Is-a vs. Has-a; Preprocessing

ITP 435 – Spring 2016
Week 3, Lecture 2
Lecturer: Sanjay Madhav

USC Viterbi
School of Engineering

University of Southern California

Is-a

In C++, “is-a” is typically implemented with inheritance:

```
// Base class Shape
class Shape
{
};

// Triangle "is-a" Shape
class Triangle: public Shape
{
};
```

Has-a

"Has-a" is usually implemented with composition:

```
class Engine
{
};
class Car
{
      Engine myEngine;
};
```

If I ever write "*Really* Effective C++"…

RULE #1: Prefer "has-a" to "is-a"

Private Inheritance

- …OR the type of inheritance you might not be aware of:

```
class Base
{
};
class Derived : private Base
{
};
```

- When you inherit privately, all public data/functions in the "parent" class become private in the "child" class
- You also *cannot* cast from the "child" class to the "base" class

```
Base* test = new Derived();
// Error: Conversion from Derived* to Base* exists, but is
// inaccessible.
```

- Because of this, private inheritance is *not* an "is-a" relationship

Private Inheritance: "has-a"

- You could implement "has-a" relationship using private inheritance.

```
class Engine
{
};

class Car : private Engine
{
};
```

- This is only recommended if the correct behavior of Engine relies on using virtual functions
- This method can also undesirably lead to multiple inheritance

"is-implemented-in-terms-of"

- Maybe the only use of private inheritance that sort of makes sense:

```
class Queue : private LinkedList
{
};
```

- It's a bit more explicit – Queue is implemented in terms of LinkedList, but we're preventing anyone from using the non-queue operations in LinkedList
- Could still just use composition, though.
- And if we really want to support multiple types of backing data structures, it's best to use templates (like STL does)

Abstract Classes

Syntax refresher for a “pure virtual” or “abstract” or “interface” class:

```
class Abstract
{
     virtual void Function() = 0;
};
```

Why use this? For example, if you’re making an interface class:

```
class IDrawable
{
     virtual void LoadContent(); // Not pure virtual
     virtual void Draw() = 0; // Pure virtual
};
```

Differences between struct and class in C++

`struct` and `class` *almost* mean the same thing.

`struct`:

Member variables/functions are public by default

```
struct A { int i; }; // i is public
```

Inheritance is public by default

```
struct B : A {};  // B inherits publicly from A
```

`class`:

Switched!

```
class A { int i; }; // i is private
class B : A {};  // B inherits privately from A
```

A Complicated Hierarchy…

- The `Animal` class:

```
// Animal - Base class of everything
struct Animal {
   virtual void eat() {
      std::cout << "Animal::eat" << std::endl;
   }
   float mWeight;
};
```

Animal (8b)
vtable*
mWeight

vtable for Animal	
Index	**Function Pointer**
0	&Animal::eat

USC Viterbi School of Engineering

University of Southern California

(Size assuming 32-bits)

A Complicated Hierarchy…

- The `Mammal` class:

```
// A Mammal is-an Animal
struct Mammal : public Animal {
   void eat() override {
      std::cout << "Mammal::eat"
         << std::endl;
   }
   virtual void breathe() {
      std::cout << "Mammal:breathe"
         << std::endl;
   }
   float mTemperature;
};
```

Mammal (12b)
vtable*
mWeight
mTemperature

vtable for Mammal	
Index	**Function Pointer**
0	&M::eat
1	&M::breathe

USC Viterbi School of Engineering

University of Southern California

(Size assuming 32-bits)

(Abbreviated Mammal as M)

A Complicated Hierarchy…

- The `WingedAnimal` class:

```
// A WingedAnimal is-an Animal
struct WingedAnimal : public Animal {
   void eat() override {
      std::cout << "WingedAnimal::eat"
         << std::endl;
   }
   virtual void flap() {
      std::cout << "WingedAnimal::flap"
         << std::endl;
   }
   float mWingSpan;
};
```

