CSE 1325: Object-Oriented Programming

Lecture 22

Polymorphism

with Scopes, Namespaces, Casting, and Maps

Mr. George F. Rice

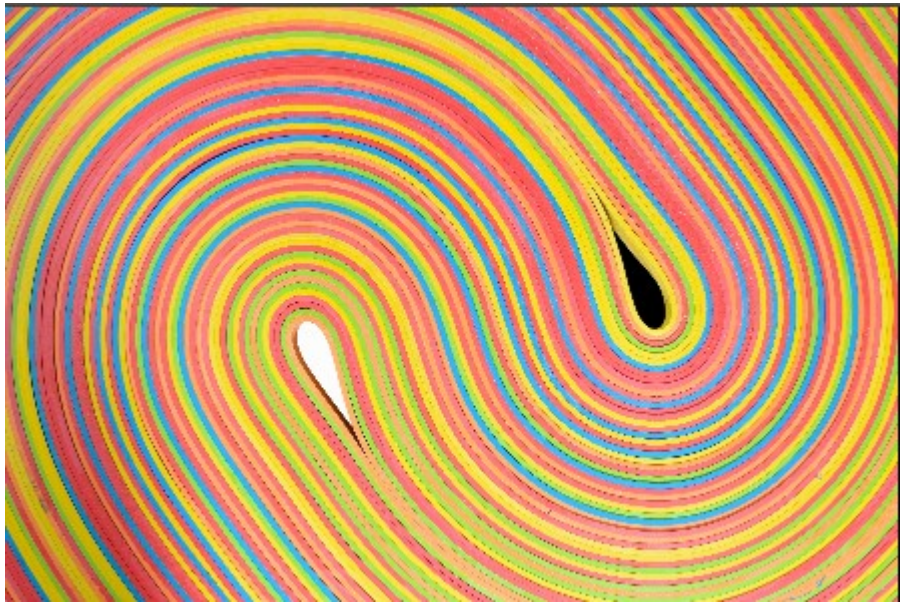
george.rice@uta.edu

Office Hours:
Prof Rice 12:30 Tuesday and Thursday in ERB 336
For TAs see this web page

Some mistakes are too much fun to only make once.

Overview: Polymorphism

- Avoiding Memory Leaks
- Scopes and Namespaces
- Polymorphism
  - Inheritance
  - Virtual methods
  - References and pointers
  - Object memory layout
- Upcasts and Downcasts
  - static_cast
  - dynamic_cast
- Maps



Review

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!
  - Many algorithms trade speed vs fragmentation

Memory Leaks

```
double* calc(int result_size, int max) {
    double* p = new double[max];
    double* result = new double[result_size];
   Sorter* sorter = new Sorter;

    // … use p and sorter to calculate result

    return result;
}

double* r = calc(200,100);
```

See any memory leaks?

Memory Leaks

```
double* calc(int result_size, int max) {
    double* p = new double[max];              // allocate max doubles from the heap
    double* result = new double[result_size]; // allocate result_size more
    Sorter* sorter = new Sorter;              // allocate Sorter object, too!

    // … use p and sorter to calculate result

    return result;
}

double* r = calc(200,100);
```

- Three “new” allocations, zero “delete” deallocations = 3 LEAKS
- Lack of de-allocation (usually called “memory leaks”) can be a serious problem in real-world programs

Fixing the Memory Leaks

```
double* calc(int result_size, int max) {
    double* p = new double[max];              // allocate max doubles from the heap
    double* result = new double[result_size]; // allocate result_size more
    Sorter* sorter = new Sorter;              // allocate Sorter object, too!
    // … use p and sorter to calculate result
    delete[] p;            // mark the heap array that p points to as free
    delete sorter;         // mark the sorter object as free
    return result;
}

double* r = calc(200,100);
delete[] r;                // this one is not at all obvious!
```

- “delete” de-allocates a simple variable or object
- “delete[ ]” de-allocates an array

Another Form of Memory Leak

```
void f()
{
    double* p = new double[27];
    // …

    p = new double[42];
    // …

    delete[] p;          // delete the allocated heap memory
}
```

See any memory leaks?
(Note that we deleted p this time!)

Another Form of Memory Leak

```
void f()
{
    double* p = new double[27];
    // …

    p = new double[42];  // allocate 42 more, overwriting the pointer
    // …

    delete[] p;          // delete the allocated heap memory
}
```

- The second allocation overwrites the first pointer
- The delete[ ] only de-allocates the second allocation
  - The double[27] is leaked

Fixing Another Form of Memory Leak

```
void f()
{
    double* p = new double[27];   // allocate 27 doubles
    // …
    delete[] p;
    p = new double[42];  // allocate 42 more
    // …

    delete[] p;      // now delete the allocated heap memory
}
```

- **Newton's Third Law of C++:** For every **`new`**, there is an equal and opposite **`delete`***

* Of *course* I made that up.
But it's still true if you don't want memory leaks!

Memory Leak Avoidance Strategies

- Use garbage collection (Hello again, Java!)
  - C++ had an optional one, but it has been deprecated due to non-use
- Don’t use new and delete if possible
  - Rely on std::vector and similar mature higher-level classes
- Use destructors… *very* carefully
  - Because it’s *very* hard to ensure the destructor *always* runs before the pointer is lost
- Use smart pointers (not required on the exam or assignments)
  - Use `std::unique_ptr<T> p{new T(42, "meaning")};` for a single pointer
  - Use `std::shared_ptr<T> p(new T(3.14, "pi"));` for a pointer that may be copied (relies on reference counting – watch out for circular references!)
- Use library-specific facilities such as Gtk::manage (which in this example is part of the graphics library gtkmm)

Scope

- A **scope** is a region of program text
  - Global scope (outside any language construct)
  - Class scope (within a class)
  - Local scope (between any { … } braces)
  - Statement scope (unique to the 3-term for statement)

- A name in a scope can be seen from within its scope and within scopes nested within that scope
  - Only after the declaration of the name ("can't look ahead" rule) EXCEPT for class scope which, like Java, is bi-directional
- A scope keeps "things" local
  - Prevents my variables, functions, etc., from interfering with yours
  - Remember: real programs have **many** thousands of entities
  - Locality is good!

Keep names as local as possible

Normal vs Class Scope

- Class scope, unlike the other 3 types, is *bidirectional*
  - Every class member can see all other members

scopes.cpp

```
// no r, i, or v here – they are “out of scope”

class Scopes {
    std::vector<int> my_v& = v; // v is visible here – class scope is bi-directional
    std::vector<int> v;         // v is in class scope
  public:
    int largest() {           // largest is in class scope
        int r = 0;            // r is in local scope to the largest method
        for (int i = 0; i<v.size(); ++i) {  // i is in statement scope
            r = max(r,abs(v[i]));
        }
        // no i here
        return r;
    }
    // no r here
};
// no v here – although the largest method is visible as My_vector::largest
//   (another name for :: is “scope resolution operator”)
```

Nested Scopes

nested_scopes.cpp

```
#include <iostream>
int x = 0;   // global variable – always avoid these
int y = 0;   // another global variable

void f() {
    int x;          // local scope variable (Note – now there are two x's)
    x = 7;          // local x, not the global x
    int& fx = x;    // keep a reference (sorta like a pointer) to f's x
    std::cout << "f's x = " << x << " but global x = " << ::x << std::endl;
    {
        int x = y;  // another local x, initialized by the global y
                    // (Now there are three x's)
        std::cout << "Before increment, nested x = " << x
                  << ", while f's x = " << fx
                  << ", while global x = " << ::x << std::endl;
        ++x;        // increment the local x in this scope
        std::cout << "After  increment, nested x = " << x
                  << ", while f's x = " << fx
                  << ", while global x = " << ::x << std::endl;
    }
    std::cout << "Now, f's x = " << x << " but global x = " << ::x << std::endl;
}
int main() {
  std::cout << "global x = " << x << std::endl;
  f();
  std::cout << "global x = " << x << std::endl;
}
```

Note that unlike Java, C++ allows arbitrary creation of new local scopes – very useful for regression testing!

