CS32 Summer 2013

Object-Oriented Programming in C++

RTTI, Advanced Inheritance, Intro to Templates

Victor Amelkin

September 4, 2013
Plan for Today

- Alternatives to Virtual Functions
  - type fields
  - dynamic_cast and RTTI
- Pure Virtual Functions and Abstract Classes
- Multiple Inheritance
- Intro to Templates
  - mainly, the topic of the last week
Alternatives to Virtual Functions

• In C++ we cannot tell the actual type of an object a pointer / reference points to (unless we do something special)

```cpp
Base *pobj = get_object_addr();
// pobj may point to Base or any its derivative
```

• What to do with it:
  – The Good: live in ignorance and use virtual functions (do not need to know the actual type of an object; the proper implementations of virtual functions are called based on the info from the virtual table)
  – The Bad: introduce “type field”
  – The OK: use dynamic_cast<T> and typeid (RTTI)
Type Field

- “Type field” is a field dedicated to keeping information about the type of every object

```cpp
class shape {
protected:
  string _type_name;
public:
  void draw() {
    if(_type_name == "triangle") {
      cout << "..drawing triangle..";
    } else if(_type_name == "circle") {
      cout << "..drawing circle..";
    } ... 
    } else { cout << "ERROR: unexpected type"; } // – new types may appear 
};

class triangle : public shape {
public:
  triangle() : _type_name("triangle") { } 
};
```

- Does not always work (what if class shape does not know how to draw a triangle?)

- Problems with extending (what if we need to add class rhombus, but we cannot change code of class shape as it is compiled in .so/.dll?)
Preliminaries on C++-Style Casts

- **C-Style casts** – same syntax for semantically different casts
  
  ```
  T var = (T)other;
  T *pvar = (T*)pother;
  ```

- **C++-Style casts:**
  
  `static_cast<T>` – used when the type is *known*
  ```
  void doit(base *pobj) {
    derived *pd = static_cast<derived*>(pobj);
  }
  ```

  `dynamic_cast<T>` – used when exact type is *unknown*
  ```
  void doit(base *pobj) {
    if(derived *pd = dynamic_cast<derived*>(pobj)) {
      pd->some_advanced_method();
    }
  }
  ```

  `const_cast<T>` – used to remove *const*'ness (bad idea)

  `reinterpret_cast<T>` – allows treating car as a cow
RTTI: dynamic_cast<T> and typeid

- **Run-Time Type Info:** Compiler can include type information – something like *type field* – in each object. (g++ enables RTTI by default.) Consequently, we have the following two constructions:

  - `dynamic_cast<T>` – *try* to cast to T; if cannot cast, return NULL

    ```cpp
    void doit(base *pobj) {
      if(derived *pd = dynamic_cast<derived*>(pobj))
        pd->some_advanced_method();
    }
    ```

  - `typeid` – like a type field

    ```cpp
    class shape {
      public:
        virtual void dummy(){} // typeid works only for polymorphic classes
        void draw() {
          // name() returns const char* "{name_length}{name}"
          string type = typeid(*this).name();
          if(type == "8triangle") { cout << "..drawing triangle.."; }
          else if(type == "6circle") { cout << "..drawing circle.."; }
          // ...
          else { cout << "Error: type not recognized"; }
        }
    };
    ```
RTTI: dynamic_cast<T> and typeid

- Use case: polymorphic partial assign

```cpp
#include <typeinfo>


class derived : public immediate_base {

public:
    void assign(const deepest_base *pother) /* override */ {
        // try to copy base state
        immediate_base::assign(pother);

        // try to copy current-level state
        const derived *pother_derived =
        dynamic_cast<const derived*>(pother);

        // if pother is of type derived
        if(pother_derived) {
            // copy pother's state's part declared in derived
            _c = pother_derived->_c;
            _d = pother_derived->_d;
        }
    }
};
```
Abstract Classes and Pure Virtual Functions

• We may not know how to implement a virtual method in a base class

```cpp
class shape {
public:
    virtual void draw() const {
        /* do not know how to draw an abstract shape */
    }
};

class triangle : public shape {
public:
    void draw() const { /* know how to draw a triangle */ }
};
```

• An empty virtual method does not enforce its implementation by the derived classes (can enforce in runtime, by throwing exceptions, but we strive for compile-time error checking)

• Sometimes we want to inherit only *interface*, not *implementation* (C++ does not have interfaces in Java/C# sense)

```
"interface" IDrawable {
    void draw() const;
    void resize(const rectangle &frame_to_fill);
}
```
Abstract Classes and Pure Virtual Functions

