Records & Classes

Assistant Prof. Dr. Turgay İBRİKÇİ
Monday, December 06, 2010

What to do with records?
- Declaring records
- Accessing records
- Accessing the field of a record
- What is a union?
- Can records be in arrays?

Records
- Recall that elements of arrays must all be of the same type
  - scores: 85, 90, 92, 57, 68, 90, ...
  - 0, 1, 2, 3, 4, 5
- In some situations, we wish to group elements of different types
  - employee: R. Jones, 123 Elm, 6/12/55, $14.75

Records
- RECORDS are used to group related components of different types
- Components of the record are called fields
- In C++
  - record called a struct (structure)
  - fields called members
  - employee: R. Jones, 123 Elm, 6/12/55, $14.75

Records
- C++ struct
  - structured data type
  - fixed number of components
  - elements accessed by name, not by index
  - components may be of different types

Declaring struct Variables
- Given
  - struct part_struct {
      char descrip [31], part_num [11];
      float unit_price;
      int qty;
  };
- Declare:
  - Use struct name as a type
  - part_struct new_part, old_part;
Accessing Components

- Use the name of the record
  the name of the member
  separated by a dot.

- The dot is called the member selector

```cpp
old_partQty = 5;
cout << new_part.descr;
```

Aggregate Operations with Structures

- Aggregate means that the operation acts on the structure as a whole whereas normal operations deal with individual members. We are able to use the equality operator because both structures are instances of the same structure type (Games).
- Recall that arrays had none (except reference parameter)
- Structures DO have aggregate operators
  - assignment statement =
  - parameter (value or reference)
  - return a structure as a function type

```cpp
aggregate mean that the operation acts on the
parameter ()
```

Limitations on aggregate operations

- no IO

```cpp
cout << old_part;
cin >> new_part;
```

- no arithmetic operations

```cpp
old_part = new_part(old_part);
```

- no comparisons

```cpp
if(old_part=new_part)
cout<<
```

Input/Output

- There are no aggregate input/output operations on struct.
  - Data in a struct variable must be read one member at a time.
  - Contents of a struct must be written one member at a time.

```cpp
struct Variables and Functions

- A struct variable can be passed as a parameter either by value or by reference.
- A function can return a value of the type struct
```
Contrast Arrays and \textbf{struct}s

<table>
<thead>
<tr>
<th>Aggregate Operation</th>
<th>Array</th>
<th>Struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Assignment</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Input/output</td>
<td>No (except strings)</td>
<td>No</td>
</tr>
<tr>
<td>Comparison</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Parameter passing</td>
<td>By reference only</td>
<td>By value or by reference</td>
</tr>
<tr>
<td>Function returning a value</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Arrays of Records

- First declare a struct (such as part\_struct)
- Then specify an array of that type
  \begin{verbatim}
  part\_struct part\_list[50];
  \end{verbatim}
- Access elements of the array, elements of the struct
  \begin{verbatim}
  for (x = 0; x <50; x++)
    cout << \textcolor{red}{part\_list[x].\textit{field}};
  \end{verbatim}

Records with Arrays

- Example
  \begin{verbatim}
  const arraySize = 1000;
  
  struct list\_Type
  {
    int elements[arraySize];
    //array containing the list
    int list\_Length;
    //length of the list
  };
  \end{verbatim}

Hierarchical Records

- Defn => records where at least one of the components is, itself, a \textbf{record}
- Example:
  \begin{verbatim}
  struct inventory\_struct {
    part\_struct part;
    int qty\_sold, re\_order\_qty;
    vendor\_struct vendor;
  };
  \end{verbatim}

Unions

- Defn => a struct that holds only one of its members at a time during program \textbf{execution}
- At run time, \texttt{\textit{how\_many}} contains either
  - an int
  - a float
  - a long ...
- But ... never all three

Enumeration Constants

- Set of integer constants represented by identifiers
- Enumeration constants are like symbolic constants whose values are automatically set
  - Values start at 0 and are incremented by 1
  - Values can be set explicitly with =
  - Need unique constant names
- Example:
  \begin{verbatim}
  enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
  \end{verbatim}
  - Creates a new type enum Months in which the identifiers are set to the integers 1 to 12
  - Enumeration variables can only assume their enumeration constant values (not the integer representations)
Example

```cpp
#include <iostream>

using namespace std;

enum Days // Declare enum type Days
{
    sunday = 0, // sunday = 0 as well
    saturday, // saturday = 0 by default
    tuesday, // tuesday = 2
    monday, // monday = 1
    case 1:
};
```

```cpp
void do_it (part_struct part);
```

Choosing Data Structures

- Strive to group logical elements of a structure together
  • calls for hierarchical structures
- Push details of entities down to lower levels of the structure
- Data Abstraction <=> separation of logical properties of a data type from its implementation

Testing and Debugging Hints

- Declaration of a `struct` type must end with a semicolon `;`
- Be sure to specify the full member selector when referencing a component of a struct variable
  • don’t leave out the struct name
- When using an array in a struct, the index goes at the end
  ```cpp
  student_rec.scores[x]
  ```
- When using an array of struct, the index goes after the struct name
  ```cpp
  parts_list[x].qty
  ```

Testing and Debugging

- Process struct members separately … the only aggregate operations will be
- Assignment =
- Parameter passing
  ```cpp
  void do_it (part_struct part);
  ```
- Function return
  ```cpp
  part_struct blanked_part { };
  ```

QUIZ

- Write a program that has a function to find zeros of the transferred array from the main function, write the result into main function.
  ```cpp
  int test [10] = { 5,0,7,8,0,19,0,8,9,0};
  ```

Object Oriented Programming
What is Object Oriented Programming?

