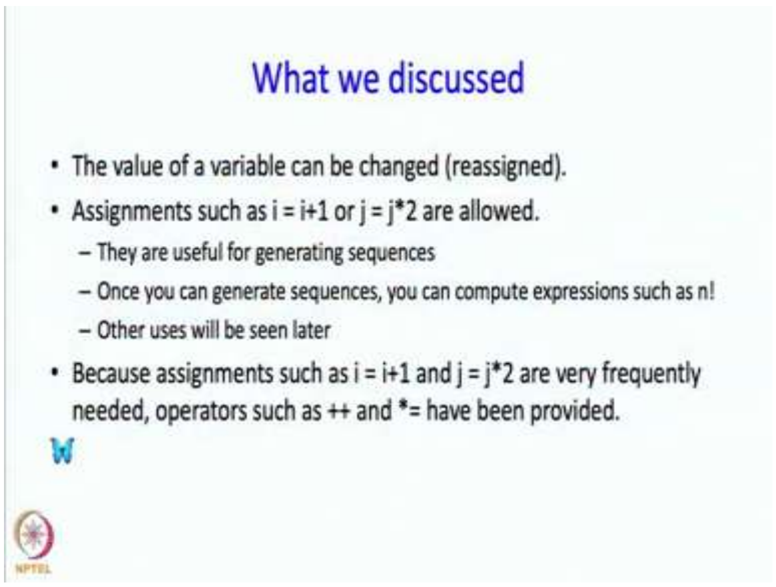
An Introduction to Programming through C++

Professor Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Lecture No. 3 Part - 4
Basic Elements of Program
Blocks and scope

(Refer Slide Time: 0:21)

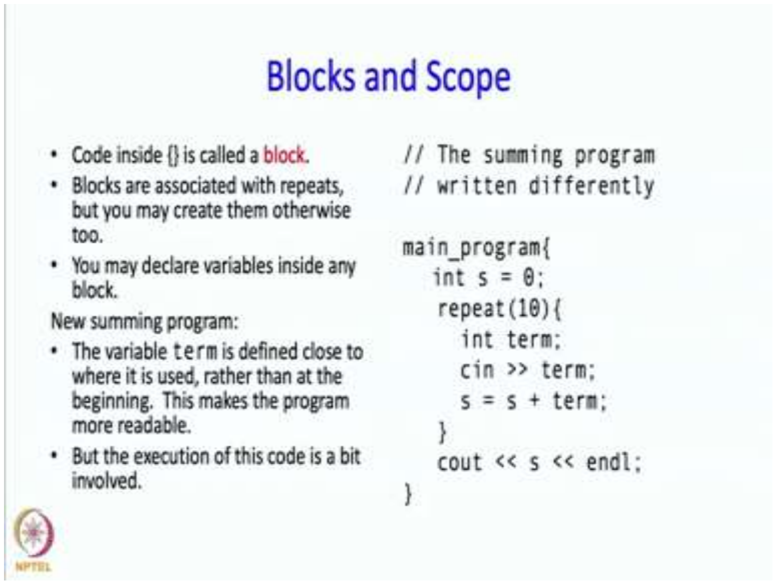


What we discussed

- 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.

In the previous segment we discussed the assignment statement further or rather we discussed how values can be reassigned and especially reassigned inside a loop we generate sequences and do interesting computation in general. And we also discussed some operators such as ++ and *= to.

(Refer Slide Time: 0:39)



Blocks and Scope

- Code inside {} is called a block.
- Blocks are associated with repeats, but you may create them otherwise too.
- You may declare variables inside any block.

New summing program:

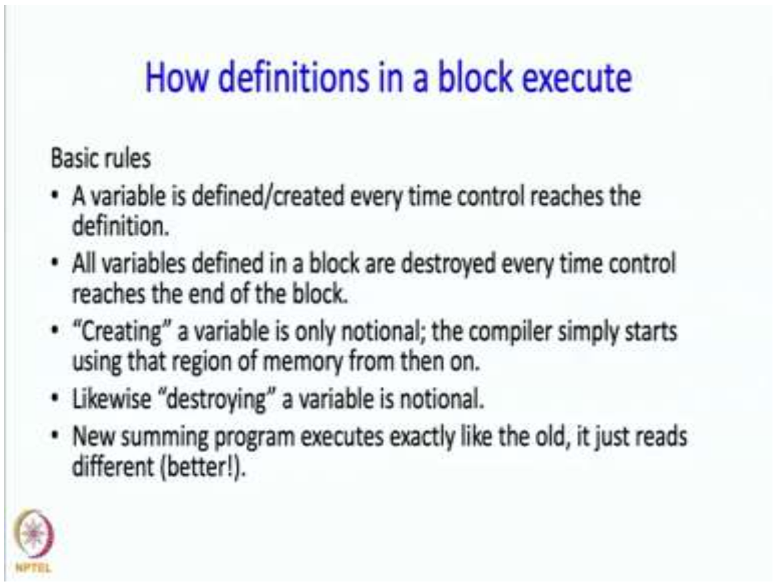
- The variable term is defined close to where it is used, rather than at the beginning. This makes the program more readable.
- But the execution of this code is a bit involved.

```
// The summing program
// written differently

main_program{
   int s = 0;
    repeat(10){
      int term;
      cin >> term;
      s = s + term;
    }
    cout << s << endl;
}
```

Now, we are going to discuss one idea which is related to how you write programs and how you define variables or rather where you define variables. So some technical terms first, the code that you write inside the braces is called the block. So, so far you have seen blocks along with repeats. So a repeat is repeat and in parentheses the count of how many times repeat which has to be followed by a block which gives you the body. But you can define blocks otherwise also and inside blocks also you can declare variables. Okay, so here is how I might choose to write the summing programs slightly differently. So in this summing program the main differences that this term variable has been define inside. Otherwise the entire program is the same. So this program is equivalent to the previous program, but it really executes slightly differently. And it really says that look I want a term variable and it is being used over here. So I might as well put the definition of it close by so I can quickly tell that it was not initialized to anything and that it is an integer variable. So this is a device for bringing the definition close to the users. So to that extent the program become a little bit more readable. But the execution of this code is a little bit tricky.

(Refer Slide Time: 2:28)



How definitions in a block execute

Basic rules

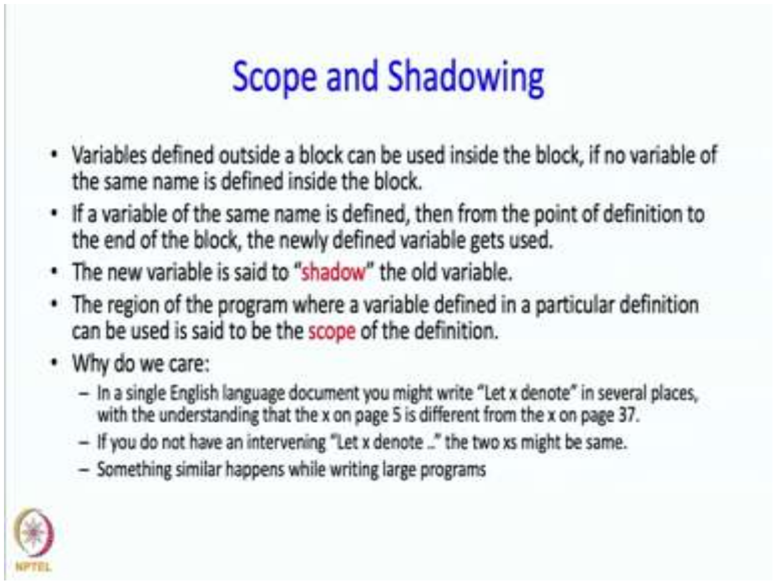
- A variable is defined/created every time control reaches the definition.
- All variables defined in a block are destroyed every time control reaches the end of the block.
- "Creating" a variable is only notional; the compiler simply starts using that region of memory from then on.
- Likewise "destroying" a variable is notional.
- New summing program executes exactly like the old, it just reads different (better!).

