and returns Hello, World!
s2 is at 0x602200 and returns Hello, World!
s3 is at 0x602200 and returns Hello, World!
ricegf@pluto:~/dev/cpp/201708/03$
```

```
// Insert at the end of main()
  Singleton& s1 = Singleton::get_instance("Hello, World!");
  Singleton& s2 = Singleton::get_instance("Goodbye, World!");
  Singleton& s3 = Singleton::get_instance();

  cout << "s1 is at " << &s1 << " and returns " << s1.to_string() << endl;
  cout << "s2 is at " << &s2 << " and returns " << s2.to_string() << endl;
  cout << "s3 is at " << &s3 << " and returns " << s3.to_string() << endl;
```

Different addresses in memory
means different objects

Git GUI

- Thus far, we've used git as a command line tool



```
ricegf@pluto:~/dev/cpp/201701/03/git_gui$ ls -a
.  ..  complex.cpp  testsort.cpp
ricegf@pluto:~/dev/cpp/201701/03/git_gui$ git status
fatal: Not a git repository (or any parent up to mount point /home)
Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
ricegf@pluto:~/dev/cpp/201701/03/git_gui$ git init
Initialized empty Git repository in /home/ricegf/dev/cpp/201701/03/git_gui/.git/
ricegf@pluto:~/dev/cpp/201701/03/git_gui$
ricegf@pluto:~/dev/cpp/201701/03/git_gui$ git add *.cpp
ricegf@pluto:~/dev/cpp/201701/03/git_gui$
ricegf@pluto:~/dev/cpp/201701/03/git_gui$
ricegf@pluto:~/dev/cpp/201701/03/git_gui$ git commit -m "Initial commit"
[master (root-commit) 281304f] Initial commit
 2 files changed, 37 insertions(+)
 create mode 100644 complex.cpp
 create mode 100644 testsort.cpp
ricegf@pluto:~/dev/cpp/201701/03/git_gui$ git status
On branch master
nothing to commit, working directory clean
ricegf@pluto:~/dev/cpp/201701/03/git_gui$ ls -a
.  ..  complex.cpp  .git  testsort.cpp
ricegf@pluto:~/dev/cpp/201701/03/git_gui$
