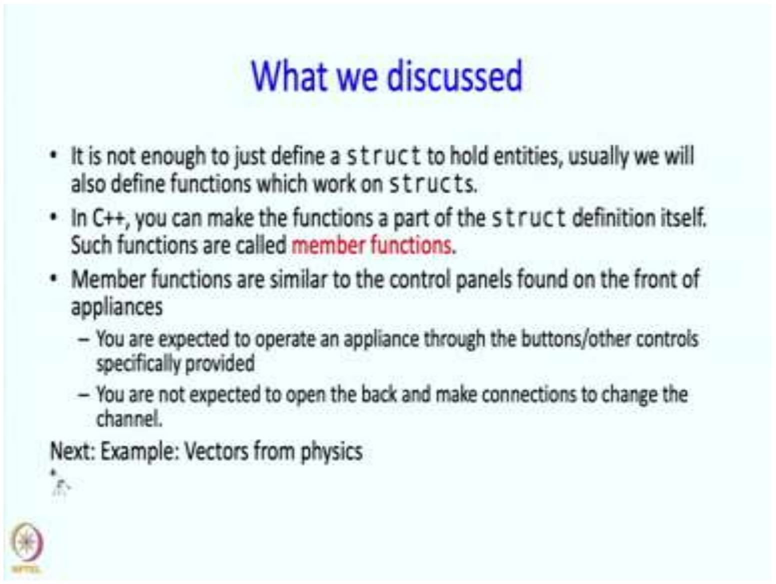
An Introduction to Programming through C++

Professor Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 20 Part 2
Structures Part 2
Vectors from Physics

Welcome back.

(Refer Slide Time: 0:34)



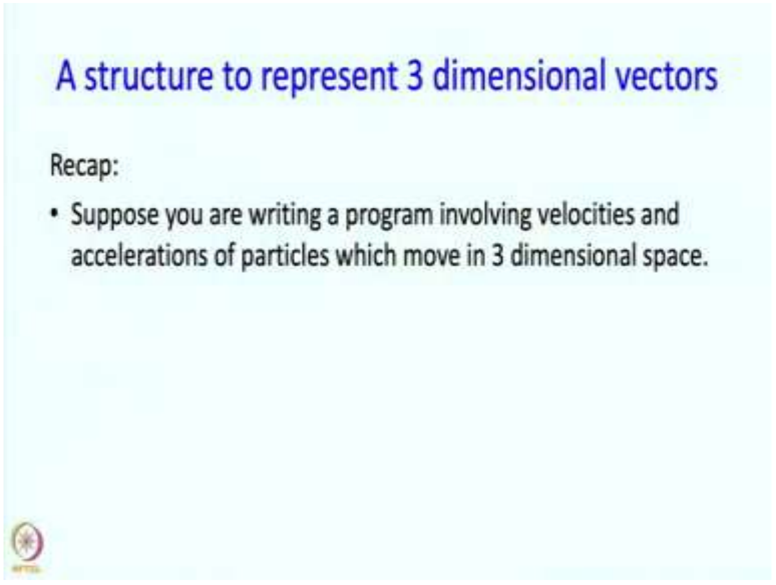
What we discussed

- It is not enough to just define a `struct` to hold entities, usually we will also define functions which work on `struct`s.
- In C++, you can make the functions a part of the `struct` definition itself. Such functions are called member functions.
- Member functions are similar to the control panels found on the front of appliances
  - You are expected to operate an appliance through the buttons/other controls specifically provided
  - You are not expected to open the back and make connections to change the channel.

Next: Example: Vectors from physics

In the previous segment, we motivated the need for member functions. In this segment, we are going to talk about vectors from physics and then later on, see how they how they can be implemented using member functions and the operations on the vectors using member functions. So, we have already talked about this.

(Refer Slide Time: 0:44)



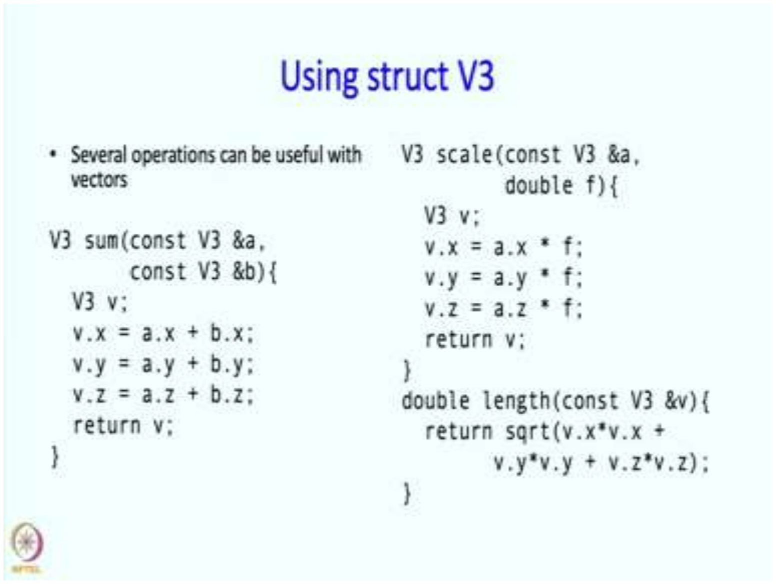
A structure to represent 3 dimensional vectors

Recap:

- Suppose you are writing a program involving velocities and accelerations of particles which move in 3 dimensional space.

