C++ (3. Vorlesung)
Inheritance

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

- Constants and Constant Expressions
- Object Oriented Programming
  - Object Model
  - Code and Type Inheritance
  - Binding: Static vs Dynamic
  - Abstract Classes and Interfaces
  - Multiple Inheritance
  - Casts
  - Implementation
C/C++ const vs Java final

- C/C++ const specifies that a given variable or object a variable points to is constant
- Java final specifies that the value of a variable cannot be changed
C/C++ `const` vs Java `final`

### C++

```cpp
const int a=17;
vector<string> v1;
v1.push_back("Hello");
const vector<string> v2;
v2.push_back("Hello");
const vector<string> *vp=&v2;
vp=&v1;
vp->push_back("Hello");
vector<string> *const vq=&v2;
vq=&v1;
vq.push_back("Hello");
vector<string> *vr=&v2;
```

### Java

```java
final int a=17;
Vector v1=new Vector();
v1.add("Hello");
final Vector v2=new Vector();
v2.add("Hello");
final Vector vp=v2;
vp=v1;
vp.add("Hello");
Vector vr=v2;
```
C/C++ const vs Java final

### C++

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>const int a=17;</td>
</tr>
<tr>
<td>OK</td>
<td>vector&lt;string&gt; v1;</td>
</tr>
<tr>
<td>OK</td>
<td>v1.push_back(&quot;Hello&quot;);</td>
</tr>
<tr>
<td>OK</td>
<td>const vector&lt;string&gt; v2;</td>
</tr>
<tr>
<td></td>
<td>v2.push_back(&quot;Hello&quot;);</td>
</tr>
<tr>
<td>OK</td>
<td>int sz=v2.size();</td>
</tr>
<tr>
<td>OK</td>
<td>const vector&lt;string&gt; *vp=&amp;v2;</td>
</tr>
<tr>
<td>OK</td>
<td>vp=&amp;v1;</td>
</tr>
<tr>
<td></td>
<td>vp.push_back(&quot;Hello&quot;);</td>
</tr>
<tr>
<td>OK</td>
<td>vector&lt;string&gt;*const vq=&amp;v1;</td>
</tr>
<tr>
<td></td>
<td>vq.push_back(&quot;Hello&quot;);</td>
</tr>
</tbody>
</table>

### Java

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>final int a=17;</td>
</tr>
<tr>
<td>OK</td>
<td>Vector v1=new Vector();</td>
</tr>
<tr>
<td></td>
<td>v1.add(&quot;Hello&quot;);</td>
</tr>
<tr>
<td>OK</td>
<td>final Vector v2=new Vector();</td>
</tr>
<tr>
<td></td>
<td>v2.add(&quot;Hello&quot;);</td>
</tr>
<tr>
<td>OK</td>
<td>int sz=v2.size();</td>
</tr>
<tr>
<td>OK</td>
<td>final Vector vp=v2;</td>
</tr>
<tr>
<td></td>
<td>vp=v1;</td>
</tr>
<tr>
<td></td>
<td>vp.add(&quot;Hello&quot;);</td>
</tr>
<tr>
<td>OK</td>
<td>Vector vr=v2;</td>
</tr>
</tbody>
</table>

OK OK OK OK OK OK OK
C/C++ const (cont’d)

OK

```cpp
const vector<string> v2;
v2.push_back("Hello");
int sz = v2.size();
```

- v2.push_back is NOT OK
- BUT v2.size is OK?
- How can the compiler tell the difference?
- The secret is the declaration of the member…

```cpp
... class vector { public:
    void push_back(T elem);  
    int size() const;
};
```
C++11: Constant Expressions

- Certain initializers require a constant expression
  - Arrays stored on the stack (e.g., char buf[256])
  - Switch expressions (same in Java)
  - Functions may not be used in such constant expressions (again, same in Java)

- C++11 allows to use functions in such expressions
  - Must be marked as constexpr
  - Must be possible to evaluate them at compile time

- Useful in combination with static assert

```cpp
constexpr int min(int a, int b) { return a<b? a : b; }
int numbers[min(consta, constb)];
```
C++11: static_assert

- Allows to assert that a given condition is met at compile time
- Alternatives
  - The assert statement is evaluated at run-time
    (not desirable if condition can be evaluated at compile time)
  - The #error directive by the C++ preprocessor
    (cannot deal properly with template instantiations)

```cpp
template<typename T, int N>
class Buffer {
  static_assert(N>16, "Buffer size too small");
  ...
};
```
Agenda

- Constants and Constant Expressions
- Object Oriented Programming
  - Object Model
  - Code and Type Inheritance
  - Binding: Static vs Dynamic
  - Abstract Classes and Interfaces
  - Multiple Inheritance
  - Casts
  - Implementation
Inheritance