Nested Scopes

```
#include <iostream>
int x = 0;   // global variable – always avoid these
int y = 0;   // another global variable

void f() {
    i
    x
    i
    s
    {
    }
    std::cout << "Now, f's x = " << x << " but global x = " << ::x << std::endl;
}
int main() {
  std::cout << "global x = " << x << std::endl;
  f();
  std::cout << "global x = " << x << std::endl;
}
```

```
ricegf@pluto:~/dev/cpp/201901/05$ make nested_scopes
g++ --std=c++17 -o nested_scopes nested_scopes.cpp
ricegf@pluto:~/dev/cpp/201901/05$ ./nested_scopes
global x = 0
f's x = 7 but global x = 0
Before increment, nested x = 0, while f's x = 7, while global x = 0
After  increment, nested x = 1, while f's x = 7, while global x = 0
Now, f's x = 7 but global x = 0
global x = 0
ricegf@pluto:~/dev/cpp/201901/05$
```

Shadowing – A variable declared in a narrower scope than that of a variable of the same name declared in a broader scope

Duplicate Definition is a Scope Problem

- Consider this code from two programmers Jack and Jill

jack.h

```
class Glob { /*…*/ };       // in Jack's header file jack.h
class Widget { /*…*/ };
```

jill.h

```
class Blob { /*…*/ };       // in Jill's header file  jill.h
class Widget { /*…*/ };
```

awesome_app.h

```
#include "jack.h"          // this is in your code
#include "jill.h"

void activate(Widget p) {   // oops! – error: multiple definitions of Widget
    // …
}
```

Namespaces Manage Scope

- The compiler will not compile multiple definitions; such clashes can occur from multiple headers.
- One way to prevent this problem is with namespaces:

jack.h

```
namespace Jack {
  class Glob { /*…*/ };      // in Jack's header file jack.h
  class Widget { /*…*/ };
}
```

jill.h

```
namespace Jill {
  class Blob { /*…*/ };      // in Jill's header file  jill.h
  class Widget { /*…*/ };
}
```

awesome_app.h

```
#include "jack.h"          // this is in your code
#include "jill.h"

void my_func(Jack::Widget p) {   // No collision!
    // …
}
```

:: means *membership* – Jack's Widget!

Namespace Declarations Append Automatically

- You may append names repeatedly to an existing namespace
  - This is how so many different files append members to namespace std

jack.cpp

```
#include <iostream>

namespace Jack {
  class Glob { /*…*/ };
  class Widget { /*…*/ };
}

class Foo {public: std::string speak() {return "Default!";}};  // In default namespace
class Bar { /*…*/ };

namespace Jack {
  class Foo {public: std::string speak() {return "Jack!";}};
  class Bar { /*…*/ }; // Jack contains 4 classes now
}

int main() {
    Foo f;
    Jack::Foo jf;
    std::cout << "Default namespace foo says " << f.speak() << std::endl;
    std::cout << "Jack's foo says " << jf.speak() << std::endl;
}
```

```
ricegf@pluto:~/dev/cpp/201901/05$ make jack
g++ --std=c++17 -o jack jack.cpp
ricegf@pluto:~/dev/cpp/201901/05$ ./jack
Default namespace foo says Default!
Jack's foo says Jack!
ricegf@pluto:~/dev/cpp/201901/05$
```

Namespaces

- A namespace is simply a named scope
- :: is also the "**namespace resolution operator**" that specifies:
  - Which namespace you are using
  - To which (of many possible) objects of the same name you are referring
- For example, since **cout** is in namespace **std**, we write:

```
std::cout << "Please enter stuff… \n";
```

- This is *very* preferable to "using namespace std; cout << …"
  - With :: we *know* to which object the code refers!
  - Avoids "namespace pollution"

Bad programmer! No donut!

Namespace – a named scope

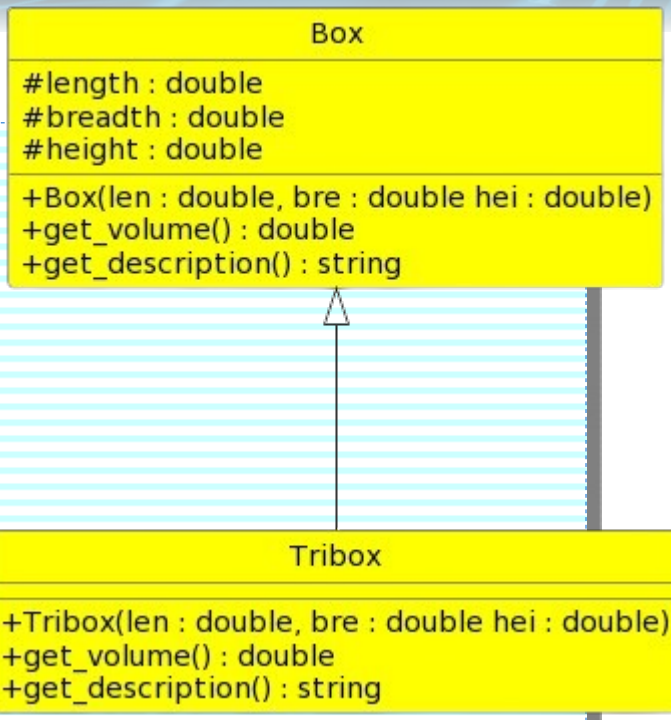
The Box Interface

- We created a Box class in Java to demonstrate polymorphism
  - Here's a similar class in C++

```
#include <iostream>

class Box {
  public:
     Box (double len, double bre, double hei)
        : length(len), breadth(bre), height(hei) { }

     double get_volume(void) {return length * breadth * height;}
     std::string get_description() {return "Rectangular box";}

  private:
     double length;      // Length of a box
     double breadth;     // Breadth of a box
     double height;      // Height of a box
};
```

Example: Inheriting the Box Interface

```
class Box {
  public:
     Box (double len, double bre, double hei)
        : length(len), breadth(bre), height(hei) { }

     double get_volume() {return length * breadth * height;}
     std::string get_description() {return "Rectangular box";}

  protected:
     double length;       // Length of a box
     double breadth;      // Breadth of a box
     double height;       // Height of a box
};

class Tribox : public Box {
   public:
      Tribox (double len, double bre, double hei)
        : Box(len, bre, hei) { }

     double get_volume() {return length * breadth * height / 2;}
     std::string get_description() {return "Triangular box";}
};
```