So, say you are writing a program involving velocities and accelerations of particles, which move in 3 dimensional space. Then you would find it natural to represent the vectors using a structure with members x, y, z. So, V3 is the structure type so for 3 dimensional vectors, and its members are x, y and z, which are all doubles. And, of course, other representations are also possible.

(Refer Slide Time: 1:21)



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;
}
```

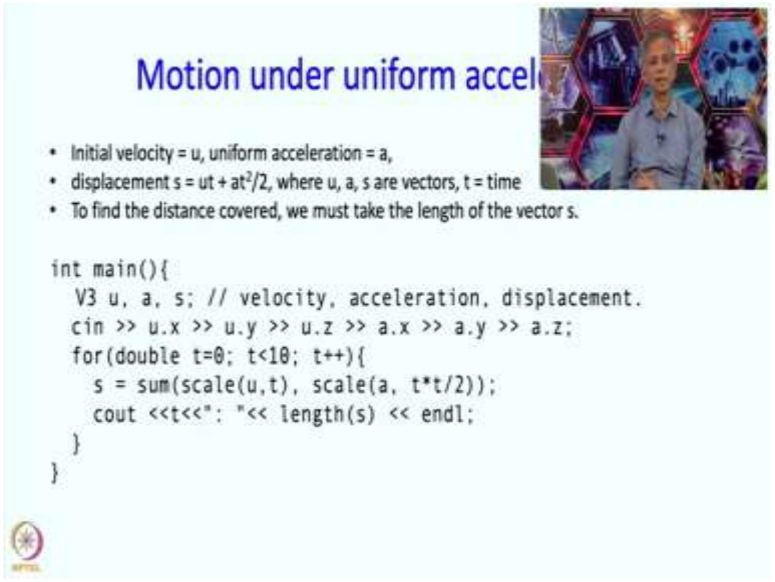
```
V3 scale(const V3 &a,
         double f){
  V3 v;
  v.x = a.x * f;
  v.y = a.y * f;
  v.z = a.z * f;
  return v;
}
double length(const V3 &v){
  return sqrt(v.x*v.x +
         v.y*v.y + v.z*v.z);
}
```

So, how do you use this struct V3? Well, there could be several operations that you might want perform. So, for example, you might take 2 vectors and add them up together. So, a function, an ordinary function which does this, would look something like this. So, you have this function name. And this function is going to return an object of type V3 or structure of type V3 and it is going to take arguments which are also of type V3 and we are taking arguments by reference.

So, first it is going to create internally, a V3 object called v and this is going to be our result. So, what is the result going to look like, well its x component is going to be the sum of the x components of these two objects similarly, the y, similarly the z. Then you could have a scaling operation which takes vector and a scale factor, and simply multiplies the scale factor that each of the members and so the resulting vector is going to be returned.

As you remember, this operation is going to return this vector that means the values will get copied in some suitable place in that calling program. Here is another possible operation that you may want with vectors which is a length operation. So, the length operation simply takes a single vector and it returns the square root of the sum of the individual members. So these might be some operations that you might want to perform on your structure type on objects of your structure type V3.

(Refer Slide Time: 3:29)



Motion under uniform accel

- 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;
  }
}
```

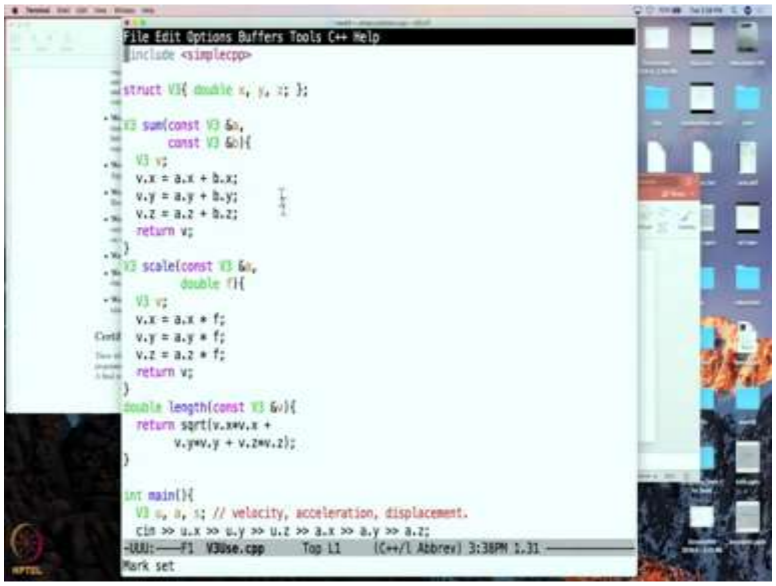
Well here is an example of use of vectors in which these operations can come in quite handy. And this is so called motion under uniform acceleration. So, we have a particle which has initial velocity u and uniform acceleration a. Then its displacement at time t, ‘s’ is given as u times t plus at square by 2 and this is applicable even if u, a and s are vectors and t is a scaler, if it is a time. To find the total distance covered or the total displacement rather we must take the length of the vector s.

