CS 101: Computer Programming and Utilization

Jan-Apr 2017

Sunita Sarawagi
(cs101@cse.iitb.ac.in)

Lecture 13: Structures and Objects

About These Slides

- Based on Chapter 17 of the book
  An Introduction to Programming Through C++
  by Abhiram Ranade (Tata McGraw Hill, 2014)

- Original slides by Abhiram Ranade
  - First update by Varsha Apte
  - Second update by Uday Khedker
  - Third update by Sunita Sarawagi

On Design

- Whenever you design something complex, it is useful to have a plan
- Example: Plan for designing a building:
  - Understand the requirements
  - Understand the constraints: budget, land area
  - Plan how many floors to have
  - What should be on each floor
- A plan/methodology is also useful when designing large programs

Object Oriented Programming: A Methodology for Designing Programs

- Clearly understand what is required and write clear specifications (needed in all methodologies)
- Identify the entities involved in the problem

  Eg. in a library management program: books, patrons
- Identify the information associated with each entity
  - Fixed information: name of the book
  - Variable information (State): who has borrowed the book at present
- Organize the code so that the entities and their actions/inter relationships are explicitly represented in the code
  - Information associated with entities: structures
  - Relationships/actions of entities: functions

Outline

- Structure
  - Basic facility provided in C++ to conveniently gather together information associated with an entity.
  - Inherited from the C language
- Member functions
  - New feature introduced in C++

Additional OOP ideas will come in later

Structures: Basics Ideas

Structure = collection of variables

Members of a structure: variables in the collection

Structure = *super variable*, denotes the memory used for all members

Each structure has a name, the name refers to the super variable, i.e. entire collection

Each structure has a type: the type defines what variables there will be in the collection

Structure Types

- You can define a structure type for each type of entity that you want to represent on the computer Structure types are defined by the programmer

  Example: To represent books, you can define a Book structure type

- When you define a structure type, you must say what variables each structure of that type will contain

  Example: In a structure to represent books, you may wish to have variables to store the name of the book, its price, etc.

Defining a structure type

General form

```
struct structure-type{
    member1-type member1-name;
    member2-type member2-name;
    ...
};      // Don't forget the semicolon!
```

Example

```
struct Book{
    char title[50];
    double price;
};
```

A structure-type is a user-defined data type, just as int, char, double are primitive data types

Structure-type name and member names can be any identifiers

Creating Structures of A Type Defined Earlier

To create a structure variable of structure type Book, just write:

```
Book p, q;
```

This creates two structures: p, q of type Book.

Each created structure has all members defined in structure type definition.

Member x of structure y can be accessed by writing y.x

```
p.price = 399;  // stores 399 into p.price.
cout << p.title; // prints the name of the book p
```

Initializing structures

Book b = {“On Education”, 399};

Stores “On Education” in b.title (null terminated as usual) and 399 into b.price

A value must be given for initializing each member

You can make a structure unmodifiable by adding the keyword const:

const Book c = {“The Outsider”, 250};

One Structure Can Contain Another

```
struct Point{
    double x,y;
};
struct Disk{
    Point center;    // contains Point
    double radius;
};
Disk d;
d.radius = 10;
d.center.x = 15;
// sets the x member of center member of d
```

Assignment

One structure can be assigned to another

- All members of right hand side copied into corresponding members on the left
- Structure name stands for entire collection unlike array name which stands for address
- A structure can be thought of as a (super) variable

```
book b = {“On Education”, 399};
book c;
c = b;    // all members copied
cout << c.price << endl; // will print 399
```

Arrays of Structures

Disk d[10];

Book library[100];

Creates arrays d, library which have elements of type Disk and Book

cin >> d[0].center.x;

Reads a value into the x coordinate of center of 0^{th} disk in array d

cout << library[5].title[3];

Prints 3^{rd} character of the title of the 5^{th} book in array library

Structures and Functions

- Structures can be passed to functions by value (members are copied), or by reference
- Structures can also be returned. This will cause members to be copied back

Parameter Passing by Value

```
struct Point{double x, y;};
Point midpoint(Point a,  Point b){
  Point mp;
  mp.x = (a.x + b.x)/2;
  mp.y = (a.y + b.y)/2;
  return mp;
}

int main(){
  Point p={10,20},  q={50,60};
  Point r = midpoint(p,q);
  cout << r.x << endl;
  cout << midpoint(p,q).x  << endl;
}
```

