ECE 590.01
C++ Programming, Data structures, and Algorithms

Object Layout, Multiple Inheritance, and Mixins
Admin

• Reading
  • On class website

• Homework 4
  • Due this Friday (April 5)

• Project
  • Due April 12
  • In class: 10 min demos to everyone last week
    • Show off what you did
  • “Normal” demos w/ Tas: same week
Talk this week: you should go!

- Todd Austin is giving a talk next week
  - Distinguished lecture series
  - Thurs 3/4 @ 3:15
  - Hudson 125

- Attend if you can
What have we been talking about?

- What did we talk about last time?
What have we been talking about?

- What did we talk about last time?
  - Algorithm Classes
    - Greedy
    - Dynamic Programming
    - Brute Force
    - Divide and Conquer
    - ...
  - Tries
Now: some advanced topics in OOP

• Advanced topics in OOP
  • Multiple Inheritance
    • Regular
    • Virtual
  • Object layout
    • How does dynamic dispatch work?
    • How do we make (multiple) inheritance work?
• Mixins
  • Classes templated in terms of what their superclass is
Multiple Inheritance

- C++ supports multiple inheritance (Java does not):

```cpp
class Cyborg : public Human, public Robot {
...
}
```

Cyborg extends both Human and Robot

- All the “normal” rules of inheritance apply
  - Polymorphism (can treat Cyborg as Human or Robot)
  - Inherits all fields/methods from both
  - Calls both parent constructors on creation
class Human { 
public:
    void speak() { cout << "Good afternoon sir" << endl; }
};
class Robot { 
public:
    void speak() {
        cout << "I am Bender, please insert girder" << endl;
    }
};
class Cyborg : public Human, public Robot{};
int main(void) {
    Cyborg c;
    c.speak();
    return 0;
}
Multiple Inheritance (cont’d)

class Human {
public:
    void speak() { cout << "Good afternoon sir" << endl; }
};
class Robot {
public:
    void speak() {
        cout << "I am Bender, please insert girder" << endl;
    }
};
class Cyborg : public Human, public Robot{};
int main(void) {
    Cyborg c;  
    c.speak();
    return 0;
}
class Human {
public:
    void speak() { cout << "Good afternoon sir" << endl; }
};
class Robot {
public:
    void speak() {
        cout << "I am Bender, please insert girder" << endl;
    }
};
class Cyborg : public Human, public Robot{};
int main(void) {
    Cyborg c;
    c.Human::speak();
    return 0;
}

Cyborg has two speak methods, need to say which one we mean
Multiple Inheritance (cont’d)

class Human {
public:
    void speak() { cout << "Good afternoon sir" << endl; }
};

class Robot {
public:
    void speak() {
        cout << "I am Bender, please insert girder" << endl;
    }
};

class Cyborg : public Human, public Robot{
public:
    void speak() {
        cout << "Resistance is futile."
        cout << "You will be assimilated" << endl;
    }
};

Note that if Cyborg overrides speak, there are now three speak methods:
- c.Human::speak()
- c.Robot::speak()
- c.Cyborg::speak() ← default if you write c.speak()
Multiple Inheritance (cont’d)

class Human {
    public:
        virtual void speak() { cout << "Hello" << endl; }
};
class Robot {
    public:
        void speak() {
            cout << "I am Bender, please insert girder" << endl;
        }
};
class Cyborg : public Human, public Robot{
    public:
        void speak() { cout << "RIF. YWBA." << endl; }
};
    Human * h = new Cyborg();    h->speak();

Now what happens?
class Human {
public:
    virtual void speak() { cout << "Hello" << endl; }
};

class Robot {
public:
    void speak() {
        cout << "I am Bender, please insert girder" << endl;
    }
};

class Cyborg : public Human, public Robot{
public:
    void speak() { cout << "RIF. YWBA." << endl; }
};