```

Is this a git repository? No

Initialize this directory as a git repository

Add all of the C++ files to tracking

Commit the added files to the repository

Verify that all is well

Note the (hidden) new .git directory – that's the actual repository

We can do this from a GUI as well

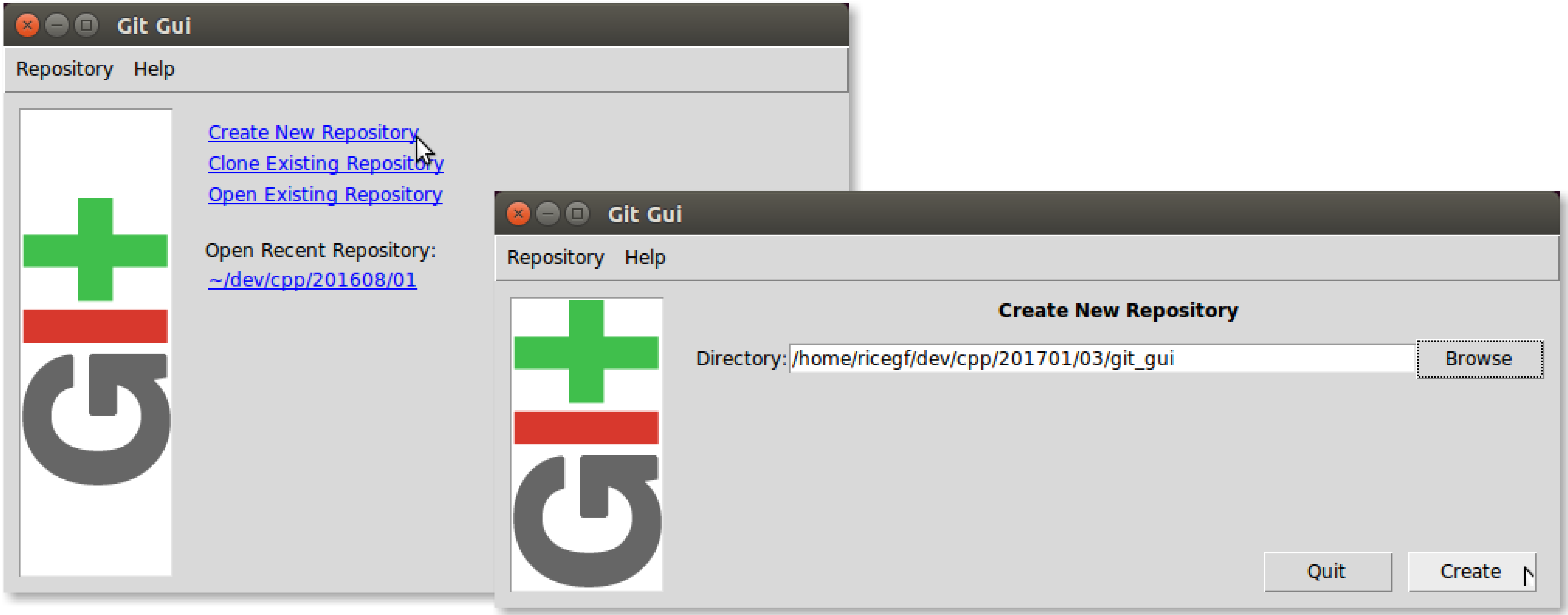
(But don't until you have the commands down pat!)

What About My IDE?

- Most Integrated Development Environments (IDE) “speak” fluent git
  - Visual Studio supports git directly starting with version 2013.1
  - Eclipse by default now includes the Egit plug-in, with excellent git support
  - CodeBlocks offers the GitBlocks plug-in, which can be installed to offer excellent git support
  - Google for other IDEs in which you have an interest
- Neither I nor the TAs offer support for IDEs