Parameter Passing by Value

- The call midpoint(p,q) causes arguments p,q to be copied to formal parameters a,b
- When midpoint executes, the members of the local structure mp are set appropriately
- The return statement sends back the value of mp, i.e. a nameless temporary structure of type Point is created in the activation frame of main, into which mp is copied
- The temporary structure is the result of the call midpoint(p,q)
- The temporary structure is copied into structure r
- r.x is printed
- The temporary structure can be used with the "." operator, as in the second call. Both will print x coordinate, 30, of the midpoint
- However, you may not modify the temporary structure. Writing midpoint(p,q).x = 100; is not allowed. The value returned is considered const

Parameter Passing by Reference

```
struct Point{double x, y;};
Point midpoint( const Point &a,  const Point &b){
    Point mp;
    mp.x = (a.x + b.x)/2;
    mp.y = (a.y + b.y)/2;
    return mp;
}
int main(){
    Point p={10,20},   q={50,60};
    Point r = midpoint(p,q);
    cout << r.x << endl;
}
```

Parameter Passing by Reference

- In the execution of midpoint(p,q) the formal parameters a,b refer to variables p, q of main
- There is no copying of p, q. This saves execution time if the structures being passed are large
- The rest of the execution is as before
- const says that a, b will not be modified inside function
  - Helps humans to understand code
  - Enables const structures to be passed by reference as arguments

  midpoint(midpoint(..,..),..)

A Structure to Represent 3 Dimensional Vectors

- Suppose you are writing a program involving velocities and accelerations of particles which move in 3 dimensional space
- These quantities will have a component each for the x, y, z directions
- Natural to represent using a structure with members x, y, z

```
struct V3{ double x, y, z; };
```

Using Struct V3

Vectors can be added or multiplied by a scalar. We might also need the length of a vector.

```
V3 sum(const V3 &a, const V3 &b){
    V3 v;
    v.x = a.x + b.x; v.y = a.y + b.y; v.z = a.z + b.z;
    return v;
}
V3 scale(const V3 &a, double f){
    V3 v;
    v.x = a.x * f; v.y = a.y * f; v.z = a.z * f;
    return v;
}
double length(const V3 &v){
    return sqrt(v.x*v.x + v.y*v.y + v.z*v.z);
}
```

Motion Under Uniform Acceleration

If a particle has an initial velocity u and moves under uniform acceleration a, then in time t it has a displacement $s = ut + at^2/2$, where u, a, s are vectors

To find the distance covered, we must take the length of the vector s

```
int main(){
    V3 u, a, s;  // velocity, acceleration, displacement
    double t;   // time
    cin >> u.x >> u.y >> u.z >>
          a.x >> a.y >> a.z >> t;
    s = sum(scale(u,t), scale(a, t*t/2));
    cout << length(s) << endl;
}
```

Remarks

- It is not enough to just define a struct to hold vectors, usually we will also define functions which work on structs
- In C++, you can make the functions a part of the struct definition itself. Such functions are called member functions. We study these next
- By collecting together relevant functions into the definition of the struct, the code becomes better organized

Structures with Member Functions

```
struct V3{
 double x, y, z;
 double length(){
     return sqrt(x*x + y*y + z*z);
 }
}

int main(){
 V3 v={1,2,2};
 cout << v.length() << endl;
}
```

Structures with Member Functions

- length is a member function
- Member function f of a structure X must be invoked on a structure s of type X by writing s.f(arguments)
- s is called receiver of the call

  Example: v.length(). In this v is the receiver
- The function executes by creating an activation frame as usual
  - The references to members in the body of the definition of the function refer to the corresponding members of the receiver
- Thus when v.length() executes, x, y, z refer to v.x, v.y, v.z
- Thus the v.length() will return sqrt($1^2+2^2+2^2$) = 3

The Complete Definition of V3

```
struct V3{
    double x, y, z;
    double length(){
        return sqrt(x*x + y*y + z*z);
    }
    V3 sum(V3 b){
        V3 v;
        v.x = x+b.x; v.y=y+b.y; v.z=z+b.z;
        return v;
    }
    V3 scale(double f){
        V3 v;
        v.x = x*f; v.y = y*f; v.z = z*f;
        return v;
    }
}
```

Main Program

```
int main(){
 V3 u, a, s;
 double t;
 cin >> u.x >> u.y >> u.z >> a.x >> a.y >> a.z >> t;
 V3 ut = u.scale(t);
 V3 at2by2 = a.scale(t*t/2);
 s = ut.sum(at2by2);
 cout << s.length() << endl;
// green statements equivalent to red:
 cout << u.scale(t).sum(a.scale(t*t/2)).length() << endl;
}
```

One More Example: Taxi Dispatch

- Problem statement: Clients arrive and have to be assigned to (earliest waiting) taxies
- An important part of the solution was a blackboard on which we wrote down the ids of the waiting taxies
- How would we implement this using OOP?