So this might be whatever program looks like. So, we have V3, we have 3 vectors of type V3 u, a and s - velocity, acceleration and displacement. And first we read values into u and a. Next, we could we could calculate this for single value of t but just for fun we will calculate it for multiple values of t. t ranging from say 0 through 9. So, what will this how we will do it? Well so s is going to be sum of these two quantities and what are these two quantities?

Well they are u scale by t and then a scale by t square upon by 2. So, what is going on here? a is a vector, scale of a and t square by 2 is also a vector. That vector and scale of u times t are being added up and the result is this sum, is this displacement s and this displacement is also a vector. So, this looks like an ordinary assignment, but really it is it is more complicated than that, when you when you look at in the ordinary assignment you are tempted to think of it as sort of basic data types.

Whereas over here, 3 components are being assigned in parallel, or 3 components are being passed to this function and this function is returning 3 components, so is this. So, then we print out as a function of t, the displacement that has happened and that is it. So, that is the program. As you can see we could have return this program by accessing each component of u, s and t, but rather than that we chose to use these functions sum, scale and length. And the reason for that is that they reflect better what operation we are actually performing. And furthermore, once you are somebody writes the operations sum, scale and length and tests them and make sure that they are right, then you can be more confident that your code is actually correct.

(Refer Slide Time: 6:48)



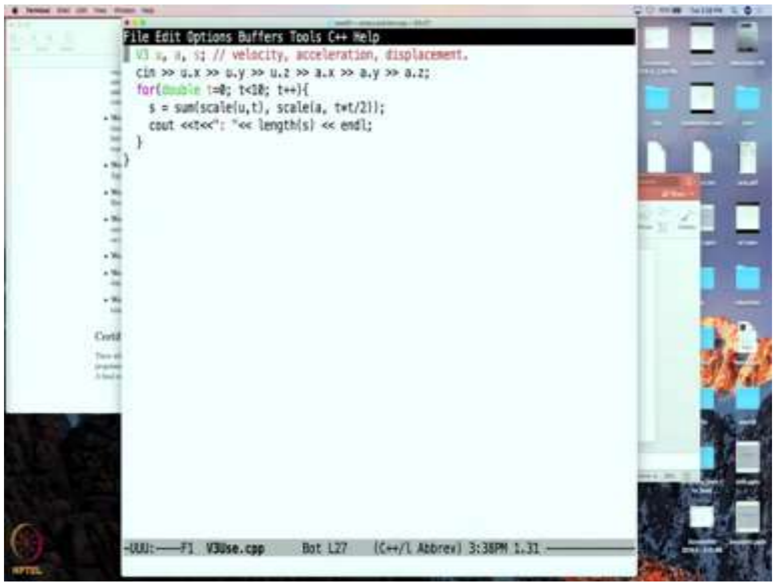
File Edit Options Buffers Tools C++ Help

```
#include <simplecpp>

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

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);
}

int main(){
  V3 u, a, s; // velocity, acceleration, displacement.
  cin >> u.x >> u.y >> u.z >> a.x >> a.y >> a.z;
-UUU:----F1  V3Use.cpp      Top L1     (C++/l Abbrev) 3:38PM 1.31 ------------
Mark set
```

