CSE 1325: Object-Oriented Programming

Exam #1 Retrospective

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

Point Allocation by Skill

I		Vocabulary	17	93%	
II		General	32	81%	
III	1&2	Patterns	7	68%	
	3	UML Use Case	6	90%	
	4	UML Extensions	4	84%	
	5a	Enum	4		
	5b	UML Class	6		
	5c	Coding .h from UML	6	86%	
	5d	Coding .cpp from .h	6		
	6	main	6	76%	
	7	git	6	80%	
B1		Intellectual Property	2	67%	
B2		References	3		
		Total	105	**86%**	**Mean**
				88.5	**Median**

Statistics and Such

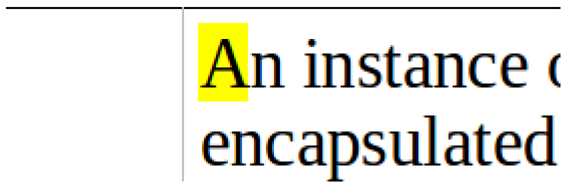
- The median grade was 88.5, 0.5 higher than last semester
  - Section I contributed 15.9 out of 17
  - Section II contributed 25.9 out of 32
  - Section III contributed 41.6 out of 51
  - The bonus contributed 3.3 out of 5
- Only a few questions were left blank
  - 97% still working after 40 minutes; 79% after 60 minutes; 35% at end, a bit longer than last semester
  - Half of the exams had been returned by 67 minutes, exactly as last time
- Minor typos (e.g., missing ;) were forgiven
- Much leeway was given on the models
  - A *reasonable* representation was sufficient for full credit
  - Rubrics were used to maintain consistency

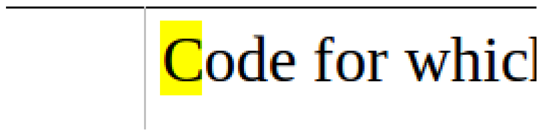
Some Problem Areas

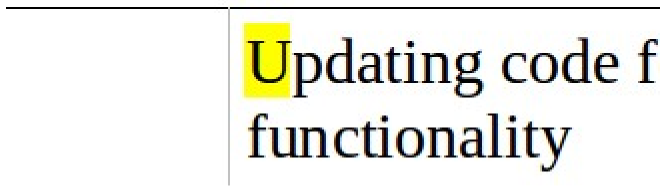
- Most missed definition: Intellectual Property
- Most missed multiple choice options:
  - Patterns are often described using the UML
  - References can’t change to point to a different variable
  - Makefiles are NOT specific to Linux
- Most students did well on modeling and coding
  - Use case and class diagramming looked great!
  - UML extensions did not look good (!)
  - Read STDIN until EOF was not good (½ pt) – any valid code was accepted (e.g., ask for number of ints)

Identifying Your Test

- Three distinct tests were given for op sec
  - The *order* of many questions changed
  - Operational security (we'll get to it!)
- The tests were named “A”, “C”, and “U” based on the 1st letter of the first definition in Section I


An instance
encapsulated


Code for whic


Updating code f
functionality

Test Markings

- Section I – red “X” marks any errors
- Section II – red “/” indicates incorrect answers, red letters or circles indicate missing answer, +N or N in left column per question indicates points gained for that question
  - Each answer is “true” or “false”. +½ if correct, +0 if not.
  - If more than 4 possible, ignore the rest
- Section III – corrections *may* be marked, +N in left column per question indicates points gained for that question
- Sum of points gained per page indicated at the bottom of each page
- Final score is on page 3 (or 1) and on Blackboard

