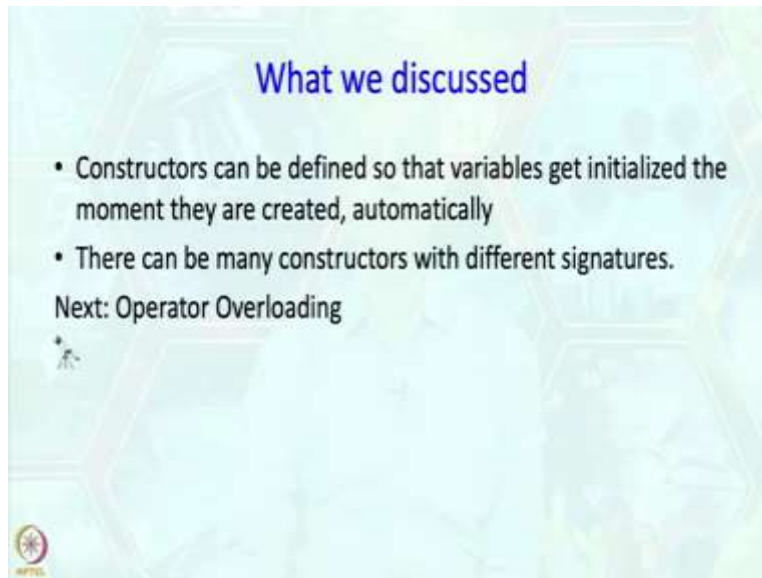


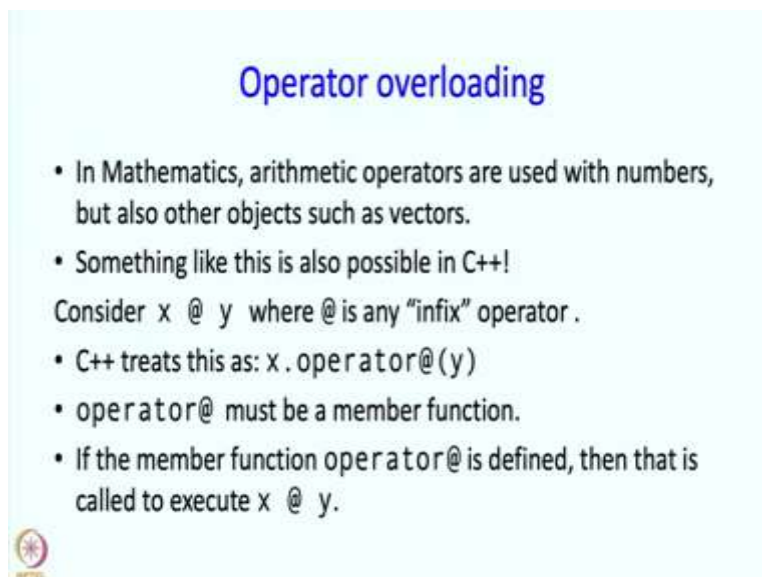
An Introduction to Programming through C++
Professor Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Lecture 21 Part-3
Classes
Operator Overloading

(Refer Slide Time: 00:18)



Welcome back. In the previous segment we talked about constructors. In this segment we are going to talk about operator overloading.

(Refer Slide Time: 00:30)



In mathematics, arithmetic operators are used with numbers, but they are also used with other objects such as vectors and the same operators in some sense mean different things in different places, but they are somehow related and therefore we somehow prefer to use the same symbol. Now, something like this can also be done in C++ and it is desirable to the extent that it helps your thinking.

So if, the program that you write is consistent with the way you think about, about the objects in your program then that is a good thing and C++ allows you to do that. How it is C++ do it? Well consider an operator @, so @ could be plus, @ could be multiply anything.

So, the operation associated is $x @ y$, so @ appears in the middle and so @ is an infix operator. So, how does C++ look at such an operation? Well C++ treats this as $x.operator@$ with argument y . So, operator @ must be a member function in the structure type x , structure type of x . So, if it is a member function then that member function is executed.

So, if the member function $operator@$ is defined, then that is called to execute $x @ y$, it is as simple as that. So, you can define for example, operator + and that code will be execute if you write $x + y$, so we will see an example of this right away.