  - We support bash, though, which is the learning focus for this semester
  - More tools in your toolkit is a feature!

Launching Git GUI

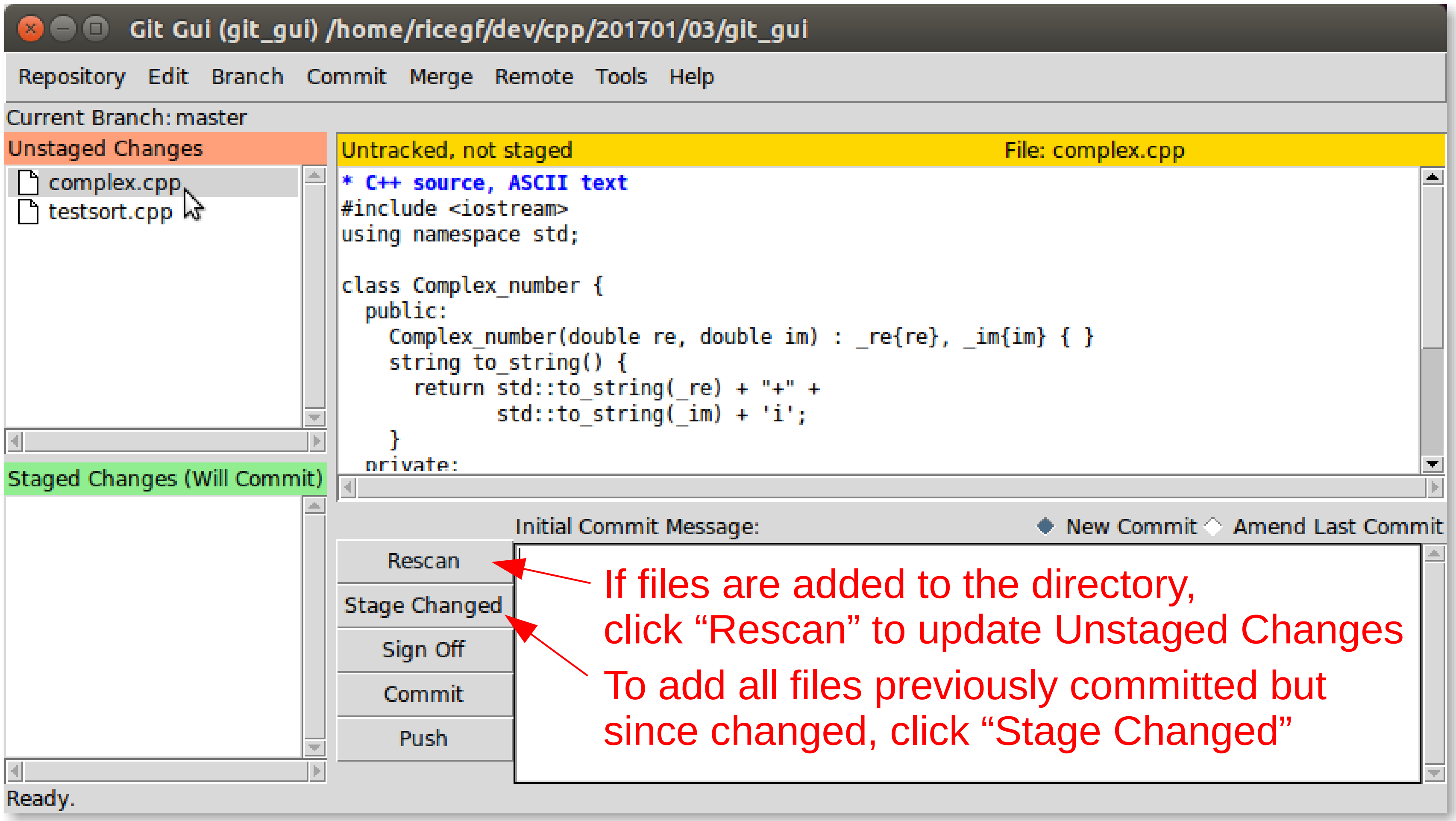
- Simply type “git gui”
- If you see this screen, no repository is present
  - Click “Create New Repository” (aka “git init”)



Directory: /home/ricegf/dev/cpp/201701/03/git_gui

Viewing and Adding Files

- Click on a file to view its contents
- Click a file's icon to stage it (aka "git add")



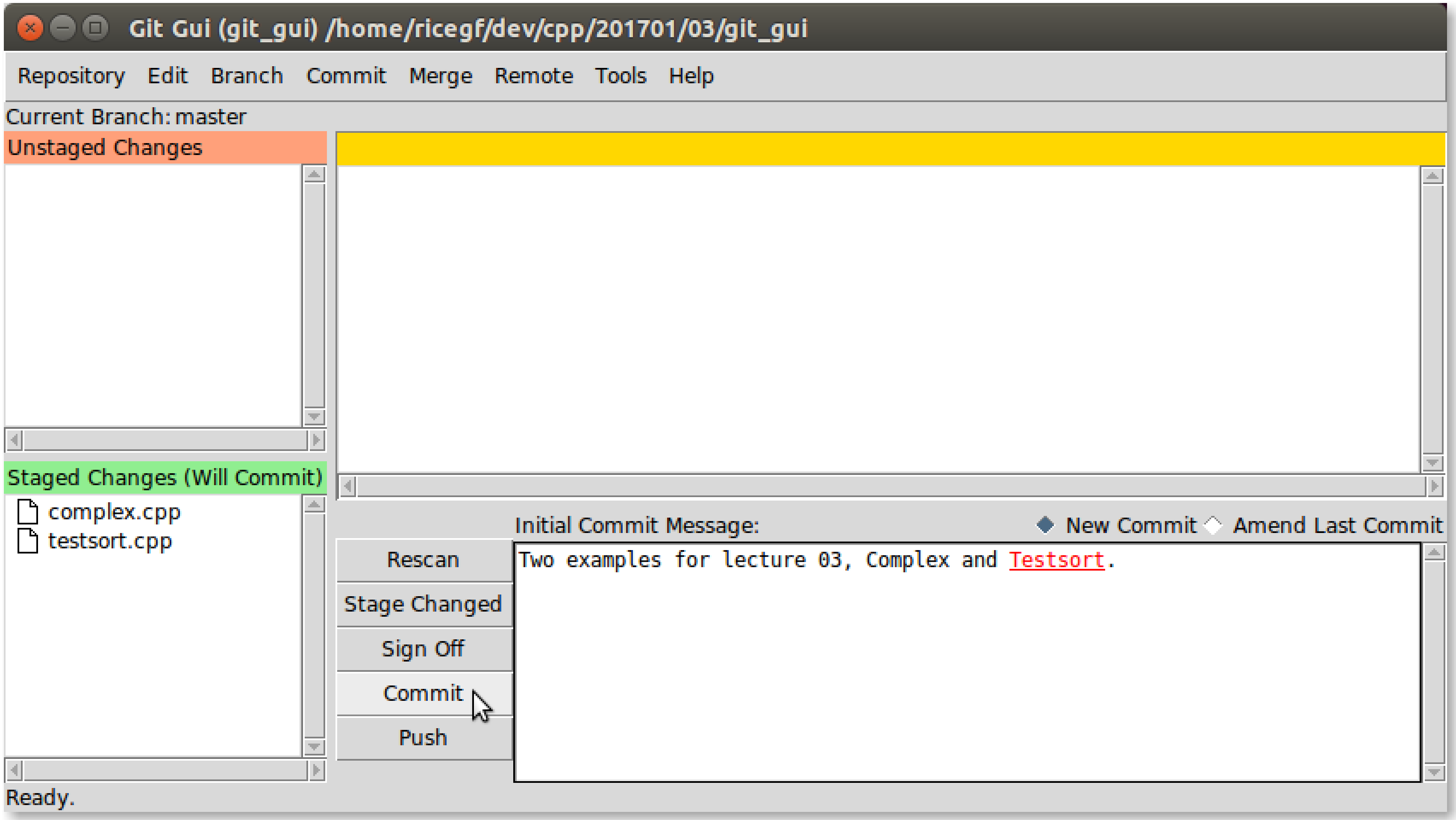
```
* C++ source, ASCII text
#include <iostream>
using namespace std;

class Complex_number {
  public:
    Complex_number(double re, double im) : _re{re}, _im{im} { }
    string to_string() {
      return std::to_string(_re) + "+" +
             std::to_string(_im) + 'i';
    }
  private:
```

If files are added to the directory, click "Rescan" to update Unstaged Changes