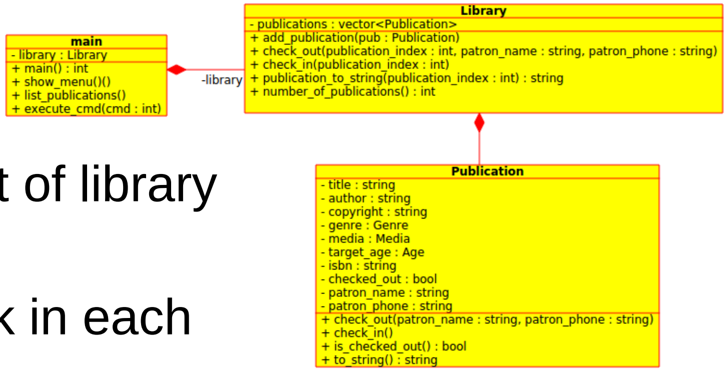
Review of the Exam Key

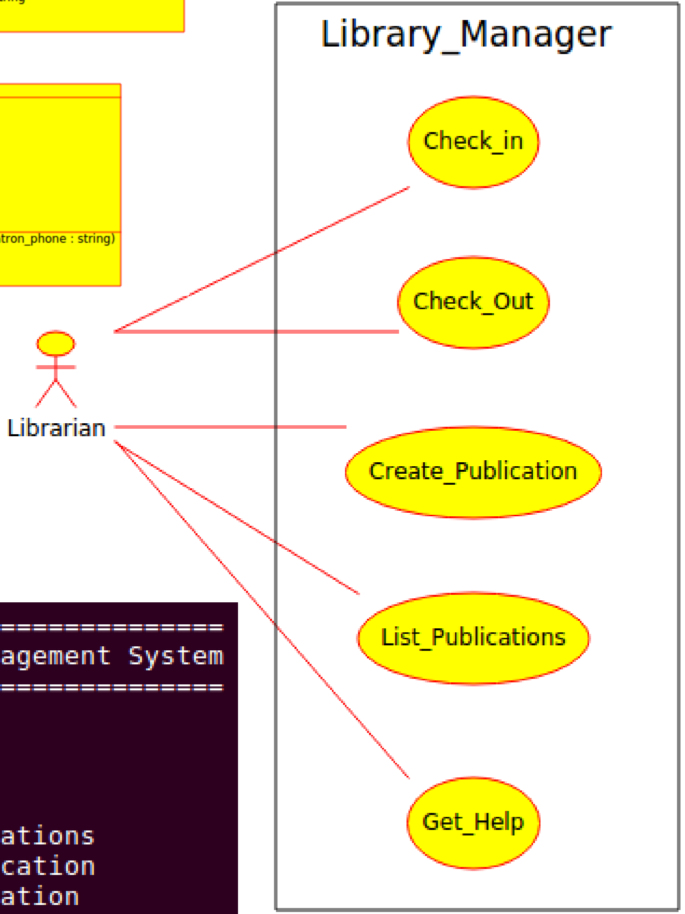
- Correct answers show by variant
  - For sections I and II, order is A / C / U
- Rubric is at the end

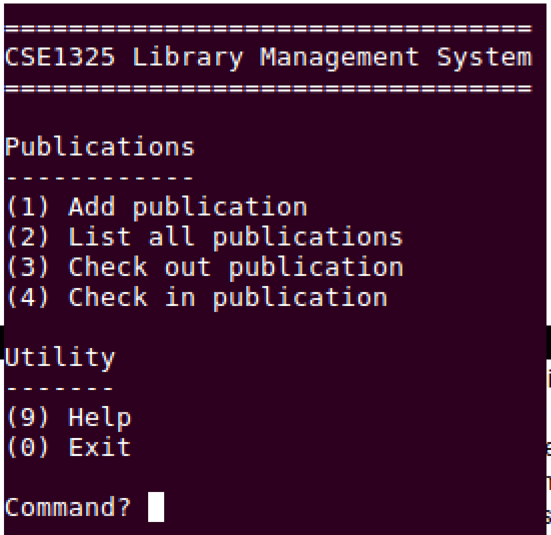
Homework Questions?

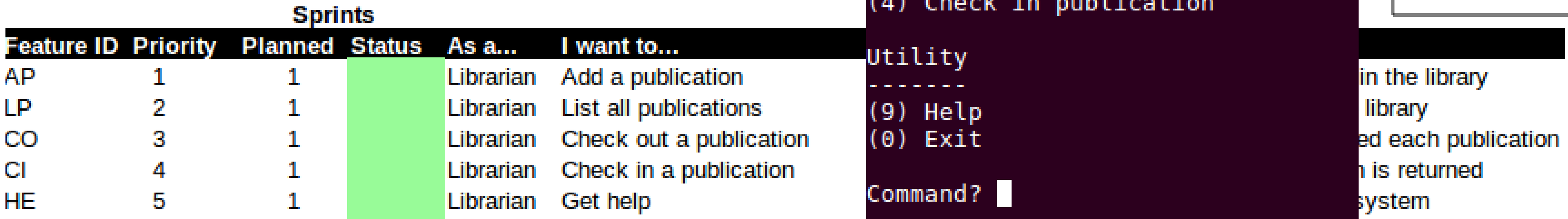
Sprint 1

- Build a simple Library Management System
  - Add publications to the list of library assets
  - Check out and check back in each publication
  - Provide basic help
- As always, use git for version management
- Manage the sprint with the Scrum spreadsheet
- Details are on Blackboard
- **Due September 28 at 8 am**


main
- library : Library
+ main() : int
+ show_menu()()
+ list_publications()
+ execute_cmd(cmd : int)
-library
Library
- publications : vector<Publication>
+ add_publication(pub : Publication)
+ check_out(publication_index : int, patron_name : string, patron_phone : string)
+ check_in(publication_index : int)
+ publication_to_string(publication_index : int) : string
+ number_of_publications() : int
Publication
- title : string
- author : string
- copyright : string
- genre : Genre
- media : Media
- target_age : Age
- isbn : string
- checked_out : bool
- patron_name : string
- patron_phone : string
+ check_out(patron_name : string, patron_phone : string)
+ check_in()
+ is_checked_out() : bool
+ to_string() : string



