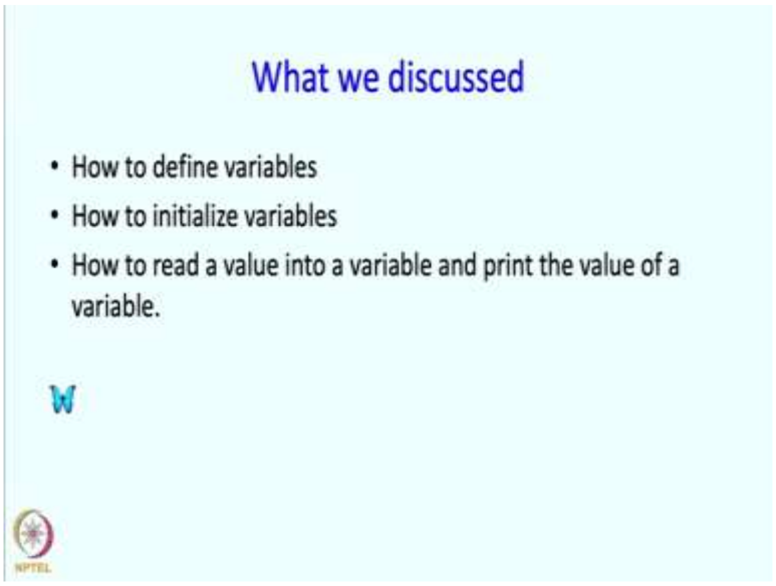
An Introduction to Programming through C++
Professor Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Lecture No. 3 Part - 2
Basic Elements of Program
Assignment statements, arithmetic expressions

(Refer Slide Time: 00:21)

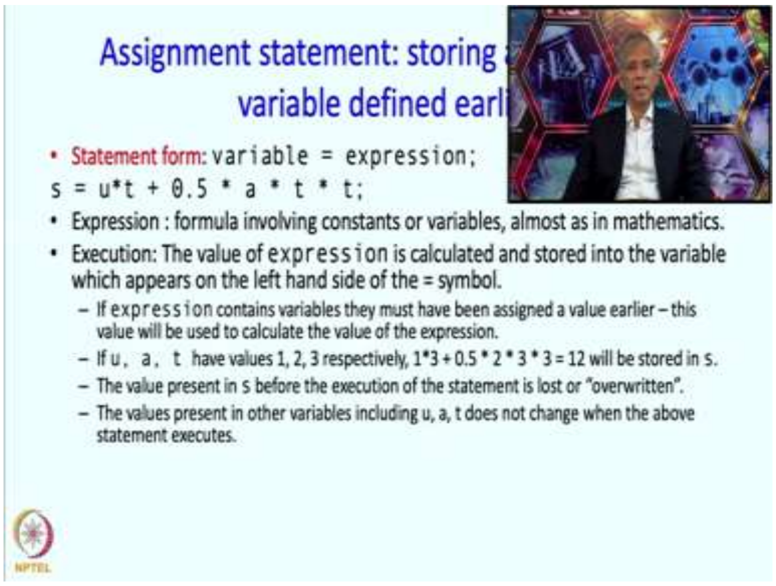


What we discussed

- How to define variables
- How to initialize variables
- How to read a value into a variable and print the value of a variable.

In the previous segment we discussed how to define variables, initialize them, read into them from the keyboard and print.

(Refer Slide Time: 00:44)



Assignment statement: storing variable defined earl

- Statement form: `variable = expression;`

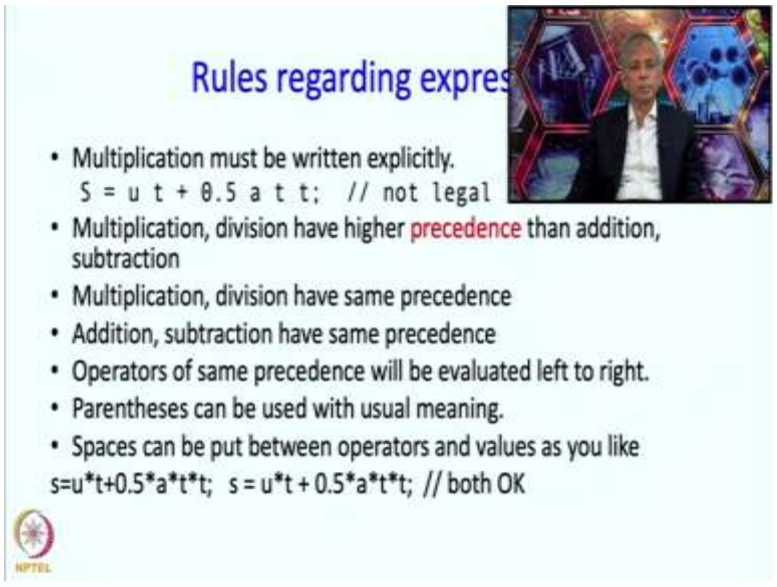
```
s = u*t + 0.5 * a * t * t;
```

- Expression : formula involving constants or variables, almost as in mathematics.
- Execution: The value of `expression` is calculated and stored into the variable which appears on the left hand side of the = symbol.
  - If `expression` contains variables they must have been assigned a value earlier – this value will be used to calculate the value of the expression.
  - If `u, a, t` have values 1, 2, 3 respectively, 1*3 + 0.5 * 2 * 3 * 3 = 12 will be stored in `s`.
  - The value present in `s` before the execution of the statement is lost or "overwritten".
  - The values present in other variables including u, a, t does not change when the above statement executes.