To add all files previously committed but since changed, click "Stage Changed"

Committing Files

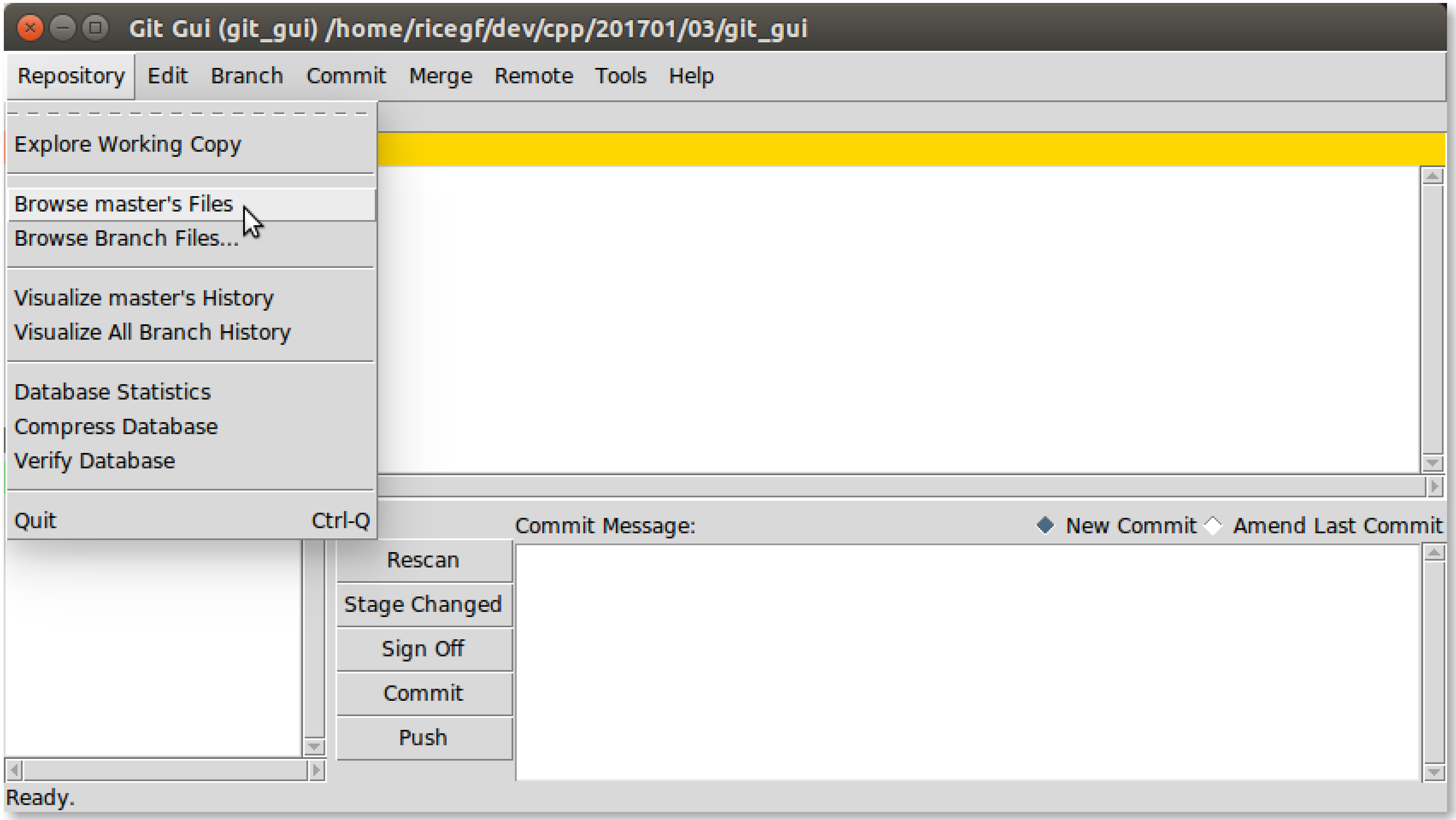
- Add a message in the lower right text box
- Click “Commit” to save the files in the repository



```
Two examples for lecture 03, Complex and Testsort.
```

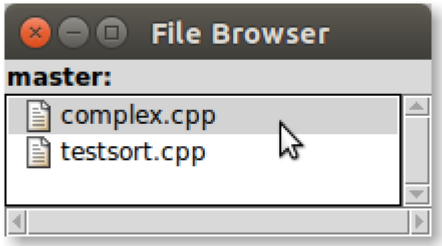
Viewing the Commit History

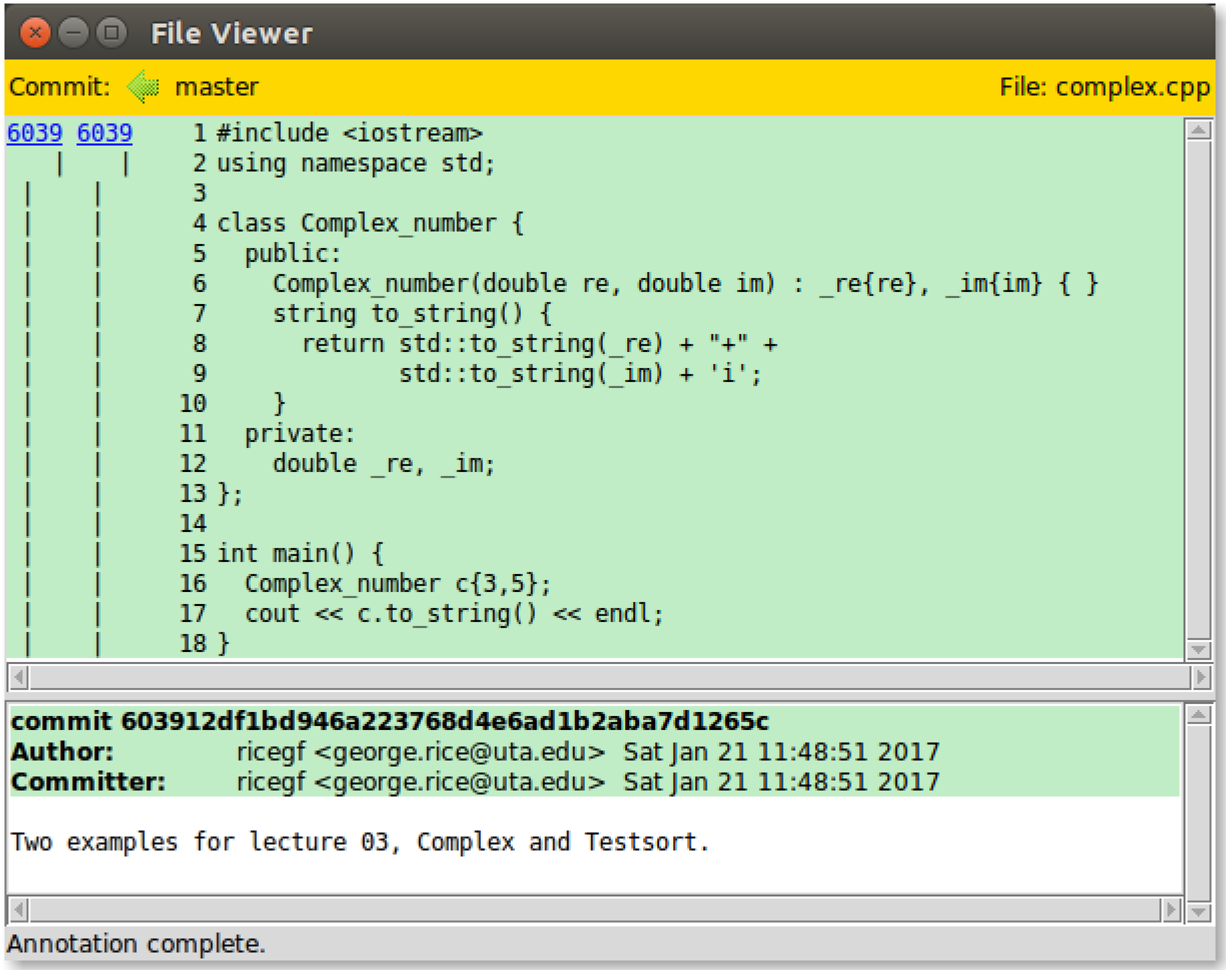
- To explore the commit history, select Repository → Browse Master's Files



Viewing a File's Origins

- To explore a file's changes, select Repository → Browse Master's Files





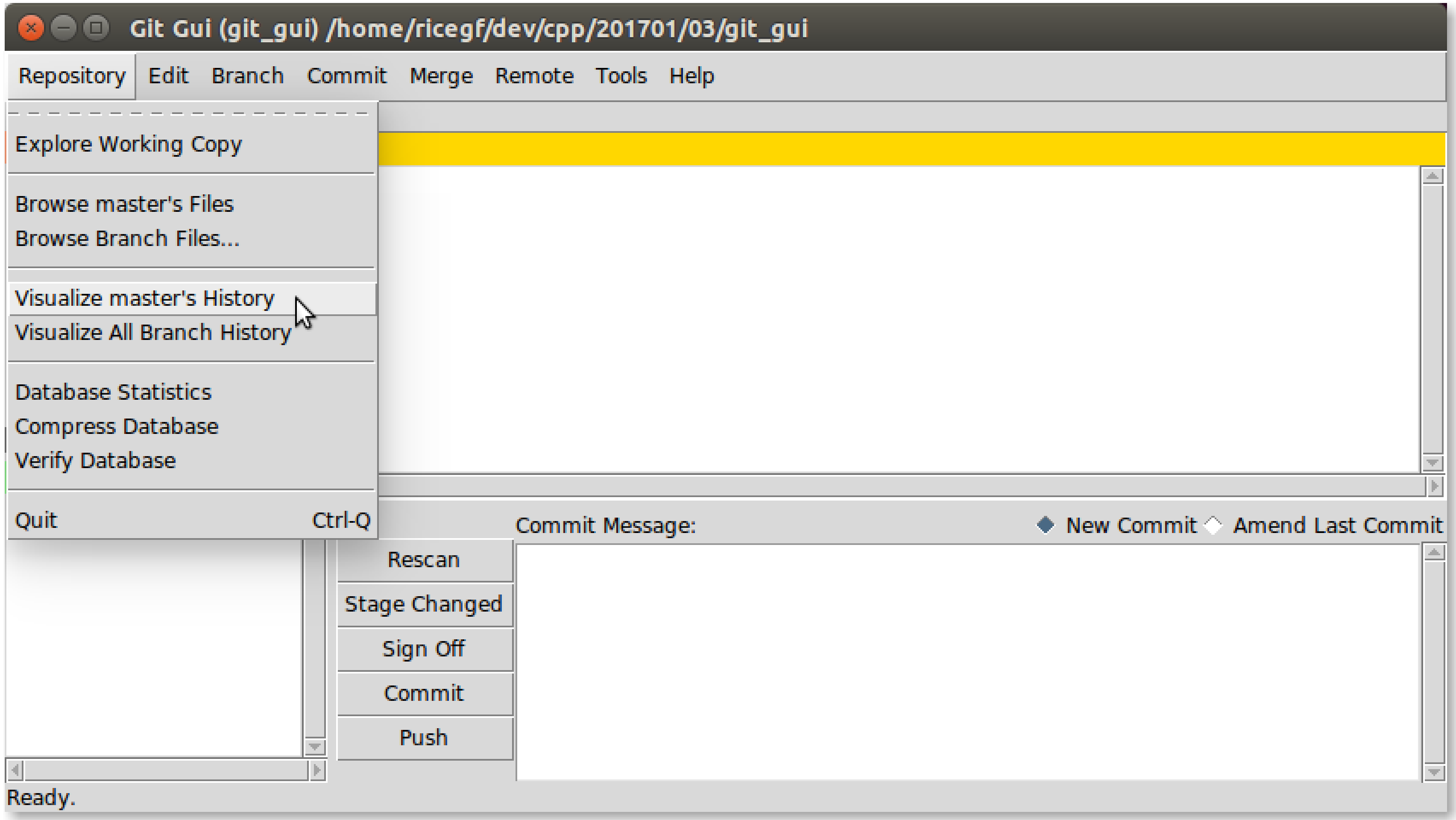
```
6039 6039  #include <iostream>
           using namespace std;

           class Complex_number {
             public:
               Complex_number(double re, double im) : _re{re}, _im{im} { }
               string to_string() {