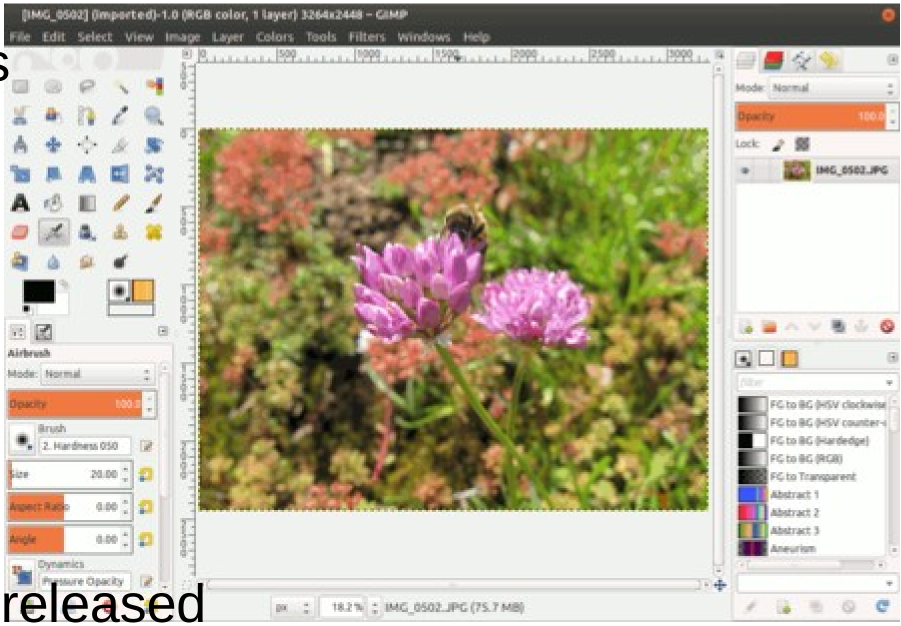

```
==================================
CSE1325 Library Management System
==================================

Publications
------------
(1) Add publication
(2) List all publications
(3) Check out publication
(4) Check in publication

Utility
-------
(9) Help
(0) Exit

Command? █
```


Sprints

Feature ID	Priority	Planned	Status	As a...	I want to...	
AP	1	1	🟩	Librarian	Add a publication	in the library
LP	2	1	🟩	Librarian	List all publications	library
CO	3	1	🟩	Librarian	Check out a publication	ed each publication
CI	4	1	🟩	Librarian	Check in a publication	n is returned
HE	5	1	🟩	Librarian	Get help	system

Review
GIMP Tool Kit + (Gtk+)

- Gtk+ is a popular cross-platform GUI toolkit written in C
  - 1995: The GIMP, a free high-end graphics program, was released
  - 1997: Its key graphics primitives were factored out, named the GNU Tool Kit (Gtk), and used to implement GIMP 0.60
  - 1998: The Gtk library was rebuilt using an object-oriented paradigm (in native C), renamed Gtk+, and used for GIMP 0.99
  - 2000: A C++ wrapper named gtkmm was released


[IMG_0502] (imported)-1.0 (RGB color, 1 layer) 3264x2448 – GIMP
File Edit Select View Image Layer Colors Tools Filters Windows Help

- Gtk+ is widely used for desktops (GNOME, Unity, Cinnamon, MATE, and Xfce) as well as all GNOME applications
  - gtkmm is used for applications such as Inkscape and VMware Workstation
- Gtk+ is licensed under the GNU Library General Public License

https://gtkmm.org/en/

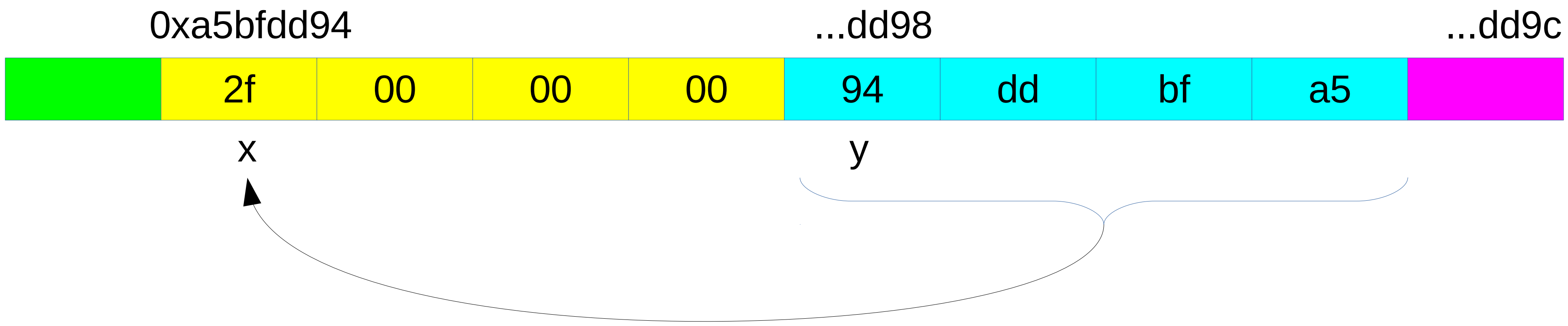
Some gtkmm Documentation Uses Pointers

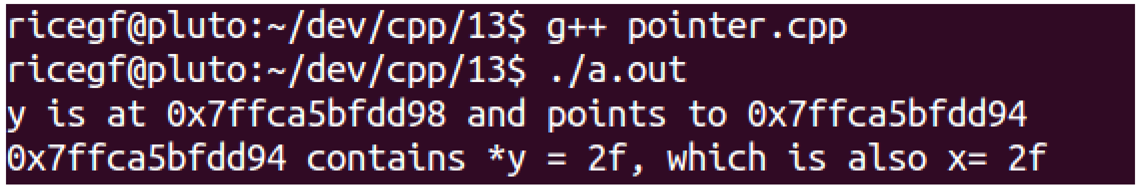
Thus we will, too

Pointers – (Likely) A Review

- A pointer is an “unsafe reference”
  - It references a memory location containing a type

```
int x = 47; // Assume the linker assigns x to address a5bfdd94
int *y;     // y contains the address of an integer – but initially, garbage
y = &x;     // Now y contains the address of x
    cout << hex << "y is at " << &y << " and points to " << y << endl;
    cout << y << " contains *y = " << *y << ", which is also x= " << x << endl;
}
```



