Third Exam
Monday, May 20
Simpler polymorphism demo
(~mikec/cs32/demos/figures)

- **Base:** Figure has `virtual` void `print()`
  - `print()` is used in `printAt(lines)`
- **Derived:** Rectangle *just overrides* `print()`
- **Which** `print()` *is used in the following code?*
  ```
  Figure *ptr = new Rectangle,
  &ref = *new Rectangle('Q', 5, 10, 4);
  ptr->printAt(1); ref.printAt(1);
  ```
- **What if** `print()` *was not declared* `virtual`?
- **What if** line 2 above just had `ref`, not `&ref`?
  - To know why, see “slicing” … a few slides from now
“Pure virtual” and abstract classes

- Actually class Figure’s print() function is useless
  - It should have been a pure virtual function:
    virtual void draw() const = 0;
  - Says not defined in this class – means any derived class must define its own version, or be abstract itself

- A class with one or more pure virtual functions is an abstract class – so it can only be a base class
  - An actual instance would be an incomplete object
  - So any instance must be a derived class instance
Types when inheritance is involved

- **Consider:** void func (Sale &x) {...} or similarly: void func (Sale *xp) {...}
  - What type of object is x (or *xp), really? Is it a Sale?
  - Or is it a DiscountSale, or even a CrazyDiscountSale?

- **Just Sale members are available**
  - But might be virtual, and Sale might even be abstract
  - & and * variables allow polymorphism to occur

- **Contrast:** void func (Sale y) {...}
  - What type of object is y? It’s a Sale. Period.
  - Derived parts are “sliced” off by Sale’s copy ctor
  - Also in this case, Sale cannot be an abstract class
Consider:

```cpp
class Pet {
public: // pls excuse bad info hiding
    string name;
    virtual void print();
};

class Dog : public Pet {
public:
    string breed;
    virtual void print();
};
```

Dog d; Pet p;
d.name = "Tiny";
d.breed = "Mutt";
p = d; // “slicing” here
– All okay – a Dog “is a” Pet

Reverse is not okay
– A Pet might be a Bird, or …

And p.breed? Nonsense!

Also see `slicing.cpp` at ~mikec/cs32/demos/
Destructors should be virtual

- Especially if class has virtual functions
- Derived classes might allocate resources via a base class reference or pointer:
  ```
  Base *ptrBase = new Derived;
  ...
  // a redefined function allocates resources
  delete ptrBase;
  ```
- If dtor not virtual, derived dtor is not run!
- If dtor is virtual – okay: run derived dtor, immediately followed by base dtor
Casting and inherited types

- Consider again: `Dog d; Pet p;`
- “Upcasting” (descendent to ancestor) is legal:
  - `p = d;` // implicitly casting “up”
  - `p = static_cast<Pet>(d);` // like `(Pet)d`
    - But objects sliced if not pointer or reference
- Other way (“downcasting”) is a different story:
  - `d = static_cast<Dog>(p);` // ILLEGAL
    - Can only do by pointer and *dynamic cast*:
      - `Pet *pptr = new Dog;` // we know it’s a Dog
      - `Dog *dptr = dynamic_cast<Dog*>(pptr)`
    - But can be dangerous, and is rarely done
Multiple inheritance and virtual

- **Idea:** A ClockRadio is a Radio *and* an AlarmClock
  - But what if class Radio and class AlarmClock are both derived from another class, say Appliance?
  - Doesn’t each derived object contain an Appliance portion?
  - So wouldn’t a Clockradio have two copies of that portion, and how can such a scheme possibly work properly?
- **Answer:** It can work, but only by using *virtual* inheritance!
  
  ```cpp
  class Radio : virtual public Appliance;
  class AlarmClock : virtual public Appliance;
  class ClockRadio : public Radio, public AlarmClock;
  ```
  - Now a Clockradio has just one Appliance portion, not two
- **See demo code in** ~mikec/cs32/demos/multi-inherit
- **But note:** Hierarchy is still messed up, and still lots of chances for ambiguity – best to avoid multi-inheritance!
How do virtual functions work?

- Not exactly magic, but safe to consider it so
- `virtual` tells compiler to “wait for instructions” until the function is used in a program
- So the compiler creates a **virtual function table** for the class, with pointers to all virtual functions
- In turn, every *object* of such a class will be made to store a pointer to its own class’s virtual function table – try .../demos/sizeofvirtual.cpp
- At runtime: follow the pointers to find the code!
Memory and C/C++ modules

From Reading #6

Will return to OOP topics (templates and library tools) soon
Compilation/linking revisited

Usually performed by gcc/g++ in one uninterrupted sequence
Layout of C/C++ programs

Source code

\[ \leftarrow \]

… becomes

Object module

\[ \rightarrow \]
A sample C program – demo.c

#include <stdio.h>

int a[10]={0,1,2,3,4,5,6,7,8,9};
int b[10];

void main(){
  int i;
  static int k = 3;

  for(i = 0; i < 10; i++) {
    printf("%d\n",a[i]);
    b[i] = k*a[i];
  }
}

- Has text section of course: the machine code
- Has initialized global data: a
- Uninitialized global data: b
- Static data: k
- Has a local variable: i
### A possible structure of demo.o

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Header section</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>124</td>
<td>number of bytes of Machine code section</td>
</tr>
<tr>
<td>4</td>
<td>44</td>
<td>number of bytes of initialized data section</td>
</tr>
</tbody>
</table>
| 8      | 40       | number of bytes of Uninitialized data section (array \(b\[\])  
  *(not part of this object module)* |
| 12     | 60       | number of bytes of Symbol table section |
| 16     | 44       | number of bytes of Relocation information section |
| **Machine code section** | (124 bytes) | |
| 20     | X        | code for the top of the for loop (36 bytes) |
| 56     | X        | code for call to \(\text{printf}\) () (22 bytes) |
| 68     | X        | code for the assignment statement (10 bytes) |
| 88     | X        | code for the bottom of the for loop (4 bytes) |
| 92     | X        | code for exiting \(\text{main}\) () (52 bytes) |
| **Initialized data section** | (44 bytes) | |
| 144    | 0        | beginning of array \(a\[\]|
| 148    | 1        | |
| 176    | 8        |  |
| 180    | 9        | end of array \(a\[\] (40 bytes) |
| 184    | 3        | variable \(k\) (4 bytes) |
| **Symbol table section** | (60 bytes) | |
| 188    | X        | array \(a\[\] : offset 0 in Initialized data section (12 bytes) |
| 200    | X        | variable \(k\) : offset 40 in Initialized data section (10 bytes) |
| 210    | X        | array \(b\[\] : offset 0 in Uninitialized data section (12 bytes) |
| 222    | X        | \(\text{main}\) : offset 0 in Machine code section (12 bytes) |
| 234    | X        | \(\text{printf}\) : external, used at offset 56 of Machine code section (14 bytes) |
| **Relocation information section** | (44 bytes) | |
| 248    | X        | relocation information |