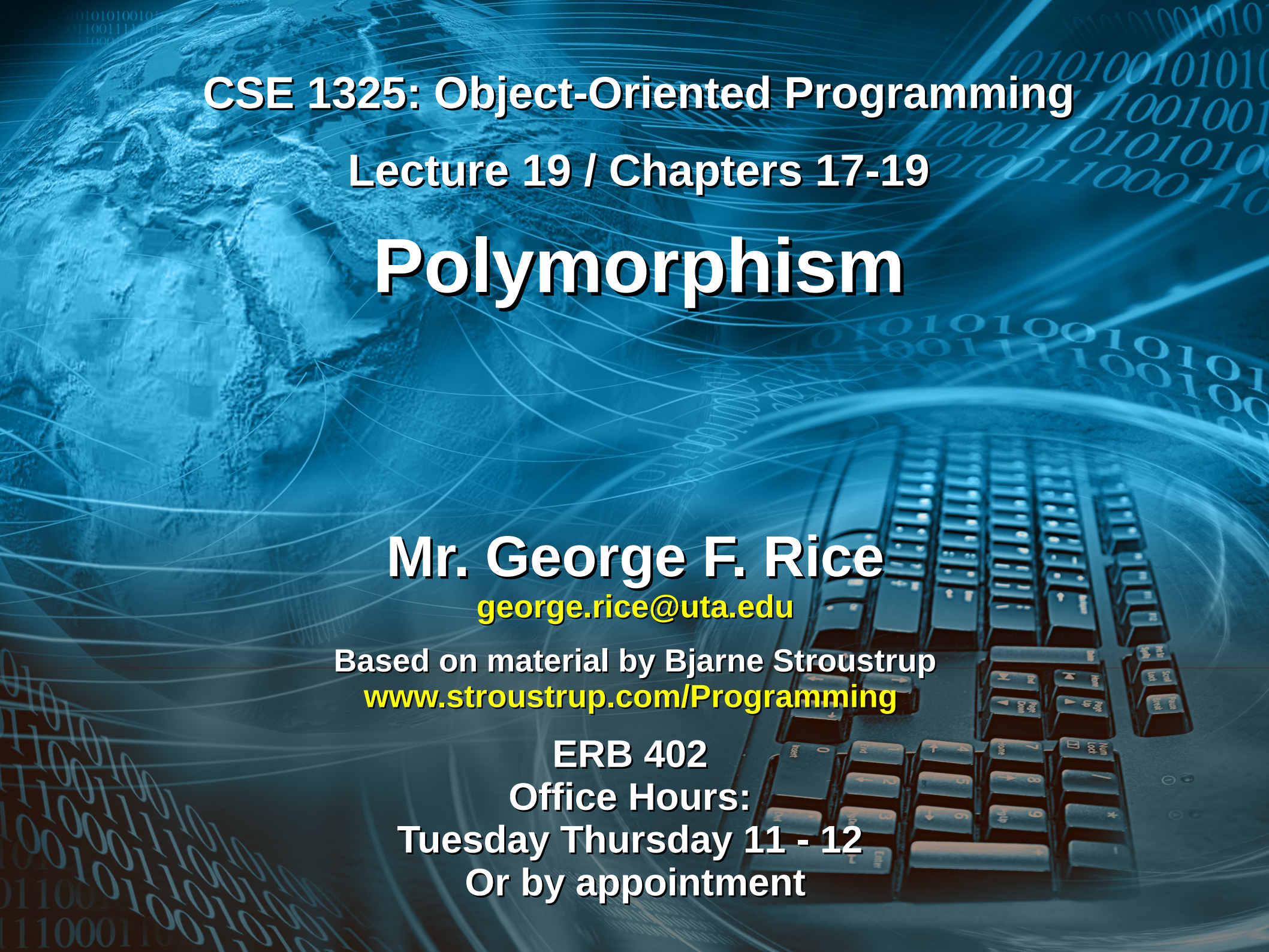


CSE 1325: Object-Oriented Programming

Lecture 19 / Chapters 17-19

Polymorphism

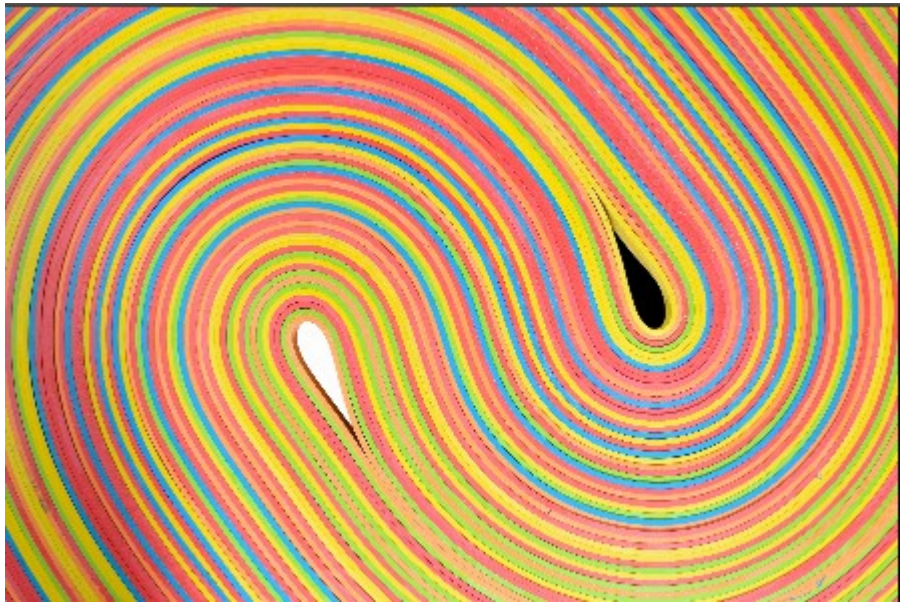
Mr. George F. Rice

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

Overview: Polymorphism



- Vectors revisited
- Pointers revisited
  - Delete and Destructors
  - Memory Leaks
  - void*
- Polymorphism
  - Inheritance
  - Virtual methods and overriding
  - References and pointers
  - Object memory layout

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

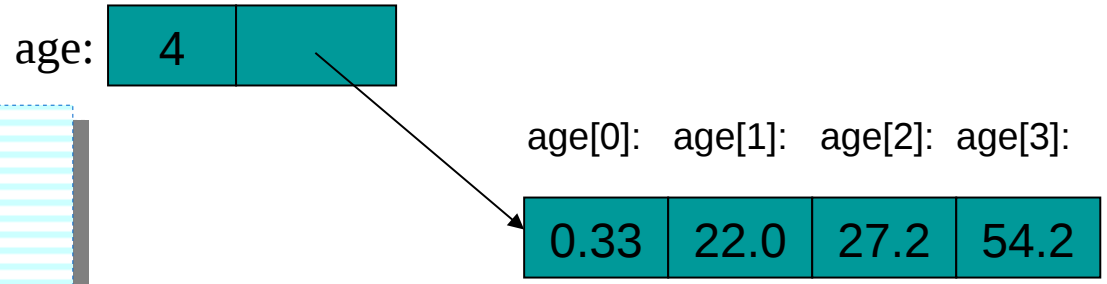
Assembly vs C++

- At the lowest level, assembly is simple and brutal
  - You have to program everything yourself
  - You have no type checking to help you
  - Run-time errors are found when data is corrupted or the program crashes
- We want to get to a higher level as quickly as we can
  - To become productive and reliable
  - To use a language “fit for humans”
- Chapters 17-19 basically show how vector is implemented
  - The alternative to understanding is to believe in “magic”
  - The techniques for building vector are the ones underlying all higher-level work with data structures

Vectors

- Vector is the “default” data structure
  - Like a Python list or Java LinkedList, it is incredibly handy for collecting random data
  - Particularly if we don’t know in advance how much data we’ll eventually collect
- Vector offers some neat features
  - Very efficient memory use and quick expansion
  - Optionally range checked
- How is vector implemented?

```
vector<double> age{4};
age[0]=.33;    age[1]=22.0;
age[2]=27.2;   age[3]=54.2;
```



Vector v0.1

```
// a very simplified vector of doubles (like vector<double>):

class vector {
    int sz;           // the number of elements (“the size”)
    double* elem;     // pointer to the first element
  public:
    vector(int s);    // constructor: allocate s elements,
                      // let elem point to them,
                      // store s in sz
    int size() const { return sz; }   // the current size
};

// Note: new here does not initialize elements (but the standard vector does)
vector::vector(int s) : sz{s},  // store the size s in sz
     elem{new double[s]} // allocate s doubles on the free store
                         // store a pointer to those doubles in elem
{ }
```