- Definition of a class on the basis of another
- Derived, sub, or child 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
Code and Type Inheritance

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

```cpp
class String2: public String {
  // type definition
  public:
  void tolower() {
    int n=length();
    for(int i=0; i<n; ++i) {
      (*this)[i]=tolower(*this[i]);
    }
  }
  // ...
}
```
“public” Derived Classes

- They are polymorph
  - They have the type of both 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
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
String2

---

```cpp
class String {  // type declaration+definition
  protected:
    char *strg; unsigned int len;

  public:
    ...
}

class String2 : public String {  // our implementation of the string class
  public:
    void tolower() {
      for(int i=0; i<len; ++i) { strg[i]=tolower(strg[i]); }
    }

    // ...
}
```

Style! In this case I would not consider this useful. Our previous implementation is not slower almost as readable and encapsulates the implementation of the string class.
Type Compatibility (“public” Sub Classes)

```cpp
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
}
```
Type Compatibility
("private" & "protected" Sub Classes)

class parent { // ... }
class priv_inh: private parent { // ... public: void priv(...); };
class prot_inh: protected parent { // ... public: void prot(); };

int main(int argc, char *argv[]) {
    parent p1(1,2);
    priv_inh pri(1,2);
    prot_inh pro(1,2);

    parent &p2=pri; // error, private base
    parent &p3=pro; // error, protected base
}

void priv_inh::priv(parent &p) {
    priv_inh pri(1,2);
    if(...) priv(pri); // ok, inheritance visible to ourself
}
Overriding Members

- In C++, members of the base class can be overridden
- Simply define the same member again in the sub class
- Like in Java

Almost!
Inheritance – An Example

```cpp
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; }
};

class Stack_with_Count: 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; }
};
```
Inheritance and Binding

```cpp
test-Stack.cc

void fill1(Stack s) { s.push(1); s.push(2); }
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;
    Stack_with_Count sc;
    sc.push(2); sc.push(4);
    fill1(s); fill1(sc);
    cout << sc.elements() << endl;
    fill2(s); fill2(sc);
    cout << sc.elements() << endl;
    fill3(&s); fill3(&sc);
    cout << sc.elements() << endl; }
```
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
  - ...
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)

```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; }
};
```

```cpp
class Stack_with_Count : 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 a good style!
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 Stack_with_Count: public Stack {
  ...
  virtual void push(int i) override;
}
```

```cpp
class Stack final {
  ...
};
```

```cpp
class Stack {
  ...
  int empty() final { return sp==256; }
};
```
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
Dynamic Binding and Overriding Methods

In C++

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

- Return value of a function can be restricted
  - `Stack Stack::clone()` can be replaced with
    `Stack_wc Stack_wc::clone()` in the subclass.
  - Again it can be preferable to return a pointer or a reference to the
    Stack/Stack_wc object
Callbacks and Hooks

class foo_client {
protected:
   // callback
   virtual foo *get_foo_from_cache(int i) { return NULL; }
   // default implementation
   virtual foo *get_foo_from_server(int i) { // ... }

public:
   virtual foo *get_foo(int i) {
      foo *f=get_foo_from_cache(i);
      if (f==NULL) f=get_foo_from_server(i);
      return f;
   }
   // ...
};
Abstract Classes

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

Examples
- Article data base
- A hash map that can store any data type
Article Database – Class Hierarchy
ArticleDBFactory

class ArticleDBFactory {
    Preferences p;

public:
    ArticleDBFactory(const char *config_file) {
        // ...
    }

    ArticleDB *getArticleDB(const char *group) {
        // determine the database to be used for
        // group
        if (caching_enabled)
            return new CachingRemoteArticleDB(...);
        else
            return new RemoteArticleDB(...);
    }
}
Article Database

```cpp
class ArticleDB {
public:
    virtual Article *getArticle(int nbr)=0;
    virtual void postArticle(int nbr, Article *a)=0;
};
```

- =0 after the member indicates that it is abstract
- If one or methods are 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
class RemoteArticleDB: public ArticleDB {
    string h; int p; // ...

public:
    RemoteArticleDB(const char *host, int port)
        : h(host), p(port) { // ...
    }

    virtual Article getArticle(int nbr) {
        Article a;
        // connect to server, if not already connected
        // retrieve article
        return a;
    }

    // ...