Human * h = new Cyborg(); h->speak();
Multiple Inheritance (cont’d)

class Human {
public:
    virtual void speak() { cout << "Hello" << endl; }
};
class Robot {
public:
    void speak() {
        cout << "I am Bender, please insert girder" << endl;
    }
};
class Cyborg : public Human, public Robot{
public:
    void speak() {cout << "RIF. YWBA." << endl; }
};
Robot * r = new Cyborg();   r->speak();

Now what happens?
Multiple Inheritance (cont’d)

class Human {
public:
    virtual void speak() { cout << "Hello" << endl; }
};

r \text{ is a Robot *}
class Robot {
    speak is NOT virtual in Robot
public:
    void speak() {
        cout << "I am Bender, please insert girder" << endl;
    }
};

class Cyborg : public Human, public Robot{
public:
    void speak() {cout << “RIF. YWBA.” << endl; }
};

Robot * r = new Cyborg(); \quad r->speak();
More Multiple Inheritance Issues

class GuiObject {
protected:
    int x, y;
};

class Button : public GuiObject {};
class ImgDisplay: public GuiObject {};
class ImgButton: public Button, public ImgDisplay {};

ImgButton inherits from two GuiObjects: Button and ImgDisplay
AuthGuard has x,y coordinates...
How many x’s and y’s does ImgButton have?
More Multiple Inheritance Issues

class GuiObject {
  protected:
    int x, y;
};

class Button : public GuiObject {};
class ImgDisplay: public GuiObject {};
class ImgButton: public Button, public ImgDisplay {};

**ImgButton inherits from two GuiObjects: Button and ImgDisplay**

GuiObject has x,y coordinates…
How many x’s and y’s does ImgButton have? 2 of each

**Button::x**  
**ImgDisplay::x**
More Multiple Inheritance Issues

class GuiObject {
    protected:
        int x, y;
};

class Button : public GuiObject {};
class ImgDisplay: public GuiObject {};
class ImgButton: public Button, public ImgDisplay {};

Drawing out the inheritance hierarchy show this
Two separate (distinct) GuiObject parents

In this case: bad for our design. Resulting object is one
GuiObject: has one (x,y) location

Asking for trouble
More Multiple Inheritance Issues

```cpp
class GuiObject {
    protected:
        int x, y;
};

class Button : public GuiObject {};
class ImgDisplay: public GuiObject {};
class ImgButton: public Button, public ImgDisplay {};
```

Can we make this work with ONE GuiObject? (as shown above)
Virtual Inheritance

class GuiObject {
protected:
    int x, y;
};

class Button : public virtual GuiObject {};
class ImgDisplay: public virtual GuiObject {};
class ImgButton: public Button, public ImgDisplay {};

Can we make this work with ONE GuiObject? (as shown above)

Yes: virtual inheritance
    Specify at Button and ImgDisplay (need on both)
    Says “Only one GuiObject if its multiply inherited by children”
How does all of this work?

• All of these high level OOP constructs sound great
  • Classes
  • Inheritance
  • Polymorphism
  • Dynamic Dispatch
  • Multiple Inheritance

• But they all must be implemented
  • Words in memory
  • Assembly instructions

• Next up: we learn how
  • Good to know in general
  • Understand performance cost associated with these features
Start simple

- No inheritance, no virtual methods

```cpp
class A {
  public:
    int x;
    int y;
    int getXplusY() { return x + y; }
};
```
Start simple

- Fields (x,y) live in object
- One copy of code (outside object), takes “this” param

```c
struct A {
    int x;
    int y;
};
```

```c
int getXplusY(A* this) {
    return this->x + this->y;
}
```

```
gGetXplusY:
    lw $t0, 0($a0)
lw $t1, 4($a0)
add $v0 <- $t0 + $t1
jr $ra
```
Start simple

• Fields \((x, y)\) live in object
• One copy of code (outside object), takes “this” param

\[ a\rightarrow \text{getXplusY}(); \]

is just

\[ \text{getXplusY}(a); \]

Note: multiple classes may have same method name. compiler “mangles” name to encode class name + method name
Inheritance: No virtual functions (yet)

class A{
    int x,y;
    int getXplusY() { return x + y;}
};
class B : public A {
    int z;
    int getXYZ(){ return getXplusY() + z;}
};

