Advanced Software Engineering with C++ Templates

Lecture 6: Exercise Discussion, Inheritance

Thomas Gschwind <thg@zurich.ibm.com>
Agenda

- Exercise Discussion
- Object Oriented Programming
  - Object Model
  - Code and Type Inheritance
  - C++ Caveat: Slicing
  - Binding: Static vs Dynamic
  - Abstract Classes and Interfaces
  - Implementation
  - Multiple Inheritance
  - Casts

Th. Gschwind. Fortgeschrittene Programmierung in C++.
Connect 4: Checking if the other player could win

- A minority of colleagues look for
  - 3 stones in a row (buggy x_xx) or
  - a pattern of 3 in a series of 4 fields (better but clumsy)
- Most colleagues drop a stone and use the has_won method
- You don’t need to copy the playfield every time, just remove the stone after the test
- Do this for every column – dead simple
- Yes, with this implementation, you can already recursively start computing the next possible move for the opponent, etc.
Connect 4: Interoperability

- **Exercises**
  - Get an implementation from your colleague – almost nobody did this
  - Do it for the next exercise when we do connect 4 with inheritance

- **A few problems you would have encountered (or will encounter)**
  - You may want to extend your playfield
  - The playfield your computer player receives, might be different if running in a colleagues implementation
  - Just use the provided methods, not more
  - If you rely on your own playfield, copy whatever you get into your own playfield that provides all the extra methods – problem solved 😊
  - Maybe you do not want to extend your main playfield but use your own playfield class inside the computer player anyway

- **The above are just some ideas to get you thinking!**
**combineops_t**

- What the heck was that good for?
  - Constructor takes three functions (function objects) as argument
    - A binary one
    - Two unary ones
  - Function object invokes the binary one with the two unary ones

- So far so good, what is it useful for?
  - Remember `find_if` and `binder1st` and `binder2nd`
  - Find the first element between 17 and 34

```cpp
void foo(vector<int> v) {
    vector<int>::iterator b=r.begin(), e=r.end(), i;
    i=find_if(b, e, combineops(logical_and,
                                bind1st(less<int>(),17),
                                bind2nd(less<int>(),34));
}
```

Yes, a lambda function would be nicer...
Agenda

- Exercise Discussion

- Object Oriented Programming
  - Object Model
  - Code and Type Inheritance
  - C++ Caveat: Slicing
  - Binding: Static vs Dynamic
  - Abstract Classes and Interfaces
  - Implementation
  - Multiple Inheritance
  - Casts
Inheritance

- Definition of a class on the basis of another
- Derived, sub, or child class
  - Inherits the type of the base or parent class
  - Inherits the implementation of the base or parent class
  - May add new methods
  - May replace existing methods

- Kinds of inheritance
  - Code inheritance
  - Interface inheritance
Object Model: C++ vs. Java

- **Java**
  - Every object inherits from the class Object
  - All objects are stored on the heap and accessed through pointers
  - Primitive types (byte, int, long, float, double, etc) are not objects

- **C++**
  - There is no Object class (i.e., there is no common base class)
  - Objects can be anywhere and passed using whatever mechanism
  - Primitive types (byte, int, long, float, double, etc) are not objects
Types of Inheritance

- **public**
  - To define an is-a relationship
  - Public and protected members will be inherited as defined in the base class

- **protected, private**
  - To define a has-a relationship (style-wise, frequently it is better to define the base class as attribute in the derived class)
  - Public and protected members become protected, respectively private, members of the derived class
  - Avoid this type of inheritance
“public” Derived Classes

- They are polymorph
  - They have the type of the derived class
  - They inherit the type of the base class(es)
  
=> May be used wherever the base class can be used

- In the base context
  - Only those attributes which are defined in the base class may be used
Code and Type Inheritance

- Extend a string class with new functions
  - `find`
  - `toupper`
  - `tolower`
  - ...

```cpp
class String2 : public String {
public:
    void tolower() {
        int n=length();
        for (int i=0; i<n; ++i) {
            (*this)[i]=tolower(*this[i]);
        }
    }

    // ...
};
```
Access

- The derived class may access the public members of the base class
- No access to private members
- Additionally, members may be declared protected
  => These members may be used by the class itself
  => These members may be used in derived classes
- Essentially, this is the same as in Java except that package does not exist
Child classes may access the protected members of the parent class

```cpp
// String.h

class String { 
protected: 
   char *strg; unsigned int len;

public: 
   ... 
}

// String2.h

class String2 : public String { 
public:
   void tolower() {
      for (int i=0;i<len;++i) { strg[i]=tolower(strg[i]); }
   }

   // ...
}
```