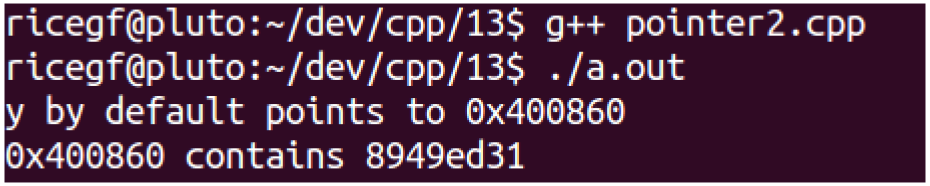


```
ricegf@pluto:~/dev/cpp/13$ g++ pointer.cpp
ricegf@pluto:~/dev/cpp/13$ ./a.out
y is at 0x7ffca5bfdd98 and points to 0x7ffca5bfdd94
0x7ffca5bfdd94 contains *y = 2f, which is also x= 2f
```

Unlike References, Pointers can Point *Anywhere*

- References must be initialized on definition
- Pointers need not be initialized...

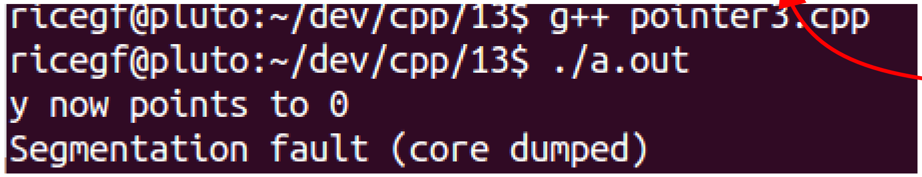
```
    int *y;
    cout << hex << "y by default points to " << y << endl;
    cout << y << " contains " << *y << endl;
```



```
ricegf@pluto:~/dev/cpp/13$ g++ pointer2.cpp
ricegf@pluto:~/dev/cpp/13$ ./a.out
y by default points to 0x400860
0x400860 contains 8949ed31
```

- ...or they can be initialized to any address

```
    int *y;
    y = 0;
    cout << hex << "y now points to " << y << endl;
    cout << y << " contains " << *y << endl;
```



```
ricegf@pluto:~/dev/cpp/13$ g++ pointer3.cpp
ricegf@pluto:~/dev/cpp/13$ ./a.out
y now points to 0
Segmentation fault (core dumped)
```

Accessing the value pointed to by a pointer is called *dereferencing*

Creating "New" Objects

- Pointers are often used to reference *unmanaged* variables
- "double x[3];" creates a *managed* variable
  - The memory is released when x goes out of scope
- "double *x = new double[3];" creates an *unmanaged* variable

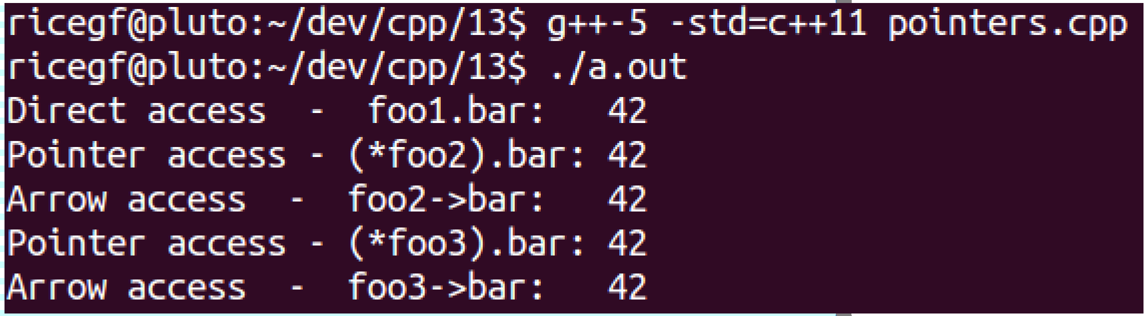
Malloc, free, et. al. are C functions –
<u>never</u> use them in an OOP C++ program

  - The memory is released when you explicitly delete it
  - Delete it using "delete[ ] x;"
- Advantage: You can create an unmanaged variable (an object) inside of a method and return it without its memory being returned when the scope exits

Accessing Object Members via Pointers

- You reference an object member with .
- You reference a pointer's object member with ->