Need to support polymorphism:
    A * a = new B();
Inheritance: No virtual functions (yet)

```cpp
class A{
    int x, y;
    int getXplusY() { return x + y; }
};
class B : public A {
    int z;
    int getXYZ(){ return getXplusY() + z; }
};
```

Need to support polymorphism:
```cpp
A * a = new B();
```

Put fields of A first: A “sub-object”
Pointer to the start of a B? Looks just like an A
Virtual Functions

class A{
    int x,y;
    virtual int getSum() { return x+ y;}
};
class B : public A {
    int z;
    virtual int getSum() { return x + y + z; }
    virtual int getProd() { return x * y * z; }
};

A * a = new B();
int s = a->getSum();  //how to figure out what to call?
//(note: compiler cannot determine)
Dynamic dispatch

- How do we solve all functionality problems?
Dynamic dispatch

• How do we solve all functionality problems?

Add a level of indirection
VTable

- Objects with virtual functions have a pointer to a “vtable”
  - “Virtual function table”
  - Holds pointers to functions
  - This is why you need `virtual` in the parent, not the child

- Dynamic dispatch:
  - Read vtable pointer from object (fixed slot, typically 0)
  - Read function pointer from vtable
    - Vtable must respect “sub-objecting” design
  - Call the function via the pointer read from the table (jalr)

- Performance Cost (vs static dispatch): two loads
  - May also prevent optimizations (e.g., inlining)
  - Could be avoided in some cases (e.g., apply CFA [Shivers, 88])
class A{
    int x, y;
    virtual int getSum() {
        return x + y;
    }
};

class B : public A{
    int z;
    virtual int getSum() {
        return x + y + z;
    }
    virtual int getProd() {
        return x * y * z;
    }
};

Note that all As share the same vtable

ECE 590.01 (Hilton): Advanced OOP
Layout of two A objects

class A{
    int x, y;
    virtual int getSum() {
        return x + y;
    }
};
class B : public A{
    int z;
    virtual int getSum() {
        return x + y + z;
    }
    virtual int getProd() {
        return x * y * z;
    }
};

Observe: sub-objecting respected
First part of B "looks like" an A
getSum points to different impl

ECE 590.01 (Hilton): Advanced OOP
Abstract functions

class A{
   int x,y;
   virtual int getSum() {
      return x + y;
   }
   virtual void foo() = 0;
};

Abstract function: Vtable entry is NULL
Should never be called (if we manage to, program segfaults)
A note about construction

- During object construction, parent constructors called
  - Vtable pointer updated as construction progresses
    - During parent object construction, vtable setup as if only parent object
      - Can trick compiler into letting you call abstract function during parent class construction 😁
    - Direct calls in constructor: compiler will detect
    - Call another function which contains the call: segfault
  - Different than Java:
    - Java sets vtable pointer to actual type being “newed” immediately
Multiple Inheritance Layout

class A{
    int x;
    virtual void f() { ... }
};
class B{
    int y;
    virtual void g() { ... }
};
class C: public A, public B {
    int z;
    virtual void f() { ... }
    virtual void g() { ... }
};

Layout of A: straightforward
Multiple Inheritance Layout

class A {
    int x;
    virtual void f() { ... }
};
class B {
    int y;
    virtual void g() { ... }
};
class C: public A, public B {
    int z;
    virtual void f() { ... }
    virtual void g() { ... }
};
Multiple Inheritance Layout

class A{
    int x;
    virtual void f() { ...}
};
class B{
    int y;
    virtual void g() { ...}
};
class C: public A, public B {
    int z;
    virtual void f() { ...}
    virtual void g() { ...}
};

