Lecture 10c
Slicing
April 1st, 2016

Announcements
• No reading due this or next week
• Recitations: Valgrind as a profiler
• PA7 is due Wednesday
• Midterm 1 week from Wednesday
• Next week: efficiency + odds & ends
• Week after: exam
  – Monday: review + hand out code
  – Wednesday: midterm #2
  – Friday: midterm post-mortem

Informal Assignment for Monday
• Run the valgrind profiler on your PA6
• Note where the time goes
• Be prepared to tell me (and the class) on Monday

Object Oriented Programming (Review)
1. Encapsulation
   – Collect data & code that operates on that data in one object
   – Provide a single, public interface
   – Changes to the implementation are local
2. Polymorphism (inheritance)
   – Abstraction via “is-a” relation
   – Write code at different levels of abstract
   – Avoid redundant code
3. Inheritance as Union
   – A child class is the union of parent and new field/methods

Collections in C++ (Review)
• Arrays, vectors, etc. collect like data items
  – We call them containers.
  – Most containers are objects
    – But arrays are not
• In C++, you can put anything in a container
  – vector<Mammal> is a vector of Mammals
  – vector<Mammal*> is a vector of references to Mammals
  – vector<Mammal*> is a vector of pointers to Mammals
• Java only supports containers of references

Inheritance as Union
• Assume Quagga inherits Equine
• Equine inherits Mammal
• Mammal inherits Animal

Slicing: an unfortunate combination of these
Implications of Storage Types

- Vector<Mammal>
  - Memory is contiguous
  - Fewer cache misses, more efficient access
  - Memory used: size() * sizeof(Mammal)
  - Push_back copies data
  - No constructor costs
  - No side effects
- Vector<Mammal&>
  - Mammals may be spread across memory
  - More cache misses
  - Memory: size() * (sizeof(Mammal) + sizeof(Mammal&))
  - Push_back does not copy data
  - No constructor costs
  - Potential side effects
  - If the source falls out of scope...
- Vector<Mammal*> -- like Vector<Mammal&> for these purposes

Polymorphism + Storage

- Polymorphism adds a wrinkle

Class Animal {
  Public:
  virtual string Type() const { return "Animal"; }
};

Class Mammal : public Animal {
  Public:
  Mammal(int span) : lifespan(span) {} 
  virtual string Type() const { return "Mammal"; }
  int Lifespan() const { return lifespan; }
  int lifespan;
};

What happens when...

```cpp
vector<Animal> avec;
vector<Animal&> aref_vec;
mammal m(30);
avec.push_back(m);
aref_vec.push_back(m);
cout << avec[0].Type() << " " << aref_vec[0].Type() << endl;
```

... on the previous slide

- What is printed out?
  - Animal Mammal
- What copy constructor was called?
  - Animal(const Animal& a);
- How many times was it called?
  - Once
- Why did avec[0] print 'Animal'?
  - Because avec[0] was created by push_back as an Animal
- Why did aref_vec[0] print 'Mammal'?
  - You are printing from the original, not the copy
  - The original was made as a Mammal

Slicing

- What is the size (in memory) of avec[0]?
  - The size of a pointer (for the VFPT)
- What happened to the lifespan of 30?
  - It was "sliced" away
  - Really, the Animal copy constructor just never copied it

More code

```cpp
Mammal* mptr = dynamic_cast<Mammal*>(&avec[0]);
cout << mptr->Lifespan() << endl;
```

- Does this compile?
  - Yes, no problem
- What does it print?
  - Nothing, it crashes
  - mptr == NULL
Expanded Class Defs

Class Animal {
    Public:
    Animal() { idptr = new IDStruct(); }  // Animal() { delete idptr; }
    virtual string Type() const { return "Animal"; }
    IDStruct* idptr;
};

Class Mammal : public Animal {
    Public:
    Mammal(int span) : lifespan(span), mptr(new Metabolism()) { }  // Mammal() { delete mptr; }
    virtual string Type() const { return "Mammal"; }
    int lifespan;  // int lifespan;
    Metabolism* mptr;
};

More Code

{  
    Mammal m(30);
}

• Is this a memory leak?
  – No
• What constructors/destructors are called in what order?
  – Animal constructor (includes new IDStruct())
  – Mammal constructor (incl. new Metabolism())
  – Mammal destructor (incl. delete mptr)
  – Animal destructor (incl. delete idptr)

Still More Code

void foo(Animal* aptr)
{
    delete aptr;
}

int main(int argc, char* arv[])
{
    Mammal* mptr = new Mammal;
    foo(mptr);
}

Questions

• Is this a memory leak?
  – Yes
• Which & how many destructors were called?
  – 1
    – ~Animal()
• What is the best solution to this problem?
  – declare ~Animal() to be virtual
• Rule of thumb:
  – if there is any chance a class might be inherited
    then make its destructor virtual

Constructors & Virtual Methods

Class Animal {
    Public:
    Animal() { idptr = new IDStruct(); cout << Type(); }  // Animal() { delete idptr; }
    virtual string Type() const { return "Animal"; }
    IDStruct* idptr;
};

Class Mammal {
    Public:
    Mammal(int span) : lifespan(span), mptr(new Metabolism()) { }  // Mammal() { delete mptr; }
    virtual string Type() const { return "Mammal"; }
    int lifespan;  // int lifespan;
    Metabolism* mptr;
};

Questions

• If I create an instance of Mammal, what will be printed?
  – Animal
• Why?
  – Mechanically:
    • Memory is allocated (if necessary) for the whole Mammal
    • The VFPT pointer is set to Animal's VFPT
    • Animal's fields are initialized (using the initialization list or defaults)
    • The Animal constructor is run (printing 'Animal')
    • The VFPT is re-set to be Mammal's VFPT
    • Mammal's fields are initialized
    • The Mammal constructor is run
  – Semantically: why?
    • Because methods of Mammal typically access fields of Mammal
    • The fields of Mammal won't have been initialized yet
    • So calling a Mammal method would be dangerous