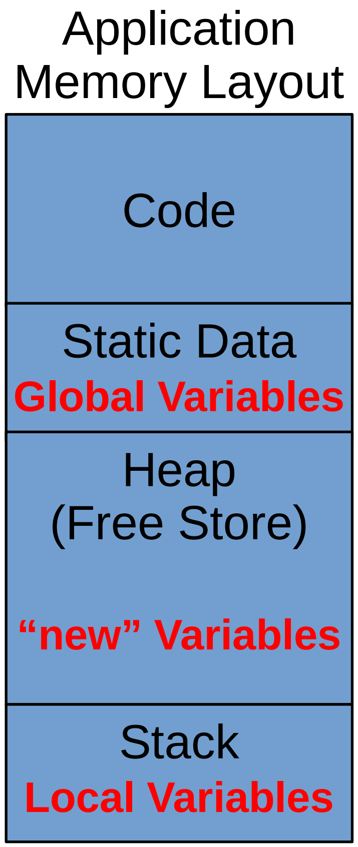
```
#include <iostream>
using namespace std;

class Foo {
  public:
    int bar = 42;
};

int main()
{
    Foo foo1;
    cout << "Direct access  -  foo1.bar:   " << foo1.bar << endl;
    Foo *foo2;
    foo2 = &foo1;
    cout << "Pointer access - (*foo2).bar: " << (*foo2).bar << endl;
    cout << "Arrow access  -  foo2->bar:   " << foo2->bar << endl;
    Foo *foo3 = new Foo{ };
    cout << "Pointer access - (*foo3).bar: " << (*foo3).bar << endl;
    cout << "Arrow access  -  foo3->bar:   " << foo3->bar << endl;
    delete[ ] foo3;
}
```



```
ricegf@pluto:~/dev/cpp/13$ g++-5 -std=c++11 pointers.cpp
ricegf@pluto:~/dev/cpp/13$ ./a.out
Direct access  -  foo1.bar:   42
Pointer access - (*foo2).bar: 42
Arrow access  -  foo2->bar:   42
Pointer access - (*foo3).bar: 42
Arrow access  -  foo3->bar:   42
```

Stack vs Heap

- **Stack** is "scratch memory" for a thread
  - LIFO (Last-In, First-Out) allocation and deallocation
    - Allocation when a scope is entered
    - Automatically deallocated when that scope exits
  - Simple to track and so rarely leaks memory
- **Heap** is memory shared by all threads for dynamic allocation
  - Allocation and deallocation can happen at any time – but only when *specifically* invoked!
    - Allocate using "new" to a pointer
    - Deallocate using "delete" ("delete[]" for vectors and arrays)



“delete” vs “delete[ ]”

```
void calc() {
    int result;
    int *i = new int{};           // allocate one integer from the heap
    double *d = new double[10];   // allocate 10 doubles in an array from the heap

    // … use i and d to calculate result

    delete[] d;    // mark the heap array that d points to as free
    delete i;      // mark the integer i on the heap as free
    return result; // result automatically deallocates when it goes out of scope!
}
```

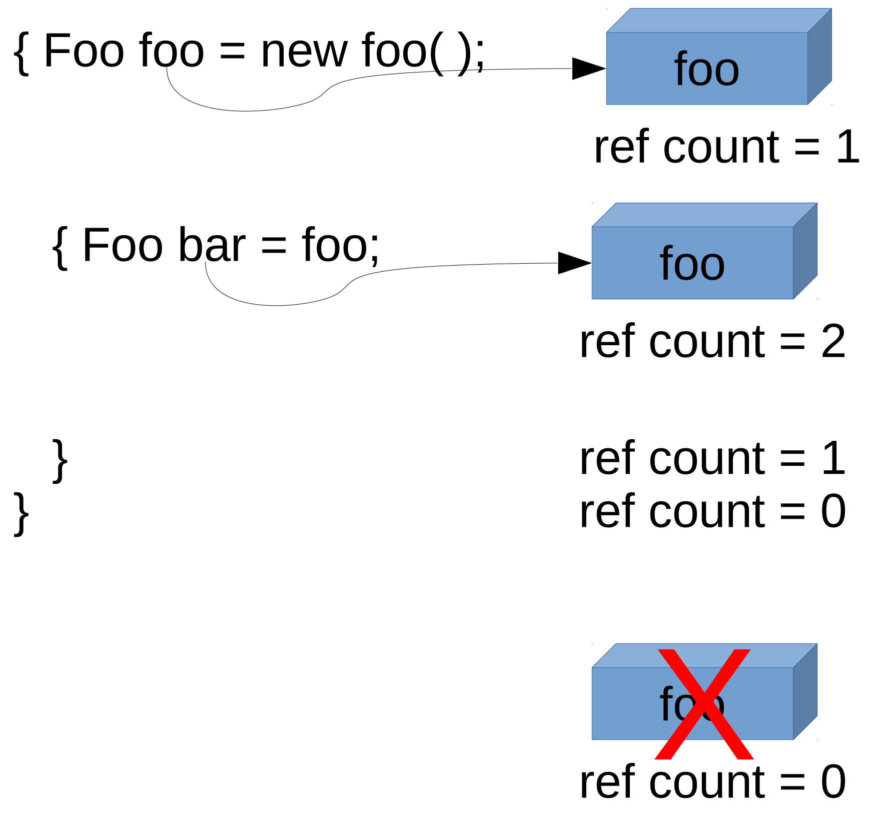
- Manually delete heap variables when you no longer need them
  - “delete” de-allocates a simple variable (like i)
  - “delete[ ]” de-allocates a vector or array (like d)
- The heap is useful for allocating memory that survives the “return” statement

Deleting Unmanaged Variables

- Problem: When do you call delete x?
  - And how can you ensure you never forget?
- Solution: In C++, it's... complicated
  - We can use “reference counted” variables via “smart pointers” (unique_ptr and shared_ptr)
    - Managed languages such as Java, Python, and C# do this (almost) always
  - We can used gtkmm’s Gtk::Manage for gtkmm objects
  - We can (and otherwise will) ignore the problem for now, as memory is freed by Ubuntu when your program exits