  Create a struct to represent each entity: customer, taxi, blackboard?

One More Example: Taxi Dispatch

- Customers are assigned taxis immediately if available

  Information about customers never needs to be stored
- Each taxi is associated with just one piece of information: id

  We can just use an int to store the id
- The blackboard however is associated with a lot of information: array of ids of waiting taxis, front/last indices into the array

  So we should create a struct to represent the blackboard

Representing the Blackboard

```
const int N=100;
struct Queue{
 int elements[N],
     nwaiting,front;
 bool insert(int v){
  …
  }
 bool remove(int &v){
  …
 }
};
```

- N = max no. of waiting taxis
- We call the struct a Queue rather than blackboard to reflect its function
- nwaiting = no. of taxis currently waiting
- front = index

  elements[front] through elements[front+nwaiting%N] holds the ids of waiting taxis
- Two operations on Queue: inserting elements and removing elements.

  These become member functions

Member Function Insert

```
struct Queue{
  …
  bool insert(int v){
     if(nWaiting >= N) return false;
     elements[(front + nWaiting)%N] = v; nWaiting++;
     return true;
     }
};
```

- A value can be inserted only if the queue has space
- The value must be inserted into the next empty index in the queue
- The number of waiting elements in the queue is updated
- Return value indicates whether operation was successful

Main Program

```
int main(){
 Queue q;
 q.front = q.nWaiting = 0;
 while(true){
  char c; cin >> c;
  if(c == ‘d’){
    int driver; cin >> driver;
    if(!q.insert(driver)) cout <<“Q is full\n”;
  }
  else if(c == ‘c’){
    int driver;
    if(!q.remove(driver)) cout <<“No taxi available.\n”;
    else cout <<“Assigning <<driver<< endl;
  }
}
```

Remarks

- The member functions only contain the logic of how to manage the queue
- The main program only contains the logic of dealing with taxis and customers
- The new program has become simpler compared to the earlier version, where the above two were mixed up together

Concluding Remarks

- Define a structure for every kind of entity you wish to represent in your program
- Structures are (super) variables which contain other (member) variables
- Members can be accessed using the "." operator
- Structure name denotes the super variable consisting of the entire collection of contained variables
- Structures can be copied using assignments. Also copied when passed by value, or returned from a function
- Member functions should be written to represent actions of the entities represented by the structure

Concluding Remarks

Arrays are also collections of variables but:

- All elements of an array must be of the same type
- Name of the array denotes the address of the 0^{th} element, whereas name of the structure denotes the entire collection
- Array elements can be accessed by an expression whose value can be computed at run time whereas structure members can be accessed by names fixed names that must be known at compile time

Objects As Software Components

- A software component can be built around a `struct`
- Just as a hardware component is useful for building big hardware systems, so is a software component for building large software systems
- A software component must be convenient to use, and also safe, i.e. help in preventing programming errors

“Packaged Software components”

- Hardware devices that you buy from the market are packaged, and made safe to use
  - Fridge, television : no danger of getting an electric shock.
  - A “control panel” is provided on the device. A user does not have to change capacitor values to change the channel on a television

“Packaged Software components”

- Analogous idea for software:
  - Make functionality associated with a `struct` available to the user only through member functions (control panel)
  - Do not allow the user to directly access the data members inside a `struct`. (Just as a user cannot touch the circuitry) The user does not need to know what goes on inside
- If you build a better fridge but keep the control panel the same as the previous model, the user does not need to relearn how to use the new fridge
  - If you build a better version of the `struct`, but keep the member functions the same, the programs that use the `struct` need not change

The Modern Version of A Struct

- Can behave like a packaged component
- Designer of the struct provides member functions
- Designer of the struct decides what happens during execution of standard operations
- Once structs are designed in this manner, using them becomes convenient and less error-prone
- Structs endowed with above features are more commonly called objects

The Modern Version of A Struct

- Designer of the struct decides what happens during execution of standard operations such as:
  - Creation of the object
  - Assignment
  - Passing the object to a function
  - Returning the object from a function
  - Destroying the object when it is not needed

  How to do this: discussed next