(Refer Slide Time: 02:38)

Example: arithmetic on V3 objects

```
int main(){
    V3 p(1,2,3), q(4,5,6);
    V3 r, s;
    r = p+q;
    // r = p.operator+(q);
    s = r * 10;
    // s = r.operator*(10);
}

struct V3{
    double x, y, z;
    V3(double a, double b, double c){
        x=a; y=b; z=c;
    }
    V3(){
    }
    V3 operator+(V3 b){ // V3 + V3
        return V3(x+b.x, y+b.y, z+b.z);
        // constructor call
    }
    V3 operator*(double f){ // V3 * number
        return V3(x*f, y*f, z*f);
    }
}
```

So, see we want to do arithmetic on v3 objects. So, this is our main, so we have v3 objects p and q and they have been called with these constructors. So, we will see those constructors as well in a minute. But let us say we have initialized those objects p and q somehow or the other. Then we

have couple of uninitialized `v3` objects and now we can write `r=p+q`. And the natural interpretation of this might be that look you add the vectors component wise and we can make that natural interpretation hold in our implementation, and that is exactly what we are going to see. So, notice that as we said earlier this just means `r=p.operator+(q)`. So, we are going to define the operator plus member function. Then we can write something like `s=r*10`.

So, so far `v3`, for `v3` star does not have any particular meaning `r` is of type `v3` and there is nothing so far we have said as to what it means to multiply a vector by a scalar. But we can simply define `r.operator*`, this expression would be effectively implemented automatically by C++ it will be interpreted as this operation and C++ will just perform this operation.

So, how does this whole thing work? So, struct `v3`, if you remember has these 3 parts `x`, `y`, `z` there are the 3 members `x`, `y`, `z` and let me just get this constructor out of the way. So, this is a constructor taking 3 arguments and setting `x`, `y`, `z` to those 3 arguments. So, this is kind of the most natural constructor you might want, I want to set my vector, I want to initialize my vector to some arbitrary 3 quantities.

Now, comes another constructor because we want to write this as well, so for this, this constructor will not work. So, this constructor is going to do nothing, so the members `x`, `y`, `z` will not be initialized in any particular manner.

Next we are going to write the member function `operator+`. So, the definition of this member function is like defining, is like the definition of any other member functions. So, this `v3` is the type of the value being returned and this is the argument. And this is effectively going to be adding 2 vectors. So, how are those 2 vectors going to come? Well 1 is going to be the receiver so we write, we wrote `p+q` that was translated implicitly by C++ to this.

So, `p` is going to be the receiver and that is going to be added to `q`. So, in this, the receiver is always there and the second the right hand side operand of the plus is going to be this `b`. So for this expression which got translated to this `p` is going to be the receiver, `q` is going to be the right hand, right hand side operand or the `b` value over here.

So, what happens? So in this, what am I supposed to do? I am supposed to return as a `v3` object having `x` co-ordinate equal to the sum of the `x` co-ordinates of the receiver and the right hand

object. So, $x+b.x$ is exactly that, $y+b.y$ is the desired y coordinate and $z+b.z$ is the desired z coordinate. So, we are calling a constructor over here and that constructor is constructing a v3 object with these members.

So, these are exactly the numbers that we wanted for component wise addition or member wise addition and that is exactly what these member function have accomplished. What about this? So, we should write a member function `r.operator*` and this would be the factor argument, so it is going to look something like this. So v3 will take a, it appears to take a single argument is operator star but that is not true there is always a receiver and the receiver is, since this is inside a struct v3 the receiver is of type v3.

So, this is an implementation of v3 times number, so f is that number. And what is going to happen over here? You should be able to write this now, you are going to return an object whose numbers are $x*f$, $y*f$ and $z*f$ and you want to return a v3 object so we just called the v3 constructor with these arguments, so that is it. So, this allows us to do arithmetic on v3 objects. Now, using this v3 arithmetic we can do something which looks quite nice.

