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 - 3
Basic Elements of Program
Reassignment, sequence generation and accumulation

(Refer Slide Time: 0:21)

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.

In the previous segment, we discussed the assignment statement and rules for expression evaluation.

(Refer Slide Time: 0:25)

Re assignment

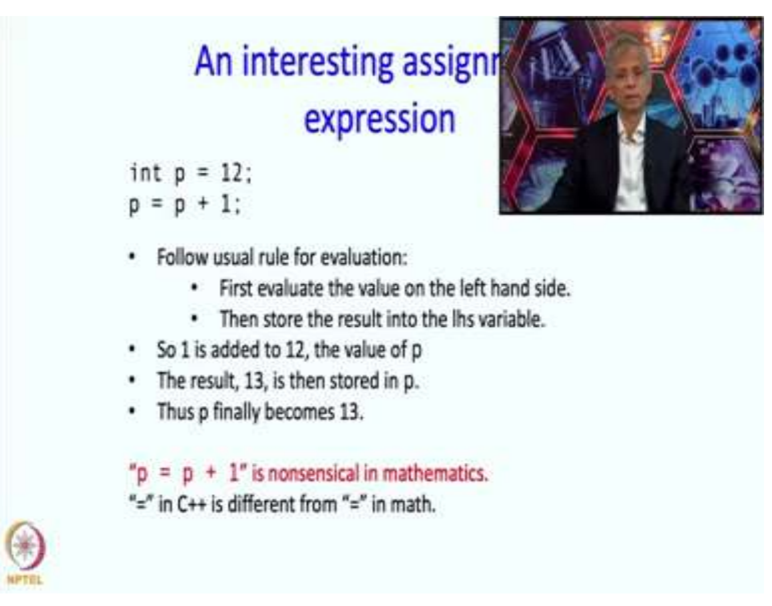
- Same variable can be assigned ag
- When a variable appears in a statement, its value at the time of the execution gets used.

```
int p=3, q=4, r;
r = p + q;          // 7 stored into r
cout << r << endl;  // 7 printed
r = p * q;          // 12 stored into r
cout << r << endl;  // 12 printed
```

So, in this segment we are going to look at reassignment. This looks simple enough basically changing the value given to a single variable ok, but this actually is turns out to be rather powerful and very very common operation and it needs to be understood properly. Ok, so let the rule for evaluating expressions is that when a variable appears in a statement its value at the time of execution is used. So suppose I write p=3 , q=4, r, and I write r=p+q. So, the values of p and q at the time of executing this are 3 and 4. So, what is going to happen is that those values are going to get used and 7 will be stored. So, if I print r out then 7 will get printed.

Now, suppose I write r=p*q, then the current values of the p and q will be used and 12 will get stored into r. Now, if I print out r the current value of r which is 12 will get printed.

(Refer Slide Time: 1:48)



Now, reassignment can lead to some interesting expressions. So, suppose I have int p=12. What will happen if I write p=p+1? Is it even legal? So, in C++ it is actually legal. So, the rule for executing such a statement is something that we have already discussed. So, the usual rule says first evaluate the value on the left hand side so, the current value of p is 12, you add that 1 so, you get 13 and then the whatever the result is stored into the variable on the left-hand side. So, 1 is added to 12, the current value of p and the result 13 gets stored in p. So, basically it will cause the value of p increase by 1.

Now, this statement is sometimes found confusing by people because writing p=p+1 is nonsensical in mathematics and the equal to operator is very very strongly linked in our minds to mathematics. So, this says that the equality in mathematics and in programming is quite different. In mathematics its more like equality, in programming it means, this says take

the value of the expression of the right-hand side and put that value into the variable whose name appears on the left-hand side. So, basically equals to in C++ is different from equal to in math and you should be aware of this very much.