- Pure virtual function – a virtual function without implementation

```cpp
#include <iostream>

class shape {
public:
  virtual void draw() const = 0;
  virtual ~shape() { }
  void some_implemented_method() { ... }
};

class pretty_shape : public shape {
  /* still have no idea how to draw() */
};

class triangle : public pretty_shape {
public:
  void draw() const { /* drawing triangle */ }
};
```

- Class having at least one pure virtual function is abstract
- Abstract classes cannot be instantiated
- Class having no pure virtual functions is concrete
Abstract Classes and Pure Virtual Functions

- C++ Interfaces = abstract classes consisting of pure virtual functions (and, usually, an empty virtual destructor)

```cpp
class IPlugin { // an interface for a browser plugin
public:
    virtual string plugin_name() = 0;
    virtual bool activate(context *pcontext) = 0;
    virtual void run() = 0;
    virtual ~IPlugin() { } // the only “impurity”
};

class YoutubeDownloaderPlugin : public IPlugin {
    /* implements methods of IPlugin */
};

IPlugin *pplugin = new YoutubeDownloaderPlugin(...);
firefox_connector.add_plugin(pplugin);
```

- If a class is derived from an interface class (“interface”), we usually say that the class implements that interface

- Class may implement many interfaces (see multiple inheritance)
Intermezzo: Composition vs. Inheritance

- Inheritance (“is-a”) – extending classes of *similar nature*

```cpp
class vehicle { ... };
class car : public vehicle { ... };
class delorean : public car { ... };
```

- Composition (“has-a”) – extending classes with *members* of possibly different nature

```cpp
class car {
private:
    engine &_engine;
    list<passenger*> _passengers;
}
```

- Composition is better in that classes do not share their *internals* with derived classes (unless you decide to never use *protected* members)

- Composition is used more often than inheritance

- Prefer (relatively) small classes with clear responsibilities to “fat” classes that do *everything*; combine small classes using composition

- “Flat” classes may be preferred when for remote calls
Multiple Inheritance

- Multiple Inheritance – class derivation from more than one base class

```c++

class Vehicle { ... };

class Mechanism { ... };

class IDrivable { ... };

class Car :
    public Vehicle,
    protected Mechanism,
    public IDrivable { ... }
```

- Interface and implementation are inherited from each base
Problems of Multiple Inheritance

- Member collision and ambiguity resolution

```cpp

class base1 {
public:
    virtual void print() const { ... }
};

class base2 {
public:
    virtual void print() const { ... }
};

class derived : public base1, public base2 {
};

derived obj;
obj.print(); // error: request for ‘print’ is ambiguous
obj.base2::print(); // ok (calling base2's print impl)
```
Problems of Multiple Inheritance

- Cyclic inheritance graph (simplest cycle – diamond):
  
Problems of Multiple Inheritance

- By default, repeated base class is replicated
Problems of Multiple Inheritance

- To prevent replication, virtual base classes are used

Multiple Inheritance

- There are many other problems with multiple inheritance
- That is why multiple inheritance is deliberately *not* included in Java and C# (exception – inheriting multiple interfaces)
- Safe uses of multiple inheritance in C++:
  - Implementing interfaces
  - Mix-in classes (– do not live on their own)

```cpp
class uncopyable { // ← mix-in class
protected:
    uncopyable() { }
private:
    uncopyable(const uncopyable &other);
    uncopyable& operator=(const uncopyable &other);
};

class myclass : ..., public uncopyable { ... }

myclass obj1;
myclass obj2(obj1); // compile-time error
```
Templates

- C++ templates allow to write generic code using types and values as parameters

```cpp
template<typename TChar>
class String {
private:
    TChar *pchars;
    int len;
public:
    String();
    explicit String(const TChar *src);
    String(const String &other);
    TChar & operator[](int i) { return pchars[i]; }
    ...
};
```

```cpp
using PlainString = String<char>;

PlainString plain_str;
String<wchar_t> unicode_str;
String<bool> boolean_str;
```
Templates

- Each time a template is used with a unique set of template arguments, a new class is generated by the compiler:

  ```cpp
  // 3 different versions of class String are generated
  String<char> plain_str;
  String<wchar_t> unicode_str;
  String<bool> boolean_str;
  ```

- This generating process is called *template instantiation*.

- Each such class generated for a particular template argument list is called *template specialization*:

  ```cpp
  vector<car> myvec; // instantiating vector<T>
  ```
Templates and STL – next week
Before you leave...

• PA4 – due Thursday night

• PA5 will be released Thursday late afternoon
  – due: in ~7-8 days (TBA)

• TA Evaluations
~ Thanks ~