(Refer Slide Time: 08:18)

```
Using V3 arithmetic

int main(){
    V3 u(1,2,3), a(4,5,6),
        s(0,0,0);
    double t=10;
    s = u*t + a*t*t*0.5;
    cout << s.x << ' ' << s.y << ' '
        << s.z << endl;
}
```

So, suppose we have v3 objects u, a and s remember our ut plus half at square example. And let us say t is a double value, now our formula was ut plus half at squared. So, we have essentially written that formula but you can see that the formula is going to do what we want, why is that? So, u times t we have defined, u times t is going to be `u.operator*(t)`, so we have defined how

this is going to happen. This is going to multiply every member, every component of u by t , so it is going to do exactly, exactly the right thing. What about this a times t ? that is again scaling, so this if it is calculated first will give me a vector which has been acceleration which has been scaled by t or acceleration which is multiplied by t . But again when I do this it will be again a v^3 though the product multiplied by t , again the v^3 multiplied by 0.5 .

So, this is going to produce exactly the half at square term that we want. So, now this is going to be a v^3 object, this is going to be a v^3 object and there is a plus over here, but we define the plus operator as well and therefore, this is going to produce the final sum that we wanted and if we put s equals this then we can get $s.x$, $s.y$, $s.z$ which is going to be half at square for t equals 10 and these values of u and a .

(Refer Slide Time: 10:01)



```
File Edit Options Buffers Tools C++ Help
#include <simplecpp>

struct V3{
    double x, y, z;
    V3(double a, double b, double c){
        x=a; y=b; z=c;
    }
    V3 operator+(V3 b){ // adding two V3s
        return V3(x+b.x, y+b.y, z+b.z);
    }
    V3 operator*(double f){ // multiplying a V3 by a number
        return V3(x*f, y*f, z*f);
    }
};

int main(){
    V3 v(1,2,3), s(4,5,6),
    t(0,0,0);
    double i=10;
    s = s + t*(i+0.5);
    cout << s.x << " " << s.y << " "
    << s.z << endl;
}
```

```
~$ g++ -std=c++17 -I. kinematics.cpp -o kinematics
~$ ./kinematics
0 0 1 0 0
0: 0
1: 0.5
Cond: 2
3: 4.5
4: 0
5: 12.5
6: 18
7: 24.5
8: 32
9: 40.5
~/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 -Mll /Users/abhiran/simplecpp/lib/libsprite.a -I/Users/abhiran/
simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lx11 -std=c++17
~/Desktop/npTEL/week9 : ./a.out
0 0 1 0 0
0: 0
1: 0.5
Cond: 2
3: 4.5
4: 0
5: 12.5
6: 18
7: 24.5
8: 32
9: 40.5
~/Desktop/npTEL/week9 : open Lecture9.3.pptx
~/Desktop/npTEL/week9 : %
emacs pointers.cpp
[1] Stopped emacs pointers.cpp
~/Desktop/npTEL/week9 : %
```

```
~/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 -I/Users/abhiran/simplecpp/lib/libsprite.a -I/Users/abhiran/
simplecpp/include -I/opt/X11/include -I/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week9 : ./a.out
0 0 1 0 0
0: 0
1: 0.5
2: 2
3: 4.5
4: 8
5: 12.5
6: 18
Coord 7: 24.5
Time 8: 32
9: 40.5
~/Desktop/nptel/week9 : open Lecture9.3.pptx
~/Desktop/nptel/week9 : %
emacs pointers.cpp

[1]+ Stopped                  emacs pointers.cpp
~/Desktop/nptel/week9 : s++ kinematics.cpp
+ g++ kinematics.cpp -Wall -I/Users/abhiran/simplecpp/lib/libsprite.a -I/Users/abh
irans/simplecpp/include -I/opt/X11/include -I/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week9 : ./a.out
210 270 330
~/Desktop/nptel/week9 : █
```

Let us do a demo of this, so this is our file, kinematics.cpp. So, inside this we have put in whatever we wanted. So, this constructor and I guess I have, I have ignored putting the simple constructor over here but I guess it is not needed over here but we could have put that in as well. So, this is the definition of operator+ so the plus operator, this the definition of times operator and you can see that I have exactly the code that we want.

So, let us look at this and let us calculate what answers are we expecting. So, t is 10, so ut we should expect in the x co-ordinate 10 over here and then half at square, so half of a in the x co-ordinate is going to be 2 and t square is 100 so this is going to contribute a 200 plus 10, so 210. So, similarly you can do the other things but lets just check whether this will indeed produce s.x to be 200, so 210. So, let us execute this and run it. Yes, 210 it is the x, x coordinate.