(Refer Slide Time: 3:40)

Repeat and reassignment

```
main_program{
  turtleSim();
  int i=1;
  repeat(10){
    forward(i*10);
    right(90);
    cout << i << endl;
    i = i + 1;
  }
  wait(5);
}
```

- First iteration:
  - 1 printed, i changes to 2.
- Second iteration:
  - 2 printed, 2 changes to 3.
- ...
- 10th iteration:
  - 10 printed, i changes to 11.
- Fundamental idiom: sequence generation.
- What does this draw?
  - "Rectangular spiral"

Now, reassignment turns out to be quite interesting with repeats. So, here is a program I am going to have int i=1 , repeat 10 times and let us say I write cout<<i<<endl; and then I write i=i+1. So, what happens when I execute this? In the first iteration, when the loop is when the repeat statement body it is entered then i will have value 1. So, because of that 1 will get printed and i=i+1 will change i to 2.

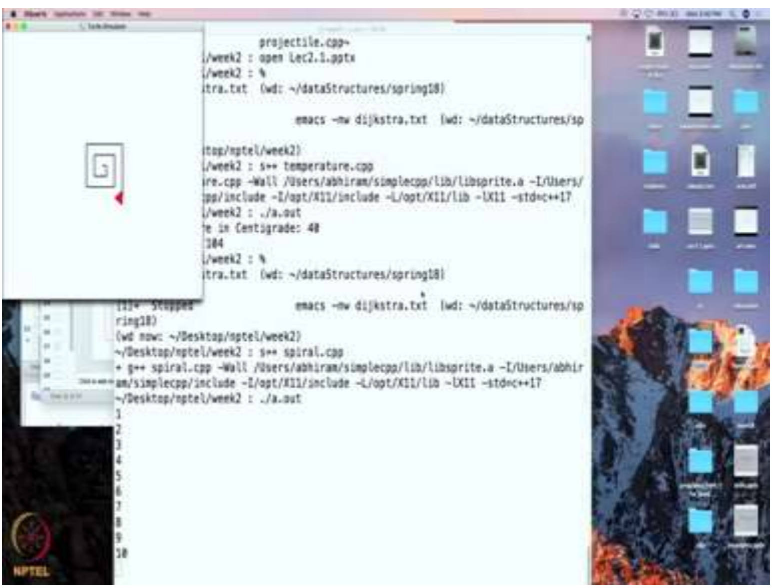
In the second iteration, what is going to happen? Well, the value that i now has is 2 so, 2 will get printed and i=i+1 will change that 2 to 3. So, at this point i will have the value 3 and if you keep on going in this manner in the 10th iteration 10 will get printed and i will change to 11. So, something quite interesting has happed over here this set of statements has been able to print out the numbers between 1 and 10 for us.

So, basically this idiom we can call a sequence generation idiom. So, if you want to ever generate sequence of numbers 1 through 10 then, this idiom will do it, which idiom? Repeat 10 times and i=i+1, the cout<<i is not really an essential part of idiom we put it there only because we wanted the sequence to be seen.

Now, suppose we put in the statement turtleSim() and we put in the statement forward(i*10), and right(90) what do you thing this does? So, notice that in this green portion we are not changing the value of i at all. So, i will have the same values in each iteration as it did earlier,

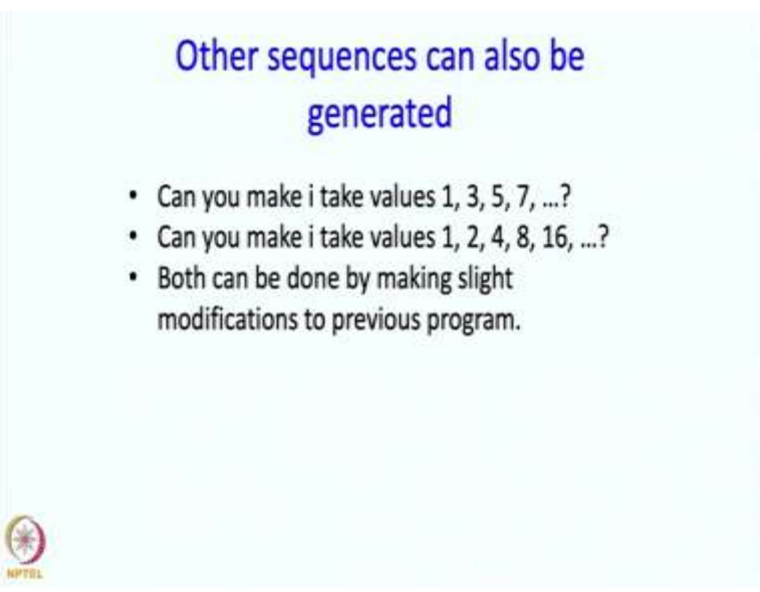
but this means that the distance by which the turtle goes forward is going to be different in each iteration. So, first it will be 10, next it will be 20, then it will be 30 and so on. So, this is sort of like drawing a square but the side length is increasing. So, what do you think happen? So, this going to cause something like a rectangular spiral, let us take a look at this.