Review

Pointers and the Heap

- You request that memory be allocated on the heap with the ***new*** operator
  - "new" returns **a pointer to** the allocated memory
  - The pointer is to the address of the **first byte** of the memory
- For example
  - int* n = new int; // allocate one uninitialized int
  - int y = *n; // assign random memory to y (don't do this)
  - int* p = new int{5}; // allocate one int initialized to 5
  - int* q = new int[7]; // allocate seven uninitialized ints
  - double* pd = new double[n]; // allocate n uninitialized doubles
- A pointer points to an object of its specified type
- A pointer does **not** "know" how many elements it points to

Pointers and Arrays

- Array members on the heap are accessed just like those on the stack (mostly)
  - int* q = new int[7]; // allocate seven uninitialized ints
  - q[0] = 42; // store 42 at the first address
  - q[1] = 13; // store 13 at the second address
  - int x = q[1]; // copy value from second address to x
  - int y = *q; // copy value from first address to y
  - y = q[0]; // ditto
  - int z = q[42]; // assign random memory to z (don’t do this)
  - z = q[-4]; // ditto (don’t do this!)

Vector v0.2

```
// a very simplified vector of doubles:
class vector {
    int sz;      // the size
    double* elem;    // a pointer to the elements
public:
    vector(int s) :sz(s), elem(new double[s]) { }  // constructor
    double get(int n) const { return elem[n]; }    // access: read
    void set(int n, double v) { elem[n]=v; }       // access: write
    int size() const { return sz; }                // the current size
};

vector v{10};
for (int i=0; i<v.size(); ++i) { v.set(i,i); cout << v.get(i) << ' '; }
```

Do you see the glaring error in the above code?

Memory Leaks

```
double* calc(int result_size, int max)
{
    double* p = new double[max];     // allocate another max doubles
                                     // i.e., get max doubles from the free store
    double* result = new double[result_size];
    // … use p to calculate results to be put in result …
    return result;
}

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

Memory Leaks

```
double* calc(int result_size, int max)
{
    double* p = new double[max];     // allocate another max doubles
                                     // i.e., get max doubles from the free store
    double* result = new double[result_size];
    // … use p to calculate results to be put in result …
    return result;
}

double* r = calc(200,100);  // p was automatically deallocated from the stack
                            //   when calc exited, “leaking” double[max] from heap!
```

- Lack of de-allocation (usually called “memory leaks”) can be a serious problem in real-world programs
- A program that must run for a long time can’t afford any memory leaks
  - It’s not a good idea even for short programs!

Fixing the Memory Leaks

```
double* calc(int result_size, int max)
{
    double* p = new double[max];     // allocate another max doubles
                                     // i.e., get max doubles from the free store
    double* result = new double[result_size];
    // … use p to calculate results to be put in result …
    delete[] p; // mark the heap array that p points to 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];   // allocate 27 doubles
    // …
    p = new double[42];  // allocate 42 more
    // …
    delete[] p;      // now delete the allocated heap memory
}
```

Another Form of Memory Leak

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

Memory Leak Avoidance Strategies

- Don’t use new and delete
  - Rely on vector and similar mature higher-level classes
- Use garbage collection
  - A scheme, e.g., reference counting, identifies allocations that are no longer in use
  - A language-provided daemon – the “garbage collector” – runs periodically to de-allocate what is no longer in use
  - Java and Python use garbage collection
    - It is still possible to leak memory in these languages – just harder
- Use destructors, smart pointers, and Gtk::manage to automate de-allocation

Allocation and Deallocation
Or, new and delete[ ]

- Allocate using new
  - “new” allocates an object on the heap, sometimes initializes it, and returns a pointer to it
    - int* pi = new int; // default initialization (none for int)
    - char* pc = new char('a'); // explicit initialization
    - double* pd = new double[10]; // allocation of (uninitialized) array
  - New throws a bad_alloc exception if it can't allocate (out of memory)
- Deallocate using delete and delete[ ]
  - delete and delete[ ] return the memory of an object allocated by new to the free store so that the free store can use it for new allocations
    - delete pi; // deallocate an individual object
    - delete pc; // deallocate an individual object
    - delete[ ] pd; // deallocate an array
  - Delete of a zero-valued pointer ("the null pointer") does nothing
    - char* p = nullptr; // new to C++11 - earlier version would “= 0”
    - delete p; // harmless

Vector v0.3