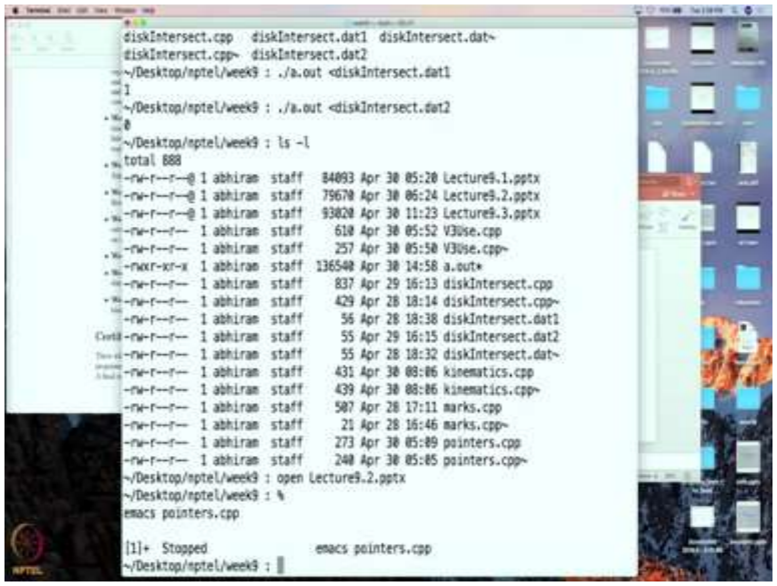


File Edit Options Buffers Tools C++ Help

```
  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;
  }
}
-UUU:----F1  V3Use.cpp      Bot L27    (C++/l Abbrev) 3:38PM 1.31 ------------
```

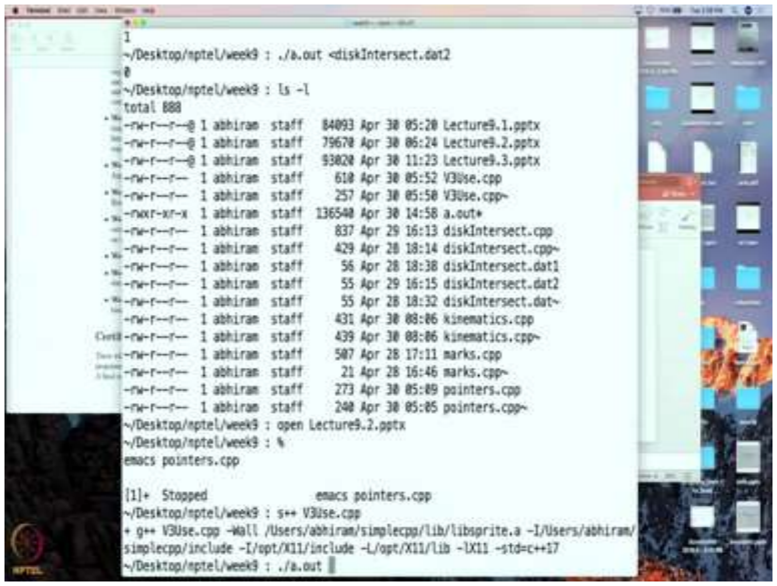
So, let us do a quick demo. So, first we have the structure type, then the sum, then the scale, length and then we have the main program. It is really what we what we saw on the slides. So, let us just run it, compile it and run it.

(Refer Slide Time: 07:03)



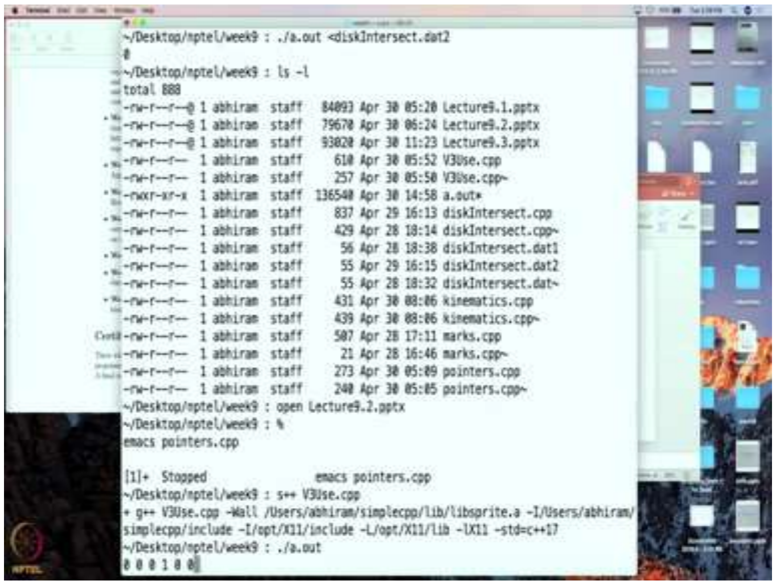
```
diskIntersect.cpp   diskIntersect.dat1  diskIntersect.dat~
diskIntersect.cpp~  diskIntersect.dat2
~/Desktop/nptel/week9 : ./a.out <diskIntersect.dat1
1
~/Desktop/nptel/week9 : ./a.out <diskIntersect.dat2
0
~/Desktop/nptel/week9 : ls -l
total 888
-rw-r--r--@ 1 abhiram  staff   84093 Apr 30 05:20 Lecture9.1.pptx
-rw-r--r--@ 1 abhiram  staff   79670 Apr 30 06:24 Lecture9.2.pptx
-rw-r--r--@ 1 abhiram  staff   93020 Apr 30 11:23 Lecture9.3.pptx
-rw-r--r--  1 abhiram  staff     610 Apr 30 05:52 V3Use.cpp
-rw-r--r--  1 abhiram  staff     257 Apr 30 05:50 V3Use.cpp~
-rwxr-xr-x  1 abhiram  staff  136540 Apr 30 14:58 a.out*
-rw-r--r--  1 abhiram  staff     837 Apr 29 16:13 diskIntersect.cpp
-rw-r--r--  1 abhiram  staff     429 Apr 28 18:14 diskIntersect.cpp~
-rw-r--r--  1 abhiram  staff      56 Apr 28 18:38 diskIntersect.dat1
-rw-r--r--  1 abhiram  staff      55 Apr 29 16:15 diskIntersect.dat2
-rw-r--r--  1 abhiram  staff      55 Apr 28 18:32 diskIntersect.dat~
-rw-r--r--  1 abhiram  staff     431 Apr 30 08:06 kinematics.cpp
-rw-r--r--  1 abhiram  staff     439 Apr 30 08:06 kinematics.cpp~
-rw-r--r--  1 abhiram  staff     507 Apr 28 17:11 marks.cpp
-rw-r--r--  1 abhiram  staff      21 Apr 28 16:46 marks.cpp~
-rw-r--r--  1 abhiram  staff     273 Apr 30 05:09 pointers.cpp
-rw-r--r--  1 abhiram  staff     240 Apr 30 05:05 pointers.cpp~
~/Desktop/nptel/week9 : open Lecture9.2.pptx
~/Desktop/nptel/week9 : %
emacs pointers.cpp

[1]+  Stopped                 emacs pointers.cpp
~/Desktop/nptel/week9 :
```