(Refer Slide Time: 6:12)



So this program also I have written and it is called spiral.cpp. So, here the program and I guess what I have not put in over here is this wait. Let me indent this properly because otherwise it does look little confusing. So, it is what we have written over there? Yes, it is so it is also going to print i but it is also going to cause movement, ok so, let us see what it does. So, let me compile it and let me run it. So, it did the spiral just as we expected.

(Refer Slide Time: 7:31)



Now, you should realise that we just do not really need to generate sequence 1 through 10, you can generate other sequences also. So, it is an easy exercise for you to generate the sequence 1, 3, 5, 7 and so on, or you can also generate the sequence 1, 2, 4, 8, 16. Basically, you can generate these sequences by making slight modification to the previous program. May be change what you add, may be instead of add do something else. So, just try it out.

(Refer Slide Time: 7:48)

Another idiom: accumu

```
// Program to read 10 numbers and add them
main_program{
   int term, s = 0;
     repeat(10){
          cin >> term;
          s = s + term;
     }
     cout << s << endl;
}
// values read get "accumulated" into s
// Accumulation happens here using +
// We could use other operators too.
```

Now, using the assignment statement or rather the reassignment statement inside a repeat loop can allow you some other things as well. So, here is a program which reads 10 numbers and adds them together. So, so we declare variables int term, s. 's' is going to be 0, s is going to have the sum, maybe we should actually call that variable sum, but that is ok. So, we are going to do repeat 10 times and so, we are going to have cin>>term so, the value goes into the term and then we add this value to s. So, whatever value we get we are going to add to s. So, notice that initially s is 0, but in the first iteration of this loop s will change and whatever you receive through keyboard is going to be added to s, what happens in the next iteration? In the next iteration, whatever the user types with the keyboard the second time around is stored into the variable term and that will get added to s. So, at the end of 2 iterations, s will have the sum of 2 variables. So, at the end of 10 iterations s will have the sum of 10 variables. So, if you type s at the end of it you will get the sum of 10 variables.

Now, s is serving like an accumulator, it accumulates by summation whatever values are read from the keyboard and so this idiom we might as well call the accumulation idiom. So, you could do other type of accumulations as well, you can accumulate by using products for example, or you take the maxima and keep accumulating the maximum. So, all kinds of other things are possible.

(Refer Slide Time: 9:52)

Composing the two idioms: program to calculate n!

```
main_program{
  int i=1, n;   cin >> n;
  int nfac=1;
  repeat(n){
     nfac = nfac * i; // multiplied by 1 to n
     i = i + 1; // i goes from 1 to n
  }
  cout << nfac << endl; // n! printed
}
```

Now, an interesting thing happens we can compose these two idioms and write a program which calculates n factorial. So, here is our basic program, ok. So, this is the first idiom. So, this is the sequence generation idiom and it is generating the sequence 1 through n. Now, what do we need to calculate n factorial? Well we want that sequence but we just do not want to leave that sequence alone we want to multiply all those numbers. So, this is easily done by writing a new variable out called nfac, it is initially 1 and then we simply write nfac=nfac*i. So, the first time around nfac will get multiplied by 1, the second time around nfac will get multiplied by 2, 3, 4 all the way till n. So, at the end if we print nfac we will get n factorial printed. So, the idioms as we can see are quite powerful, we can sort of mix them up as well.

