Defining New Types

Lecture 21

Hartmut Kaiser

hkaiser@cct.lsu.edu

http://www.cct.lsu.edu/~hkaiser/spring_2015/csc1254.html
Programming Principle of the Day

• Hide Implementation Details
  • Hiding implementation details allows change to the implementation of a code component while minimally affecting any other modules that use that component.
  • Encapsulation is key
    • A language mechanism for restricting access to some of the object's components
    • A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data

Defining New Types

• C++ has two kinds of types
  • built-in types
    • int, double, char, etc.
  • class types (user defined types)
    • string, vector, istream
  • Rely entirely on built-in types and other class types and generally available language facilities

• Design of C++ rests on the idea of creating user defined types usable similar to built-in types
  • Requires substantial language support
  • Taste and judgment in class design
student_info Type Revisted

- We wrote the student_info type and related functions
  - However, these were not well suited for other programmers to use
    - Any newly created student_info object required first to read data into it
      - Otherwise the values would have been uninitialized
    - No way to check whether held data was valid
      - Well, without looking at member values
      - Requires internal knowledge of student_info
  - Assumption was that after grades were read they wouldn’t change
  - Interface to student_info was scattered (separate, unrelated functions)
Class (User Defined) Types

- Class types allow to group several data items into one entity

- That’s what we used:

```cpp
struct student_info
{
    std::string name;
    double midterm, final;
    std::vector<double> homework;
};
```

- Four data elements: data members
  - A string, two double’s, and a vector<double>
Class (User Defined) Types

- Programmers may – and must – handle those data items directly
  - They may – because nothing is preventing them
  - They must – because there is no other way

- It would be better to hide implementation details of how things are stored
  - Manipulate members through functions only
  - This forms the interface of our class
Class (User Defined) Types

- Important general remarks
  - Always use fully qualified names in headers
    - There is no guarantee that a using directive is in effect
    - But adding a using directive yourself to a header might break other code
  - Never put a using directive into a header
### Member Functions

- Let’s start with adding interface function to read the student grades and to calculate the overall grade:
  ```
  struct student_info
  {
    std::string name;
    double midterm, final;
    std::vector<double> homework;

    std::istream& read(std::istream&); // added
    double grade() const; // added
  };
  ```

- We added two member functions
  - The ‘const’ means, that grade will not change any members
Member Functions

• A member function is a function that is a member of a class object
  • We have already seen those (v.size(), v.begin(), etc.)

• In order to call a member function, our users must nominate the object of which the function to be called is a member:

  student_info s;
  s.read(cin); // calling ‘read()’ for object instance ‘s’
  cout << s.grade(); // calling ‘grade()’ for ‘s’
Member Functions

• Member function implementation:
  
  ```cpp
  istream& student_info::read(istream& in)
  {
    in >> name >> midterm >> final;
    read_hw(in, homework);
    return in;
  }
  ```

• The name of the function is different
  • It’s `student_info::read` instead of plain `read`

• This function is a member of a `student_info` object
  • No need to pass a `student_info` object as an argument

• We access the data elements of our object directly
  • Instead to `s.midterm` we refer to just `midterm`.
Member Function Specifics

• student_info::read instead of read
  • ‘::’ scoping operator (like in string::size_type)
  • That means read is now part of student_info, i.e. a member function
  • No need to pass a student_info object as an argument
    • The student_info instance is implicit for every call
    • The instance we want to invoke the function for is specified using the dot-notation:

```
student_info s;
std::cin >> s;  // read from stdin
```
Member Function Specifics

- The references to the members inside read are unqualified
  - They are references to members of the object instance on which we are operating
  - If we call s.read(cin) for a student_info object named s, then we are operating on object s.
  - When read() uses midterm, final, and homework, it will be using s.midterm, s.final, and s.homework respectively
Member Functions

• Member function implementation:

```cpp
double student_info::grade() const
{
    return ::grade(midterm, final, homework);
}
```

• Same considerations: name, implicit student_info instance, and unqualified members

• Using ‘::’ in front of a name forces using a version of that name in global namespace
  • Without it the compiler would refer to student_info::grade itself, resulting in a compilation error (why?)
Member Function Constness

• Appended const – why do we need it?