Reference Counting

- Java, Python, and (mostly) C# implement managed memory via reference counters
  - When an object is instanced, memory is allocated and a **“reference counter”** is set to 1
  - When a reference to that object is created, the counter is incremented
  - When a reference to that object is deleted (e.g., goes out of scope), the counter is decremented
  - When more memory is needed, the **“garbage collector”** finds all reference counters == 0, frees the associated memory, and moves all in-use objects to free up a contiguous block of memory for future use



```
{ Foo foo = new foo( );
  { Foo bar = foo;
  }
}
```

C++ Offers Managed Memory for Pointers via Library Classes

- std::unique_ptr<T> ensures the (single referenced) memory allocated from heap is released when the managing object goes out of scope

```
{   T *p = new T{42, "meaning"};  // T is a class type that we defined earlier
    my_function(p);
    delete[] p;
}
```

my_function may throw an exception, skipping the delete[] and "leaking" memory

```
using namespace std;
{
    unique_ptr<T> p{new T(42, "meaning")};
    my_function(p);
} // p's destructor is guaranteed to run "here", calling delete
```

Even if my_function throws an exception, memory is freed when local scope exits

- std::shared_ptr<T> implements a reference counter and releases the memory when the counter reaches 0

```
{
    std::shared_ptr<T> p(new T(3.14, "pi"));
    my_object.may_add(p);
} // p's destructor will only delete the T if number_storage didn't copy
```

may_add may keep a reference to p. If so, memory is freed only when both references are deleted.

Gtk::Manage

- Gtkmm will manage nested widgets for you, deleting them when their container is deleted
  - This greatly simplifies memory management
  - Works with all gtkmm widgets
  - Just create the new widget as a parameter to the static method Gtk::manage, e.g.,
    - Gtk::Box *box = **Gtk::manage(**new Gtk::Box{ }**)**;

Once added to a container, the (gtkmm) object's destructor will be called as part of the container's destructor code.

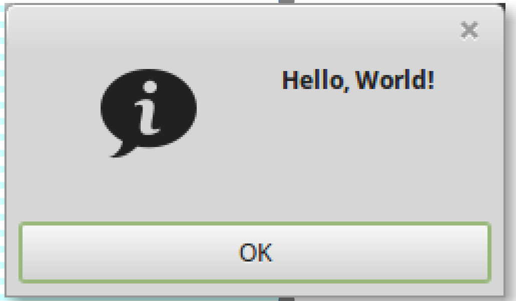
Create a new "box" object on the heap, and return its address (i.e., pointer)

Enough Pointers
Let's Write a GUI!

Writing “Hello, World” in gtkmm

Without pointers

```
#include <gtkmm.h>

int main(int argc, char *argv[])
{
    // Initialize GTK
    Gtk::Main kit(argc, argv);

    // Create a simple dialog containing "Hello, World!"
    Gtk::MessageDialog dialog("Hello, World!");

    // Turn control over to gtkmm until the user clicks OK
    dialog.run();
}
```


Hello, World!

With pointers

```
#include <gtkmm.h>

int main(int argc, char *argv[])
{
    // Initialize GTK
    Gtk::Main kit(argc, argv);

    // Create a simple dialog containing "Hello, World!"
    Gtk::MessageDialog *dialog = new Gtk::MessageDialog{"Hello, World!"};

    // Turn control over to gtkmm until the user clicks OK
    dialog->run();
}
```

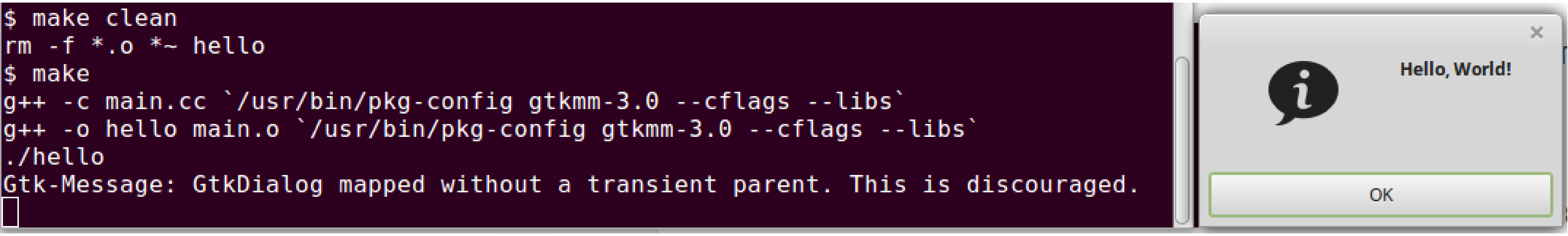
Thoughts on “Hello, World”

- The Gtk::Main instance is traditionally called kit
  - Because it’s a tool *kit*, I suppose
  - Gtk::Main is a Singleton class – every time you “instance” it, you actually get a reference to the same object
- We will use pointers going forward
  - It’s common to create long-lived windows and widgets in a narrow scope
  - If created on the stack, they are automatically deleted when their creating scope exits
  - If created on the heap (i.e., “new” with pointers), the are never automatically deleted
    - But don’t forget to delete them when they are no longer used!