Naïve approach does not work

Why not?
Multiple Inheritance Layout

class A{
    int x;
    virtual void f() { ...}
};
class B{
    int y;
    virtual void g() { ...}
};
class C: public A, public B {
    int z;
    virtual void f() { ...}
    virtual void g() { ...}
};
class A{
    int x;
    virtual void f() { ... }
};
class B{
    int y;
    virtual void g() { ... }
};
class C: public A, public B {
    int z;
    virtual void f() { ... }
    virtual void g() { ... }
};
Multiple Inheritance Layout

class A{
    int x;
    virtual void f() { ...}
};
class B{
    int y;
    virtual void g() {...}
};
class C: public A, public B{
    int z;
    virtual void f() {...}
    virtual void g() {...}
};

Solution:
Have sub-object, not at start
Multiple Inheritance Layout

\[
C * c = \text{new } C();
\]
Multiple Inheritance Layout

C * c = new C();
B* b = c;

Cast (implicit or explicit) to non-primary parent class type results in different pointer

Cost: add immediate instruction
Multiple Inheritance Layout

C * c = new C();
B* b = c;
b->g();

But wait…
What g? Does this dispatch to?
Multiple Inheritance Layout

C * c = new C();
B* b = c;
b->g();

class C: public A, public B {
    int z;
    virtual void f() {…}
    virtual void g() {…}
};

Dynamic dispatch to C’s g()

What type does it expect for “this”?
Multiple Inheritance Layout

C * c = new C();
B* b = c;

b->g();

class C: public A, public B {
    int z;
    virtual void f() {...}
    virtual void g() {...}
};

Dynamic dispatch to C's g()

What type does it expect for "this"?
Multiple Inheritance Layout

```cpp
C * c = new C();
B* b = c;

b->g();

class C: public A, public B {
    int z;
    virtual void f() {...}
    virtual void g() {...}
};
```

g() is in C, so expects a C* for this
b is a B*
Was not a problem when pointing at start
Multiple Inheritance Layout

```c
C * c = new C();
B* b = c;

b->g();
```

- Adjustment to “this” before function
- C’s B sub-object vtable points at fixup
- C’s regular vtable points at “normal” code
Earlier example re-visited

class GuiObject {
    int x, y;
    virtual void f() { .. }
};
class Button : public GuiObject {
    int z;
    virtual void g() { ... }
};
class ImgDisplay: public GuiObject {
    int q;
    virtual void h() { ... }
};
class ImgButton: public Button, public ImgDisplay {
    int k;

Try to lay this out
(Not virtual inheritance—yet)
Earlier example re-visited

class GuiObject {
    int x, y;
    virtual void f() {...}
};
class Button : public GuiObject {
    int z;
    virtual void g() {...}
};
class ImgDisplay: public GuiObject {
    int q;
    virtual void h() {...}
};
class ImgButton: public Button, public ImgDisplay {
    int k;

GuiObj is straight forward
Earlier example re-visited

class GuiObject {
    int x, y;
    virtual void f() { .. }
};

class Button : public GuiObject {
    int z;
    virtual void g() { ... }
};

class ImgDisplay : public GuiObject {
    int q;
    virtual void h() { ... }
};

class ImgButton : public Button, public ImgDisplay {
    int k;

Button is simple inheritance
Earlier example re-visited

class GuiObject {
    int x, y;
    virtual void f() {...}
}

class Button : public GuiObject {
    int z;
    virtual void g() {...}
}

class ImgDisplay: public GuiObject {
    int q;
    virtual void h() {...}
}

class ImgButton: public Button, public ImgDisplay {
    int k;

    \textbf{ImgDisplay is too}
class GuiObject {
    int x, y;
    virtual void f() { .. }  
};

class Button : public GuiObject {
    int z;
    virtual void g() { ... }
};

class ImgDisplay: public GuiObject {
    int q;
    virtual void h() { ... }
};

class ImgButton: public Button, public ImgDisplay {
    int k;

ImgButton follows MI rules we just learned (note 2 GuiObjects)
Virtual Multiple Inheritance

class GuiObject {
    int x, y;
    virtual void f() {..}
};
class Button : public virtual GuiObject {
    int z;
    virtual void g() {...}
};
class ImgDisplay: public virtual GuiObject {
    int q;
    virtual void h() {...}
};
class ImgButton: public Button, public ImgDisplay {
    int k;