Testing Box and Tribox

```
student@cse1325:/media/sf_dev/19$ make box_nonpoly
g++ -std=c++17 -o box_nonpoly box_nonpoly.cpp
student@cse1325:/media/sf_dev/19$ ./box_nonpoly
Rectangular box
Box 1: volume 210
Box 2: volume 1560

Triangular box
Box 1: volume 105
Box 2: volume 780
```

box_nonpoly.cpp

```
// Test the Box superclass
// and Tribox subclass

int main( ) {
   Box box1(6.0, 7.0, 5.0);
   Box box2(12.0, 13.0, 10.0);

   std::cout << box1.get_description() << std::endl;
   std::cout << "Box 1: volume " << box1.get_volume() << std::endl;
   std::cout << "Box 2: volume " << box2.get_volume() << std::endl << std::endl;

   Tribox tbox1(6.0, 7.0, 5.0);
   Tribox tbox2(12.0, 13.0, 10.0);

   std::cout << tbox1.get_description() << std::endl;
   std::cout << "Box 1: volume " << tbox1.get_volume() << std::endl;
   std::cout << "Box 2: volume " << tbox2.get_volume() << std::endl << std::endl;
}
```

Here we create objects of the subclass and use them directly. Some of the features of the superclass inherit to subclass scope – the protected data, for example – and some do not – the get_volume() method. We could also add additional fields and methods to the subclass as needed.

Do the Tribox Methods Override?

box_nonpoly_override.cpp

```
class Box {
  public:
     Box (double len, double bre, double hei)
        : length(len), breadth(bre), height(hei) { }
     double get_volume(void) {return length * breadth * height;}
     std::string get_description() {return "Rectangular box";}

  protected:
     double length;      // Length of a box
     double breadth;     // Breadth of a box
     double height;      // Height of a box
};

class Tribox : public Box {
   public:
      Tribox (double len, double bre, double hei)
        : Box(len, bre, hei) { }
     double get_volume(void) override {return length * breadth * height / 2;}
     std::string get_description() override {return "Triangular box";}
};
```

Override – A subclass replacing its superclass' implementation of a method

Question: Do Tribox::get_volume and Tribox::get_description *override* the same methods in the superclass box, or are they distinct methods? Why?

Non-Virtual Methods Never Override in C++

```
class Box {
  public:
     Box (double len, double bre, double hei)
        : length(len), breadth(bre), height(hei) { }
     double get_volume(void) {return length * breadth * height;}
     std::string get_description() {return "Rectangular box";}

  protected:
     double length;      // Length of a box
     double breadth;     // Breadth of a box
     double height;      // Height of a box
};

class Tribox : public Box {
   public:
      Tribox (double len, double bre, double hei)
        : Box(len, bre, hei) { }
     double get_volume(void) override {return length * breadth * height / 2;}
     std::string get_description() override {return "Triangular box";}
};
```

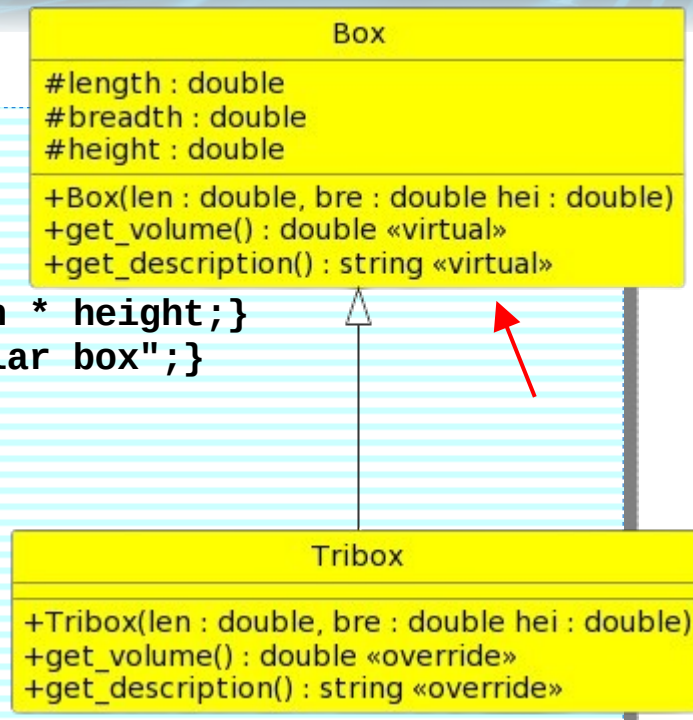
They are distinct methods.

```
ricegf@antares:~/dev/202308/20-polymorphism/code_from_slides$ c17 box_nonpoly_override.cpp
box_nonpoly_override.cpp:22:13: error: 'double Tribox::get_volume()' marked 'override', but does not override
   22 |      double get_volume(void) override {return length * breadth * height / 2;}
      |             ^~~~~~~~~~
box_nonpoly_override.cpp:23:18: error: 'std::string Tribox::get_description()' marked 'override', but does not override
   23 |      std::string get_description() override {return "Triangular box";}
      |                  ^~~~~~~~~~~~~~~
ricegf@antares:~/dev/202308/20-polymorphism/code_from_slides@
```

Virtual Methods Override in C++

box_override.cpp

```
class Box {
  public:
     Box (double len, double bre, double hei)
        : length(len), breadth(bre), height(hei) { }
     virtual double get_volume(void) {return length * breadth * height;}
     Virtual std::string get_description() {return "Rectangular box";}

  protected:
     double length;      // Length of a box
     double breadth;     // Breadth of a box
     double height;      // Height of a box
};

class Tribox : public Box {
   public:
      Tribox (double len, double bre, double hei)
        : Box(len, bre, hei) { }
     double get_volume(void) override {return length * breadth * height / 2;}
     std::string get_description() override {return "Triangular box";}
};
```



In C++, we fix this by declaring the superclass methods to be *virtual*. This means access to those methods will be via a table of pointers (the "vtable") rather than direct.

Virtual Methods Override in C++

```
class Box {
  public:
     Box (double len, double bre, double hei)
        : length(len), breadth(bre), height(hei) { }
     virtual double get_volume(void) {return length * breadth * height;}
     virtual std::string get_description() {return "Rectangular box";}

  protected:
     double length;      // Length of a box
     double breadth;     // Breadth of a box
     double height;      // Height of a box
};

class Tribox : public Box {
   public:
      Tribox (double len, double bre, double hei)
        : Box(len, bre, hei) { }
     double get_volume(void) override {return length * breadth * height / 2;}
     std::string get_description() override {return "Triangular box";}
};
```

They now override.

```
student@cse1325:/media/sf_dev/19$ make box_override
g++ -std=c++17  -o box_override box_override.cpp
student@cse1325:/media/sf_dev/19$ ./box_override
Rectangular box
Box 1: volume 210
Box 2: volume 1560

Triangular box
Box 1: volume 105
Box 2: volume 780
```

Storing a Subclass Object in a Superclass Variable

box_nonpoly_assign.cpp

```
int main( ) {
   Box box1(6.0, 7.0, 5.0);
   Box box2(12.0, 13.0, 10.0);

   std::cout << box1.get_description() << std::endl;
   std::cout << "Box 1: volume " << box1.get_volume() << std::endl;
   std::cout << "Box 2: volume " << box2.get_volume() << std::endl << std::endl;

   Tribox tbox1(6.0, 7.0, 5.0);
   Tribox tbox2(12.0, 13.0, 10.0);

   std::cout << tbox1.get_description() << std::endl;
   std::cout << "Box 1: volume " << tbox1.get_volume() << std::endl;
   std::cout << "Box 2: volume " << tbox2.get_volume() << std::endl << std::endl;

   box1 = tbox1;  // Since a Tribox “isa” Box, this is a legal assignment
   box2 = tbox2;  // But do box1 and box2 now contain boxes or triboxes?

   std::cout << box1.get_description() << std::endl;
   std::cout << "Box 1: volume " << box1.get_volume() << std::endl;
   std::cout << "Box 2: volume " << box2.get_volume() << std::endl << std::endl;

}
```

So what will the lines in red print?

