An Introduction to Programming though C++

Abhiram G. Ranade

Lecture 9.1

Ch. 17: Structures

Where are we in the course...

You have learned enough to write essentially any program.

- Basic control statements
- Functions
- Arrays

Next lectures: Language features that will make writing programs more convenient, safer, modular.

- These terms will become clearer soon.

A difficulty

- A large program will have lots of variables.
- Just managing all the variables is tiring.
  - "Lots of papers strewn over the table"
  - We can bring some neatness by putting related papers into files.
  - Can we do something like that with variables?
- The solution: "structures"
  - The main topic of this lecture

Structures – high level idea

Most entities we deal with in programming have lots of attributes.

- If our program is about simulating movement of stars
  - Each star has a position, velocity, mass, ...
- If our program is about managing books in a library
  - Each book has author, library number, who has borrowed it, ...
- **Key idea**: collect together all information about an entity into a group/supervariable = structure

Outline for this lecture

- “Structure”
  - Basic facility provided in C++ to conveniently gather together information associated with an entity.
  - Inherited from the C language
- Operations on structures
- Examples.

Structures: overview

- 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.
  - "Programmer defined type"
- 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, …

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 and member names can be any identifiers.

Creating structures of a type defined earlier

- To create a structure 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
```

What we discussed

- Creating structure types
- Creating variables/instances of an already created structure type.

Next: Operations on structures

Initializing structures during creation

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

Structures and functions

- Structures can be passed to functions by value
  - members are copied
- Structures can also be passed by reference.
  - Same structure is used in called function
- Structures can also be returned.
  - All data members are copied back to a temporary structure in the calling program

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,r).x << endl;
}
```

- Call ***midpoint(p,q)*** : ***p,q*** copied to parameters ***a,b***.
- ***midpoint*** creates local structure ***mp***.
- The value of ***mp*** is returned:

A nameless temporary structure of type ***Point*** is created in the activation frame of ***main***.

mp is copied into the temporary structure

- The temporary structure is copied into structure ***r***.
- ***r.x*** is printed.
- We can use the "." operator on temporary structures, as in the second call.

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;
}
```

- In execution of ***midpoint(p,q)*** parameters ***a,b*** refer to variables ***p,q*** of ***main***.
- There is no copying of ***p,q***.
- Saves execution time if the structures being passed are large.
- The rest of the execution is as before.
- Normally, reference parameters are expected to be variables.
- ***const*** says that ***a,b*** will not be modified inside function.
- Enables ***const*** structures to be passed as arguments.

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

Arrays of structures

```
Disk d[10];      // d[0], ..., d[9]
Book lib[100];   // lib[0],..., lib[99]
```

- Creates arrays with appropriate structures as elements.

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

- Reads a value into the ***x*** coordinate of ***center*** of 0th disk in array ***d***.

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

- Prints 3rd character of the title of the 5th book in array ***library***.

What we discussed

- Various operations on structures:
  - Initialization
  - Nesting
  - Passing and returning from functions
  - Creating arrays

Next: detailed example

Example

Given a n disks in the plane, determine if they intersect.

- We have solved this earlier
- Structs enable us to write this in a nicer manner.
- Basic struct that we need:

```
struct disk{
  double centerx, centery, radius;
};
```

Background

If you have not seen this problem, a quick background that is adequate to proceed further. ….

Disk □ Circle, characterized by center (x,y coordinates) and radius (r)

Problem statement: Given a n disks in the plane, determine if they (any) intersect

Output: true is any pair intersect, false if no pair intersects

Idea: For a pair of circles C1 and C2, distance between centers,

$\text{sqrt}((x1-x2)^2 + (y1-y2)^2) < (r1+r2)$ then they intersect

The main program

```
int main(){
  const int n=5;
  disk disks[n];
  readData(disks,n);
  cout << checkAllPairs(disks, n) << endl;
}
```

- Without structs the function calls would require **centerx, centery, radius** to be passed separately.
- In this program we are "thinking at a high level" – not worrying about what is contained inside **disk**.

Reading in the data

```
void readData(disk disks[], int n){
  for(int i=0; i<n; i++)
    cin >> disks[i].centerx
        >> disks[i].centery
        >> disks[i].radius;
}
```

Checking intersections

```
bool checkAllPairs(disk disks[], int n){
  for(int i=0; i<n-1; i++)
    for(int j=i+1; j<n; j++)
      if(intersect(disks[i], disks[j])) return true;
  return false;
}

bool intersect(disk d1, disk d2){
  return pow(d1.centerx-d2.centerx,2) +
         pow(d1.centery-d2.centery,2)
         < pow(d1.radius+d2.radius, 2);
}
```

Demo

- diskIntersect.cpp

What we discussed

- A detailed example
- By using structures, our functions have fewer arguments.
- Better readability, less clutter
- Our overall program = many small functions.
- Small functions are easier to understand at a glance.
- Main program is at high level
- Other functions deal with disk details.

Next: Pointers with structures, conclusion of lecture.

Pointers to structures

Disk d1={{2,3},4}, *dptr;

- ***dptr** is defined to have type **Disk**, so **dptr** is a pointer to a variable of type **Disk**.

- Normal pointer operations are allowed on structure pointers.

```
dptr = &d1;
(*dptr).radius = 5;
//changes the radius of d1.
```

- Operator **->**
  - **(*x).y** is same as **x->y**

```
dptr->radius = 5; // same effect as above.
```

Pointers as structure members

```
struct Disk2{
        double radius;
        Point *centerptr;
}

Point p={10,20};
Disk2 d;
d.centerptr = &p;
cout << d.centerptr->x; // will print 10.

Disk2 f;
f.centerptr = &p; // center is shared
            // possible motivation for using pointers.
f.centerptr->x = 15;
cout << d.centerptr->x;
// will print 15
```

NULL pointers

- NULL is a keyword in C++
- It can be assigned to any pointer, to indicate that the pointer does not point to anything meaningful.
- You may check if a pointer does not point to anything by writing ***ptr == NULL***

Pointers to structure of same type

```
struct Employee{
   char title[20];
   Employee* boss;
 };
Employee e1={"President",NULL},
        e2={"MD", &e1}, e3={"ED", &e1};
cout << e2.boss->title <<endl;
// will print "President"
cout << e3.boss->boss->title <<endl;
// Error: cannot take title of NULL
```

Concluding Remarks

- Structures enable us to group together variables of different types
  - Arrays also group together variables, but of same type.
- A structure name behaves like a (super-)variable
  - If you copy it or pass it to a function, the members are copied.
- By grouping we can reduce the clutter in our argument lists, and overall organize our data better.