Now, we are going to discuss the assignment statement which is useful for storing a value into a variable defined earlier. So you might have initialized it, but you can change the value and that is what the assignment statement does. The statement has the form ‘variable=expression’. So here is an example, I might have variables s, u, t, a defined earlier, I should have such variables and if I have these variables I can write a statement ‘s=u*t + 0.5*a*t*t’ and what this does is what you might expect. So the text on the right hand side of the assignment is what is called an expression and it is a formula involving constants or variables, essentially like you write formulae in mathematics and in the execution, the value of expression is calculated and that value is stored into the variable which appears on the left hand side of the equal to symbol. Now, if expression contains variables then they must have been assigned values earlier. If they are not, then it is wrong to write such a statement. Ok. And if for example, ‘a’ has been assigned a value then the value used in place of ‘a’ in this expression will be whatever that value is.

So for example, suppose u, a and t have values 1, 2, 3 respectively assigned before the statement is executed. Then when the statement executes then this expression will be calculated. So the value 1 of u will be used, the value 3 of t will be used and the value 2 of a will be used. So we will have the evaluation ‘1*3 + 0.5*2*3*3’ and the result of this 12 will get stored into the variable s. Now, when you store a value into a variable then whatever value was there before you did the store goes away. Because, after all the variable contains some capacitors and now you are saying, look I want a different value to be stored in these capacitors. So basically, whatever value was present is lost or its also said that the new value overwrites the old value. When you evaluate an expression which contains references to variables, then the values of those variables are taken from those variables but they continue to remain in those variables as well. So for example, in this case while we will need to use the value stored in u, a and t, just because we are using the values, does not mean that they are going to be destroyed. So even after the statement is executed the variables u, a, t will continue to have continue to have those values that we were given earlier. In fact, this statement only changes the value of the variable s. Whatever other variables we might have defined those variables will remain as they are.

(Refer Slide Time: 4:09)



Rules regarding expres

- Multiplication must be written explicitly.
  `S = u t + 0.5 a t t;  // not legal`
- Multiplication, division have higher precedence than addition, subtraction
- Multiplication, division have same precedence
- Addition, subtraction have same precedence
- Operators of same precedence will be evaluated left to right.
- Parentheses can be used with usual meaning.
- Spaces can be put between operators and values as you like

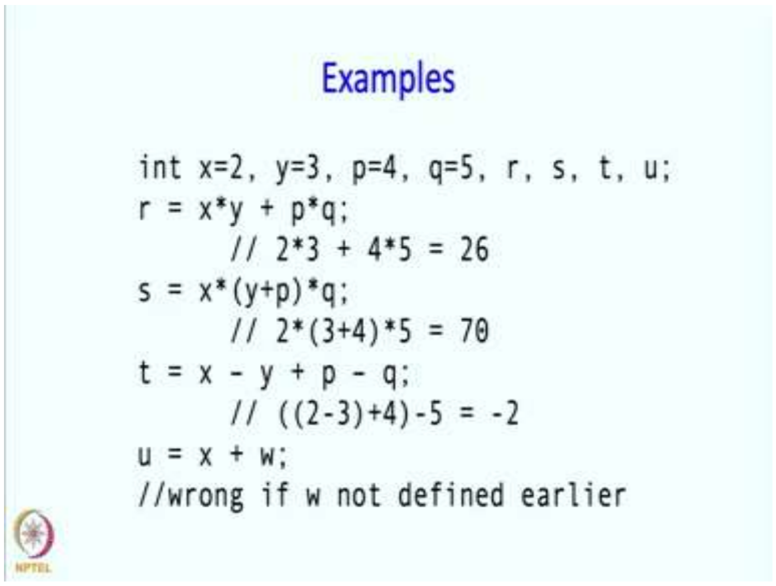
`s=u*t+0.5*a*t*t;  s = u*t + 0.5*a*t*t; // both OK`

Now, when we write expressions there are several rules. Some of which are slightly unusual, but you will see that they are quite natural. The first rule is that multiplication must be written out explicitly. So you cannot write 's = ut + 0.5att' while in mathematics if we want to write x times y we just write xy this is not allowed on a computer. Simply because if we write u and t together then there is a question, does it mean u multiplied by t? Or does it mean a single name ut of some variable? So because of such considerations the designer designers of C++ and many languages decided that if you want multiplication to happen you have to say multiplication so you have to explicitly indicate the multiplication operator. So between u and t and between the other things that you want multiplied you must put a '*' operator.

Now, multiplication and division have higher precedence than addition. So this is like mathematics. So for example, in this case u*t will be computed then 0.5*a*t*t will get computed provided you had put in the multiplication operators and only after that will the addition be computed. So, between multiplication and division the precedence is the same and between addition and subtraction also the precedence is the same. So operators of same precedence if they appear and you have to make a choice between them then evaluation happens left to right. So the operator that is on the left side will get evaluated first.

So we will see examples of all this and inside expressions you can use parentheses with the usual meaning. Now, spaces can be put between operators and values if you like, but of course if you have an identifier which consists of several letters, you cannot split it with a space. So the first example there are no spaces but in the second example there are spaces between on either side of the equality symbol as well as the plus. So whatever you think is easier to read is fine. So that is the choice left to you.