(Refer Slide Time: 10:58)

Some additional operators

- The fragment "i = i + 1" appears very frequently, and so can be abbreviated as "i++".
- ++ : increment operator. Unary
- Similarly we may write "j--" which means "j = j - 1"
- -- : decrement operator

Now, because the assignment statement and the reassignment statement are so commonly used C++ defines the addition operator. So, here for example i=i+1 is something which appears very frequently and so, in C++ it can be abbreviated as i++. So, ++ is an operator and and it is a unary operator and it can be written as i++ which simply means i=i+1. So, it is an increment operator it is a unary. And similarly, you can have --, which means j=j-1 and -- is a decrement operator.

(Refer Slide Time: 11:43)

Intricacies of ++ and --

- ++ and -- can be written after the variable, and this also cause the variable to increment or decrement.
- Turns out that expressions such as k = ++i; and k = i++; are legal in C++ and produce different results.
- Such assignments are described in the book for completeness.
- But they are somewhat hard to read, and so it is recommended you do not use them.
- Similarly with --.

Now, ++ and -- are actually quite tricky. ++ and -- can be written after the variable, or before the variable. So, I can write ++i or I can write i++. Both are allowed. Now, furthermore, I can also write expressions such as k=++i and k=i++. So, these expressions are legal and as it

happens, they produce different results and so, here ++i and i++ are different. If I just write i++ and ++i in isolation then they change the value of i and they change the value of i in a similar manner, but here they produce some different results as per as the k is concerned.

Now, such assignments are described in the book, but, I believe, and many programmers believe that such expressions are really difficult to understand. So, it is like using very complicated words when you do not need to use them. So it said that when you speak, you should use simple words as simple words as possible. So, what has happened over there is that these variables were defined in the language but more and more people are saying do not use these complicated forms like k=++i and k=i++.

So, indeed in this course we are not going to use them and I will recommend to you that do not use them in your carrier as well so just stick to the simple forms, same for --.

(Refer Slide Time: 13:43)

Compound assignment

- The fragments of the form "sum = sum + expression;" occur frequently, and so C++ allows them to be shortened to "sum += expression;"
- Likewise you may have *=, -=, ...
- Example

```
int x=5, y=6, z=7, w=8;
x += z;   // x becomes x+z = 12
y *= z+w; // y becomes y*(z+w), = 90
```

- Note: z, w do not change.

Now, the fragment of the form sum=sum+expression also occurs frequently, so C++ allows such fragments to be shortened to sum+=expression. So you can also have *=, -=, to represent sum=sum*expression or sum=sum-expression. So, if I write x+=z, then really it is x=x+z so x is going to become 12 because x is 5 and z is 7.

Similarly, I can write star equal to, so here, if I write y*=z+w note that z and w do not change, I mean this is not this is not something new that I am telling you if you get confused remember that it is really exactly equivalent to writing y=y*z+w and this when you evaluate the expression z and w do not change.

(Refer Slide Time: 14:50)

Exercise

- What does the following program do?

```
unsigned int n=7589, m=0;
repeat(5){
  m = 10*m + (n % 10);
  n = n/10;
}
```

- What are the values of the variables after the following statements execute

```
int i=1,j=2,k=3;
i=j;
j=k;
k=i;
k++;
i--;
```

So here are some exercises to you to try out based on the material that we are seen in this segment.

(Refer Slide Time: 14:56)

What we discussed

- The value of a variable can be changed (reassigned).
- Assignments such as i = i+1 or j = j*2 are allowed.
  - They are useful for generating sequences
  - Once you can generate sequences, you can compute expressions such as n!
  - Other uses will be seen later
- Because assignments such as i = i+1 and j = j*2 are very frequently needed, operators such as ++ and *= have been provided.

So, what have we discussed in this segment? Well we said that the value of a variable can be changed and assignments such as i=i+1 and j=j*2 are allowed they are useful for generating sequences, then once you are able to generated the sequences you can compute the expression such as n factorial and several other users are there. And because we do things like i=i+1 and j=j*2 operators such as ++ or *= have also been provided. So, we will stop at this point we can continue later.