```
// a very simplified vector of doubles:
class vector {
   int sz;        // the size
   double* elem; // a pointer to the elements
public:
   vector(int s) // constructor: allocates/acquires memory
         :sz(s), elem{new double[s]} { }
   ~vector()     // destructor: de-allocates/releases memory
         { delete[ ] elem; }
    // …
};
```

- Note: this is an example of a general, important technique:
  - Acquire resources in a constructor
  - Release them in the destructor
- Examples of resources: memory, files, locks, threads, sockets
- The textbook continues to improve the vector class, while we'll follow a rabbit trail for a bit instead

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
  - Example: the 2nd parameter of FLTK’s Fl_Callback function
  - You can point to *any data* when you register for the callback
  - The callback function must “cast” the void* to the correct type, then access the data via ->
  - Very versatile, and more than a little scary!

A (Very) Little FLTK

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

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

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() does not return anything
    // f() does not return an object of type void

C-style cast and static_cast

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

```
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 can't dereference a void*
                     // good! (The int 7 is not represented  like the double 7.0)
    pv[2] = 9;       // error: you can't subscript a void*
    pv++;            // error: you can't increment a void*
    int* pi_cast = (int*) pv;                 // ok: C-style cast
    int* pi_scast = static_cast<int*>(pv); // ok: C++-style static cast
}
```

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

C-style cast vs static_cast

- static_cast performs *some* sanity checks

```
class A {};
class B {};

int main() {
  A* a = new A;
  B* b = static_cast<B*>(a); // ERROR – A and B are unrelated, so won't compile
  B* c = (B*)(a);            // Compiles, but nonsensical – this will not end well
}
```

```
ricegf@pluto:~/dev/cpp/201801/19$ g++ --std=c++14 casting.cpp
casting.cpp: In function 'int main()':
casting.cpp:6:27: error: invalid static_cast from type 'A*' to type 'B*'
   B* b = static_cast<B*>(a);
                          ^
ricegf@pluto:~/dev/cpp/201801/19$
```

- A C-style cast is powerful but easy to misuse (see nonsense above), and is not appropriate for a C++ program
- A static_cast more safely converts a void* to a pointer to object type – or more generally, a pointer of one type to a pointer of a compatible type
- We'll get to dynamic casts shortly

From Lecture 6

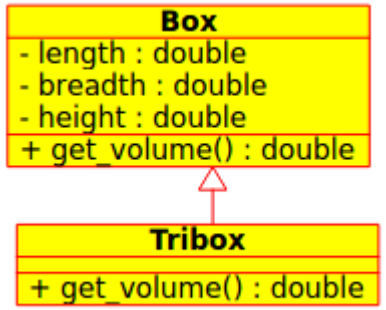
The Box Interface

- In Lecture 6, we created a Box class to demonstrate operator overloading
- We'll use a similar class for polymorphism

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

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



Protected is scoped between public (everyone sees it) and private (only this class and its friends see it) – useful for inheritance. Protected members are also visible to derived class members.

A “Tribox” is a triangular rather than rectangular box. Its volume is simply half that of a rectangular box of the same dimensions.

Since constructors don't inherit, we explicitly call the base constructor. We then (attempt to) override getVolume.

Testing Box and Tribox

```
ricegf@pluto:~/dev/cpp/201801/19$ g++ --std=c++14 box_nonpoly.cpp
ricegf@pluto:~/dev/cpp/201801/19$ ./a.out
Rectangular box
Box 1: volume 210
Box 2: volume 1560

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

```
// Test the Box base class
// and Tribox derived class

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 derived class and use them directly. Some of the features of the base class inherit – the protected data, for example – and some do not – the get_volume() method. We could also add additional fields and methods to the derived class as needed.

Do the Tribox Methods Override?