(Refer Slide Time: 06:48)



Examples

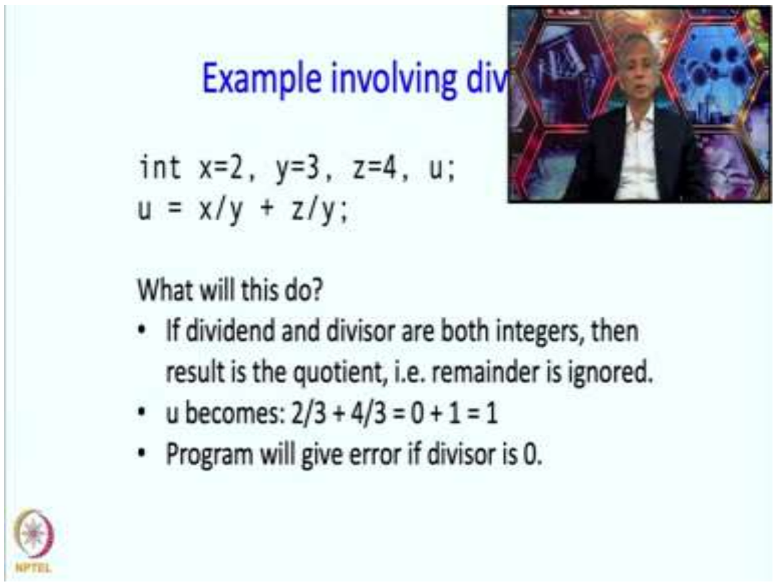
```
int x=2, y=3, p=4, q=5, r, s, t, u;
r = x*y + p*q;
      // 2*3 + 4*5 = 26
s = x*(y+p)*q;
      // 2*(3+4)*5 = 70
t = x - y + p - q;
      // ((2-3)+4)-5 = -2
u = x + w;
//wrong if w not defined earlier
```

Some examples, so suppose we create these variables and initialize some of them. So now, what happens if I write r=x*y+p*q? So in this expression, we want to pick x and y first and perform their product. Then, the next multiplication has higher precedence so that product will be computed and finally the two products will be added. So since x and y have value 2 and 3 and p, q have value 4, 5 we are going to multiply 2*3+4*5 and get 26. So r will get the value 26 if you execute these two statements.

If you execute s=x*(y+p)*q then notice that here the parentheses are evaluated first. So y+p will be calculated first. So the result will be 2*(3+4)*p or in other words s will get the value 70. If you write t=x-y+p-q, the evaluation is left to right. So 2-3 is evaluated first which will give you -1, plus p or 4 will get added so you will get 3 and then you will do minus 5 so you will get minus 2. So minus 2 will go into t.

If you write u=x+w in this context this statement is a little questionable. Because it is not clear whether w has been defined before or not. So if w has not been defined before, then this statement is wrong. Otherwise whatever the value of w is, and it had better have been given a value that value will be added to x and the result will be stored in u.

(Refer Slide Time: 08:51)



Example involving div

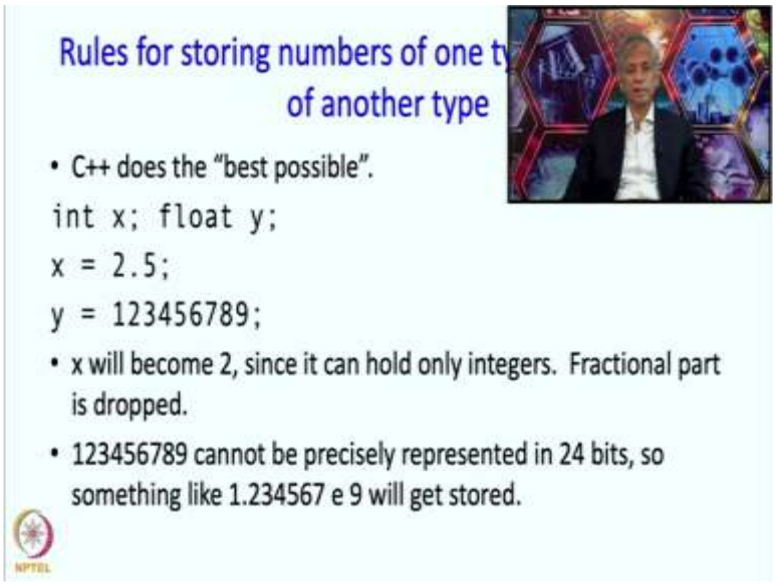
```
int x=2, y=3, z=4, u;
u = x/y + z/y;
```

What will this do?

- If dividend and divisor are both integers, then result is the quotient, i.e. remainder is ignored.
- u becomes: 2/3 + 4/3 = 0 + 1 = 1
- Program will give error if divisor is 0.

Now, division is a little tricky and suppose we have these four variables and if I write u=x/y + z/y. So notice that x and y are both integers and z and y are also integers. So in this case C++ has a somewhat unusual rule. So if the dividend and divisor are both integers then C++ will somehow produce an answer which is also integral. So the most natural answer to produce is to use the quotient. So basically, the division will be performed and the remainder will be ignored. So in this case 2/3 will produce the quotient 0, 4/3 will produce the quotient 1 and so the answer will be 0+1 or 1. So u will become 1 after this. And if you divide an integer by 0 then you will get error because within integers division by 0 is not defined.

