An Introduction to Programming though C++

Abhiram G. Ranade

Lecture 9.2

Ch. 17: Structures, part 2

Some reflections on structures

- We proposed that all attributes of each entity in our program should be grouped into a structure.
- Benefits
  - Reduce clutter overall
  - Make it easy to pass the entity to a function.
  - Easier to write small functions
  - Main program is at high level, other function deal with low level details

An observation

- Typically the attributes of an entity are accessed in only a few different ways.

Example 1: Vector from physics:

Example 2: Queue in taxi dispatch

- Both vectors and queues will have many attributes/members.
- We do not read/update the individual members in isolation.
- A natural operation on vectors or queues requires us to simultaneously access/update several attributes/members in a very specific manner.

Vector from physics

- A vector may be used to represent velocities, positions, displacements, electric fields, ...
- A vector can be indicated by its components in x, y, z directions.
- So we may represent a vector as

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

- There could be other representations, e.g. polar.
- Key observation:
  - You will rarely read just the x component, or update the y component.
  - Most likely you will do something with all 3, e.g. add two vectors, scale a vector.

The insight

- We should put all the attributes of an entity into a structure, and,
- We should provide functions with which to access the attributes.

"Member functions" : Main topic of this lecture

A structure to represent 3 dimensional vectors

Recap:

- Suppose you are writing a program involving velocities and accelerations of particles which move in 3 dimensional space.
- Natural to represent using a structure with members x, y, z.
  ```
  struct V3{ double x, y, z; };
  ```
- Other representations are also possible.

Using struct V3

- Several operations can be useful with vectors

```
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

- Initial velocity = u, uniform acceleration = a,
- displacement s = ut + $at^2/2$, where u, a, s are vectors, t = time
- To find the distance covered, we must take the length of the vector s.

```
int main(){
  V3 u, a, s; // velocity, acceleration, displacement.
  cin >> u.x >> u.y >> u.z >> a.x >> a.y >> a.z;
  for(double t=0; t<10; t++){
    s = sum(scale(u,t), scale(a, t*t/2));
    cout <<t<<": "<< length(s) << endl;
  }
}
```

Demo

- V3use.cpp

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;
}
```

- ***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.
- 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
- Member functions can modify receiver members.
  receiver is passed by reference

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;
  }
}
```

```
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;
}
```

What we discussed

- Syntax of member functions
- A member function is physically placed inside the struct definition.
- A call to a member function looks like:

receiver.function-name(…)

- The member function executes like an ordinary function, with receiver being an additional argument.
  - The syntax highlights the special relationship of the function to the receiver.
  - Receiver is implicitly passed by reference, i.e. the call can modify the members in the receiver.
- The benefits of this syntax will become clear soon.