```
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 derived class replacing its base class' implementation of a method

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

Non-Virtual Methods Don't 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@pluto:~/dev/cpp/201801/19$ g++ --std=c++14 box_nonpoly.cpp
box_nonpoly.cpp:22:13: error: 'double Tribox::get_volume()' marked 'override', but does not ov
erride
      double get_volume(void) override {return length * breadth * height / 2;}
             ^

box_nonpoly.cpp:23:18: error: 'std::__cxx11::string Tribox::get_description()' marked 'overrid
e', but does not override
      std::string get_description() override {return "Triangular box";}
                  ^
```

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";}
};
```

In C++, we fix this by declared the base class 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.

```
ricegf@pluto:~/dev/cpp/201801/19$
ricegf@pluto:~/dev/cpp/201801/19$ g++ --std=c++14 box_override.cpp
ricegf@pluto:~/dev/cpp/201801/19$
```

Storing a Derived Object in a Base 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 << "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 Derived Object in a Base Variable

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

    std::cout << box1.get_desc
    std::cout << "Box 1: volum
    std::cout << "Box 2: volum

    Tribox tbox1(6.0, 7.0, 5.0
    Tribox tbox2(12.0, 13.0, 1

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

    box1 = tbox1;
    box2 = tbox2;

    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;
}
```

```
ricegf@pluto:~/dev/cpp/201801/19$ g++ --std=c++14 box_nonpoly.cpp
ricegf@pluto:~/dev/cpp/201801/19$ ./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
```

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

In C++, an object instanced from a derived type but *directly* assigned to a variable of the base class becomes an object of the base class. 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

```
void print_volume(Box b) {
   std::cout << b.get_description() << " volume is " << b.get_volume() << std::endl;
}
```

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

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

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

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

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

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

Direct variable version is first.

```
   std::cout << "Access derived class methods directly:" << std::endl;
   print_volume(box1);
   print_volume(tbox1);
```

Reference variable version is second.

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

Pointer variable version is third.

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

Two out of Three Isn't Bad?

```
void print_volume(Box b) {
   std::cout << b.get_descripti
}

void print_volume_ref(Box& b) {
   std::cout << b.get_descripti
}

void print_volume_p(Box* b) {
   std::cout << b->get_descript
}

// 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 derived class methods directly:" << std::endl;
   print_volume(box1);
   print_volume(tbox1);

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

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

```
ricegf@pluto:~/dev/cpp/201801/19$ g++ --std=c++14 box_poly.cpp
ricegf@pluto:~/dev/cpp/201801/19$ ./a.out

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
ricegf@pluto:~/dev/cpp/201801/19$
```

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 derived classes, enabling the same method call to invoke different derived methods to generate different results.