(Refer Slide Time: 10:15)



Rules for storing numbers of one t of another type

- C++ does the "best possible".

```
int x; float y;
x = 2.5;
y = 123456789;
```

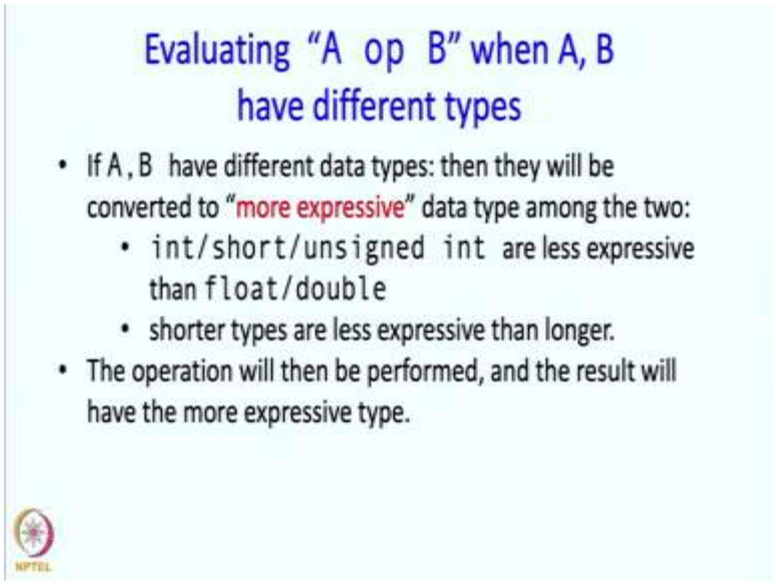
- x will become 2, since it can hold only integers. Fractional part is dropped.
- 123456789 cannot be precisely represented in 24 bits, so something like 1.234567 e 9 will get stored.

Now, C++ allows numbers of one type to be stored into variables of another type. And in such cases C++ tries to do the "best possible" it can under the circumstances. So suppose I have int x, float y and suppose I say x=2.5. Well x is integer. I cannot really store 2.5 into it. So what is the best possible? So here C++ says that I am going to be simplistic and I am just going to take the integer part of it. You could have said, why not round it? But C++ does not do that. C++ just rounds it down or takes the integer part.

If I am storing something into a floating point number, again the value that gets stored will be in in that scientific notation and further more if I have float y, then that only has about 7 digits of precision. Ok. So 123456789 has 9 digits which cannot be represented. So probably something like 1.234567 or may be even 1.23456 will get stored. But the magnitude will be roughly right and the exponent will be 9. So what is going to be stored here is 123456700. Ok. So instead of 123456789, 123456700 will get stored. So this is sort of an inhabitable consequence of using the scientific notation or using the float data type.

So these remarks will apply also to the double data type. But the only thing is that the double data type can store far larger number of digits. Something like 17, 18 digits can be stored. So in this case the representation will still be exact. If you had used a double.

(Refer Slide Time: 12:23)

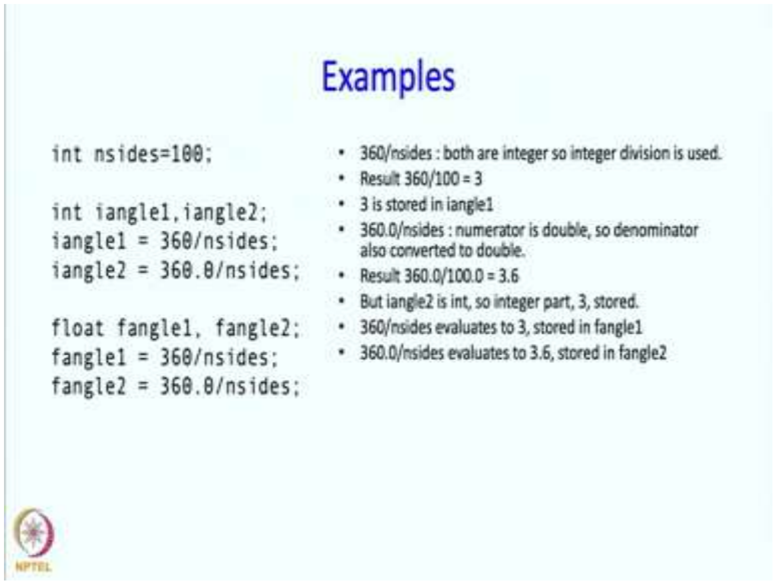


Evaluating "A op B" when A, B have different types

- If A , B have different data types: then they will be converted to "more expressive" data type among the two:
  - `int/short/unsigned int` are less expressive than `float/double`
  - shorter types are less expressive than longer.
- The operation will then be performed, and the result will have the more expressive type.

Now, C++ also allows you to mix different kinds of numbers when you operate on them. So I could write an expression, A some operator B where A and B have different types. So here are the rules that C++ uses to evaluate such expressions. So if A and B have different data types then they will be converted into the more expressive of those types. So, what does that mean? Well basically integers are less expressive than floating point. So floating point floating point can potentially represent integers exactly but integers can very rarely represent floating point numbers exactly. So that is kind of a reason to say that float or double are more expressive. And furthermore shorter types are less expressive than longer types. So something which use as 64 bits is certainly more expressive. So A and B you use, then both will get converted to the more expressive data type, you perform the operation and the result will have the type of the more expressive type.

(Refer Slide Time: 13:28)



Examples

```
int nsides=100;

int iangle1,iangle2;
iangle1 = 360/nsides;
iangle2 = 360.0/nsides;

float fangle1, fangle2;
fangle1 = 360/nsides;
fangle2 = 360.0/nsides;
```

- 360/nsides : both are integer so integer division is used.
- Result 360/100 = 3
- 3 is stored in iangle1
- 360.0/nsides : numerator is double, so denominator also converted to double.
- Result 360.0/100.0 = 3.6
- But iangle2 is int, so integer part, 3, stored.
- 360/nsides evaluates to 3, stored in fangle1
- 360.0/nsides evaluates to 3.6, stored in fangle2