Storing a Subclass Object in a Superclass Variable

```
int main( ) {
    Box box1(6.0, 7.0, 5.0);
    Box box2(12.0, 13.0, 10.0);

    std::cout << box1.get_description() << std::endl;
    std::cout << "Box 1: volume " << box1.get_volume() << std::endl;
    std::cout
    Tribox tbo
    Tribox tbo
    std::cout
    std::cout
    std::cout
    box1 = tbo
    box2 = tbo
    std::cout
    std::cout
    std::cout << "Box 2: volume " << box2.get_volume() << std::endl << std::endl;
}
```

```
ricegf@antares:~/dev/202308/20-polymorphism/code_from_slides$ c17 box_nonpoly_assign.cpp
ricegf@antares:~/dev/202308/20-polymorphism/code_from_slides$ ./a.out
Rectangular box
Box 1: volume 210
Box 2: volume 1560

Triangular box
Box 1: volume 105
Box 2: volume 780

Rectangular box
Box 1: volume 210
Box 2: volume 1560

ricegf@antares:~/dev/202308/20-polymorphism/code_from_slides$
```

tbox1 and tbox2 became Box objects when assigned to variables of type Box!

In C++, an object instanced from a subclass type but *directly* assigned to a variable of the superclass becomes an object of the superclass. We'll explain why in a moment. This is different from other OO languages such as Java and Python, and may be counter-intuitive.

Review

Pointers vs References

- A reference is...
  - An automatically dereferenced pointer
  - A constant pointer whose address can’t change

```
int x = 7;
int y = 8;
int* p = &x; *p = 9;
p = &y;  // ok
int& r = x;  x = 10;
r = &y;  // error
```

  - An alias for another object

```
class View {
  public:
    View(Model& model) : _model{model} { }
  private:
    Model& _model;
}
```

Inheriting Outside the Box

box_poly.cpp

```
void print_volume(Box b) {
   std::cout << b.get_description() << " volume is " << b.get_volume() << std::endl;
}
void print_volume_ref(Box& b) {
   std::cout << b.get_description() << " volume is " << b.get_volume() << std::endl;
}
void print_volume_p(Box* b) {
   std::cout << b->get_description() << " volume is " << b->get_volume() << std::endl;
}

// Test the Box class
int main( ) {
   Box box1(6.0, 7.0, 5.0);
   Tribox tbox1(6.0, 7.0, 5.0);

   std::cout << "Access subclass methods directly:" << std::endl;
   print_volume(box1);
   print_volume(tbox1);

   std::cout << "Access subclass methods via reference:" << std::endl;
   print_volume_ref(box1);
   print_volume_ref(tbox1);

   std::cout << "Access subclass methods via pointers:" << std::endl;
   print_volume_p(&box1);
   print_volume_p(&tbox1);
}
```

Accept a **copy** of a Box and print its volume.

Accept a **reference** to a Box and print its volume.

Accept a **pointer** to a Box and print its volume.

Direct variable version is first.

Reference variable version is second.

Pointer variable version is third.

Two out of Three Isn't Bad?

```
void print_volume(Box b) {
   std::cout << b.get_description() << " vol
}

void print_volume_ref(Box& b) {
   std::cout << b.get_description() << " vol
}

void print_volume_p(Box* b) {
   std::cout << b->get_description() << " vo
}

// Test the Box class
int main( ) {
   Box box1(6.0, 7.0, 5.0);
   Tribox tbox1(6.0, 7.0, 5.0);

   std::cout << "Access subclass methods directly:" << std::endl;
   print_volume(box1);
   print_volume(tbox1);

   std::cout << "Access subclass methods via reference:" << std::endl;
   print_volume_ref(box1);
   print_volume_ref(tbox1);

   std::cout << "Access subclass methods via pointers:" << std::endl;
   print_volume_p(&box1);
   print_volume_p(&tbox1);
}
```

```
student@cse1325:/media/sf_dev/19$ make box_poly
g++ -std=c++17  -o box_poly box_poly.cpp
student@cse1325:/media/sf_dev/19$ ./box_poly

Access derived class methods directly:
Rectangular box volume is 210
Rectangular box volume is 210

Access derived class methods via reference:
Rectangular box volume is 210
Triangular box volume is 105

Access derived class methods via pointers:
Rectangular box volume is 210
Triangular box volume is 105
student@cse1325:/media/sf_dev/19$
```

Direct variable version is first.

Reference variable version is second.

Pointer variable version is third.

Polymorphism

- **Polymorphism** – The provision of a single interface to multiple subclasses, enabling the same method call to invoke different subclass methods to generate different results.

- In C++, this requires:
  - Virtual methods in the superclass to allow overriding
  - A superclass type variable referencing (or pointing to) an object of the subtype

Object Layout in Memory

mem_layout.cpp

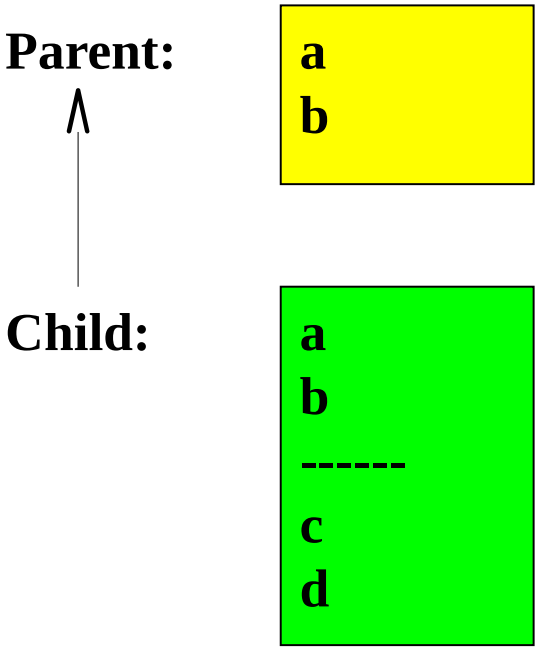
```cpp
#include <iostream>

class Parent{
public:
    int a, b;
    virtual void foo(){std::cout << "parent" << std::endl;}
};

class Child : public Parent {
public:
    int c, d;
    virtual void foo()
      {std::cout << "child" << std::endl;}
};

int main() {
    Parent p; p.foo();
    Child c; c.foo();

    std::cout << "Parent Offset a = " << (size_t)&p.a - (size_t)&p << std::endl;
    std::cout << "Parent Offset b = " << (size_t)&p.b - (size_t)&p << std::endl;

    std::cout << "Child Offset a = " << (size_t)&c.a - (size_t)&c << std::endl;
    std::cout << "Child Offset b = " << (size_t)&c.b - (size_t)&c << std::endl;
    std::cout << "Child Offset c = " << (size_t)&c.c - (size_t)&c << std::endl;
    std::cout << "Child Offset d = " << (size_t)&c.d - (size_t)&c << std::endl;
}
```