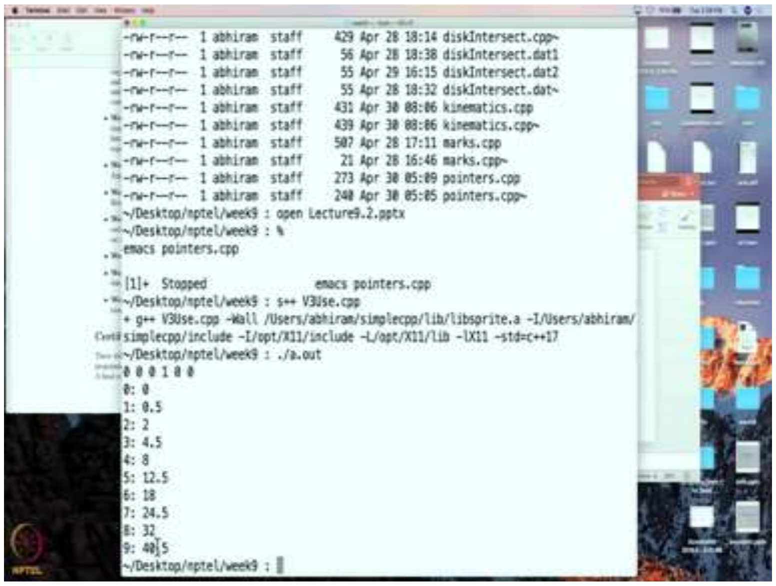
```
1
~/Desktop/nptel/week9 : ./a.out <diskIntersect.dat2
0
~/Desktop/nptel/week9 : ls -l
total 888
-rw-r--r--@ 1 abhiram  staff   84093 Apr 30 05:20 Lecture9.1.pptx
-rw-r--r--@ 1 abhiram  staff   79670 Apr 30 06:24 Lecture9.2.pptx
-rw-r--r--@ 1 abhiram  staff   93020 Apr 30 11:23 Lecture9.3.pptx
-rw-r--r--  1 abhiram  staff     610 Apr 30 05:52 V3Use.cpp
-rw-r--r--  1 abhiram  staff     257 Apr 30 05:50 V3Use.cpp~
-rwxr-xr-x  1 abhiram  staff  136540 Apr 30 14:58 a.out*
-rw-r--r--  1 abhiram  staff     837 Apr 29 16:13 diskIntersect.cpp
-rw-r--r--  1 abhiram  staff     429 Apr 28 18:14 diskIntersect.cpp~
-rw-r--r--  1 abhiram  staff      56 Apr 28 18:38 diskIntersect.dat1
-rw-r--r--  1 abhiram  staff      55 Apr 29 16:15 diskIntersect.dat2
-rw-r--r--  1 abhiram  staff      55 Apr 28 18:32 diskIntersect.dat~
-rw-r--r--  1 abhiram  staff     431 Apr 30 08:06 kinematics.cpp
-rw-r--r--  1 abhiram  staff     439 Apr 30 08:06 kinematics.cpp~
-rw-r--r--  1 abhiram  staff     507 Apr 28 17:11 marks.cpp
-rw-r--r--  1 abhiram  staff      21 Apr 28 16:46 marks.cpp~
-rw-r--r--  1 abhiram  staff     273 Apr 30 05:09 pointers.cpp
-rw-r--r--  1 abhiram  staff     240 Apr 30 05:05 pointers.cpp~
~/Desktop/nptel/week9 : open Lecture9.2.pptx
~/Desktop/nptel/week9 : %
emacs pointers.cpp

[1]+  Stopped                 emacs pointers.cpp
~/Desktop/nptel/week9 : s++ V3Use.cpp
+ g++ V3Use.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhiram/
simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week9 : ./a.out
```



```
~/Desktop/nptel/week9 : ./a.out <diskIntersect.dat2
0
~/Desktop/nptel/week9 : ls -l
total 888
-rw-r--r--@ 1 abhiram  staff   84093 Apr 30 05:20 Lecture9.1.pptx
-rw-r--r--@ 1 abhiram  staff   79670 Apr 30 06:24 Lecture9.2.pptx
-rw-r--r--@ 1 abhiram  staff   93020 Apr 30 11:23 Lecture9.3.pptx
-rw-r--r--  1 abhiram  staff     610 Apr 30 05:52 V3Use.cpp
-rw-r--r--  1 abhiram  staff     257 Apr 30 05:50 V3Use.cpp~
-rwxr-xr-x  1 abhiram  staff  136540 Apr 30 14:58 a.out*
-rw-r--r--  1 abhiram  staff     837 Apr 29 16:13 diskIntersect.cpp
-rw-r--r--  1 abhiram  staff     429 Apr 28 18:14 diskIntersect.cpp~
-rw-r--r--  1 abhiram  staff      56 Apr 28 18:38 diskIntersect.dat1
-rw-r--r--  1 abhiram  staff      55 Apr 29 16:15 diskIntersect.dat2
-rw-r--r--  1 abhiram  staff      55 Apr 28 18:32 diskIntersect.dat~
-rw-r--r--  1 abhiram  staff     431 Apr 30 08:06 kinematics.cpp
-rw-r--r--  1 abhiram  staff     439 Apr 30 08:06 kinematics.cpp~
-rw-r--r--  1 abhiram  staff     507 Apr 28 17:11 marks.cpp
-rw-r--r--  1 abhiram  staff      21 Apr 28 16:46 marks.cpp~
-rw-r--r--  1 abhiram  staff     273 Apr 30 05:09 pointers.cpp
-rw-r--r--  1 abhiram  staff     240 Apr 30 05:05 pointers.cpp~
~/Desktop/nptel/week9 : open Lecture9.2.pptx
~/Desktop/nptel/week9 : %
emacs pointers.cpp

[1]+  Stopped                 emacs pointers.cpp
~/Desktop/nptel/week9 : s++ V3Use.cpp
+ g++ V3Use.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhiram/
simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week9 : ./a.out
0 0 0 1 0 0
```