                 return std::to_string(_re) + "+" +
                        std::to_string(_im) + 'i';
               }
             private:
               double _re, _im;
           };

           int main() {
             Complex_number c{3,5};
             cout << c.to_string() << endl;
           }
```

```
commit 603912df1bd946a223768d4e6ad1b2aba7d1265c
Author:       ricegf <george.rice@uta.edu>  Sat Jan 21 11:48:51 2017
Committer:    ricegf <george.rice@uta.edu>  Sat Jan 21 11:48:51 2017

Two examples for lecture 03, Complex and Testsort.
```

Annotation complete.

Viewing the Commit History

- To explore the commit history, select Repository → Browse Master's Files

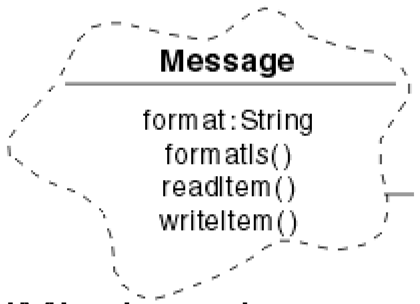


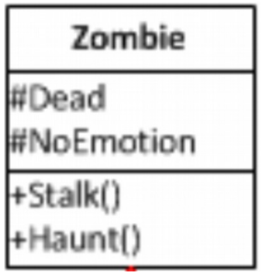
Quick Review

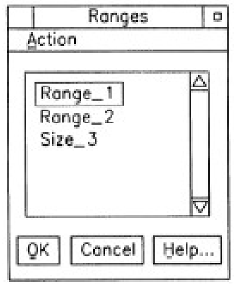
- Dividing big computations into many little ones is called __________.
- Providing a higher-level concept that hides detail is called ____________.
- Why are concise operators (+=) usually preferable to infix operators (= … + …)?
- Which of the following is a C++ statement?
  a. An expression terminated with a semicolon
  b. A declaration
  c. A “control statement” that determines the flow of control
- Describe the two different “for” loops in C++, and when the use of each is preferable?
- A procedure for solving a specific problem, expressed in terms of an ordered set of actions to execute, is called a(n) _____________.
- The git GUI can be launched from the command line with ____ ____.

Quick Review

- Which of these represents a "class" in UML?

  a. 

  b. 

  c. 

- In a UML class diagram, public visibility of a field or method is indicated by a ___, while private visibility by a ___.