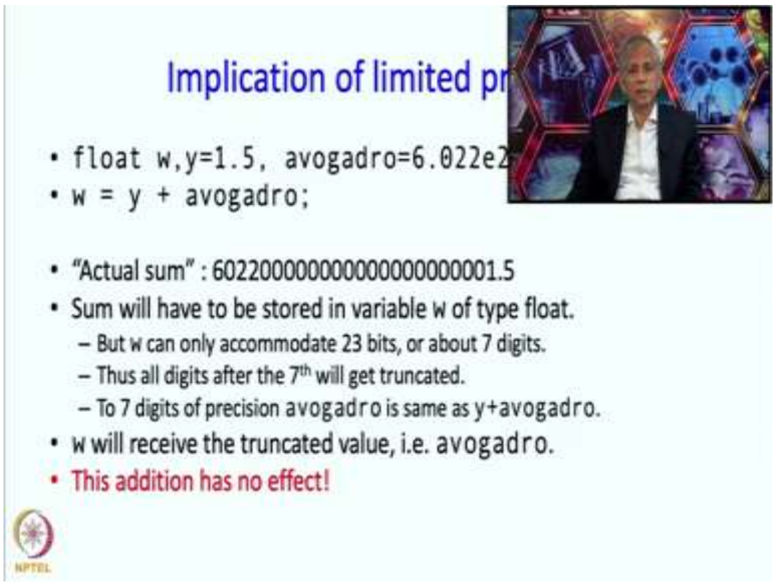
Some examples, so int nsides equal to 100, but anyway so suppose we have that and after that we write int iangle1=iangle2 and now if I write iangle1=360/nsides. So let us see what happens here, so 360 is int, nsides is int, so the result of division will be an int, integer division will be used and the result will be 3. Because the it is only going to take the quotient. So 3 will get stored into iangle1 which is an integer ok.

Then iangle2 suppose it is 360.0 divided by nsides. So then what happens over here is a little tricky ok. So 360 upon nsides the numerator is double. So if you if you write any constant by default it is double. So the numerator here is double, so denominator is also converted to double so 100 is converted to double then we do the division. So the result in this case will be 3.6. So the result of the expression on the right hand side 360.0 divided by nsides is actually going to be 3.6 now.

However, notice that iangle2 is integer. So while storing only the integer part will get stored. So we are going to do the same thing when the destination, the left hand side variables are going to be floats. So suppose you write fangle=360/nsides. So, what happens? So 360 and nsides are both integer so the result for the expression is 3 and 3 will get stored into fangle1.

If on the other hand we write 360.0/nsides then the result will be 3.6 and that will be stored in fangle2. So if you are expecting 3.6 to get stored then that will get stored only in fangle2. So this is something that you have to be careful about; especially when you mix integers and floating point values. One solution to this which is kind of a simple solution, but is a solution of worth, is really just to work with the data type double. So it has 16, 17 bits of precession and it will not do this type of truncation. So that is that is something that you can think about. But anyway very likely you will use doubles and ends mix together. And in that case you do need to know these rules.

(Refer Slide Time: 16:45)



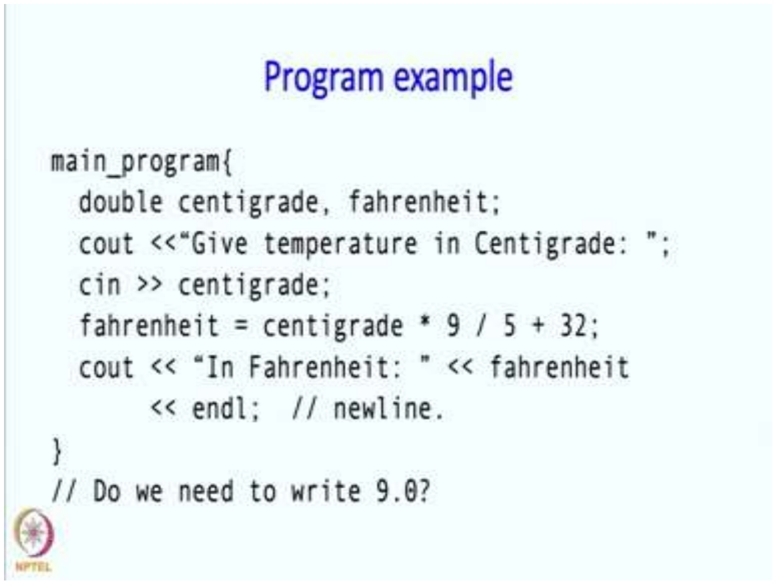
Implication of limited pr

- `float w,y=1.5, avogadro=6.022e2`
- `w = y + avogadro;`

- "Actual sum" : 6022000000000000000000001.5
- Sum will have to be stored in variable w of type float.
  - But w can only accommodate 23 bits, or about 7 digits.
  - Thus all digits after the 7th will get truncated.
  - To 7 digits of precision `avogadro` is same as `y+avogadro`.
- w will receive the truncated value, i.e. `avogadro`.
- This addition has no effect!

Alright, now when you use either double or float or the equivalent of scientific notation then there are some other things that you have to watch out for. So suppose for example, I define the variables w, y and Avogadro and I initialize y to 1.5 and Avogadro to 6.022e23. Now, what happens if I write w=y+Avogadro. Well, ideally what might happen is that I would have to write that entire Avogadro out in whatever 23 digits or so and then add 1.5 to it and the actual sum will be something like this. However, this actual sum is going to be stored in w which is a floating point number. So w can only accommodate 23 bits, which is about 7 digits. So really only the first 7 digits will be considered. Ok. So everything after the 7th will get truncated. So to 7 digits of precision Avogadro is really the same as y+Avogadro. 'w' will receive the truncated value that is the Avogadro itself. Ok. So basically if you are adding a very small number to a very large number then the addition may not have any effect at all.