WingedA… (12b)
vtable*
mWeight
mWingSpan

vtable for WingedAnimal	
Index	**Function Pointer**
0	&WA::eat
1	&WA::flap

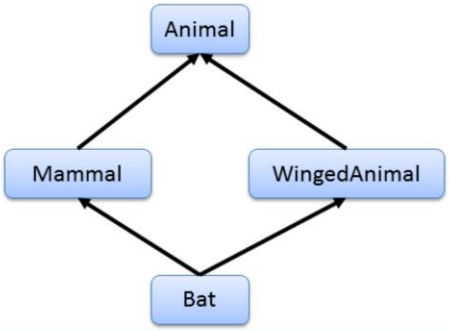
(Size assuming 32-bits)

(Abbreviated Mammal as M)

The Diamond Problem

- The Bat class:

```
// A Bat is-a Mammal AND is-a WingedAnimal
// (C++ Supports Multiple Inheritance)
struct Bat : public Mammal,
             public WingedAnimal {
};
```



Let's look at the parent classes

- `WingedAnimal` and `Mammal` are both the parents of `Bat`…

Mammal (12b)
vtable*
mWeight
mTemperature

WingedA… (12b)
vtable*
mWeight
mWingSpan

vtable for Mammal	
Index	**Function Pointer**
0	&M::eat
1	&M::breathe

vtable for WingedAnimal	
Index	**Function Pointer**
0	&WA::eat
1	&WA::flap

Diamond Problem

- `Bat` will end up with two copies of common ancestor variables…

Mammal (12b)
vtable*
mWeight
mTemperature

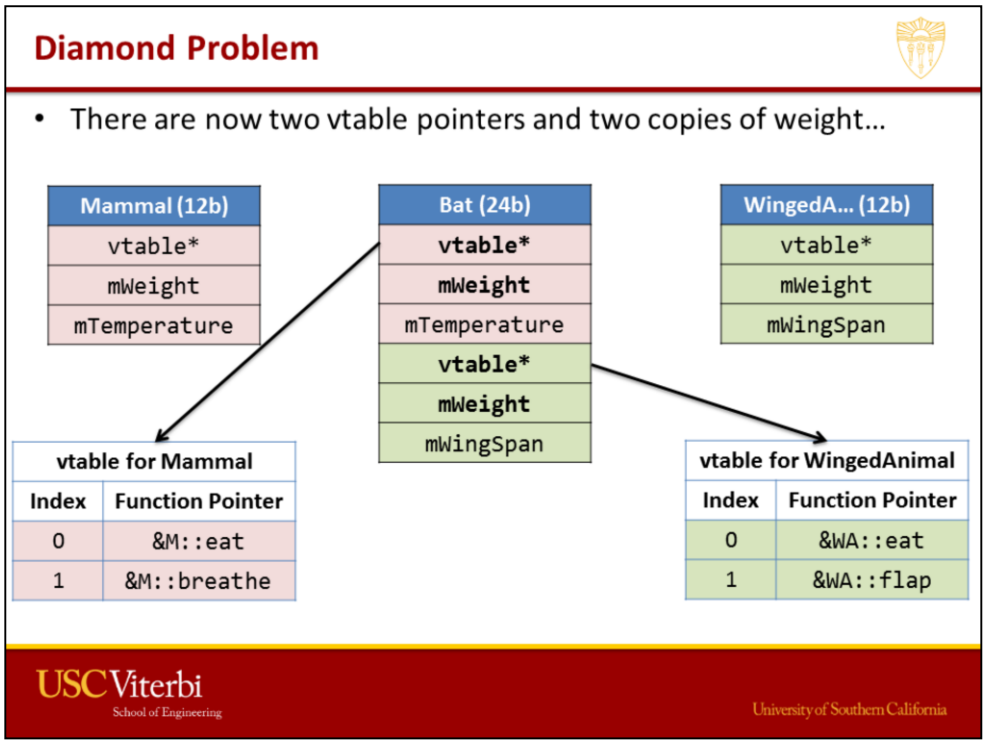
Bat (24b)
vtable*
mWeight
mTemperature
vtable*
mWeight
mWingSpan