- Identifying objects and assigning responsibilities to these objects.
- Objects communicate to other objects by sending messages.
- Messages are received by the methods of an object.

What is an object?

- Tangible Things: as a car, printer, ...
- Roles: as employee, boss, ...
- Incidents: as flight, overflow, ...
- Interactions: as contract, sale, ...
- Specifications: as colour, shape, ...

So, what are objects?

- An object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain.
- An "object" is anything to which a concept applies.

Why do we care about objects?

- Modularity - large software projects can be split up in smaller pieces.
- Reuseability - Programs can be assembled from pre-written software components.
- Extensibility - New software components can be written or developed from existing ones.

Example: The Person class

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

class Person{
  char name[20];
  int yearOfBirth;
public:
  void displayDetails() {
    cout << name << " born in 
    << yearOfBirth << endl;
  }
};
```

The two parts of an object

Object = Data + Methods

or to say the same differently:

An object has the responsibility to know and the responsibility to do.
**Basic Terminology**

- **Abstraction** is the representation of the essential features of an object. These are ‘encapsulated’ into an abstract data type.
- **Encapsulation** is the practice of including in an object everything it needs hidden from other objects. The internal state is usually not accessible by other objects.

**Basic Terminology: Inheritance**

- Inheritance means that one class inherits the characteristics of another class. This is also called a “is a” relationship.

```
A car is a vehicle
A dog is an animal
A teacher is a person
```

**Basic Terminology: Polymorphism**

- Polymorphism means “having many forms”. It allows different objects to respond to the same message in different ways, the response specific to the type of the object.

```
E.g. the message displayDetails() of the Person class should give different results when send to a Student object (e.g. the enrolment number).
```

**Basic Terminology: Aggregation**

- Aggregation describes a “has a” relationship. One object is a part of another object.

```
A car has wheels.
```

**Basic Terminology: Behaviour and Messages**

- The most important aspect of an object is its behaviour (the things it can do). A behaviour is initiated by sending a message to the object (usually by calling a method).

**The two steps of Object Oriented Programming**

- **Making Classes**: Creating, extending or reusing abstract data types.

- **Making Objects interact**: Creating objects from abstract data types and defining their relationships.
A class is a collection of a fixed number of components
- The components of a class are called members of the class
- The general syntax of defining a class is:
  ```
  class classIdentifier
  {
    classMemberList
  }
  ```
- A member of a class can either be a variable (that is, to store some data) or a function

> If a member of a class is a variable, it is declared just like any other variable
- In the definition of the class, you cannot initialize a variable when you declare it
- If a member of a class is a function, typically the function prototype is used to define that member
- A missing semicolon, therefore, will result in a syntax error

> If a member of a class is a function,
  - it can (directly) access any member of the class—data members and function members
  - when you write the definition of the member function, you can directly access any data member of the class without passing it as a parameter
  - the only obvious condition is that you must declare an identifier before you can use it
- Class is a reserved word and it only defines a data type, no memory is allocated
- The semicolon after the right brace is part of the syntax

The members of a class are classified into three categories:
- private
- public
- protected

Following are some facts about public and private members of a class:
- By default, all members of a class are private
- If a member of a class is private, you cannot access it outside the class

- A public member is accessible outside the class
- To make a member of a class public, you use the label public with a colon
- In C++, private, protected, and public are reserved words
**Electronics Engineering Dept**

**Example**

class Rectangle {
    int x, y;
    public:
        void set_values (int k, int l);
        int area () {return (x*y);}
};

Rectangle::set_values(int k, int l)
{ x = k;
  y = l;
}

**Classes, Objects, Members**

- Class ⇒ a structured type in a programming language
  - used to represent an abstract data type
- Class Member ⇒ Component of a class
  - functions
  - data

```cpp
class Complex
{ public:
    void assign (float a, float b);
    float real_part ();

    private:
    float a, b;
};
```

**What is an Object?**

- Object is a variable.
- Object is a function?
- Object is data and functions
- Object is an abstraction of something that has attributes (property) and operations (function calls).

**Built In Class Operations**

- Programmer defined classes can be like built in types
- Declare as many objects of a class as you like
- Pass as parameters to functions
  ```cpp
  void do_whatever (Complex z);
  ```
- Return as function values
  ```cpp
  Complex new_value ( ... );
  ```

**Built In Class Operations**

- Arrays of class objects
  ```cpp
  complex c1, c2;
  Complex c_list [20];
  ```
- Can be automatic or static
- Assign operator and dot . operator both work
  ```cpp
  complex c1, c2;
  ... 
  c1 = c2;
  cout << c1.real_part();
  ```

**Classes, Objects, Members**

- Class Object ↔ class instance
  - variable of a class type
  ```cpp
  complex c1, c2;
  cout << c1.real_part();
  ```
- Client ⇒ Software that declares and manipulates objects of a particular class:
**Class Scope**

- Name of a class member is **local** to the class.