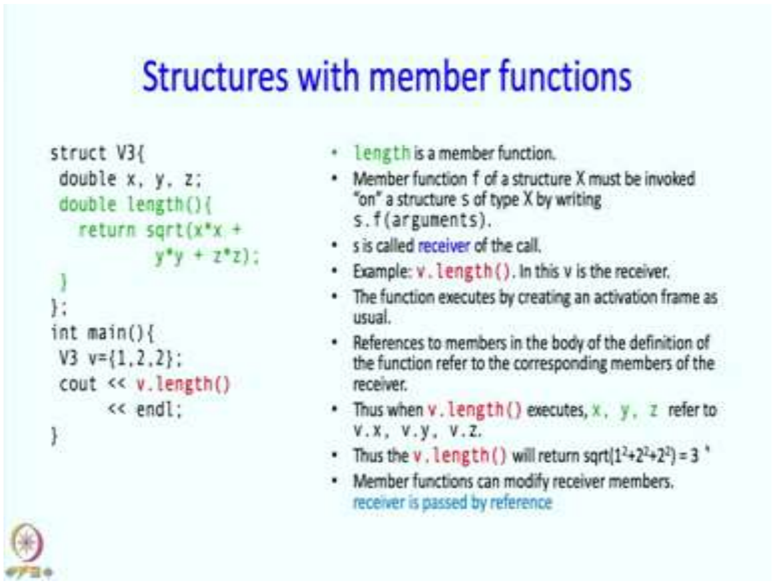


```
-rw-r--r--  1 abhiram  staff     429 Apr 28 18:14 diskIntersect.cpp~
-rw-r--r--  1 abhiram  staff      56 Apr 28 18:38 diskIntersect.dat1
-rw-r--r--  1 abhiram  staff      55 Apr 29 16:15 diskIntersect.dat2
-rw-r--r--  1 abhiram  staff      55 Apr 28 18:32 diskIntersect.dat~
-rw-r--r--  1 abhiram  staff     431 Apr 30 08:06 kinematics.cpp
-rw-r--r--  1 abhiram  staff     439 Apr 30 08:06 kinematics.cpp~
-rw-r--r--  1 abhiram  staff     507 Apr 28 17:11 marks.cpp
-rw-r--r--  1 abhiram  staff      21 Apr 28 16:46 marks.cpp~
-rw-r--r--  1 abhiram  staff     273 Apr 30 05:09 pointers.cpp
-rw-r--r--  1 abhiram  staff     240 Apr 30 05:05 pointers.cpp~
~/Desktop/nptel/week9 : open Lecture9.2.pptx
~/Desktop/nptel/week9 : %
emacs pointers.cpp

[1]+  Stopped                 emacs pointers.cpp
~/Desktop/nptel/week9 : s++ V3Use.cpp
+ g++ V3Use.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhiram/
simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week9 : ./a.out
0 0 0 1 0 0
0: 0
1: 0.5
2: 2
3: 4.5
4: 8
5: 12.5
6: 18
7: 24.5
8: 32
9: 40.5
~/Desktop/nptel/week9 :
```