So, how the definitions in a block execute? Well the rule are as follows. So a variable which is define inside a block will get created every time the control reaches the definition. Then all the variable define in the block are destroyed every time control reaches the end of the block. Sso this term variable which we just saw will get created at this point and get destroyed again at this point. So this as you can see does not really hurt us, because it really is useful only within this region. So you read in value and you add it to s, s stays around because s is not defined inside this, s was defined outside. So the rule of getting destroyed does not apply to S. But whatever value we read in we actually got into s, so we do not care if term gets destroyed. This means this program is going to be work exactly like the way we wanted to. 's' will accumulate the values that get typed, the sum will get placed in s. But notice that there is one more implementation that the variable term will not available over here. Okay, so in some sense it is sort of a private or a local variable for this loop. That is in fact also a term that often get used-that term is now made local to the body of this loop.

Now, when you talk about creating, you may think is not that of work. But in this case creating is not really a much work or is not really any work, basically the compiler from that point onwards starts using a certain piece of memory which it has any way. Okay, destroying is also not any work because the compiler stops using that piece of memory. But yes, in some other cases which we might see later, creating variables inside might has some cost. And for that reason you may choose to not has variables define inside in but rather have the variables defined once and for all outside if that is indeed what you want.

But in this case, it is a good idea to define the variables inside because it really says that that variable term is not really important for the final result. It is something that is being use only locally, whereas s is an important variable. It is variable which is more global.

(Refer Slide Time: 5:21)



Scope and Shadowing

- Variables defined outside a block can be used inside the block, if no variable of the same name is defined inside the block.
- If a variable of the same name is defined, then from the point of definition to the end of the block, the newly defined variable gets used.
- The new variable is said to "shadow" the old variable.
- The region of the program where a variable defined in a particular definition can be used is said to be the scope of the definition.
- Why do we care:
  - In a single English language document you might write "Let x denote" in several places, with the understanding that the x on page 5 is different from the x on page 37.
  - If you do not have an intervening "Let x denote .." the two xs might be same.
  - Something similar happens while writing large programs

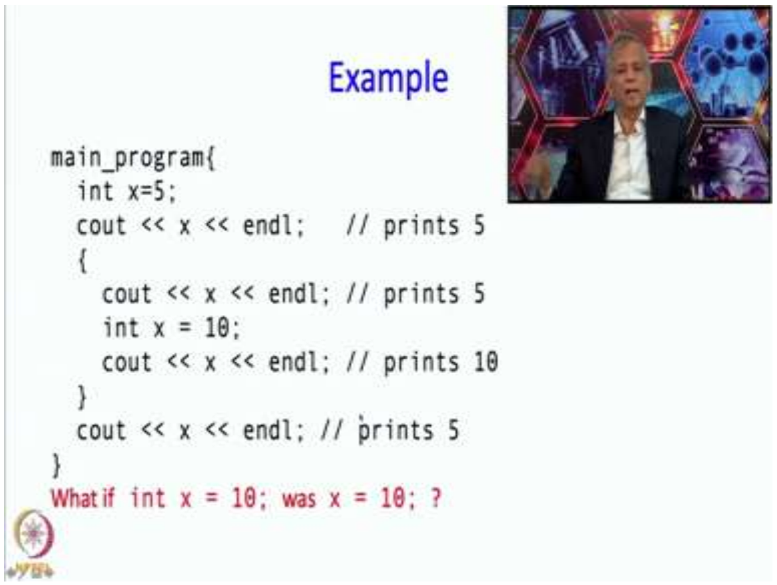
Alright, now, once we have mechanisms to define variables at different places, we need to carefully understand what a certain variable name refers to. So, if a variable is define outside a block then it can be used inside a block just as the variable S was defined outside and it got used inside the block. However, this can happen only if no variable of the same name is defined inside the block. If a variable of the same name is defined, then from the point of the definition to the end of the block the newly defined variable get used. So what is happening is this the new variable is said to "shadow" the old variable. The region of a program where the variable defined in a particular definition can be used is said to be the scope of the definition. So the scope of the variable defined inside a block starts at the point of the definition and ends at the end of the block.

Well why do we are care of all these things? So in some sense the programming is also like writing text. Say you are a writing a long document. So in a single English language document you might write "let x denote" in several places, so if you write on page 5 and on page 7 then there is simplicity to understanding that the x on page 5 is different from the x on page 7. So essentially we can say that the scope of that definition on page 5 is maybe from the page 5 to page 37 or something like that or maybe it is from the page 5 to the end of the chapter or maybe even the end of the section.