```cpp
complex c1, c2;
int a, b, c;
... c1 = c2;
// cout << c1.real_part();
// but NOT
// cout << real_part
```

- Same identifier **declared outside** the class will be unrelated;

**Information Hiding**

- Class object has an "invisible wall" • called the abstraction barrier
- Protects private data and functions • client code cannot access private elements • client can access only public members
- Think of a class as a "black box" • it will act on the data • but you need not know how it works

**Information Hiding**

**Class implementation details are hidden from the client's view. This is called information hiding.**

Public functions of a class provide the **interface** between the client code and the class objects.

**Implementation File**

- Contains all the function definitions • includes function heading • includes body of function
- Similar to function definitions below `main ( )` • except is in different file
- Function headings in this file must match prototypes in `.h` file
Familiar Class Instances and Function Members

- The member selection operator (.) selects either data members or function members.
- Header files `<iostream>` and `<fstream>` declare the istream, ostream, and ifstream I/O classes.
- Both `cin` and `cout` are class objects.
- Both `get` and `ignore` are function members.

```
cin.get(someChar);  
cin.ignore(100, '<n>');  
```

Constructors

- Constructors
  - We can guarantee the initialization of the data members of a class by using constructors
  - There are two types of constructors:
    - with parameters
    - without parameters
  - The constructor without parameters is called the default constructor
  - A constructor that has no parameters, or has all default parameters, is called the default constructor

```
ifstream myInfile;  
myInfile.open( "A:\mydata.dat" );  
```

Invoking a Constructor

- A constructor is automatically executed when a class variable is declared
- A class might have more than one constructor, including the default constructor

```
Invoking a Constructor
```
**Invoking the Default Constructor**

- The syntax to invoke the default constructor is:
  ```
  className classVariableName;
  ```
- The statement:
  ```
  clockType yourClock;
  ```
  declares yourClock to be a variable of the type clockType
- In this case, the default constructor is executed and the data members of yourClock will be initialized to zero

**Constructor with Parameters**

- The syntax to invoke a constructor with parameters is:
  ```
  className classVariableName(argument1, argument2, ...);
  ```
- where argument1, argument2, etc. is either a variable or an expression

**Destructor**

- Like constructors, destructors are also functions
- The name of a destructor is the character '~' followed by the name of the class
- The name of the destructor for the class clockType is:
  ```
  ~clockType();
  ```
- A class can have only one destructor and it has no parameters
- The destructor is automatically executed when the class object goes out of scope

---

```cpp
#include <iostream>
using namespace std;

class Dunya
{
public:
    Dunya() { cout << "Merhaba'n"
    ~Dunya() { cout << "Güle güle'n"

    int main()
    {
        Dunya dunya;
        return 0;
    }
};
```
Guidelines for Constructors:

- A constructor cannot return a value
  - no return value type is declared
- Multiple constructors are allowed
  - compiler chooses appropriate one
  - according to number & data types of parameters
- Parameters passed to a constructor
  - place actual parameter list after name of class object being declared

Guidelines for Constructors:

- If class object declared WITHOUT a parameter list
  - results depend on what constructors are provided
  - even if NONE provided, there is a default constructor which allocates memory for private data elements
- When array of class objects declared, default constructor is invoked

Testing and Debugging Hints:

- Don’t forget semicolon ; at end of class type declaration
- Function declaration in .h file (specification) end with semicolons
- Function definitions in .cpp file (implementation) do NOT end with ;
- In implementation, don’t forget to prefix function name with
  - name of class
  - double colon ::

Testing and Debugging Hints:

- Only built-in operations (we know about) are
  - the dot . for member selection
  - the assignment =
- Functions which inspect (but do not modify) should be const member functions

Quiz-Normal:

- Write a program that takes a book values
  name 20 characters,
  Publish year-dd/mm/yyyy 8 character,
  Page number integer,
  Price float
- Take inputs from keyboard, then find the price average of 10 books with using a function.
- Print the average of the price on the screen into main function.
  - Hint: Use function, struct and an array
Write a program that takes a student values
name 20 characters,
ID 10 character,
midterm float,
final grade int
Take inputs from keyboard, then find the
midterm average of the class which has 5
students. Print the average of the class on the
screen.
• Hint : Use struct and an array

Write a program that insert the given value(dat)
somewhere-take this position into given
array(test).
• int test [10] = { 5,6,7,8,9,19,7,8,};
• int dat = 15;