(Refer Slide Time: 18:12)



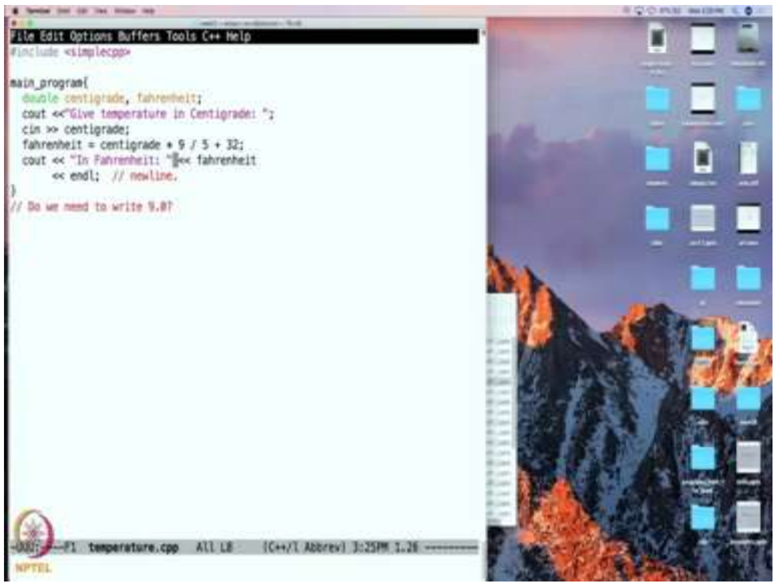
Program example

```
main_program{
  double centigrade, fahrenheit;
  cout <<"Give temperature in Centigrade: ";
  cin >> centigrade;
  fahrenheit = centigrade * 9 / 5 + 32;
  cout << "In Fahrenheit: " << fahrenheit
       << endl;  // newline.
}
// Do we need to write 9.0?
```

So let us put together whatever you have learnt or some of what we have learnt to write a simple program. So in this program we are going to define two variables; centigrade and Fahrenheit and you may you can guess what I am going to put into them. So I am going to print out a message saying "give the temperature in centigrade". So after that I will wait for the user to type in some value and the Cin statement will cause that value typed by the user to be stored in centigrade. Now, I can calculate Fahrenheit, right? So the formula is the Fahrenheit value is centigrade value times 9 upon 5 plus 32. So that is what I have written down over here. So as a result of this the variable Fahrenheit will actually get the equivalent value of temperature in Fahrenheit equivalent to whatever was typed by the user which we interpreted as a centigrade temperature value. And finally this statement is going to print out a message saying that in Fahrenheit the temperature is so and so. And we are putting in endl which forces the value to go out and of course an endl to a line to also appear.

Now, let me just ask you, ok, do I need to write 9.0 or will this work? Well think about it for a second before you hear my answer perhaps; so here centigrade has been defined as a double variable, so when we do the multiplication itself the result will be a double. So again when we do the division it will be a double divided by 5. But when you divide a double then the result is going to be a double and therefore, we will not have any truncation and will indeed get the result calculated correctly. Ok.

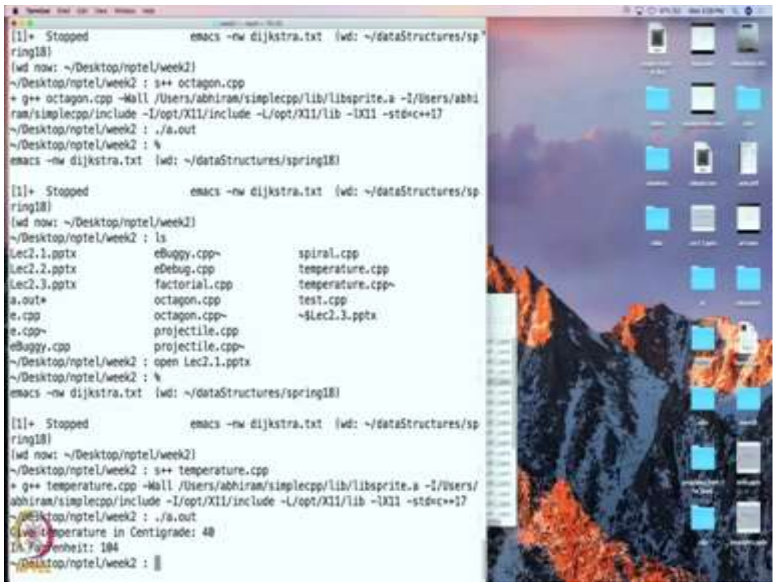
(Refer Slide Time: 20:20)



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

main_program{
  double centigrade, fahrenheit;
  cout <<"Give temperature in Centigrade: ";
  cin >> centigrade;
  fahrenheit = centigrade * 9 / 5 + 32;
  cout << "In Fahrenheit: "<< fahrenheit
       << endl;  // newline.
}
// Do we need to write 9.0?

