Classes II: Type Conversion, Friends, …

For : COP 3330.
Object oriented Programming (Using C++)
http://www.compgeom.com/~piyush/teach/3330

Piyush Kumar

Abstraction and Encapsulation

- Abstraction: Separation of interface from implementation
- Encapsulation: Combining lower level elements to form higher-level entity.
- Access Labels (public/private/protected) enforce abstraction and encapsulation.

Concrete Types.

- A concrete class exposes, rather than hides, its implementation
- Example : pair<> (defined in <utility>)
- Exists to bundle two data members.

```
#include<iostream>
#include<utility>
#include<string>

int main(){
  std::pair<std::string, int> pr1; //-- Declare a pair variable
  std::pair<std::string, int> pr2("heaven", 7); // Declare - initialize with constructor.
  std::cout << pr2.first << "=" << pr2.second << std::endl; // Prints: heaven=7
  return 0;
}
```

Benefits of Abstraction & Encapsulation

- Class internals are protected from user-lever errors.
- Class implementation may evolve over time without requiring change in user-level code.

More on Class definitions

```
class Screen {
public:

private:
        std::string contents;
        std::string::size_type cursor;
        std::string::size_type height,width;
…
```

Using Typedefs to streamline classes.

```
class Screen {
public:
        using index = std::string::size_type;
private:
        std::string contents;
        index cursor;
        index height,width;
…
};

inline Screen::index Screen::get_cursor() const{
        return cursor;
}
```

Class declaration

- class Screen; // declaration of the class
- Forward declaration: Introduces the name Screen into the program and indicates that Screen refers to a class name.
- Incomplete Type: After declaration, before definition, Screen is an incomplete type. It's known screen is a type but not known what members that type contains.

Class declaration for class members.

- Because a class is not defined until its class body is complete, a class cannot have data members of its own type.
- A class can have data members that are pointers or references to its own type.

```
class Human {
        Screen window;
        Human *bestfriend;
        Human *father, *mother;
        …
}
```

Using **this** pointer

- Implement Screen class so that:
  - myScreen.move(4,0).set('#');
- Can replace the following two lines:
  - myScreen.set(4,0);
  - myScreen.set('#');

Using this pointer

- Return reference to Screen in the member functions.
  - Screen& move(index r, index c);
  - Screen& set(char);
  - Implementation:

```
Screen& Screen::move(index r,index c){
   index row = r* width; cursor = row +c;
   return *this;
}
```

Using this pointer

- Beware of const:

```
const Screen& Screen::display(ostream &os) const
{
        os << contents;
        return *this;
 }
```

- myScreen.move(4,0).set('#').display(cout)

Mutable data members

- "Sometimes", you might want to modify a variable inside a const member function.
- A mutable data member is a member that is never const (even when it is a member of a const object).
- **mutable** => removes **const** qualification

Mutable data members

```
class Screen {
public:

private:
        mutable size_t access_ctr;

};

void Screen::do_display(std::ostream& os) const {
        ++access_ctr;  // keep count of calls to any member func.
        os << contents;
}
```

Guideline.

- Never repeat code.
- If you have member functions that need to have repeated code, abstract it out in another function and make them call it (maybe inline it).

Type conversion: Revisited

- int i; float f;
- f = i; // implicit conversion
- f = (float)i; // explicit conversion
- f = float(i); // explicit conversion

Type conversions: revisited

- Can convert from one type into “this” type with constructor
  - Bitset(const unsigned long x);
- How do we convert from “this” type to something else?
  - Create an operator to output the other type
  - Later.

Implicit Type conversion

- A constructor that can be called with a single argument defines an implicit conversion from the parameter type to the class type.

```
Class SalesItem {
Public:
        SalesItem(const std::string &book = “ “)
          : isbn (book), units_sold(0), revenue(0.0) {}
…
}

String null_book = “9-999-9999-9”;
Item.sale_isbn(null_book); // implicit type conversion..
```

Beware:

```
class String {
public:
        String( int ); // Allocation constructor
        // ...
};

// Function that receives an object of type String as an argument
void foo( const String& );

// Here we call this function with an int as argument
int x = 100;
foo( x ); // Implicit conversion => foo( String( x ) );
```

User defined implicit type conversion

```
#include<iostream>
#include<string>
class String{
  std::string str;
  public:
    String(char * cp){str = cp;};
    operator const char * () const;
};

String::operator const char * () const{
  std::cout << "type conversion" << std::endl;
  const char * out = str.c_str();
  return out;
}

void foo(const String& aString){};
void bar(const char * fewChars){};

int main(){
  foo("hello"); // Implicit type conversion char* -> String
  String peter = "pan";  // Calls parametrized constructor. Treats "pan" as char *
  static_cast<const char *>(peter); // implicit type conversion String -> const char*
  bar(peter);
  return 0;
}
```

Suppressing implicit conversions.