- A UML _________ relationship indicates a reference from one class to another.
- The notation "« interface »" (a name inside of guillemets) is called a __________. This is one of 3 mechanisms to customize UML.
- A general reusable algorithm solving a common problem is called a __________ _________ __________.
  - How are new ones created?
- The ____________ ____________ restricts a class to instancing a single object, typically to coordinate actions across a large system.
  - What is the key to implementing this?

For Next Class

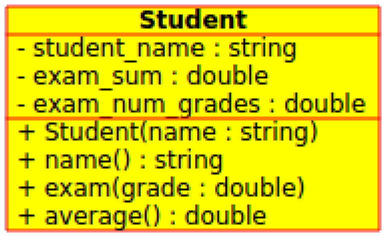
- (Optional) Read Chapter 4 in Stroustrop
  - Do the Drills!
  - Skim Chapter 5 and (optionally) Chapter 26 for next lecture
- (Optional) Read Chapter 3 (through "Static View") pp 25-29 in Rumbaugh
  - Skim Chapters 1-2 for background

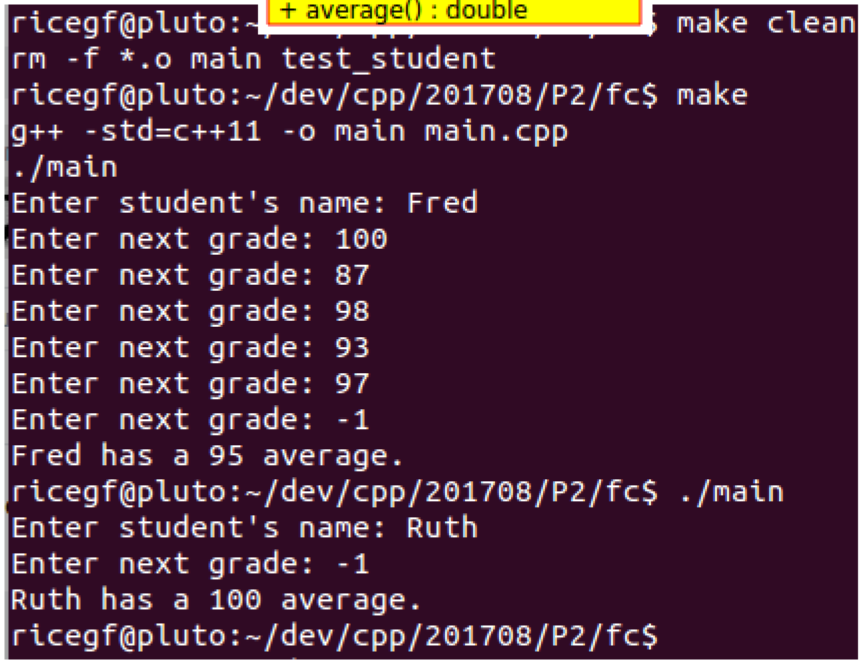
You learn to program by programming!

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

Homework #2: Gradebook

- Create an OO-based program that accepts a student’s name and exam grades, and outputs their semester average.
  - **Bonus:** Also include homework grades in their semester average.
  - **Extreme Bonus:** Read the data from a file, and write the same file in the same format but with additional information added.
- As always, details are on Blackboard.
  - Requirements are in a ZIP archive along with test data





```
ricegf@pluto:~                              $ make clean
rm -f *.o main test_student
ricegf@pluto:~/dev/cpp/201708/P2/fc$ make
g++ -std=c++11 -o main main.cpp
./main
Enter student's name: Fred
Enter next grade: 100
Enter next grade: 87
Enter next grade: 98
Enter next grade: 93
Enter next grade: 97
Enter next grade: -1
Fred has a 95 average.
ricegf@pluto:~/dev/cpp/201708/P2/fc$ ./main
Enter student's name: Ruth
Enter next grade: -1
Ruth has a 100 average.
ricegf@pluto:~/dev/cpp/201708/P2/fc$
```

“Now is the time for all good persons to come to the aid of their final average.”
Do the bonus and extreme bonus! – Patrick Henry*

*OK, maybe it was Charles E. Weller. And he didn’t say “final average”. Or “persons”. Or possibly anything like this at all.