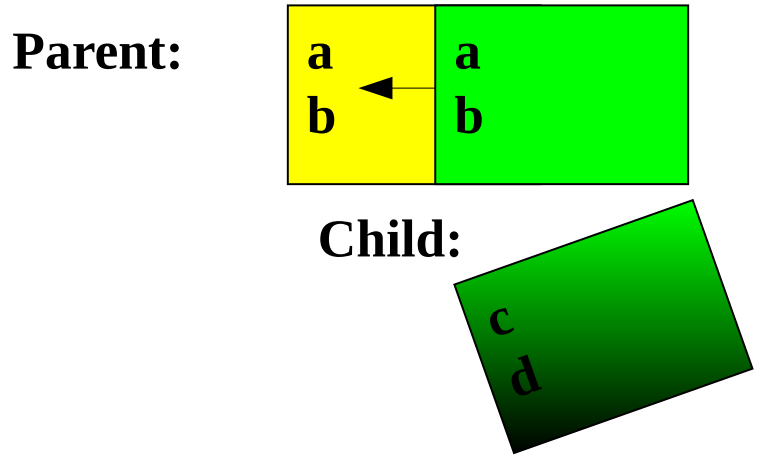
```
student@cse1325:/media/sf_dev/19$ make mem_layout
g++ -std=c++17    mem_layout.cpp   -o mem_layout
student@cse1325:/media/sf_dev/19$ ./mem_layout
parent
child
Parent Offset a = 8
Parent Offset b = 12
Child Offset a = 8
Child Offset b = 12
Child Offset c = 16
Child Offset d = 20
student@cse1325:/media/sf_dev/19$
```

http://stackoverflow.com/questions/8672218/memory-layout-of-inherited-class

Object Layout - Copy

- The data members of a subclass are simply added at the end of its superclass
- If a Child object is stored in a Parent variable, only enough memory exists to hold the Parent's fields, so (by default*) only those fields are copied – the rest are discarded

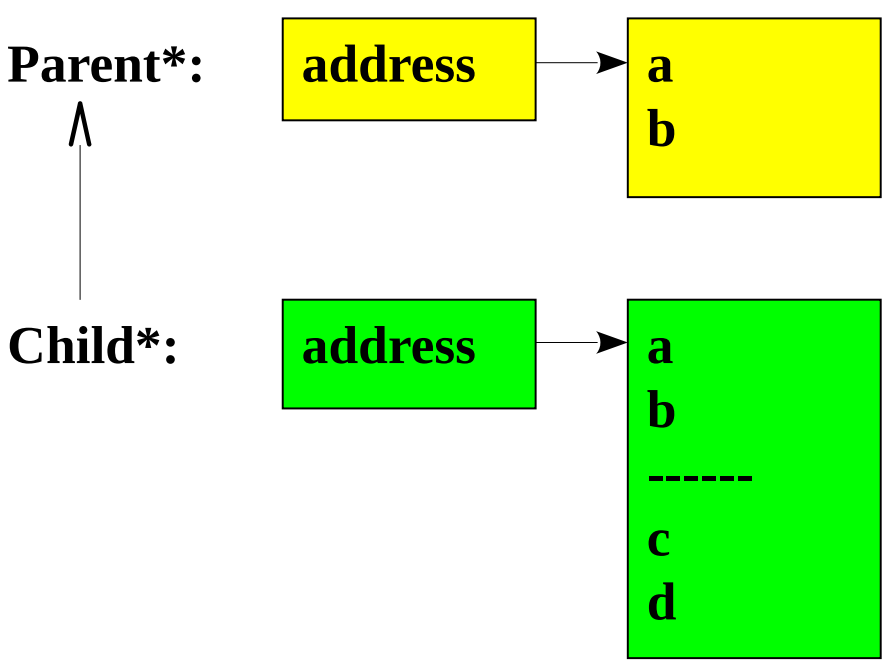


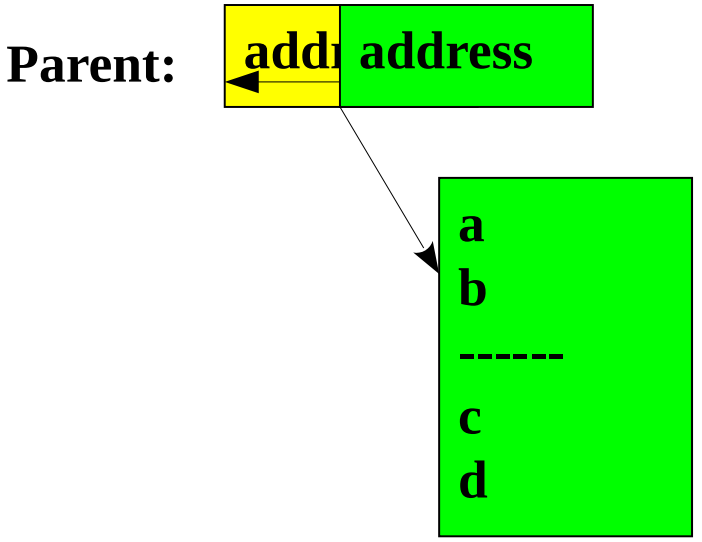


* We'll cover non-default options in a future lecture

Object Layout - Pointer

- When using a pointer or reference, only the Child object's address is copied to the Parent variable. None of the Child's data is lost.





Note that references work as well as pointers here.

C++ (Incorrect) Upcast

- Upcasting as in Java no longer works

upcast_bad.cpp

```
#include <iostream>

class Superclass {
  public:
    virtual std::string who_am_i() {
        return "Superclass";
    }
};
class Subclass : public Superclass {
  public:
    std::string who_am_i() override {
        return "Subclass";
    }
};
int main() {
    Subclass subclass{};
    std::cout << "Subclass   variable, Subclass object: "
              << subclass.who_am_i() << std::endl;

    // upcast
    superclass superclass = subclass;
    std::cout << "Superclass variable, Subclass object: "
              << superclass.who_am_i() << std::endl;
}
```

`virtual` helps, but isn't enough...

When a subclass object
is assigned to a superclass variable,
it *becomes* a superclass object!

```
Subclass   variable, Subclass object: Subclass
Superclass variable, Subclass object: Superclass
```

C++ (Correct!) Upcast

- Upcasting works only with pointers / references!

upcast.cpp

`virtual` and *pointers* for the win!

```
#include <iostream>

class Superclass {
  public:
    virtual std::string who_am_i() {
        return "Superclass";
    }
};
class Subclass : public Superclass {
  public:
    std::string who_am_i() override {
        return "Subclass";
    }
};
int main() {
    Subclass* subclass = new Subclass;
    std::cout << "Subclass   variable, Subclass object: "
              << subclass->who_am_i() << std::endl;

    // upcast
    Superclass* superclass = subclass;
    std::cout << "Superclass variable, Subclass object: "
              << superclass->who_am_i() << std::endl;
}
```

When a *pointer* to a subclass object is assigned to a superclass *pointer*, it is preserved and upcasting works!

```
Subclass   variable, Subclass object: Subclass
Superclass variable, Subclass object: Subclass
```

Direct Downcasts Fail
(Of course!)