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

Object Layout in Memory

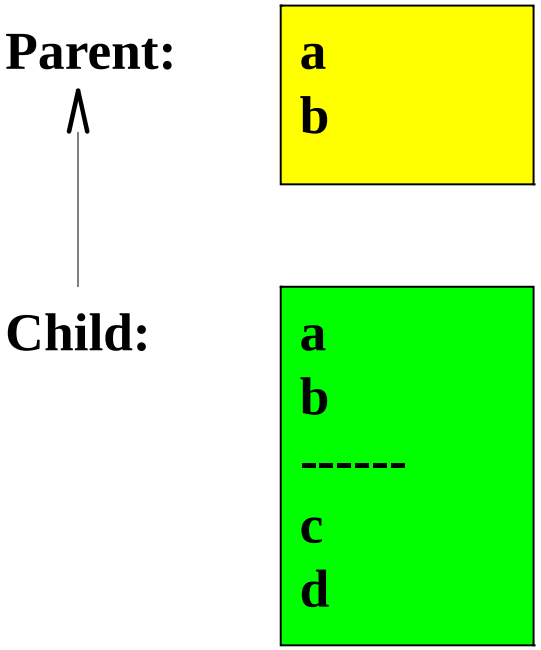
```
class Parent{
public:
    int a, b;
    virtual void foo(){cout << "parent" << endl;}
};

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

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

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

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

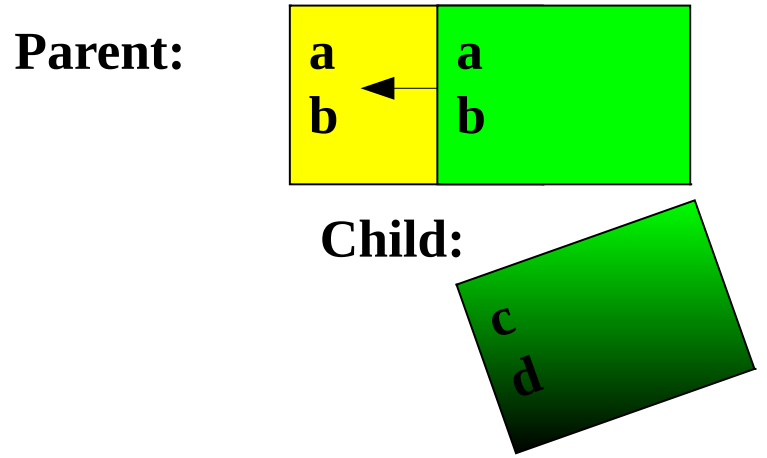
```
ricegf@pluto:~/dev/cpp/19$ g++ -o mem_layout mem_layout.cpp
ricegf@pluto:~/dev/cpp/19$ ./mem_layout
parent
child
Parent Offset a = 8
Parent Offset b = 12
Child Offset a = 8
Child Offset b = 12
Child Offset c = 16
Child Offset d = 20
ricegf@pluto:~/dev/cpp/19$
```

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

Object Layout - Copy

- The data members of a derived class are simply added at the end of its base class
- 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

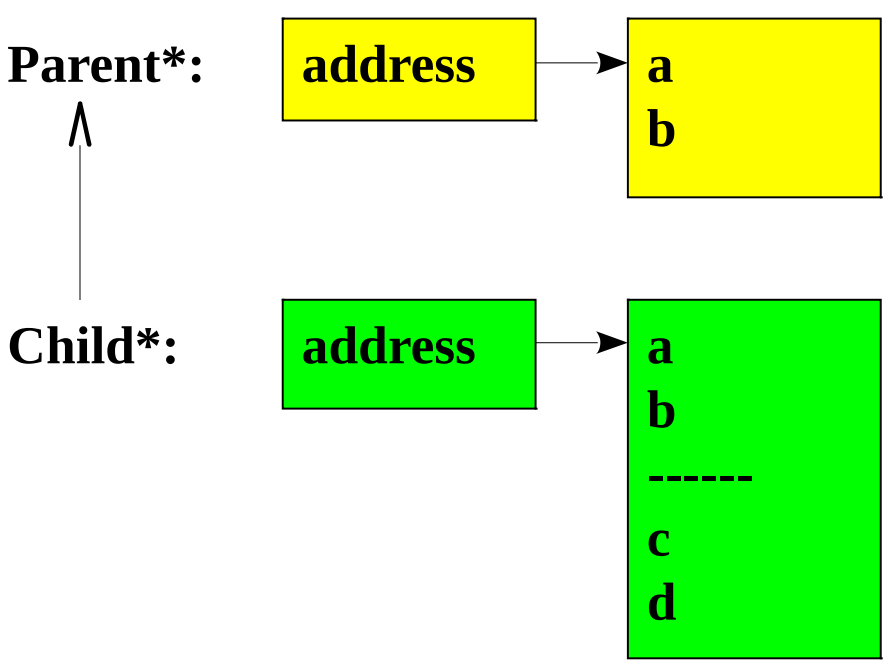


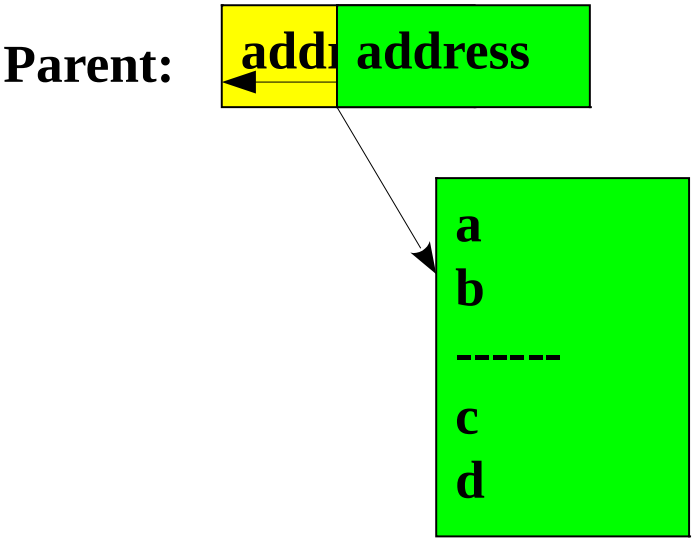


* We'll cover non-default options in a future lecture

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-style cast and static_cast
dynamic_cast

- dynamic_cast performs additional sanity checks, *as long as the source type is polymorphic*
  - If source type is not polymorphic, compiler error