- Use “explicit” before conversion constructors.

explicit String(char* cp); // Constructor

Friends.

- **friend** function/classes
  - Can access **private** and **protected** (more later) members of another class
  - **friend** functions are not member functions of class
    - Defined outside of class scope
  - A Friend declaration begins with the keyword “friend”

Friends

- Properties
  - Friendship is granted, not taken
  - NOT symmetric
    - if B a **friend** of A, A not necessarily a **friend** of B
  - NOT transitive
    - if A a **friend** of B, B a **friend** of C, A not necessarily a **friend** of C.

Friends

- **`friend`** declarations
  - **`friend`** function
    - Keyword **`friend`** before function prototype in class that is giving friendship.
      - **`friend int myfunc( int x );`**
      - Appears in the class granting friendship
  - **`friend`** class
    - Type **`friend class`** *Classname* in class granting friendship
    - If **`ClassOne`** granting friendship to **`ClassTwo`**,

      `friend class ClassTwo;`

      appears in **`ClassOne`**'s definition

Friends

- Why use friends?
  - to provide more efficient access to data members than the function call
  - to accommodate operator functions with easy access to private data members
- Be careful: Friends can have access to everything, which defeats data hiding.
- Friends have permission to change the internal state from outside the class. Always use member functions instead of friends to change state

An example

```
#include <iostream>
#include <string>

class SalesItem {
    friend bool operator==(const Sales_item&, const SalesItem&);
    friend std::istream& operator>>(std::istream&, SalesItem&);
    friend std::ostream& operator<<(std::ostream&, const SalesItem&);
// other members as before
public:
    // added constructors to initialize from a string or an istream
    SalesItem(const std::string &book):
            isbn(book), units_sold(0), revenue(0.0) { }
    SalesItem(std::istream &is) { is >> *this; }
public:
    // operations on SalesItem objects
    // member binary operator: left-hand operand bound to implicit this pointer
    SalesItem& operator+=(const SalesItem&);
    // other members as before
…
```

Class to handle an integer sequence

```
/* File: numbers.hpp */

#include<iostream>
#include<string>
#include<ostream>

class Sequence{
  private:
    int * numbers;
    int length;
  public:
    Sequence();
    Sequence(const int, std::string &);   // Parameterized constructor
    Sequence(const Sequence&);            // Copy constructor
    ~Sequence();                          // Destructor

    void operator=(const Sequence &);  // Overloaded assignmnt
    int operator()(int);
    friend std::ostream& operator<<(std::ostream&, const Sequence&);
};
```

```
/* File: part of numbers.cpp */

#include "numbers.hpp"
#include<iostream>
#include<string>
#include<sstream>
#include<ostream>

Sequence::Sequence(){
  numbers = nullptr;
  length = 0;
}

Sequence::Sequence(const int n, std::string & instring){
  std::stringstream ss;
  ss << instring;
  length = n;
  numbers = new int[length];
  for(int i = 0; i < length; i++){
    ss >> numbers[i];
  }
}
```

```
/* File: part of numbers.cpp */

Sequence::Sequence(const Sequence & inseq){
  std::cout << "Copy constructor" << std::endl;
  length = inseq.length;
  numbers = new int[length];
  for(int i = 0; i < length; i++){
    numbers[i] = inseq.numbers[i];
  }
}

Sequence::~Sequence(){
  if(numbers != nullptr){
    delete [] numbers;
    numbers = nullptr;
  }
}
```

```
/* File: part of numbers.cpp */

int Sequence::operator()(int index){
  return numbers[index];
}


std::ostream& operator<<(std::ostream& out, const
Sequence& seq){
  out << "Overloaded <<" << std::endl;
  for(int i = 0; i < seq.length; i++){
    out << seq.numbers[i] << " ";
  }
  out << std::endl;
  return out;
}
```

```
/* File: main.cpp */

#include<iostream>
#include "numbers.hpp"

int main(){
  std::string instr = "12 34 56 78 88 77";
  Sequence w(4, instr);
  std::cout << w(2) << std::endl;  // overloaded ()

  std::cout << w;  // overloaded << operator
}
```

Output:
56
Overloaded <<
12 34 56 78