    ~RemoteArticleDB() { // close connection, if still open }
};
class CachingRemoteArticleDB: public RemoteArticleDB {
  string h; int p; // ...
  Map<int,Article> cache;

public:
  RemoteArticleDB(const char *host, int port) : h(host), p(port) { // ...

  virtual Article getArticle(int nbr) {
    Article a;
    try { a=cache.get(nbr) }
    catch (NotFoundException) {
      cache[nbr]=a=RemoteArticleDB::getArticle(nbr);
    }
    return a;
  }
  // ...
}
Using the Article Database

```cpp
int main(int argc, char *argv[]) {
    ArticleDBFactory f(".../.config");
    const char *grp;

    while(grp=getselectionfromuser()) {
        ArticleDB *db=f.getArticleDB(grp);

        // interact with database db
        // ...

        delete db; // only calls the destructor of
        // class ArticleDB!
        // the connection to our server won’t
        // be closed
    }
}
```
Solution

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

**Hint:** As soon as you have a virtual member function, implement a virtual destructor
Virtual or not Virtual?

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

- Not Virtual
  - If not useful to derive a class from your class
  - Performance *is* important (unlikely)
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 Car {
    virtual void draw() = 0;
};

struct Player {
    virtual void play() = 0;
};

class CarPlayer: public Car, public Player {
    // ...
};
```
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 CarPlayer: public Car, public Player {
    string to_string() {
        return "("+Car::to_string()+","+
            Player::to_string()+")";
    }
}
```
Multiple Inheritance: “Diamond Shape” Inheritance

- If Base has no state, there is no problem

  ![Class Diagram]

- 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;
Casts

- Casts allow to “convert” an object of type FOO into a different type BAR.
- `static_cast<T>(o)`
- `reinterpret_cast<T>(o)`
- `dynamic_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
  ```
  fraction fr(1,2);
  double f=static_cast<double>(fr);
  ```
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*>(o);
  malloc(n*sizeof(char));
  ```
dynamic_cast<T>(o)

Checks whether o is of type T (down/crosscast)

- Dynamically verified (i.e., during run-time)
- Uses Run Time Type Information (RTTI, generated for classes with a virtual method)
- Returns NULL if o is not of type T

Example

child c(...), *c1;
parent *p=&c;
c1=dynamic_cast<child*>p;
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
Slicing

```cpp
void fill1(Stack s) { s.push(1); s.push(2); }
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;
    Stack_with_Count sc;

    sc.push(2); sc.push(4);
    fill1(s); fill1(sc);
    cout << sc.elements() << endl;

    fill2(s); fill2(sc);
    cout << sc.elements() << endl;

    fill3(&s); fill3(&sc);
    cout << sc.elements() << endl;
}
```

The stack will be passed by value. In order to be able to do this, it will be “sliced”.
Slicing (cont’d)

- That is the problem with the `fill1(Stack s)` function
- Using this signature, `fill1` allocates enough memory to store an object of type `Stack`
- `Stack_wc` uses more memory than `Stack`!
- If `Stack_wc` is not implemented nicely we may get an inconsistent object (if the attributes of `Stack` are not kept in accordance with the guidelines of `Stack`)
Dynamic Binding – Implementation

- How is dynamic binding implemented?
- How do we know which member function to execute?
Virtual Method Table

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

class Stack_wc : public Stack {
    virtual void push(int i);
    virtual int pop();
};
Summary

- Constants and Constant Expressions
- Object Oriented Programming
  - Object Model
  - Code and Type Inheritance
  - Binding: Static vs Dynamic
  - Abstract Classes and Interfaces
  - Multiple Inheritance
  - Casts
  - Implementation
Exercise 1

- In what order are constructors and destructors being executed when an object is constructed or destructed respectively.
- Show the same in case of multiple inheritance.
- In case of multiple inheritance, what we can do in order to avoid the inheritance of multiple copies from the base class (diamond inheritance relationship).
- Justify your answer with a correct program.
Exercise 2 – Connect 4 with Inheritance

- Implement a simple version of “Vier Gewinnt”
  - You have a playing field of 7 columns by 6 rows.
  - Player 1 and player 2 in turns type in the column where to throw in their stone in
  - Gravity pulls the stone towards the lowest row

- After each turn display the field using ASCII art.

- Once a player has 4 of his stones in a row (horizontal, vertical, diagonal), he has won

- Try to implement the game in a modular way as we will extend it in future classes

- Interfaces to be used, to be published
Exercise 3 – Connect 4 with Inheritance (1\textsuperscript{st} Extension)

- Implement a computer player
- The computer first checks whether the player can get 4 in a row and if so tries to stop the player from winning
- If the player cannot win in the next turn, the computer throws in his stone into a random column
- You are free to improve your computer player per your liking
- Interface to be used, to be published
Next Lecture

- Liskov Substitution Principle
- Templates

Have fun solving the examples!

See you next week!