// original
double grade(student_info const&) { ... }
// member-function version
double student_info::grade() const { ... }

• const member function
  • Does not change any of the members
  • This const is part of the function signature
    • Needs to be specified for the declaration and the definition
Protection

• Users of student_info do not need to manipulate the data members of our class
  • But they still could!

• Hide the data members (make them inaccessible):

```cpp
class student_info
{
   public:
      // interface goes here
      double grade() const;
      std::istream& read(std::istream&);

   private:
      // implementation goes here
      std::string name;
      double midterm, final;
      std::vector<double> homework;
};
```
Protection

- class instead struct
  - Essentially, 100% equivalent
  - Different defaults in terms of accessibility of members

- Protection labels (key words): ‘public’, ‘private’
  - Public: accessible by non-members
  - Private: not accessible by non-members
Accessor Functions

- We hid data members from modification but also from reading :P
- Let’s make name available again:

```cpp
class student_info
{
public:
    // interface goes here
    double grade() const;
    std::istream& read(std::istream&);
    std::string name() const { return n; } // added!

private:
    // implementation goes here
    std::string n; // name change!
    double midterm, final;
    std::vector<double> homework;
};
```
Comparing student_info Instances

• In order to compare (sort!) student_info instances we need to rewrite compare():

```cpp
bool compare(student_info const& x, student_info const& y)
{
    return x.name() < y.name();
}
```

• Still a global function

• Nevertheless it’s part of the interface of student_info: add it to the same header file
Checking for Validity

• This code will throw (why?):
  student_info s;
  cout << s.grade() << endl; // exception: s has no data

• We need a way to check whether a student_info instance ‘is valid’ (has homework):
  class student_info
  {
    public:
    bool valid() const { return !homework.empty(); } 
    // ...as before
  };

• Allows to check before calling grade
Constructors

• What happens when objects are created?
  • Constructor is called – leaves object in defined state

• Special member functions that define how objects are initialized
  • No way to call a constructor explicitly
  • Called automatically by the compiler whenever object instance is created

• If no constructor is defined, compiler synthesizes one (default constructor) which calls the default constructors of all members
Constructors

• We define 2 constructors
  • Default constructor (no argument)
  • Construct from istream

```cpp
class student_info
{
public:
    // construct an empty student_info object
    student_info();
    // construct one by reading from a stream
    student_info(std::istream&);

    // ... as before
};
```
The default Constructor

- No argument constructor is called ‘default constructor’:
  
  ```cpp
  student_info::student_info()
    : midterm(0), final(0)
  {}  
  ```

- Initialize built in data types
  - Otherwise would contain garbage

- Compiler (default) initializes all class types, but could be explicit:
  
  ```cpp
  student_info::student_info()
    : n(), midterm(0), final(0), homework()
  {}  
  ```
The Constructor (any Constructor)

• When we create a new class object, several steps happen in sequence:
  1. The implementation allocates memory to hold the object.
  2. It initializes the object, as directed by the constructor's initializer list.
  3. It executes the constructor body.

• Initialization of all members happens before the constructor body begins execution
  • Regardless whether there is a initializer list or not
Constructors with Arguments

• Any constructor can take arguments allowing to initialize the object in different ways:

```cpp
student_info::student_info(std::istream& is)
{
    read(is);
}
```

• Even if there is no initializer list, ‘n’ and ‘homework’ are default constructed before body is executed
Using the New student_info Class

```cpp
int main()
{
    vector<student_info> students;
    student_info record;
    string::size_type maxlen = 0;
    // read and store the data
    while (record.read(cin)) {
        // changed:
        maxlen = max(maxlen, record.name().size());  // changed
        students.push_back(record);
    }
    // alphabetize the student records
    sort(students.begin(), students.end(), compare);
    // ...
}
```
Using the New student_info Class

```cpp
// write the names and grades
for (vector<student_info>::size_type i = 0;
     i != students.size(); ++i)
{
    cout << students[i].name() // changed
        << string(maxlen + 1 - students[i].name().size(), ' ');
    try {
        double final_grade = students[i].grade(); // changed
        streamsize prec = cout.precision();
        cout << setprecision(3) << final_grade
             << setprecision(prec) << endl;
    } catch (domain_error e) {
        cout << e.what() << endl;
    }
}
return 0;
```