```
casting.cpp:8:28: error: cannot dynamic_cast 'a' (of type 'class A*') to type 'class B*' (sour
ce type is not polymorphic)
   B* d = dynamic_cast<B*>(a);
                           ^
```

```
#include <iostream>
class Parent {
  public: virtual void hi() {std::cout << "Parent" << std::endl;}
};

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

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

// Continued on next page
```

dynamic_cast Example

```
// Continued from previous page
int main() {
  Parent* a = new A;

  A* a1 = (A*)(a);              // Compiles, perfectly valid
  if (a1 == nullptr) std::cout << "a1 is null" << std::endl;
  else a1->hi();

  A* a2 = static_cast<A*>(a);  // Compiles, perfectly valid
  if (a2 == nullptr) std::cout << "a2 is null" << std::endl;
  else a2->hi();

  A* a3 = dynamic_cast<A*>(a); // Compiles, perfectly valid but a little slower
  if (a3 == nullptr) std::cout << "a3 is null" << std::endl;
  else a3->hi();

  B* b1 = (B*)(a);              // Compiles, but undefined - 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 - A is not a B
  if (b2 == nullptr) std::cout << "b2 is null" << std::endl;
  else b2->hi();

  B* b3 = dynamic_cast<B*>(a); // ERROR - A is not a B, b3 is null
  if (b3 == nullptr) std::cout << "b3 is null" << std::endl;
  else b3->hi();
}
```

dynamic_cast Error Detection

```
// Continued from previous page
int main() {
  Parent* a = new A;

  A* a1 = (A*)(a);
  if (a1 == nullptr) std:
  else a1->hi();

  A* a2 = static_cast<A*>
  if (a2 == nullptr) std:
  else a2->hi();

  A* a3 = dynamic_cast<A*>(a); // Compiles, perfectly valid but a little slower
  if (a3 == nullptr) std::cout << "a3 is null" << std::endl;
  else a3->hi();

  B* b1 = (B*)(a);             // Compiles, but undefined - 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 - A is not a B
  if (b2 == nullptr) std::cout << "b2 is null" << std::endl;
  else b2->hi();

  B* b3 = dynamic_cast<B*>(a); // ERROR - A is not a B, b3 is null
  if (b3 == nullptr) std::cout << "b3 is null" << std::endl;
  else b3->hi();
}
```

```
ricegf@pluto:~/dev/cpp/201801/19$ g++ --std=c++14 dynamic_cast.cpp
ricegf@pluto:~/dev/cpp/201801/19$ ./a.out
A
A
A
A
A
b3 is null
ricegf@pluto:~/dev/cpp/201801/19$
```

Undefined (Error Not Detected)

Error Detected

Casting Bottom Line

- C-style casting is naïve
  - Not for 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
  - Preferred for C++ polymorphic types
- Other casts (e.g., const_cast and reinterpret_cast) exist, but those won't be covered this semester

Quick Review

- The provision of a single interface to multiple derived classes, enabling the same method call to invoke different derived methods to generate different results is called _____.
- In C++, this requires the _____ keyword for base class methods, and also the use of _____ or _____ for base class variables storing derived objects.
- The closest feature that C++ has to providing raw memory is the _____.
- Changing the type of an object is called a _____. Describe 3 types, and when each should be used in a C++ program.
- Define the difference between stack and heap memory, and how each is allocated.
  - If new cannot allocate memory (for example, because there is not enough memory available), what does C++ do?
- Give 2 examples of errors that may cause a memory leak.
- Identify 3 strategies to avoid memory leaks.
- ____ de-allocates arrays from the heap, while ____ de-allocates normal variables.

For Next Class

- (Optional) Read Chapters 17-19 in Stroustrop
  - Do the Drills!
- Next class we will discuss Templates and Iterators
- **Sprint #1 is due Thursday morning at 8 am**