


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


```

      //-- Declare a pair variable.
      pair<string, int> pr1;
      //-- Declare and initialize with constructor.
      pair<string, int> pr2("heaven", 7);
      cout << pr2.first << " = " << pr2.second << endl;
      // Prints heaven=7
      
```



## Benefits of Abstraction & Encapsulation

- Class internals are protected from user-level 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:
    typedef std::string::size_type index;
private:
    std::string contents;
    index cursor;
    index height,width;
    ...
}

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

```

## Class declaration

- o `class Screen; // declaration of the class`
- o Forward declaration: Introduces the name `Screen` into the program and indicates that `Screen` refers to a class name.
- o 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.

- o Because a class is not defined until its class body is complete, a class cannot have data members of its own type.
- o 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

- o How do we implement the `Screen` class so that the following is allowed:
  - `myScreen.move(4,0).set('#')`
  - Equivalent to: `myScreen.set(4,0); myScreen.set('#')`

## Using this pointer

- o 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

- o Beware of `const`:

```
const Screen& Screen::display(ostream &os) const  
{  
    os << contents;  
    return *this;  
}
```
- o `myScreen.move(4,0).set('#').display(cout)`

## Mutable data members

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

## 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 Sales_item {
Public:
    Sales_item(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 length ); // Allocation constructor
    // ...
};

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

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

## Use of implicit type conversion

```
class String {
public:
    String( char* cp ); // Constructor
    operator const char* () const;
    // Conversion operator to const char*
    // ...
};

void foo( const String& aString );
void bar( const char* someChars );

// main.cc
int main() {
    foo( "hello" ); // Implicit type conversion char* -> String
    String peter = "pan";
    bar( peter ); // Implicit type conversion String -> const char*
}
```

## Suppressing implicit conversions.

- o Use "explicit" before conversion constructors.

```
explicit String( char* cp ); // Constructor
```

## Friends.

- o **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

- o 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

- o 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
- o Be careful: Friends can have access to everything, which defeats data hiding.
- o 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 Sales_item {
    friend bool operator==(const Sales_item&, const Sales_item&);
    friend std::istream& operator>>(std::istream&, Sales_item&);
    friend std::ostream& operator<<(std::ostream&, const Sales_item&);
    // other members as before
public:
    // added constructors to initialize from a string or an istream
    Sales_item(const std::string &book):
        isbn(book), units_sold(0), revenue(0.0) { }
    Sales_item(std::istream &is) { is >> *this; }
public:
    // operations on Sales_item objects
    // member binary operator: left-hand operand bound to implicit this pointer
    Sales_item& operator+=(const Sales_item&);
    // other members as before
    ...
};
```