Style! In this case, the previous implementation is nicer. It is not slower, almost as readable, but encapsulates the implementation of the string class.
void foo(String2 s2) {
    s2.tolower();
    cout << s2 << endl;
}

int main(int argc, char *argv[]) {
    String s1="Hello ";
    String2 s2="World!";
    String &s3=s2; // ok, every String2 is a String
    cout << s1[0]; // ok, String implements operator[]
    cout << s2[0]; // ok, String2 inherits operator[
    cout << s3[0]; // ok, String implements operator[
    foo(s1); // error s1 is a String
    foo(s2); // ok, s2 is of type String2
    foo(s3); // error, type of s3 is String
}
“protected” and “private” Derived Classes

- The parent’s type is not inherited outside the class
- Defines a code inheritance relationship
  - Avoid, there are other ways to use a class’s code
  - Use aggregation instead
    - Define a member with the type of the intended parent instead
    - Use that for the functionality you otherwise would have inherited
"protected" & "private" Derived ...: Type Compatibility

Th. Gschwind. Fortgeschrittene Programmierung in C++.
Overriding Members

- In C++, member functions of the base class can be overridden
- Simply define the same member again in the sub class
- Almost like in Java
  - However, many C++ novices typically need to figure out two issues: Slicing and Binding
Overriding Members: Example

- We have a trivial Stack class
- From this Stack class we want to create a child class that counts the elements on the stack through an extra counter

```
Stack.cc

class Stack {
    int buf[256], sp;

public:
    Stack() { sp=256; }  
    void push(int i) { s[--sp]=i; }
    int pop() { return s[sp++]; }
    int empty() { return sp==256; }
};

CountingStack.cc

class CountingStack: public Stack {
    int cnt;

public:
    Stack_with_Count() { cnt=0; }  
    void push(int i) {
        Stack::push(i); ++cnt; }
    int pop() {
        --cnt; return Stack::pop(); }
    int elements() { return cnt; }
};
```

This is probably not what we want!
Why? Binding!
Using Inheritance: Slicing

- Naively, we might use our new CountingStack as shown below
- CountingStack is passed by value where Stack is expected
  - C++ only has space for a Stack on the stack
  - Hence it will only copy the Stack attributes of CountingStack (slicing)
  - In this case it actually works but is probably not what you expected

```cpp
void fill(Stack s) {
    cout << s.pop() << endl;
    s.push(2);
}

int main(int argc, char *argv[]) {
    CountingStack cs;
    cs.push(2); cs.push(4);
    fill(s); fill(sc);
    cout << sc.elements() << endl;
}
```

This is “fine” because CountingStack did not change the internal representation of the Stack. Had buf and sp been protected and had we decided to reverse the order in buf, we would have had a big surprise.
Slicing

- Using this signature, `fill` allocates enough memory to store an object of type `Stack`
  - `CountingStack` uses more memory because it adds the counter
  - Obviously the counter will not fit, hence it is omitted

- If the subclass is not implemented nicely we may get an inconsistent object
  - That is, if the attributes of the subclass are not kept consistent with their use in the base class

- Lesson learned, subclass are typically best passed by pointer or reference
  - This way the class will not be copied and hence all its state can be preserved
Using Inheritance: Binding

- References and Pointers fix the first issue
- However, the new program still will not produce what we expect?
- Why? Binding!

```c++
void fill2(Stack &s) { s.push(1); s.push(2); }
void fill3(Stack *s) { s->push(1); s->push(2); }
int main(int argc, char *argv[]) { 
  Stack s;
  CountingStack cs;
  cs.push(2); cs.push(4);
  fill2(s); fill2(cs);
  cout << cs.elements() << endl;
  fill3(&s); fill3(&cs);
  cout << cs.elements() << endl; }
```

Before fill2, CountingStack is:
buf={…, 4, 2}, sp=254, cnt=2
After fill2 calls, CountingStack will be:
buf={…, 2, 1, 4, 2}, sp=252, cnt=2
After fill3 calls, CountingStack will be:
buf={…, 2, 1, 2, 1, 4, 2}, sp=250, cnt=2
 Binding

- Binding refers to the process of binding different entities to a specific symbol
  - For instance a value to a variable
  - An implementation of a function to a function name
  - ...