Compiling and Running gtkmm Programs

Makefile

```
# Remember to use the leading TABS, not spaces, for commands!
main: main.o
	g++ -o hello main.o `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
	./hello

main.o: main.cc
	g++ -c main.cc `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`

clean:
	-rm -f *.o *~ hello
```



```
$ make clean
rm -f *.o *~ hello
$ make
g++ -c main.cc `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
g++ -o hello main.o `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
./hello
Gtk-Message: GtkDialog mapped without a transient parent. This is discouraged.
```

Note: The “Gtk-Message” is complaining that we’re displaying a *dialog* without first displaying a *main window* (where the menus and tool bars would be). We’ll ignore this warning until we get to main windows a couple of lectures hence.

Review
A Turtle Graphics Façade

```
class Line {    // Define a line from (x1, y1) to (x2, y2)
     double _x1, _y1, _x2, _y2;
   public:
     Line(double p_x1, double p_y1,double p_x2, double p_y2)
       : _x1{p_x1}, _y1{p_y1}, _x2{p_x2}, _y2{p_y2} { }
     double x1() {return _x1;}
     double y1() {return _y1;}
     double x2() {return _x2;}
     double y2() {return _y2;}
};
class Turtle_graphics {
       const double d2r = M_PI/180.0; // Degrees to radians
       double x, y, angle; // Current turtle position
       bool pen_is_down;
       vector<Line> lines;
    public:
        void penUp() {pen_is_down = false;}
        void penDown() {pen_is_down = true;}
        void turn(double degrees) {angle += degrees * d2r;}
        void forward(double distance) {
          double x2 = x + distance*cos(angle);
          double y2 = y + distance*sin(angle);
          if (pen_is_down) lines.push_back(Line(x, y, x2, y2));
          x = x2;
          y = y2;
        }
        vector<Line> get_lines() {return lines;}
};
```

gtkmm's line drawing capability
is needed for an actual implementation
(now...)

Let's take a quick overview of how to use our façade with the gtkmm library!

Much more gtkmm to follow

Create a gtkmm Drawing Area...

view.h

```
#ifndef _VIEW_H
#define _VIEW_H

#include "line.h"
#include <vector>
#include <gtkmm/drawingarea.h>

using namespace std;

class View : public Gtk::DrawingArea {
     vector<Line> lines;
  public:
     View(vector<Line> L);
     bool on_draw(const Cairo::RefPtr<Cairo::Context>& cr) override;
};
#endif
```

Class View "is a" DrawingArea (much more on this shortly!)

It stores the vector of lines generated by our Turtle Graphics class

It *overrides* (replaces) DrawingArea's drawing method with our method

When Ubuntu needs for a DrawingArea to redraw itself, such as when we resize the window, or minimize and then restore the window, or put another window over it and then bring it back to the front, it calls the DrawingArea's on_draw method.

IMPORTANT: We don't **call** on_draw – we just **write** on_draw.
It's automatically called as needed.

...and Draw On It

view.cpp

```
bool View::on_draw(const Cairo::RefPtr<Cairo::Context>& cr) {
    // If nothing to draw, we're done – EDGE CASE!
    if (lines.size() == 0) return true;

    // Create a Cairomm context to manage our drawing
    Gtk::Allocation allocation = get_allocation();

    // Center the drawing in the window
    const double width = allocation.get_width();
    const double height = allocation.get_height();
    cr->translate(width / 2, height / 2);

    // Use a 3 pixel wide red line
    cr->set_line_width(3.0);
    cr->set_source_rgb(0.8, 0.0, 0.0); // Set lines to red

    // Draw the lines
    for(Line l : lines) {
        cr->move_to(l.x1(), l.y1());
        cr->line_to(l.x2(), l.y2());
    }

    // Apply the colors to the window
    cr->stroke();

    // Drawing was successful
    return true;
}
```

A “context” tracks our “crayon” - color, width, dash pattern, etc.

These are straight lines, but we can also draw (and fill) curves and complex shapes

Nothing appears until we “stroke” (fill in) the lines.

Finally and inevitably, main()

main.cpp

```cpp
int main(int argc, char** argv)
{
   // Create the application
   auto app = Gtk::Application::create(argc, argv, "edu.uta.cse1325.turtle1");

   // Instance a window
   Gtk::Window win;
   win.set_title("Turtle Graphics");

   // Create the vector of lines using Turtle Graphics
   Turtle_graphics g;
   g.penDown();
   for (double d=0; d<20; ++d) {
     g.forward(d*15.0);
     g.turn(144);
   }

   // Instance a drawing surface containing the lines and add to the window
   View view{g.get_lines()};
   win.add(view);
   view.show();

   return app->run(win);
}
```