-UUU:---F1  temperature.cpp   All L8     (C++/l Abbrev) 3:25PM 1.26 ---------
```



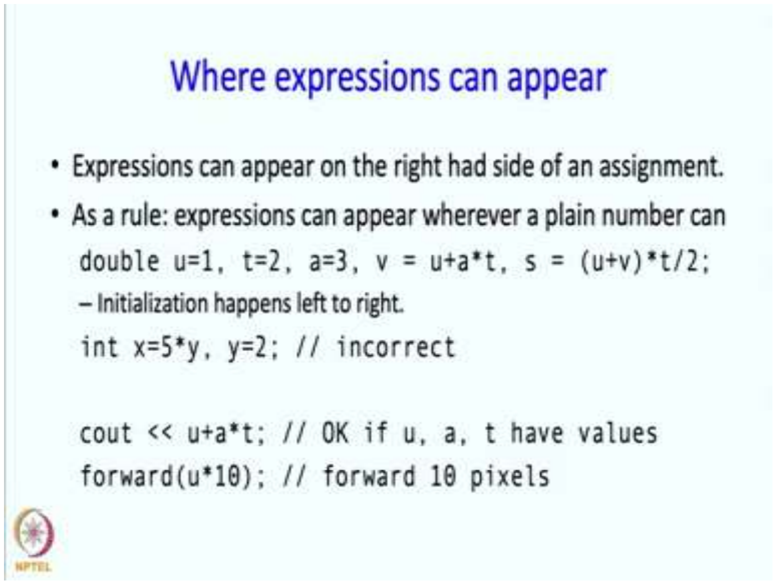
```
[1]+  Stopped                 emacs -nw dijkstra.txt  (wd: ~/dataStructures/sp
ring18)
(wd now: ~/Desktop/nptel/week2)
~/Desktop/nptel/week2 : s++ octagon.cpp
+ g++ octagon.cpp -Wall /Users/abhiram/simplecpp/lib/libsprite.a -I/Users/abhi
ram/simplecpp/include -I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week2 : ./a.out
~/Desktop/nptel/week2 : %
emacs -nw dijkstra.txt  (wd: ~/dataStructures/spring18)

[1]+  Stopped                 emacs -nw dijkstra.txt  (wd: ~/dataStructures/sp
ring18)
(wd now: ~/Desktop/nptel/week2)
~/Desktop/nptel/week2 : ls
Lec2.1.pptx           eBuggy.cpp~           spiral.cpp
Lec2.2.pptx           eDebug.cpp            temperature.cpp
Lec2.3.pptx           factorial.cpp         temperature.cpp~
a.out*                octagon.cpp           test.cpp
e.cpp                 octagon.cpp~          ~$Lec2.3.pptx
e.cpp~                projectile.cpp
eBuggy.cpp            projectile.cpp~
~/Desktop/nptel/week2 : open Lec2.1.pptx
~/Desktop/nptel/week2 : %
emacs -nw dijkstra.txt  (wd: ~/dataStructures/spring18)

[1]+  Stopped                 emacs -nw dijkstra.txt  (wd: ~/dataStructures/sp
ring18)
(wd now: ~/Desktop/nptel/week2)
~/Desktop/nptel/week2 : s++ temperature.cpp
+ g++ temperature.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/week2 : ./a.out
Give temperature in Centigrade: 40
In Fahrenheit: 104
~/Desktop/nptel/week2 :
```

So I am going to stop the slides for a second and I am going to show you a demo of this program. Ok, so here I have the program that I showed you earlier all typed out. So I am going to now compile it. So I go back to my shell again and now I am going to compile it. So, it has compiled and now I can execute it. So as soon as I execute it, as soon as I run it, I get the message, “give temperature in centigrade”. So,we can give some temperature, so I guess if you want to check whether the program is behaving properly. May be we should type in some value whose Fahrenheit equivalent we know. So say we type in 40 whose Fahrenheit equivalent is 104. So indeed it does print 104 and so our program is in fact running correctly.

(Refer Slide Time: 21:17)



Where expressions can appear

- Expressions can appear on the right had side of an assignment.
- As a rule: expressions can appear wherever a plain number can

```
double u=1, t=2, a=3, v = u+a*t, s = (u+v)*t/2;
```

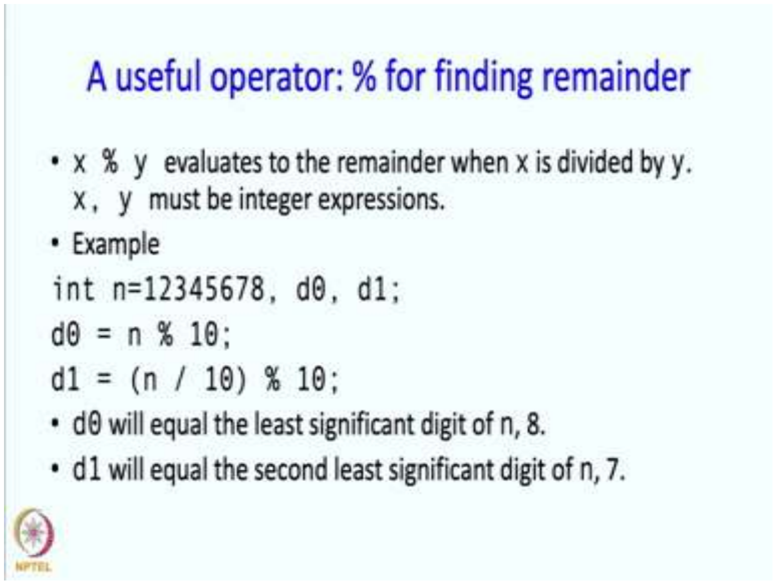
  – Initialization happens left to right.