- This fails in all languages I know
  - **`-fpermissive`** flag would make it work – but don't!

downcast_direct.cpp

```
int main() {
    Superclass* superclass = new Subclass{};
    std::cout << "Superclass variable, Subclass object: "
              << superclass->who_am_i() << std::endl;

    // Downcast directly
    Subclass* subclass = superclass;
    std::cout << "Subclass   variable, Subclass object: "
              << subclass->who_am_i() << std::endl;
}
```

```
ricegf@antares:~/dev/202308/19-c++-oop/code_from_slides/cpp_inheritance$ c17 downcast_direct.cpp
downcast_direct.cpp: In function 'int main()':
downcast_direct.cpp:22:26: error: invalid conversion from 'Superclass*' to 'Subclass*' [-fpermissive]
   22 |     Subclass* subclass = superclass;
      |                          ^~~~~~~~~~
      |                          |
      |                          Superclass*
ricegf@antares:~/dev/202308/19-c++-oop/code_from_slides/cpp_inheritance@
```

C-Style Downcasts Work
(Of course!)

- But cue the Jaws music anyway – danger lurks just beneath the surface...

downcast_c_style.cpp

```
int main() {
    Superclass* superclass = new Subclass{};
    std::cout << "Superclass variable, Subclass object: "
              << superclass->who_am_i() << std::endl;

    // C-Style Downcast
    Subclass* subclass = (Subclass*) superclass;
    std::cout << "Subclass   variable, Subclass object: "
              << subclass->who_am_i() << std::endl;
}
```

It works here...

```
ricegf@antares:~/dev/202308/19-c++-oop/code_from_slides/cpp_inheritance@ c17 downcast_c_style.cpp
ricegf@antares:~/dev/202308/19-c++-oop/code_from_slides/cpp_inheritance$ ./a.out
Superclass variable, Subclass object: Subclass
Subclass   variable, Subclass object: Subclass
ricegf@antares:~/dev/202308/19-c++-oop/code_from_slides/cpp_inheritance$
```

C-Style Downcasts Work
(WAY too often!)

- C-style downcasts are *too* flexible
  - They work when they shouldn't!

downcast_bad_pointer.cpp

```
int main() {
    std::vector<std::string>* v = new std::vector<std::string>
        {"This", "will", "end", "badly"};
    for(auto s : *v) std::cout << s << ' ';
    std::cout << std::endl;

    // C-style downcast with pointers - compiles! What could go wrong?
    std::string* s = (std::string*) (v);
    std::cout << *s << std::endl;
}
```

Wait… what??? Why can you cast a vector of string to a string? NOT the same!

```
ricegf@antares:~/dev/202308/19-c++-oop/code_from_slides/cpp_inheritance$ c17 downcast_bad_pointer.cpp
ricegf@antares:~/dev/202308/19-c++-oop/code_from_slides/cpp_inheritance$ ./a.out
This will end badly
;VThis;Vwill ;Vend@;VbadlyThis will end badly
ricegf@antares:~/dev/202308/19-c++-oop/code_from_slides/cpp_inheritance$
```

Ouch!

C++-Style Downcasts Work
(But only when they should!)

- C++-style downcasts perform additional checks
  - They properly fail when the types aren't compatible

downcast_bad_static_cast.cpp

```
int main() {
    std::vector<std::string>* v = new std::vector<std::string>
        {"This", "will", "end", "badly"};
    for(auto s : *v) std::cout << s << ' ';
    std::cout << std::endl;

    // C++-style static_cast - generates a compile error even with pointers
    std::string* s = static_cast<std::string*> (v);
    std::cout << *s << std::endl;
}
```

```
ricegf@antares:~/dev/202308/19-c++-oop/code_from_slides/cpp_inheritance$ c17 downcast_bad_static_cast.cpp
downcast_bad_static_cast.cpp: In function 'int main()':
downcast_bad_static_cast.cpp:12:22: error: invalid 'static_cast' from type 'std::vector<std::__cxx11::basic_string<char> >*'
 to type 'std::string*' {aka 'std::__cxx11::basic_string<char>*'}
   12 |     std::string* s = static_cast<std::string*>(v);
      |                      ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
ricegf@antares:~/dev/202308/19-c++-oop/code_from_slides/cpp_inheritance@
```

C++-Style static_cast Works
(Even better than C-style downcasts!)

- *Always* downcast with static_cast
  - Except for polymorphic downcasts – next!

```
int main() {
    Superclass* superclass = new Subclass{};
    std::cout << "Superclass variable, Subclass object: "
              << superclass->who_am_i() << std::endl;

    // C-Style Downcast
    Subclass* subclass = (Subclass*) superclass;
    std::cout << "Subclass   variable, Subclass object: "
              << subclass->who_am_i() << std::endl;
}
```

C-style
(do NOT use!)

```
int main() {
    Superclass* superclass = new Subclass{};
    std::cout << "Superclass variable, Subclass object: "
              << superclass->who_am_i() << std::endl;

    // C++-style downcast
    Subclass* subclass = static_cast<Subclass*> (superclass);
    std::cout << "Subclass   variable, Subclass object: "
              << subclass->who_am_i() << std::endl;
}
```

C++-style

C-style cast and static_cast and... dynamic_cast

- Consider this example
  - Class A single-inherits Superclass, while class B multiple-inherits Superclass2 and then Superclass
  - Thus the method table is *different* for classes A & B!

downcast_dynamic2.cpp

```
class Superclass {
  public: virtual void hi() {std::cout << "Superclass" << std::endl;}
};

class Superclass2 {
  public: virtual void hey() {std::cout << "Superclass too!" << std::endl;}
};


class A : public Superclass {
  public: void hi() override {std::cout << "A" << std::endl;}
};

class B : public Superclass2, public Superclass {
  public: void hi() override {std::cout << "B" << std::endl;}
  public: void hey() override {std::cout << "B2" << std::endl;}
};
```