WingedA… (12b)
vtable*
mWeight
mWingSpan

vtable for Mammal	
Index	Function Pointer
0	&M::eat
1	&M::breathe

vtable for WingedAnimal	
Index	Function Pointer
0	&WA::eat
1	&WA::flap

Diamond Problem



- There are now two vtable pointers and two copies of weight…

Mammal (12b)
vtable*
mWeight
mTemperature

Bat (24b)
vtable*
mWeight
mTemperature
vtable*
mWeight
mWingSpan

WingedA… (12b)
vtable*
mWeight
mWingSpan

vtable for Mammal	
Index	**Function Pointer**
0	&M::eat
1	&M::breathe

vtable for WingedAnimal	
Index	**Function Pointer**
0	&WA::eat
1	&WA::flap

Bat Amibiguity

- The diamond structure causes some ambiguity…

```
Bat* myBat = new Bat();
// Error: ambiguous access of 'mWeight'
// could be the 'mWeight' in base 'Animal'
// or could be 'mWeight' in base 'WingedAnimal'
myBat->mWeight = 5.0f;

// Error: ambiguous access of 'eat'
// could be the 'eat' in base 'Animal'
// or could be 'eat' in base 'WingedAnimal'
myBat->eat();
```

Solution

- There are two parts to solving these errors…
- First, use ***virtual*** inheritance for `Mammal`/`WingedMammal`:

```
// A Mammal is-an Animal
struct Mammal : public virtual Animal {
   void eat() override {
      std::cout << "Mammal::eat" << std::endl;
   }
   virtual void breathe() {
      std::cout << "Mammal:breathe" << std::endl;
   }
   float mTemperature;
};

// A WingedAnimal is-an Animal
struct WingedAnimal : public virtual Animal {
   void eat() override {
      std::cout << "WingedAnimal::eat" << std::endl;
   }
   virtual void flap() {
      std::cout << "WingedAnimal::flap" << std::endl;
   }
   float mWingSpan;
};
```

This guarantees that there will only be one shared copy of “Animal” data for all Mammals/WingedAnimals

Solution, Cont'd

- `Bat` needs to overload eat so that a call to eat is not ambiguous

```
// A Bat is-a Mammal AND is-a WingedAnimal
struct Bat : public Mammal, public WingedAnimal {
   // Overload to prevent amibiguity
   void eat() override {
      std::cout << "Bat::eat" << std::endl;
   }
};
```

Size of Classes

- Once you mix in virtual inheritance, the size of the classes gets quite confusing

```
sizeof(Animal) == 8
sizeof(Mammal) == 20
sizeof(WingedAnimal) == 20
sizeof(Bat) == 32
```


I'm not even sure why the sizes ended up like that (in VS 2013 32-bit mode)

Destructors

Given these classes:

```
class Shape
{
public:
     Shape(); // Dynamically allocates memory for Shape
     ~Shape(); // Deallocates memory for Shape
};

class Triangle : public Shape
{
public:
     Triangle(); // Dynamically allocates memory for Triangle
     ~Triangle(); // Deallocates memory for Triangle
};
```

Destructors, cont'd

Q: Suppose you have a Shape pointer as such:

```
Shape* myShape = new Triangle();
```

What happens when you execute the following?

```
delete myShape;
```

A: Because ~Shape is *not* virtual, ~Triangle will *not* be called (bad!!)

Solution: Whenever you have a class with virtual functions, the destructor should always be virtual too.

Preprocessor

- Processes all # directives to generate the final C++ code which will be compiled
- The resulting code is often called a “translation unit”
- Example 1:

```
#include "dbg_assert.h"
// dbg_assert.h code is essentially copy/pasted at this line
```

- Example 2:

```
// Compile this only in a "debug" build
#ifdef _DEBUG
// Random debug code…
#endif
```

- Example 3:

```
// Replaces "MAX_POOL_SIZE" with 256 in code
// (Breaks Rule #2 in Effective C++)
#define MAX_POOL_SIZE 256
```

Be careful with #include

- Don't make an "everything.h" that you include everywhere:

```
// INCLUDE EVERYTHING
#include <algorithm>
#include <bitset>
#include <cassert>
#include <cctype>
#include <cerrno>
#include <cfloat>
// ...
```

- Only include files you really need to include!

