

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. #includecutility #includecstring int main(){ int main(){ std::pair<std::string, int> pr2("heaven", 7); // Declare - initialize with constructor. std::pair<std::string, int> pr2("heaven", 7); // Declare - initialize with constructor. std::cout << pr2.first << "=" << pr2.second << std::end1; // 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.

```
class Screen {
public:
private:
std::string::size_type cursor;
std::string::size_type height,width;
...
```

Class declaration

- o 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.

Using this pointer

- o Implement Screen class so that:
 - myScreen.move(4,0).set('#');
- o 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

o Beware of const:

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

• • 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).
- o 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;
}</pre>
```

• • Guideline.

- o 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

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

Item.sale_isbn(null_book); // implicit type conversion..

Beware:

User defined implicit type conversion #include-datream#include-

Suppressing implicit conversions.

 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

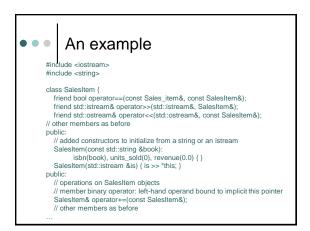
- Properties
 - Friendship is granted, not taken
 - NOT symmetric
 - if B a friend of A, A not necessarily a friend of B
 - NOT transitive
 - if A a friend of B, B a friend of C, A not necessarily a friend of C.

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
- 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



```
Class to handle an integer sequence

/* File: numbers.hpp */

#include-ciostreams-
#include-cstreams-
#include-cstreams-
#include-cstreams-
class Sequence(
private:
    int *numbers;
    int length;
    public:
    Sequence();
    Sequence();
    Sequence(const int, std::string &); // Parameterized constructor
    Sequence(const Sequence(s); // Copy constructor
    Sequence(const Sequence(s); // Copy constructor
    Sequence(const Sequence(s); // Overloaded assignmnt int operator(pfint);
    irriend std::ostream& operator<-(std::ostream&, const Sequence&);
};
```

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

#include numbers.pp*

#include-catings
#i
```