dynamic_cast Example

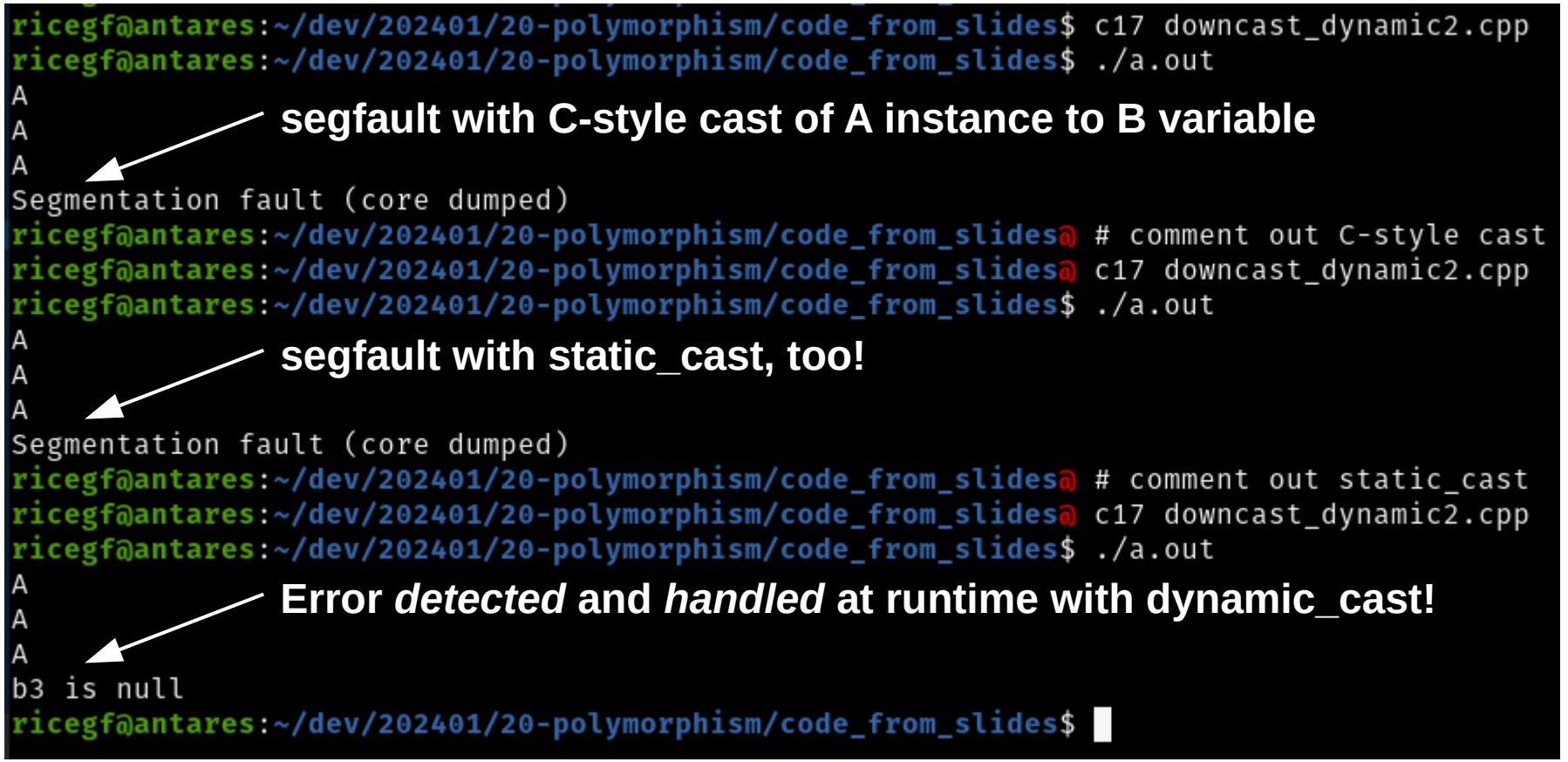
downcast_dynamic2.cpp

Valid downcasts

Invalid downcasts!

```
int main() {
  Superclass* a = new A;        // Superclass variable points to subclass object A
                                //    (polymorphism!)
  A* a1 = (A*)(a);                          // C-style downcast compiles
  if (a1 == nullptr) std::cout << "a1 is null" << std::endl; // perfectly valid
  else a1->hi();

  A* a2 = static_cast<A*>(a);               // C++-style static downcast compiles
  if (a2 == nullptr) std::cout << "a2 is null" << std::endl; // perfectly valid
  else a2->hi();

  A* a3 = dynamic_cast<A*>(a);              // C++-style dynamic downcast compiles
  if (a3 == nullptr) std::cout << "a3 is null" << std::endl; // perfectly valid
  else a3->hi();                                             // but a little slower

                                // Now (incorrectly!) cast object A to variable of B
  B* b1 = (B*)(a);              // Compiles, but undefined at runtime - A is NOT a B
  if (b1 == nullptr) std::cout << "b1 is null" << std::endl;
  else b1->hi();

  B* b2 = static_cast<B*>(a);  // Compiles, but undefined at runtime - A is NOT a B
  if (b2 == nullptr) std::cout << "b2 is null" << std::endl;
  else b2->hi();

  B* b3 = dynamic_cast<B*>(a); // Compiles, but ERROR reported at runtime: b3 == null
  if (b3 == nullptr) std::cout << "b3 is null" << std::endl;
  else b3->hi();
}
```

dynamic_cast Error Detection



(Note: If you are using polymorphism through references rather than pointers, dynamic_cast will throw a **`std::bad_cast`** exception instead of returning null, since a reference cannot be null.)

Casting Bottom Line

- C-style casting is naïve
  - Best to never use in C++ programs
- **static_cast** does some additional error checking
  - Preferred for C++ non-polymorphic types
- **dynamic_cast** does more extensive error checking, which takes some additional time
  - But the source type must be polymorphic (at least one virtual method, accessed via pointer or reference)
  - Preferred for C++ polymorphic types
- Other casts exist (e.g., const_cast and reinterpret_cast), but those won't be covered this semester

std::map

- **map** is a collection of key-value pairs sorted by keys (like Java's TreeMap)
  - Essentially a vector of objects with (almost) any type as the key *including* primitives, and always sorted
  - Elements are accessed like a vector, e.g., m["bar"]
- You may need to provide a comparator method such as **operator<**
  - This is equivalent to Java's Comparable interface
  - The spaceship may be your friend! <=>

Common std::map Operations

- **m.empty()** is true if the map contains no pairs
  - **m.clear()** removes all pairs from the map
- **m.size()** returns the number of pairs in the map
- **x = m[key]** provides random access by key, **adding a default constructor value to the map** if the key isn't already in the map
  - **m.at(key)** is like operator[], but instead throws an std::out_of_range exception if the key isn't in the map
- **m[key] = x** silently overwrites the existing value (if any) for a key
  - **m.count(key)** returns 1 if the key exists, 0 otherwise
  - **m.erase(key)** removes the key and associated value from the map
- **for(auto& [key, value] : m)** iterates over the map, setting variables key and value to the (key, value) pair of each map entry
- **begin() and end()** return "iterators" to the first and last key/value pair, which behave like pointers (soon!)

Simple Map Example

```
student@cse1325:/media/sf_dev/20/code_from_slides/maps$ make map_easy
g++ -std=c++17 -o map_easy map_easy.cpp
student@cse1325:/media/sf_dev/20/code_from_slides/maps$ ./map_easy
0 = Maps rock
earth = 5.97
student@cse1325:/media/sf_dev/20/code_from_slides/maps$
```

```cpp
#include <iostream>
#include <vector>
#include <map>

int main() {

    // With vectors (using int as the index type)
    std::vector<std::string> s;
    s.push_back("Maps rock");
    for (int i=0; i < s.size(); ++i)
       std::cout << i << " = " << s[i] << std::endl;

    // With maps (using a std::string as the index type or key)
    std::map<std::string, double> m;
    m["earth"] = 5.97219; // ronnagrams, of course
    for (auto& [ planet, mass ] : m )
        std::cout << planet << " = " << mass << std::endl;
}
    // With Java TreeMaps this would be
    //   TreeMap<String, Double> m = new TreeMap<>();
    //   m.put("earth", 5.97);
    //   for(var planet : m.keySet())
    //       System.out.println(planet + " = " + m.get(planet));
```

I find this for-each iteration over maps *very* elegant!