This helps ensure the compile time stays reasonable

Include "Guard"

- May have seen this before:

```
#ifndef _MYFILE_H_
#define _MYFILE_H_

// stuff here

#endif // _MYFILE_H_
```

- Not really recommended anymore. Preferred method is to put this at the start of the header (works in Visual Studio, Clang, and GCC):

```
#pragma once
```

Other useful (Visual Studio) #pragmas


```
// Emits warning
#pragma message("Warning: Ugly hack here.")

// Emits error
#pragma message("Error: Don't use this!")

// Disables optimization for any subsequent code in this file
#pragma optimize("", off)

// Disable a particular warning message (dangerous!!)
#pragma warning(disable : 4705)
```


Most #pragmas are compiler-specific

Macros

- Not only can we define values like this:

```
#define MY_VALUE 10
```

- We can replace one expression with another:

```
#define max(a,b) (((a) > (b)) ? (a) : (b))
```

- So if you write code like this:

```
std::cout << max(5, 6);
```

- Preprocessor replaces max with the defined code and our parameters:

```
std::cout << (((5) > (6)) ? (5) : (6));
```

Macros, Part 2

- Problem 1: Macros can clash with other functions/classes with confusing errors.
- What if I later declare…

```
void max();
```

- Error messages:

```
warning C4003: not enough actual parameters for macro 'max'
error C2059: syntax error : ')'
error C2059: syntax error : ')'
error C2059: syntax error : ')'
```

Macros, Part 3

- Problem 2: Must be very careful with parenthesis

```
#define MULT(x, y) x * y
// What if I do this?
int z = MULT(3 + 2, 4 + 2);
```

- Preprocessor evaluates it to:

```
int z = 3 + 2 * 4 + 2;
```

- Instead, you need a lot of parenthesis

```
#define MULT(x, y) ((x) * (y))
```

- So the preprocessor gives you:

```
int z = ((3 + 2) * (4 + 2));
```

Macro Counter-Point

- Instead of this (which is type-agnostic)

```
#define max(a,b) (((a) > (b)) ? (a) : (b))
```

- C++ lets us do (which is also type-agnostic):

```
template <class T>
T max(T a, T b)
{
     return ((a > b) ? a : b);
}
```

Macro Counter-Point Counter-Point

- What about something like this:

```
class CBlackjackView : public CWindowImpl<CBlackjackView>
{
public:
   DECLARE_WND_CLASS(NULL)

   BEGIN_MSG_MAP(CBlackjackView)
    MESSAGE_HANDLER(WM_PAINT, OnPaint)
    MESSAGE_HANDLER(WM_CREATE, OnCreate)
   END_MSG_MAP()
// ...
};
```

DECLARE_WND_CLASS Macro


```
#define DECLARE_WND_CLASS(WndClassName) \
static ATL::CWndClassInfo& GetWndClassInfo() \
{ \
static ATL::CWndClassInfo wc = \
{ \
{ sizeof(WNDCLASSEX), CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS,
   StartWindowProc, \
  0, 0, NULL, NULL, NULL, (HBRUSH)(COLOR_WINDOW + 1), NULL,
   WndClassName, NULL }, \
NULL, NULL, IDC_ARROW, TRUE, 0, _T("") \
}; \
return wc; \
}
```

constexpr

- C++11 feature to allow *compile-time* computation
- Example:

```
constexpr int max(int a, int b)
{
     return (a > b ? a : b);
}
```

- Then if we have code like this:

```
int a = max(5, 6);
```

- The *compiler* will replace it with:

```
int a = 6;
```

USC Viterbi School of Engineering

University of Southern California

Basic support for this in Visual Studio will be in 2015. Xcode has supported this for some time.