Th. Gschwind. Fortgeschrittene Programmierung in C++.
Static Binding

- Member function to be executed is determined on the basis of the variable’s static type
  - Used in our current example
  - Member may be inlined
  - Faster code

- In object-oriented programming frequently not desired
Dynamic Binding

- Member function to be executed is determined on the basis of the object’s runtime type (“dynamic binding”)
- Member functions where dynamic binding is to be used need to be declared as virtual
  - A member function declared virtual will stay virtual for all subclasses
  - Style: repeat the virtual keyword in all the subclasses
- Member function to be executed determined via virtual method table
  - No inlining
  - A bit slower (generally, don’t worry about this)
Dynamic Binding (cont’d)

Use `virtual` to indicate a member function should be bound dynamically

```cpp
class Stack {
    int s[256], sp;

public:
    Stack() : sp(256) {}
    virtual void push(int i) { s[--sp]=i; }
    virtual int pop() { return s[sp++]; }
    virtual int empty() { return sp==256; }
};

class CountingStack : public Stack {
    int cnt;

public:
    Stack_with_Count() : cnt(0) {}
    virtual void push(int i) { Stack::push(i); ++cnt; }
    virtual int pop() { --cnt; return Stack::pop(); }
    virtual int elements() { return cnt; }
};
```

Not necessary to repeat `virtual` here, but it’s considered good style!
Dynamic Binding – Implementation

- How is dynamic binding implemented?
- How do we know which member function to execute?
- Put member functions into a table (virtual member table) and look them up during runtime

```
class Stack {
    virtual void push(int i);
    virtual int pop();
    virtual int empty();
};
```

```
class CountingStack : public Stack {
    virtual void push(int i);
    virtual int pop();
};
```
Dynamic Binding and Overriding Methods

In C++

- Arguments are fixed
  - `void push(int x)` cannot be overridden with `void push(long x)` in the subclass.

- Return value of a function can be restricted
  - `Stack Stack::clone()` can be replaced with `CountingStack CountingStack::clone()` in the subclass.
  - Again it can be preferable to return a pointer or a reference to the `Stack/CountingStack` object.
Where are Virtual Members Useful?

- Default implementations
- Callbacks and hooks
- In abstract classes or interfaces
- Never implement a member that only returns an error
  - Use abstract methods
Abstract Classes

- Express a concept
- Define an interface
- Cannot be instantiated
- Can use pointers and references to such classes

Examples
- An abstract player implementation for our Connect 4 Example
class PlayerFactory {
    ...

public:
    PlayerFactory(const char *config_file) { /* ... */ }
    Player *getPlayer(const string &type) {
        if (type=="human") return new HumanPlayer();
        else if (type=="computer1") return new Comp1Player();
    }
}
Connect 4: Player Class

class Player {
public:
    virtual string getName(void) = 0;
    virtual int play(const playfield &field) = 0;
};

- =0 after the member indicates that it is abstract
- If at least one method is abstract, the entire class is abstract
- Abstract classes cannot be instantiated
- A method without state and all members being virtual defines a pure virtual class and is equivalent to a Java interface
It would be nice to have our own computer player

- Keeps some advanced state (next moves)
- Uses an extra thread to compute moves during the other player’s turn
- Need to shut this down in the constructor

```cpp
int run(const string &p1, const string &p2) {
    PlayerFactory pf(...);
    Player *players[2];
    players[0]=pf.getPlayer(p1);
    players[1]=pf.getPlayer(p2);
    ...
    while (canPlay) {
        ...
    }
    delete players[0];
    delete players[1];
}
```

```cpp
class Comp2Player {
    SomeDataStructure *nextMoves;

public:
    Comp2Player(void) {
        nextMoves=new ...;
    }

    ~Comp2Player(void) {
        delete nextMoves;
    }

    v. int play(const playfield &f) {
        // compute best moves
        // store nextMoves in nextMoves
        return move;
    }
}
```

This will not free any player resources or stop its thread.
Solution

- Define a virtual destructor in the base class
  ```cpp
template virtual ~Player() {}
  ```
- Now the virtual destructor is called and the derived class has a chance to free its resources

As soon as you have a virtual member, implement a virtual destructor.

Thomas Gschwind
Virtual or not Virtual?

- If possible/useful, define an interface
  - In C++ a pure virtual class defines an interface
  - C++ does not need a special syntax for interfaces
    C++ has multiple inheritance

- Not Virtual
  - If not useful to derive a class from your class
  - Performance *is* important (unlikely)
Dynamic Binding: C++11 Features

- **override**
  A member function may be marked as override to signify it overrides a non-final virtual member function of the base class

- **final**
  A member function or class may be marked final to indicate it may not be overridden

```cpp
class CountingStack : public Stack {
    ...
    virtual void push(int i) override;
};

class Stack final {
    ...
};

class Stack {
    ...
    int empty() final { return sp==256; }
};
```
Multiple Inheritance: Interfaces