(Refer Slide Time: 11:30)

Exercise

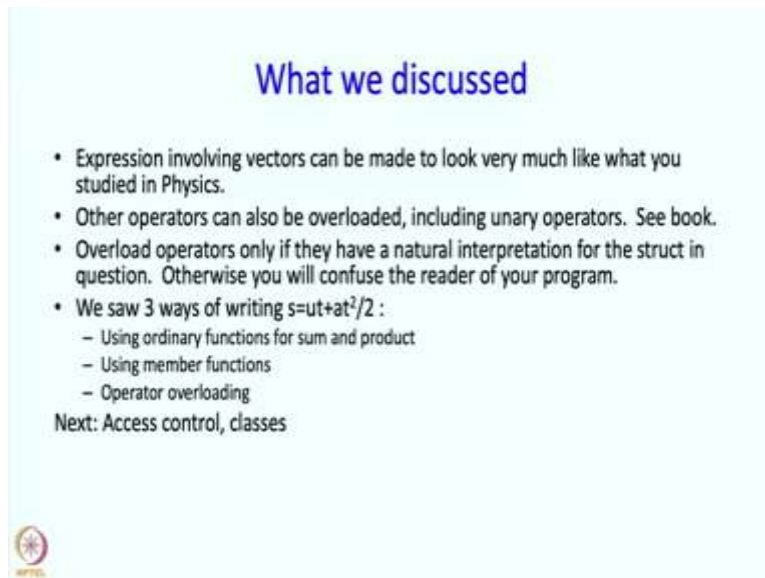
- Overload multiplication to define the dot product of two vectors.
- This can sit next to our previous overloading of multiplication – that had a different signature because it was a product between a vector and a scalar.



So, here are some exercises, so overload the multiplication to define the dot product of 2 vectors. As you might remember this is also something that is needed in physics a lot. Now, we already have overloaded the multiplication but that was a multiplication between a $v3$ object and a number.

This is going to be $v3$ object and another $v3$ object and that is perfectly fine that can sit next to our old definitions because the signatures are different and so when you write it in your code C++ will know exactly which of those functions to use because it will look at the signatures and it will look at how many arguments, what kinds of arguments you have specified.


(Refer Slide Time: 12:19)



What we discussed

- Expression involving vectors can be made to look very much like what you studied in Physics.
- Other operators can also be overloaded, including unary operators. See book.
- Overload operators only if they have a natural interpretation for the struct in question. Otherwise you will confuse the reader of your program.
- We saw 3 ways of writing $s=ut+at^2/2$:
 - Using ordinary functions for sum and product
 - Using member functions
 - Operator overloading

Next: Access control, classes



So, what have we discussed? So, we have said that expressions involving vectors can be made to look very much like what you studied in physics. The exact formulae you can almost see in your code. Other operators can also be overloaded, including unary operators and this is discussed at length in the book. Although for this course you will not really worry so much about all such detail overloading. So, if you understand plus and times that is quite enough for this course and may be the few other operators which we will specifically point out a little bit later.

Now, overloading operators seems very cute but it is also a little bit strange, I mean so in principle you could take the plus operator and make it do multiplication. So, that could be amusing but in practice it is a terrible idea because you are fooling people and that is the last thing that you want to do when you are cooperating, when you are trying to co-operatively solve the problem.

So, you have to be careful with this kind of overloading. You should use overloading when it really, when the operators are really consistent with the intuition that you have or intuition that in general people have, otherwise you will just end up confusing the reader of your program and believe me you will also confuse yourself. But if the intuition is consistent then you should overload because one of the ideas in modern programming is that, your programs should look like the language should be like the language you used in dealing with that domain.

So, if while doing the physics you think of multiplying vectors by scalars then if you can make that happen in programming it will also be a good thing, it will increase the readability of your code and generally make things more appealing. So, I just want to point out that we saw 3 ways of writing s equal to ut plus half a square, using ordinary functions for sum and product, using member functions and operator overloading. And I guess you probably agree with me that operator overloaded looked very very natural. The next topic is going to be access control and classes but before that let us take a short break.