So, now we are supposed to type in the velocities and the accelerations. So just so that we can see what is going on. Let me type the initial velocity to be 0 and let me type the acceleration to be say 1. So this is a plot of what happens to the particle as a function of time. So by making these things simple, you should be able to check that these are in fact the correct answers. So, what is going on over here is, I guess, you can think of it as a particle which is initially at rest is moving may be falling down to gravity or something like that. Now we could ask, can we do the same thing with member functions? And this is how it looks like.

(Refer Slide Time: 8:23)



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

So, here we have a member function, which calculates the length and that member function is being called over here. This red text is the call of that member function. So, length is a member function. You can see that it is inside the structure definition. And a member function f of a structure type x should be invoked on a structure s of this type x by writing s.f(arguments), which is exactly what has happened over here.

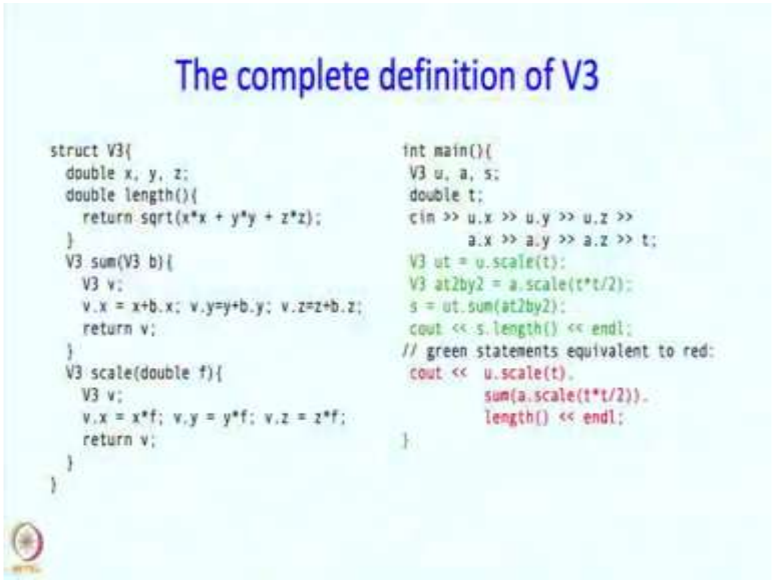
So, V is a structure of type of V3 and we are invoking this function on v by writing v. length. There are no arguments specified over here. So, no arguments are given here as well. Now, this s or this v is called the receiver of the call. So, there is another analogy here as well that this part is sometimes even called a message being sent. A message is being sent to this object and that object responds, so that is that is another metaphor that is used.

But anyway we are going to call this thing before the dot as the receiver of that call. So, in v.length, v is the receiver, and it is a function call. So, all function calls it executes by creating an activation frame. And the references to the members of the body of the definition of the function refer to the corresponding members of the receivers. So see this you have x.x, y.y, z.z. What are these x, y, z? Well they are these, but as far as these calls are concerned they are members of the receiver. So this x.x, when this call is being made will actually refer to v.x, this is refer to v.y, this is refer to v.z. So, then v.length executes x, y, z refer to v.x, v.y, v.z. So, as a result of this v.length is going to return one square plus two square plus three square plus two square again. It is square root so 1 plus 4 plus 9 square root of that is 3.