- Unlike Java, C++ does not provide an interface concept
- Multiple inheritance provides the same functionality
- Multiple inheritance is more powerful
- C++ provides multiple inheritance

```cpp
struct Vehicle {
  virtual void driveTo(...) = 0;
};

struct Kitchen {
  virtual void cook(...) = 0;
};

class PizzaTruck: public Vehicle, public Kitchen {
  // ...
};
```
Multiple Inheritance: Name Collisions

- Two base classes/interfaces provide the “same” function
  - Members have different semantics, needs to be looked at individually
  - Interfaces and same semantics of the members, no problem
  - Classes with implementation, resolve explicitly

- Let’s assume, both Car and Player provide each a to_string() member function, each with an implementation

```cpp
class PizzaTruck: public Vehicle, public Kitchen {

    string to_string() {
        return "("+Vehicle::to_string()+","+
                      Kitchen::to_string()+")";
    }
}
```
Multiple Inheritance: “Diamond Shape” Inheritance

- If Base has no state, there is no problem

- If Base has a state, this becomes tricky (generally, avoid)
  - Shall Base be inherited twice: no change necessary to the above
  - Shall Base be shared among the sub-classes: inherit virtually
    - class A: public virtual Base;
    - class B: public virtual Base;

Th. Gschwind. Fortgeschrittene Programmierung in C++. 38
Casts allow to “convert” an object of type FOO into an object of type BAR.

- `static_cast<T>(o)`
- `dynamic_cast<T>(o)`
- `reinterpret_cast<T>(o)`
- `const_cast<T>(o)`
- `(T)o /* C-Cast */`
static_cast<T>(o)

- Converts an Object o into a given type T
- Is statically verified (i.e., during compile time)
- Pre-defined conversion
- User-defined conversion
- Example
  ```cpp
fraction fr(1,2);
double f = static_cast<double>(fr);
```
\texttt{\textbf{dynamic\_cast}\langle T\rangle\ (o)}

- Checks whether $o$ is of type $T$ (down/crosscast)
- Dynamically verified (i.e., during run-time)
- Uses Run Time Type Information (RTTI)
  - RTTI is generated for any class with a virtual method
- **Returns** \texttt{NULL} if $o$ is not of type $T$
- **Example**
  ```c++
  child c(...), *c1;
pARENT *p=&c;
c1=dynamic\_cast\langle\text{child}\rangle\ p;
  ```
reinterpret_cast<T>(o)

- The value (bit-pattern) of object o will be interpreted to be of type T
- Not verified
- Example
  ```cpp
  char *mem = reinterpret_cast<char*>(malloc(n*sizeof(char)));
  ```
Squares and Rectangles

- I had a colleague who was unsure whether to derive rectangle from square or vice-versa. What would you recommend to him? Implement a set of sample programs illustrating various options and your recommendation! The programs should demonstrate why your solution is better than the other solutions.
Squares and Rectangles

- Class hierarchy for a vector graphics program
- We have an abstract class `Shape` from which various geometric shapes are being derived
- Task: Add the following classes: Square, Rectangle
Summary

- Exercise Discussion
- Object Oriented Programming
  - Object Model
  - Code and Type Inheritance
  - C++ Caveat: Slicing
  - Binding: Static vs Dynamic
  - Abstract Classes and Interfaces
  - Implementation
  - Multiple Inheritance
  - Casts
Exercise 1 – Connect 4 with Inheritance

- Use inheritance to reimplement the “Connect 4” game
- Reuse your template based implementation for the implementation of the inheritance-based version of “Connect 4”
- Take care of both, the human, and the computer player

- Interfaces to be used, to be published

- If you had not implemented “Connect 4” for the first lab submission, implement it now and it will count 80% for the lab submission 1
Exercise 2

- What is the purpose of std::mem_fun, show how it can be used in a small sample.
Exercise 3

- Improve the computer player of your “Connect 4” implementation and make sure you can accommodate computer players from your colleagues. Get at least 2 computer players from other colleagues and see who is better. In order to be able to do this you need to do the following (see the web for details):
  - You need to implement a class religiously implementing the playfield interface that gives the computer player access to the playfield through the stoneat(x,y) function.
  - Make sure that your program works in such a way that two computer players can play against each other.
  - Do not change the playfield or player interfaces or otherwise your implementation will not work with your colleagues’ implementations.
Exercise 4: RPN<complex>

- Make sure your RPN calculator runs fine with complex numbers
- Sounds trivial? It is not hard but ...
- ... how do you solve the computation of the minimum with for_each?
- Use only concepts we had discussed so far in the lecture!
Next Lecture

- Liskov Substitution Principle
- Exceptions

Have a nice weekend, see you next week