```
int x=5*y, y=2; // incorrect

cout << u+a*t; // OK if u, a, t have values
forward(u*10); // forward 10 pixels
```

Ok, so now I want to make some comments about expressions, specifically, where can expressions appear in the program? So basically expressions can appear on the right hand side of an assignment that we have already seen; but in general, you could say that expressions can appear wherever a plain number can. So for example, in this code I can put numbers to initialize; but I can also put expressions. Ok. So instead of putting in a number I have putting in a expression. Instead of putting in a number over here I have put in an expression. So now, the initialization happens left to right. Ok. So u will get initialized, t will get initialized, a will get initialized. And then v and s will get initialized. So this is important because when you initialize v, you are using the values of u, a and t. So because the initialization is happening left to right, the correct values will be available over here and similarly for this. So for example, if you define variables in this manner where you are referencing a variable which has not yet been defined in the left to right order, then this is incorrect. To have this correct, you should really write y=2 first and then x=5*y. Now, you could have printed just a number if you wanted or you could have printed a variable but you can also print out an expression. So this will cause this expression to be evaluated and that value will get printed. Similarly, when you are passing an argument to a command you could place a value, a numerical value or you could place the name of a variable or you can put an expression. So before that value is actually sent off to that command, the expression is evaluated and only the resulting value will get sent off.

(Refer Slide Time: 23:38)



A useful operator: % for finding remainder

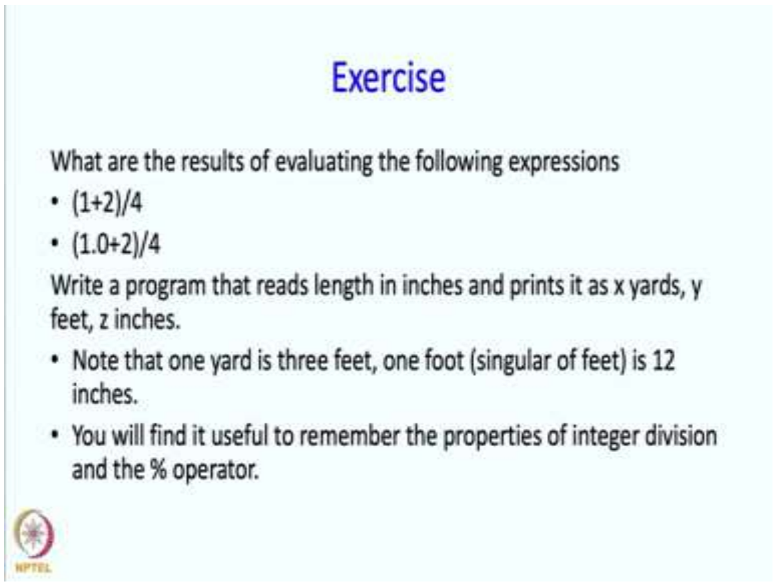
- x % y evaluates to the remainder when x is divided by y. x, y must be integer expressions.
- Example

```
int n=12345678, d0, d1;
d0 = n % 10;
d1 = (n / 10) % 10;
```

- d0 will equal the least significant digit of n, 8.
- d1 will equal the second least significant digit of n, 7.

Now, very often we might want to find the remainder and in C++ the remainder is found by using the operator ‘%’. The character percent is actually the remainder operator. Ok. So x%y evaluates to the remainder when x is divided by y and x, y must be integer expressions. So for example, I might write something like this, int n=12345678, d0, d1. So if I write d0=n%10, d0 will get the remainder of n modular 10. So it will get 8. d1 I am writing as (n/10)%10. So what will n by 10 be? n by 10 will just be the quotient when divided by 10, so that will be 1234567. That when taken modular 10 again will get be 7. So basically what is going on over here is that by using the remainder operator we have a way of extracting the digits of a number, by taking remainder and dividing by 10 as many times as we want.

(Refer Slide Time: 24:34)



Exercise

What are the results of evaluating the following expressions

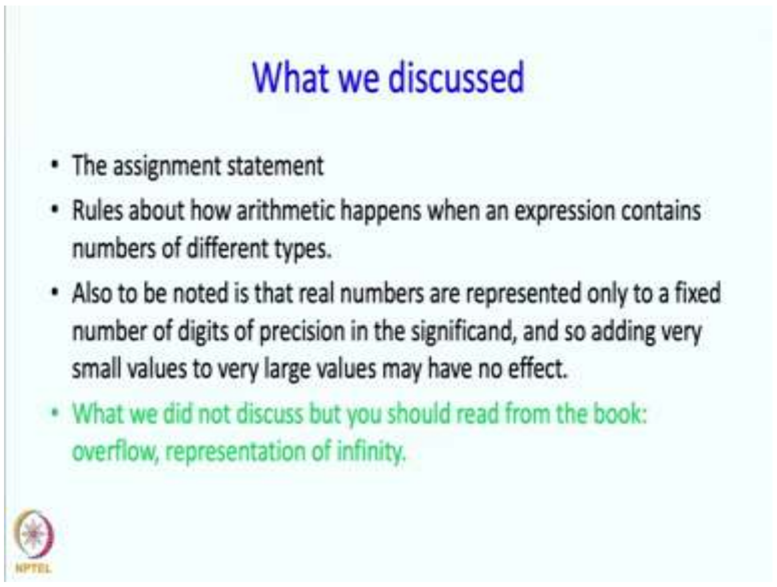
- (1+2)/4
- (1.0+2)/4

Write a program that reads length in inches and prints it as x yards, y feet, z inches.

- Note that one yard is three feet, one foot (singular of feet) is 12 inches.
- You will find it useful to remember the properties of integer division and the % operator.

So here are some exercises, these are going to test whatever you have learnt in this segment.

(Refer Slide Time: 24:41)



What we discussed

- The assignment statement
- Rules about how arithmetic happens when an expression contains numbers of different types.
- Also to be noted is that real numbers are represented only to a fixed number of digits of precision in the significand, and so adding very small values to very large values may have no effect.
- What we did not discuss but you should read from the book: overflow, representation of infinity.

So, what have we learnt? So we have learnt about assignment statement, we have learnt about how arithmetic happens when expressions contain numbers of different types and we should also note that real numbers are represented only to a fix number of digits of precision. So adding very small values to very large values may not have may have no effect. What we did not discuss but what you should read from book are things like overflow and representation of infinity. So, we will stop this segment will end this segment here.