Now make it virtual
Virtual Multiple Inheritance

class GuiObject {
    int x, y;
    virtual void f() { .. }
};
class Button : public virtual GuiObject {
    int z;
    virtual void g() { ... }
};
Virtual Multiple Inheritance

class GuiObject {
    int x, y;
    virtual void f() { .. }
};
class Button : public virtual GuiObject {
    int z;
    virtual void g() { ... }
};

Now make it virtual
GuiObject is same

\[
\begin{array}{c}
<vtbl>
\hline
f \\
\hline
x \\
\hline
y \\
\hline
\end{array}
\]
Virtual Multiple Inheritance

class GuiObject {
    int x, y;
    virtual void f() { .. }
};

class Button : public virtual GuiObject {
    int z;
    virtual void g() { .. }
};

Now make it virtual

Button has virtual inheritance
Vtable has offset to parent
Virtual Multiple Inheritance

class GuiObject {
    int x, y;
    virtual void f() { .. }
};
class Button : public virtual GuiObject {
    int z;
    virtual void g() { .. }
};

Button * b = ...
GuiObject * g = b;

Polymorphism now requires

Load vtable pointer
Read offset from vtable
Add offset to “this”

<vtbl>
  z

<vtbl>
  x
  y

<parent>  +8
  f
  g

f
Virtual Multiple Inheritance

class GuiObject {
    int x, y;
    virtual void f() {...}
};
class Button : public virtual GuiObject {
    int z;
    virtual void g() {...}
};

Button * b = ...
GuiObject * g = b;
g->f();

Dynamically dispatched calls may require fixup
f’s implementation may be in a subclass (e.g. B)
GuiObject’s vtable points to fixup before “normal” codes
Virtual Multiple Inheritance

class GuiObject {
    int x, y;
    virtual void f() { .. }
};
class ImgDisplay: public virtual GuiObject {
    int q;
    virtual void h() { .. }
};

Other parent class (ImgDisplay) is similar
Virtual Multiple Inheritance

class GuiObject {
    int x, y;
    virtual void f() { .. }
};
class Button : public virtual GuiObject {
    int z;
    virtual void g() { ... }
};
class ImgDisplay: public virtual GuiObject {
    int q;
    virtual void h() { ... }
};
class ImgButton: public Button, public ImgDisplay {
    int k;
}
Virtual Multiple Inheritance

ImgButton layout
Level of indirection allows “holes”
Virtual Multiple Inheritance

ImgButton layout
Level of indirection allows “holes”

Cast as a Button
GuiObject access: look in vtable

Access to x?
  Load vtbl pointer
  Load parent_offset
  Compute parent = parent_offset + this
  Load parent_offset + field offset

Constrast with “Normal”: just (this + field offset)
Multiple Inheritance Summary

- Multiple Inheritance
  - Two options: regular + virtual
  - Performance costs
    - Understand them if you want to use it
    - Are they worth it?
      - Designer time usually more important than run time...

- Java does not have MI
  - Avoids the entire can of worms
  - Does have interfaces
    - No fields, only methods (avoids field layout issues)
    - Still need to solve vtable layout issues
Mixins

• Imagine working on a GUI

• Writing a class that puts fancy borders around any other component:
  • Would like to extend the component we are putting borders around
    • Effectively be a new component just like it, but with fancy borders

• Use a **mixin**: template over parent class

```cpp
template<class P> class FancyBorders: public P {
  ...
};
```
Mixins

• More generally:
  • Add functionality to something with a given interface
  • ...typically keeping the same interface

• Can be layered together as needed

• Read Yannis Smaragdakis’s paper for more
Wrap up for today

- Multiple Inheritance
- Object layout
  - Inheritance
  - Polymorphism
  - Dynamic Dispatch
  - Multiple Inheritance
- Mixins

- Next time:
  - A bit about garbage collection