Casting and polymorphism Enumeration

Shahram Rahatlou

SAPIENZA
UNIVERSITÀ DI ROMA

http://www.roma1.infn.it/people/rahatlou/programmazione++/

Corso di Programmazione++

Roma, 25 May 2009

Polymorphic vector of Person

```
vector<Person*> createPeople() {
  vector<Person*> people; // owns its objects

  people.push_back(new Person("John"));
  people.push_back(new Student("Susan", 123456));
  people.push_back(new GraduateStudent("Paolo", 9856, "Physics"));

  return people; // transfers ownership of content to client
}


int main() {

  vector<Person*> people = createPeople();

  // polymorphic call to ::print()
  for(int i=0; i< people.size(); ++i) {
    people[i]->print();
  }

  // we are responsible for content of people
  for(int i=0; i< people.size(); ++i) {
    delete people[i];
  }

  return 0;
}
```

Vector filled with objects of different sub-classes

At runtime the compiler calls the appropriate **`::print()`** method based on type

Vector filled with objects of different sub-classes

- Suppose I want to changing the major of each **`GraduateStudent`** to be Chemistry by calling **`GraduateStudent::setMajor(const string& major)`**
- I cannot call **`people[i]->setMajor(“Chemistry”)`** … why?

Downcasting via **dynamic_cast**

- The problem is that not all **`people[i]`** are of type GraduateStudent
- Downcasting consists in specifying a sub-type in inheritance hierarchy for base-class pointer

```
int main() {

  vector<Person*> people = createPeople();

  // polymorphic call to ::print()
  for(int i=0; i< people.size(); ++i) {
    people[i]->print();

    GraduateStudent* gs =
     dynamic_cast<GraduateStudent*>(people[i]);

    // check whether gs is a valid pointer
    // gs == 0 for all classes except GraduateStudent
    if( gs != 0 ) gs->setMajor(“Chemistry”);
  }

  // we are responsible for content of people
  for(int i=0; i< people.size(); ++i) {
    delete people[i];
  }

  return 0;
}
```

- Operator **`dynamic_cast`** provides a pointer to the specified sub-class
- It’s user’s responsibility to check the validity of the pointer
- Null pointer means that pointed object is not of the specified sub-class
- A valid pointer allows us to call sub-class methods

Don’t abuse `dynamic_cast`!

- Abusing downcasting quickly and easily breaks polymorphism

- In order to down cast you need to specify the subtype in the code

- Remember: polymorphism means you can use objects of any sub-type indifferently in your code
  - Downcasting breaks this!

- Use downcasting only if nothing else can be done
  - You are provided with a set of base-class pointers e.g from a database

Enumerators

- Enumerators are set of integers referred to by identifiers

- There is natural need for enumerators in programming
  - Months: Jan, Feb, Mar, …, Dec
  - Fit Status: Successful, Failed, Problems, Converged
  - Shapes: Circle, Square, Rectangle, …
  - Colors: Red, Blue, Black, Green, …

- Enumerators make the code more user friendly
  - Easier to understand human identifiers instead of hardwired numbers in your code!

- You can redefine the value associated to an identifier w/o changing your code

Example of Enumeration

```
// enum1.cpp
#include <iostream>
using namespace std;

int main() {
  enum FitStatus  { Succesful, Failed, Problems, Converged };

  FitStatus status;

  status = Succesful;
  cout << "Status: " << status << endl;

  status = Converged;
  cout << "Status: " << status << endl;

  return 0;
}
```

By default the first identifier is assigned value 0

Don't forget this one!

```
$ g++ -o enum1 enum1.cpp
$ ./enum1
Status: 0
Status: 3
```

enums can be used as integers but not vice versa!

Another Example of Enumeration

```
// enum2.cpp
#include <iostream>
using namespace std;

int main() {
  enum Color  { Red=1, Blue=45, Yellow=17, Black=342 };

 Color col;

  col = Red;
  cout << "Color: " << col << endl;

  col = Black;
  cout << "Color: " << col << endl;


  return 0;
}
```

You can use arbitrary int values for each of your identifiers

```
$ g++ -o enum2 enum2.cpp
$ ./enum2
Color: 1
Color: 342
```

Common errors with enumeration

```
// enum2.cpp
#include <iostream>
using namespace std;

int main() {
  enum Color  { Red=1, Blue=45, Yellow=17, Black=342 };

 Color col;

  col = Red;
  cout << "Color: " << col << endl;

  col = Black;
  cout << "Color: " << col << endl;

  col = 45; //assign int to enum

  int i = Red;

  return 0;
}
```

Can't assign an int to an enum!

But you can assign an enum to an int

```
$ g++ -o enum3 enum3.cpp
enum3.cpp: In function `int main()':
enum3.cpp:16: error: invalid conversion from `int' to `main()::Color'
```

Enumeration in Classes

```
#ifndef Fitter_h_
#define Fitter_h_
// Fitter.h
namespace analysis {

  class Fitter {
    public:
      enum Status { Succesful=0,
                    Failed,
                    Problems };

      Fitter() { };

      Status fit() {
        return Succesful;
      }
    private:
  }; // class Fitter
} //namespace
#endif
```

```
//enum4.cpp
#include "Fitter.h"
#include <iostream>
using namespace std;

int main() {

  analysis::Fitter myFitter;

  analysis::Fitter::Status stat =
                             myFitter.fit();

  if( stat == analysis::Fitter::Succesful ) {
    cout << "fit succesful!" << endl;
  } else {
    cout << "Fit had problems ... status = "
         << stat << endl;
  }

  return 0;
}
```

Use complete qualifier including namespace and class to use PUBLIC enumerators

```
$ g++ -o enum4 enum4.cpp
$ ./enum4
fit succesful!
```

Enumeration and Strings

```
// color.cpp

#include <iostream>
#include <map>
using std::cout;
using std::endl;

int main() {
  enum Color  { Red=1, Blue=45,
                Yellow=17, Black=342 };

 Color col;

  std::string strCol[400]; // quick ‘n’ dirty

  strCol[Red] = std::string("Red");

  //std::map<Color,std::string> colorMap;
  //colorMap[Red] = std::string("Red");
  //colorMap[Black] = std::string("Black");

  // better solution
  std::map<int,std::string> colorMap;
  colorMap[Red] = std::string("Red");
  colorMap[Black] = std::string("Black");

  col = Red;
  //cout << "Color: " << strCol[col] << endl;
  cout << "Color: " << colorMap[col] << endl;

  return 0;
}
```

- No automatic conversion from enumeration to strings
- You can use vectors of strings or std::map to assign string names to enumeration states

Can’t use Color as index

Why?

… because there is no way to compare two states of color.
Is Red smaller than Blue?
Compiler doesn’t know!
must provide a comparison function

```
$ g++ -o color color.cpp
$ ./color
Color: Red
```