constexpr, Part 2

- Better example:

```
constexpr int factorial(int x)
{
     if (x > 0)
     {
          return x * factorial(x - 1);
     }
     else
     {
          return 1;
     }
}
```

Preprocessor Trick – Stringify

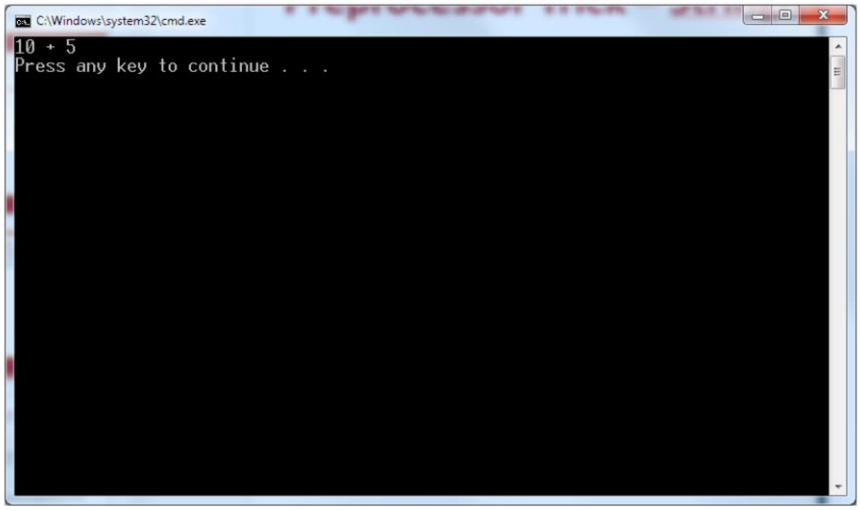
- You can convert any token passed to the preprocessor to a string using # in front of the parameter name

```
#include <iostream>

#define TO_STRING(str) #str

int main() {
   std::cout << TO_STRING(10 + 5) << std::endl;
   return 0;
}
```

Stringify in Action



Token Concatenation

- You can use ## to concatenate a preprocessor token to a set value, for example:

```
#include <iostream>

#define DECLARE_VAR(var) static int var##_s = 5;

int main() {
   DECLARE_VAR(hello);
   std::cout << hello_s << std::endl;
   return 0;
}
```

X-Macros

- One very useful (but advanced) macro design pattern is X-Macros
- An ***X-Macro*** can be used to generate a list of repetitive code constructs at preprocessor time
- Can save a lot of annoying repetition, though they are a little confusing to use

An example – Tokens.def


```
// Expression Operators
TOKEN(Assign,"=",1)
TOKEN(Plus,"+",1)
TOKEN(Minus,"-",1)
TOKEN(Mult,"*",1)
TOKEN(Div,"/",1)
TOKEN(Mod,"%",1)
TOKEN(Inc,"++",2)
TOKEN(Dec,"--",2)
TOKEN(LBracket,"[",1)
TOKEN(RBracket,"]",1)
TOKEN(EqualTo,"==",2)
TOKEN(NotEqual,"!=",2)
TOKEN(Or,"||",2)
TOKEN(And,"&&",2)
TOKEN(Not,"!",1)
TOKEN(LessThan,"<",1)
TOKEN(GreaterThan,">",1)
TOKEN(LParen,"(",1)
TOKEN(RParen,")",1)
TOKEN(Addr,"&",1)
```

Using Tokens.def to Generate an enum…


```
enum Tokens
{
   #define TOKEN(a,b,c) a,
   #include "Tokens.def"
   #undef TOKEN
};
```

Using Tokens.def to generate arrays of data


```
static const char* Names_data[] =
{
    #define TOKEN(a,b,c) #a,
    #include "Tokens.def"
    #undef TOKEN
};

static const char* Values_data[] =
{
    #define TOKEN(a,b,c) b,
    #include "Tokens.def"
    #undef TOKEN
};

static const int Lengths_data[] =
{
    #define TOKEN(a,b,c) c,
    #include "Tokens.def"
    #undef TOKEN
};
```