So in English language the scope is not very formally specified it is sort of left as understood. But in programming since the computer has to do something with it, we have to be really careful and talk about the scope. Okay and further more if you do not have an intervening “x denote” then you can say that maybe the two x’ are the same of course you really do not want the same x to be used so far apart or at least if you are using so far apart then you will warn in an English language document, but in a program unless there is an intervening “x denote” and if the x is not inside a block which is like a chapter where the scope ends, the two x’ really can be considered to refer to the same concept or in case of the program the same variable.

So this is the motivation why we are being so finicky about scope and shadowing. If you do not use the same names then this really does not apply but you will see that it does make sense to use the same names and so this discussion will actually come in useful.

(Refer Slide Time: 8:32)



Example

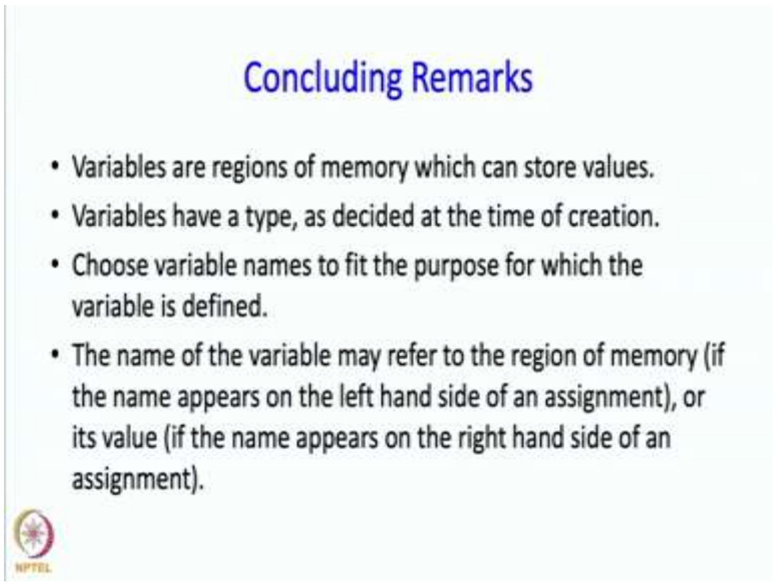
```
main_program{
  int x=5;
  cout << x << endl;   // prints 5
  {
    cout << x << endl; // prints 5
    int x = 10;
    cout << x << endl; // prints 10
  }
  cout << x << endl; // prints 5
}
What if int x = 10; was x = 10; ?
```

So here we can take an example, so we have a variable x so if we print it out then 5 will get printed. And this x will refer to that definition. Now, if you print it again, again 5 will printed but now if you write int x=10 since it a new block we are started a block this definition is actually allowed. If we have not started a new block then this definition is actually not allowed. C++ will say that look you already have variable called x how can you define another variable?

But since we have started a new block, C++ most that you can have another variable. So now, if you print x it will print 10. So say the block ends over here and suppose over here you print x in this case 5 will get printed.

Okay, why? Because the scope of this x is limited to this region and over here or this point is in the scope of this definition so 5 will get printed. So when you come out, name x will start to referring to this variable over here. Now, I will leave you with a question if instead of this int x=10 suppose I wrote x=10, then what to change? I would like to think this and write this program and check your answer I am not going to tell you the answer.

(Refer Slide Time: 10:04)