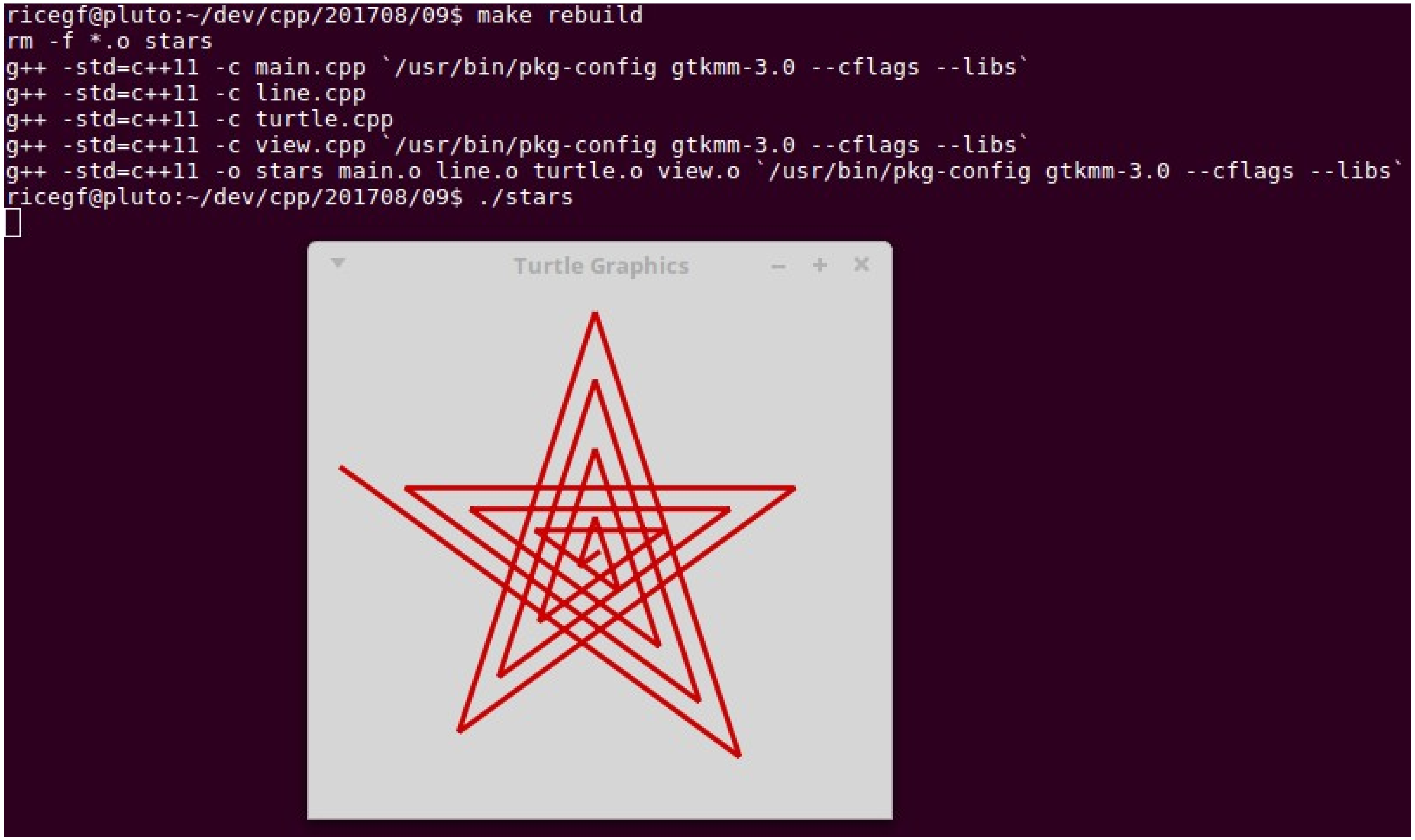
Gtk::Application::create is a static method that handles routine gtkmm application initialization (including Gtk::Main kit).

We draw the same "star" shape as when we introduced the Turtle Graphics façade.

Then we use those lines to instance our turtle's drawing area, add it to the window, and make it visible

Type "auto" just means "use whatever type the assignment is assigning". Convenient!

Turtles in Action!



```
ricegf@pluto:~/dev/cpp/201708/09$ make rebuild
rm -f *.o stars
g++ -std=c++11 -c main.cpp `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
g++ -std=c++11 -c line.cpp
g++ -std=c++11 -c turtle.cpp
g++ -std=c++11 -c view.cpp `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
g++ -std=c++11 -o stars main.o line.o turtle.o view.o `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
ricegf@pluto:~/dev/cpp/201708/09$ ./stars
```

Again, this is just an introductory overview.
We'll cover this a few more times. Don't Panic!

Quick Review

- Put the following user interface technologies in chronological order of introduction: Voice, GUI, Touch / Gesture, CLI, Punch Card, Paper Tape
  - How do web apps fit in?
- What is the Principle of Least Astonishment?
- A pointer variable contains the ________ of the value of interest. Accessing the value of interest via the pointer value is called ________________.
- Memory for the value of interest for pointer variables is usually allocated from the _____ using the _____ keyword, and freed using the ___ keyword.
- Access a member of an object via a pointer uses the ___ operator.
- The __________ pattern implements a simplified interface to a complex class or package.

Quick Review

- What is the primary philosophical difference between CLI and GUI applications?
- Why is the main program loop part of gtkmm rather than written by you?
- To ensure your gtkmm program is compiled and linked using the right libraries, use the ______ tool with a _________.
- How is memory allocated from the stack? How (and when) is it subsequently deallocated?
- How is memory allocated from the heap? How (and when) is it subsequently deallocated?
- When are the two variants of “delete” used?

Next Class

- Review chapter 12 in Stroustrop
  - Notice he is using a façade!
  - Do the drills
- We continue Graphical User Interfaces
  - Create a Dialogs class – display text messages and images, and get text and buttons input from the user
  - We'll use these Dialogs to "GUI-ize" the Library Management System in Sprint #2
- Sprint 1 of Homework 4 (or "the project") is due Thursday, September 28 at 8 am