Now, the receiver itself is also an argument otherwise how would you know what does x mean. So, this receiver is an argument and it is it is kind of a hidden argument. Well it is not really hidden. It is actually upfront, it is it is really at the front. So, it is a kind of a special argument. But it is not there in the argument list, in the parameter list. Alright, but it is an argument nevertheless. And it is an argument which is passed by reference.

So in this code, if you change x then the x member of v is going to get changed. So, it is this v is being passed by reference that is why this happens. If v is v is passed by value, then if you change x that would not be that would not happen. But indeed, the receiver is passed by reference. So, we can write the other functions as member functions and here is the complete definition.

(Refer Slide Time: 12:15)



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;
}
```

So, this is our struct V3, the members then this is a first member function which we just saw. This is the sum operation written as a member function, so sum of V3. Well V3 in what, we need two arguments to make this sum. Now, that sum, the first argument is the receiver itself. So, coming over here, so we are doing ut.sum of at/2, so ut is one V3 which is being added to at by 2 which is another V3, so in this case this v is going to be at/2 and ut is going to be the receiver for this.

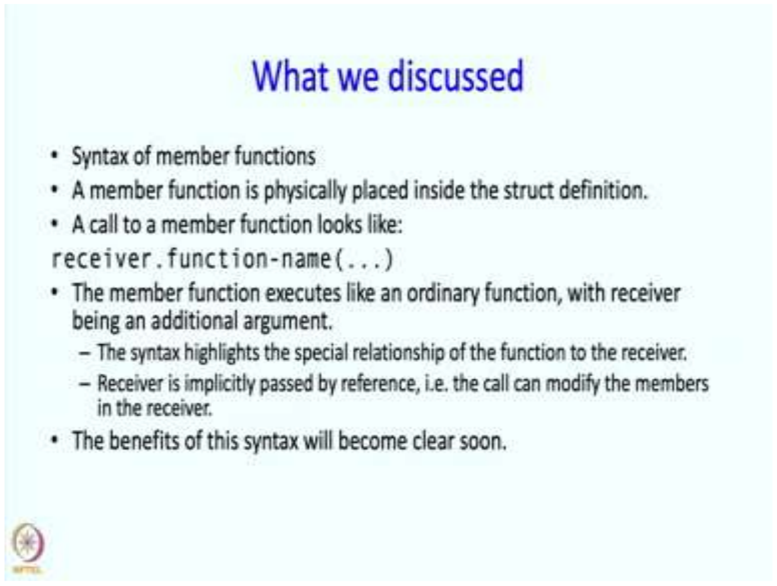
Alright, so now what how does this functions work. We are supposed to add up the 2 vectors and pass the resulting vector. So we are going to create the result that over here. V3 of v is going to be our result vector. Now, the x member of this should be the sum of this ut x and at/2 x. So this is ut x and this is at by 2 x; similarly y, similarly z. And then we return this vector so that is how this sum works.

Scale is similar, for the scale so we have u.scale(t). So, u is the receiver and t is the argument. So t is, the value of t is placed in f, so again what we are we calculating, we are supposed to calculate the result which is V3 vector, which is the scale version of the receiver. So, to implement that this is the x member of the receiver, we scale it up by f and put it in the x member of our result V3. And so we construct all V3, the 3 vx, vy, vz and then we return v.

How does it look like in this function? Well we wanted to calculate u times t so that it is scaling so we partial result we put over here. Then we wanted to calculate at square so that is a times scaling by t square by 2. So a.scale of t square by 2 and then we have to add up these two things. So, that is what is happening over here. Finally, we have to take the sum of the length of that sum and so that is happening over here.

And I should say that we do not necessarily need to have so many variables if we do not want. What is happening over here is we are going to scale and then the result is going to be summed with the scale version of this. And that we can take length, we can apply the length member function to this entire result. And we are going to, we will then get the length. So, here we can think of this as one long complicated expression, instead of that we have broken up that expression into 3 separate expressions which are of course related.

(Refer Slide Time: 15:40)



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.

So, what we have discussed? So we discussed the syntax of member functions. We have said that member function is physically placed inside the struct definition. And we said that the call to a member function looks like receiver.function_name followed by the arguments. The member function executes like an ordinary function with the receiver being an additional argument and the syntax makes the receiver look very different and indeed it is very different. It has a special relationship.

There is nominal special relationship in that the receiver is being definitely passed by reference but in addition to that somehow if I write say v.something, then I know this is something which is happening and it has some special meaning in the context of v. Usually. Or it tells me look this the code for this function will be found wherever v is itself defined.

We also said that the receiver is implicitly passed by reference and so the call can actually be modified by the members in the receiver. And this syntax is reasonable but you might ask well we really need it? We have used the usual function called syntax, so that is that is not quite clear yet, but it will become clear quite soon. So we will take a quick break over here.