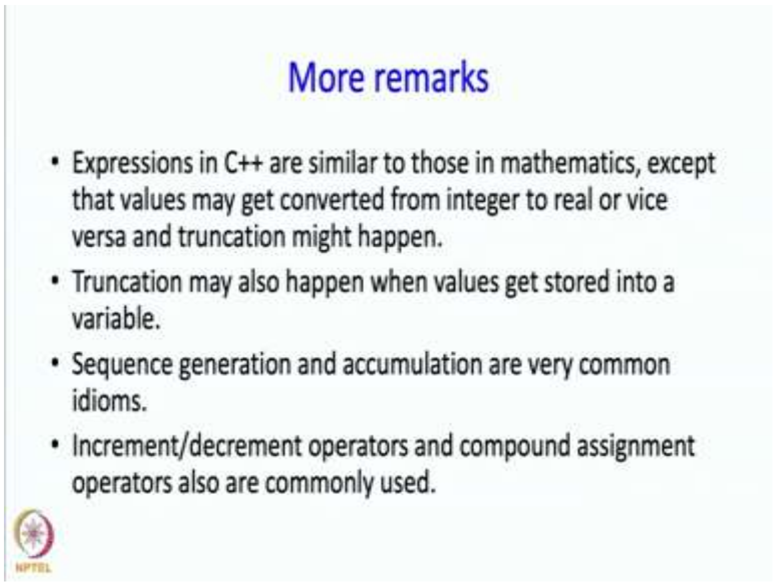


Concluding Remarks

- Variables are regions of memory which can store values.
- Variables have a type, as decided at the time of creation.
- Choose variable names to fit the purpose for which the variable is defined.
- The name of the variable may refer to the region of memory (if the name appears on the left hand side of an assignment), or its value (if the name appears on the right hand side of an assignment).

So some remarks, we have come to the end of this lecture and I want to summarize what we have done in this. So first we defined the variables are regions of memory which can store the values. Variables have types and this type is used to interpret how the bits stored in that region are going to be interpreted. Then, you are advised to choose variable names so that they describe the purpose for which the variable is defined. And we said that when we use a variable when that name if it appears at left hand side of an assignment actually refers to the variable itself it refers to the region of memory. On the other hand, if it appears on the right hand side then you really do not care of about the memory but you are saying give me its value. So there is sort of dual view or this name sort of is interpreted depending on its context and you should keep this is in the mind or may be this is in your mind but anyway.

(Refer Slide Time: 11:33)

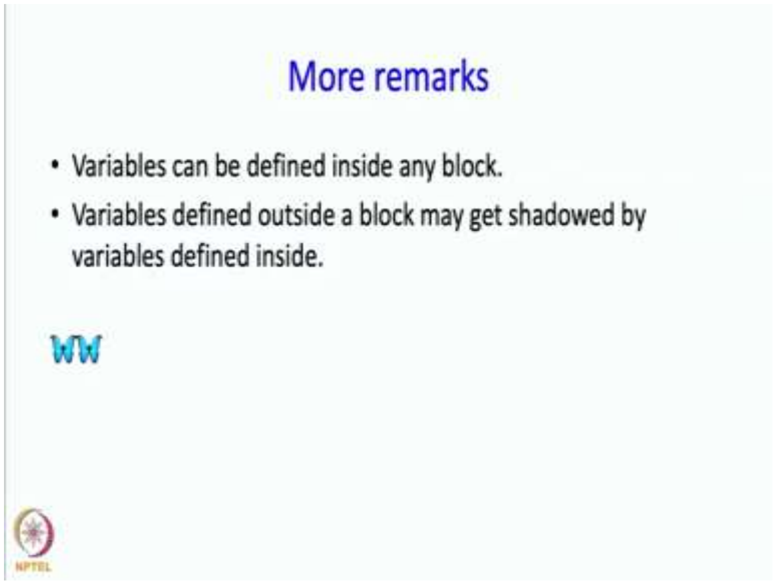


More remarks

- Expressions in C++ are similar to those in mathematics, except that values may get converted from integer to real or vice versa and truncation might happen.
- Truncation may also happen when values get stored into a variable.
- Sequence generation and accumulation are very common idioms.
- Increment/decrement operators and compound assignment operators also are commonly used.

So more remark, expression in C++ are similar in those in mathematics, except that value may get converted from integer to real or vice versa and truncation might happen. Truncation may also happen when values get stored into a variable. And sequence generation and accumulation are very common idioms. Increment, decrement operators and compound assignment operators also are commonly used.

(Refer Slide Time: 12:01)



More remarks

- Variables can be defined inside any block.
- Variables defined outside a block may get shadowed by variables defined inside.

And then we said variables can be defined inside any block. Variables defined outside a block may get shadowed by variables defined inside. And this is, these are basically the main ideas covered in this lecture. And at this point we have seen some rather interesting programs that you could write and indeed by now you have to the point at which you can write several quite interesting programs and we will see these programs in the next lecture. Thank you.