Note: This code only works on C++ 17 and later – see GitHub for earlier code options

Practical Map Example: Gradebook as a Ragged Array

Note that the keys are sorted!

```
student@cse1325:/media/sf_dev/20/maps$ make map_practical
g++ -std=c++17 -o map_practical map_practical.cpp
student@cse1325:/media/sf_dev/20/maps$ ./map_practical
Student Ajay grades: 98 88 92 100
Student Juan grades: 91 73 110 100
Student Li grades: 100 98
Student Sophia grades: 77 69 75 84 91
student@cse1325:/media/sf_dev/20/maps$
```

```cpp
#include <iostream>
#include <vector>
#include <map>

typedef std::string Student;
typedef std::vector<int> Grades;

int main() {
    std::map<Student, Grades> gradebook = {
        {"Li",     {100,98}        },
        {"Ajay",   {98,88,92,100}  },
        {"Juan",   {91,73,110,100} },
        {"Sophia", {77,69,75,84,91}},
    };

    for (const auto& [student, grades] : gradebook) {
        std::cout << "Student " << student << " grades: ";
        for (int grade : grades) std::cout << grade << ' ';
        std::cout << std::endl;
    }
}
```

Practical Map Example: Improved with Op Overloads!

```
student@cse1325:/media/sf_dev/20/code_from_slides/maps$ make map_practical_plus
g++ -std=c++17 -o map_practical_plus map_practical_plus.cpp
student@cse1325:/media/sf_dev/20/code_from_slides/maps$ ./map_practical_plus
Student Ajay grades: 98 88 92 100
Student Juan grades: 91 73 110 100
Student Li grades: 100 98
Student Sophia grades: 77 69 75 84 91
student@cse1325:/media/sf_dev/20/code_from_slides/maps$
```

```cpp
#include <iostream>
#include <vector>
#include <map>

typedef std::string Student;
typedef std::vector<int> Grades;

std::ostream& operator<<(std::ostream& ost, const Grades& grades) {
    for (int grade : grades) ost << grade << ' ';
    return ost;
}

int main() {
    std::map<Student, Grades> gradebook = {
        {"Li",     {100,98}        },
        {"Ajay",   {98,88,92,100}  },
        {"Juan",   {91,73,110,100} },
        {"Sophia", {77,69,75,84,91}},
    };

    for (const auto& [student, grades] : gradebook) {
        std::cout << "Student " << student << " grades: " << grades << std::endl;
    }
}
```

Operator overloading simplifies output

Practical Map Example: *Overloaded* with Op Overloads!!!

```
#include <iostream>
#include <vector>
#include <map>

typedef std::vector<int> Grades;
std::ostream& operator<<(std::ostream& ost, const Grades& grades) {
    for (int grade : grades) ost << grade << ' ';
    return ost;
}

typedef std::string Student;
typedef std::map<Student, Grades> Gradebook;
std::ostream& operator<<(std::ostream& ost, const Gradebook& gradebook) {
    for (const auto& [student, grades] : gradebook)
        std::cout << "Student " << student << " grades: " << grades << std::endl;
    return ost;
}

int main() {
    Gradebook gradebook = {
        {"Li",     {100,98}       },
        {"Ajay",   {98,88,92,100}  },
        {"Juan",   {91,73,110,100} },
        {"Sophia", {77,69,75,84,91}},
    };

    std::cout << gradebook << std::endl;
}
```

Gradebook deserves its own type and <<, right?

Can't get much simpler output than this!

Variations

- C++ also has an unsorted map like Java's HashMap
  - Called unordered_map
  - Not as commonly used in C++ as HashMap in Java
- C++ also has versions of map and unordered_map that accept duplicate values
  - Called multimap and unordered_multimap
  - These require iterators to manage, since a key may access many values
  - Multimap is actually quite useful!

Summary

- C++ memory leaks usually result from missing **`delete`** calls
  - Use smart pointers or specific library classes to avoid
- Scopes control variable visibility
  - “Declare before use”: global, local, 3-term for
  - “Bidirectional”: class (just like in Java)
  - Scopes can be arbitrarily created at any point with { and } (unlike Java)
- C++ supports both single and multiple inheritance of classes (unlike Java which supports single only, but C++ lacks interfaces)
- Polymorphism only works with virtual superclass methods and pointers
- Casting rules are a bit more complicated than your other languages
  - Upcasts work only with pointers or references
  - Avoid C-style / Java-style downcasts entirely due to poor error detection
  - Downcasts should usually be done through **`static_cast`**
  - Downcasts through **`dynamic_cast`** are better for classes that have at least one virtual method and are accessed via pointers
- C++ std::map is like Java TreeMap


The End

OPTIONAL TOPIC
void*

- void* means "pointer to raw memory"
  - Very useful in embedded programming
- Use void* to transmit an address between pieces of code that really don't know each other's types – only the programmer knows
  - You can point to *any data* with a void*
  - Your code must "cast" the void* to the *correct* type before accessing the data via ->
  - Versatile, but more than a little scary!
  - Reminiscent of C's union

```
// Buffer can store any of
// 3 data types in the same
// memory location (yikes!)
union Buffer {
   int i[256];
   double f[128];
   char str[1024];
};
```

This topic will NOT be on the exam,
even as a bonus question!

Example Function with void* Parameter

- To use the void* variable, we first “cast” a type

void_star.cpp

```
void f(void* pv) {
    void* pv2 = pv;   // copying is ok (copying is what void*s are for)
    double* pd = pv;  // ERROR: can't implicitly convert void* to double*
    *pv = 7;          // ERROR: you (thankfully) can't dereference a void*
    pv[2] = 9;        // ERROR: you can't subscript a void*
    pv++;             // ERROR: you can't increment a void* or do pointer math
    std::cout << *pv; // ERROR: you can't stream a void*

    int* ip = (int*) pv;              // OK to use a C-style cast
    int* ips = static_cast<int*>(pv); // BETTER to use a C++-style static_cast
    std::cout << "Parameter is " << *ips << std::endl; // We can stream an int!
}
```

- Most operations fail for void* variables
  - Key exceptions are assignment to another void* variable and the various casts
- Once cast to a non-void type, normal operations apply (assuming the void* really *was* that type!)

A Real Example

Fast, Light ToolKit (FLTK) is a C++ graphics library

FLTK defines its callbacks with an “any data goes here” void* parameter

void_star_fltk.cpp

```
// Handle sending commands to child when button pressed
void HandleButton_CB(Fl_Widget*, void* data) {
    std::cout << static_cast<std::string*>(data) << std::endl;
}

class MyApp {
  public:
    MyApp() {
        win = new Fl_Window(720, 486);
        Fl_Button a(10, 10, 20, 20, "A"); a.callback(HandleButton_CB, (void*)"a\n");
        Fl_Button b(30, 10, 20, 20, "B"); b.callback(HandleButton_CB, (void*)"b\n");
        Fl_Button c(50, 10, 20, 20, "C"); c.callback(HandleButton_CB, (void*)"c\n");
        Fl_Button q(70, 10, 20, 20, "q"); q.callback(CleanExit_CB,    (void*)"q\n");
        win->end();
        win->show();
    }
};

// MAIN
int main() {
    MyApp app;
    return(Fl::run());
}
```

The handler must cast the void* to the same type as was registered to avoid a segfault

When the button is clicked, call the _CB method which always accepts exactly one void* parameter

Pointers generate segfaults.
Void* generates *many* segfaults!

More on void*

- Any pointer to object can be assigned to a void*
  - int* pi = new int; void* pv1 = pi;
  - double* pd = new double[10]; void* pv2 = pd;
- void* is rarely necessary – it's presence may indicate a poor software design
- Important: No objects of type "void" exist in C++
  - void v; // error – no such type
  - void f(); // f() doesn't return anything
    // f() does NOT return an object of type void!