Lecture 10a
Slicing and Casting

Oct. 24th, 2017

Announcements

• Midterm #2: Next Tuesday (Oct. 31st)
• Recitations this week: optional
  – Help centers, as you prepare for the midterm
• PA7 not assigned until next week
• PA6 due today
  – Any questions?

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...

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 the Animal constructor (called by push_back) and is 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
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() const { return 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
    – 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() const { return 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

New topic: type conversion

- Process of converting one type to another type
  - Not defined for all type combinations
- Required for typed languages
  - As you recall from cs270, internal representation of values differ
  - int – 2's complement
  - float – ieee754
- Conversion occurs during assignment
  - Including parameter assignment

Types of Type Conversion

- Converting from one data representation to another
  - E.g. int to double
- Converting declared types
  - Polymorphism
    - E.g. Quagga to Equine
  - Adding (or removing) 'const'
- Bit reinterpretation
  - These are different operations – but all called type conversion...

Examples of type conversion

```
i = 48;
char c = 'A';
float f = 3.14;
double d = 6.02E+23;
i = c; // is this safe? yes
    sizeof(int) >= sizeof(char)
c = i; // is this safe? no, 32 bits may not fit in 8 bits
i = f; // is this safe? no, a floating point value may not fit in an int
f = i; // is this safe? no, not necessarily (~23 bit mantissa)
f = d; // is this safe? no, lose precision, may be out of range
```

When is type conversion safe?

- Type conversion is safe when the receiving type can hold all the information in the source type
  - If you can convert from type 'A' to type 'B' and then back to 'A' without losing anything, the conversion is safe.
- Unsafe conversions generate warnings.
  - Information may be lost
  - Performance may vary across machines

Examples of type conversion

```
i = 48;
char c = 'A';
float f = 3.14;
double d = 6.02E+23;
i = c; // is this safe? yes – sizeof(int) >= sizeof(char)
c = i; // is this safe? no, 32 bits may not fit in 8 bits
i = f; // is this safe? no, a floating point value may not fit in an int
f = i; // is this safe? no, not necessarily (~23 bit mantissa)
f = d; // is this safe? no, lose precision, may be out of range
i = d; // is this safe? no, 64 bits may not fit in 32 bits
d = i; // is this safe? yes, double has more bits in mantissa than int has
```
Type Conversion Syntax

- Type conversion is a mess in C++
  - Assume 'j' is an int, 'd' is a double
- Four different syntactic forms:
  1. Implicit: j = d; (generates warning if unsafe)
  2. C Style: j = (int)d;
  3. Functional style: j = int(d);
     // struggles with unsigned types
  4. Modern style: j = static_cast<int>(d);

Type Conversion Style Note

- Code writing style (my suggestions)
  - Use implicit when safe
  - Use modern style otherwise
- Code reading
  - Recognize all four styles
    - All exist in legacy code
    - Use –Wall to detect unsafe implicit casts

Type casting – C style

```c
int i;
float f = 3.14;
float big = 6.02E+23;
i = (int) f;  // I know I’m losing the fractional part
i = (int) big;  // what is the result?
some = (some’s Type) other;  // general form
```

Modern Style Type Casting C++

- The C++ define these type of casts
  - static_cast<type>
  - dynamic_cast<type>
  - const_cast<type>
  - reinterpret_cast<type>

C++ static_cast<type>  

- Conversion done at compile time
- May actually generate code

```c++
int i = -1;
unsigned int u;
float f = 3.14;
u = static_cast<unsigned int> (i);  // any code?
i = static_cast<int> (f);  // any code generated?
```

C++ dynamic_cast<type>

- Used for casting polymorphic pointer types (i.e. base class and derived classes)
- Only required when going down the inheritance chain
- Combines Java’s instanceof and casting together
```c++
class Base;
class Derived : public Base;
Base* bp = new Derived();
Derived* dp = dynamic_cast<Derived*> (bp);
// what if it isn’t?
```
C++ `dynamic_cast<type>`

- The result of `dynamic_cast<type>` returns NULL if the type of the object is not compatible with the type is the cast.
- Best practice is to always check the result of the cast.
- Dynamic cast relies on run time type information (RTTI).

C++ `const_cast<ptrType>`

- Used to remove `const` from pointer or reference.
  - Use with caution!!!!!!!!!
  - Red flag for code quality
- Needed for legacy functions that should use `const`, but don’t.

```cpp
void func (Foo* f); // function doesn't actually change object
void bar (const Foo* cfp) {
  func(const_cast<Foo*> (cfp)); // warning without cast
}
```

C++ `reinterpret_cast<type>`

- Leave the bits alone, but treat them differently.
- Used in low level code.
  - E.g. device drivers

```cpp
float f = 3.14;
int i = static_cast<int> (f);       // what is i?
int j = reinterpret_cast<int> (f); // what is j?
```

C++ `narrow_cast<type>`

- Proposed extension, not yet part of standard
  - Verify at run time that `static_cast<type>` does not lose data
  - Succeeds, or throws runtime exception
- Available from
  - Boost
  - Stroustrup’s book