# An Introduction to Programming through C++

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

**Lecture No. 6 Part – 5**
**Conditional Execution**
**Switch statement and logical data**

(Refer Time Slide: 0:34)

## What we discussed

- A detailed example

Next: The switch statement and logical data.

Hello welcome back in the last segment we did a detailed example somewhat a complete program using IF statements. Now, we are going to discuss the Switch Statement and Logical Data.

(Refer Time Slide: 0:48)

## The switch statement

**Execution**

The expression is evaluated.

The resulting value is compared with constant_1, constant_2,....

If some constant_i is found equal:

- then all statements starting with group(i) statements are executed till the end of the switch statement. If a break statement is found, then execution stops.
- If any group of statements does not contain a break then the next group is executed.

If no constant_i is found equal to expression:

- then the default-group of statements is executed.

**General form:**

```
switch (expression){
  case constant_1:
     group(1) of statements usually
     ending with ``break;''
  case constant_2:
     group(2) of statements usually
     ending with ``break;''
  ...
  default:
     default-group of statements
}
```

- The expression and constants must be integers.

So first the Switch Statement. So this is a little bit complicated and so will have bear with me as I describe it. So I will start by describing the general form. So it looks like this Switch followed by an expression then there is the keyword case and then there is constant 1 so here you are expected to have a number like 5, 7, 1, 2, 3 and then you are going to have a group of statements. Okay, usually these statements will end with a break. Then there is a another constant, another group of statements and so on. And then there will be a key word default and there will be another group of statements which will be called default group of statements. So that is what the Switch Statement looks like in general. And expressions and constants are required to be integers. How does this execute? So first the expression is evaluated, so after the previous statement is executed when we want to execute the Switch Statement first the expression is evaluated. Now, this value is compared with constant 1, constant 2 and so on. If some constant i is found equal say may be constant 2 is found equal and constant 1 is not equal. So the first constant that is found equal let us say that is constant i or in this case constant 2, then all statements starting group i are executed. So from here everything is executed okay. If a break statement is found, so we said that it usually ends with a break is found then we deem that this switch statement execution is over and the control goes on to the next statement following the switch. If there is no break statement then it executes the next group of statements and the next group of statements and so on okay until a break statement appears or you execute the last statement in whatever group. So that is what the switch statement looks like. If no constant i is found equal to this expression then the default group of statements will execute, alright?

(Refer Time Slide: 3:24)

Turtle controller using sw

```
main_program{
  char command;
  turtleSim();

  repeat(100){
    cin >> command;
    switch(command){
      case 'f': forward(100);
                break;
      case 'r': right(90);
                break;
      case 'l': left(90);
                break;
      default: cout << "Not a proper command, " << command << endl;
    }
  }
}
```

Okay, so this statement looks like it is designed for our turtle controller. So remember what was happening in the turtle controller? We were reading in a command and command is a character but you know that a character is actually an integer. A character is a numeric type. So a character only when you print it gets printed behaves like a character. But for calculation purposes it is actually an integer. So this is a perfectly fine this is a perfectly fine numeric type as we required it to be. So this is we wanted this switch command we wanted an integer to come out of this and the command being a character, its value is in fact an integer. If the value of this integer is equal to ‘f’ the ASCII value of F then forward(100) will be executed, but after forward(100) is executed because there is a break then the control will go on to the next statement following this. So it will go on to the next part, next repeat . If command was ‘r’ then this will be executed, if command was ‘l’ then this will be executed okay. And if none of these were present then the default this will be executed. So notice that this looks quite nice.

(Refer Time Slide: 5:00)

## Remarks

- Statement is error-prone, because you may forget to write break.

On the other hand there is a slight problem, what is the problem? You may forget to write the break, okay so go back here.

(Refer Time Slide: 5:06)

## Turtle controller using switch

```
main_program{
  char command;
  turtleSim();

  repeat(100){
    cin >> command;
    switch(command){
      case 'f': forward(100);
                break;
      case 'r': right(90);
                break;
      case 'l': left(90);
                break;
      default: cout << "Not a proper command, " << command << endl;
    }
  }
}
```

It is very it is very likely that you may just forget it if you just forget say this break, what happens? You execute this and you also execute this, so that is not such a good idea, okay.

(Refer Time Slide: 5:20)

## Remarks

- Statement is error-prone, because you may forget to write `break`.
- Book gives examples where leaving out `break` is actually useful.

So this statement is actually such as that you do not use this statement, instead just use the if then else, okay. So the statement looks elegant because of the way it is designed because of this break part it is actually a little bit dangerous to use because you may forget the break. And you may say that look sometimes it might be necessary to use to execute several groups and the book indeed gives you an example just for completeness, but really my recommendation is do not use the switch statement. And in fact in this course I will not be asking you anything about the switch statement, I just described this because for the reasons of completeness if you read real code, code written by somebody else, you should not get surprise when you see this switch statement.

(Refer Time Slide: 6:20)

## Logical Data

- We have seen that we can "evaluate" conditions, combine conditions.
- Why not allow storing the results (true or false) of such computations?
- Indeed, C++ has data type `bool` into which values of conditions can be stored.
- `bool` is named after George Bool, who formalized the manipulation of conditions/logical data.

Okay, so the next topic I want to cover in this segment is Logical Data. So we have seen that we can evaluate conditions and we can also combine together conditions, so we can write something like if x greater than 5 and y less than 6. So we are really performing, we are really doing operations on results on conditions as well okay. So then you might ask why should we not allow storing the results true or false of such computations. So we evaluate a condition it came out to be true or false so let us say I want to remembers that that condition came out to be true or false. So can I remember it can it put it in some variable? Indeed I can, so C++ has a data type bool into which values of conditions can be stored. Bool is named after scientist George Bool who formulised the manipulations of conditions and logical data. In fact he invented something called a Boolean Algebra. So we are not going to see Boolean Algebra as such, but we will discuss this bool data type next.

(Refer Time Slide: 7:44)

## The data type bool

```
bool highincome, lowincome;
```

- Defines variables highincome and lowincome of type bool.

```
highincome = (income > 800000);
bool fun = true;
```

- Will set highincome to true if the variable income contains value larger than 800000.
- true and false : boolean constants.
- boolean variables which have a value can be used wherever "conditions" are expected, e.g.

```
if(highincome) tax = …
```

So here is how we can declare a bool type of data. So I can write bool high income or low income. Of course these are going to be standard identifiers with the usual rules for identifiers okay. I can assign to this okay and I can write something like high income equals and I can put down a condition over here. So this condition will get evaluated and depending upon the value of the condition either a true or a false will go in over here. I can also write something like declare a Boolean variable fun and simultaneously initialize it to true if I want. If this true is a constant or a key word. So the first statement will set high income to true if there is a variable income contains value larger than 800000 and the second statement will just set fun to true and true and false are Boolean constants. And what is the use of these variables? Well Boolean variables can be used wherever conditions are expected. So I can write something like if(highincome), tax equal to so and so. Of course before this it is expected that I will have set high income to true or false by writing a statement of this kind. Okay, so now I want to give you and exercise which is write a program to test if a given number n is prime. So we are going to do this and let us see how we would do it. So first of all we would do this, so we would ask look how we do this manually?

(Refer Time Slide: 9:38)

## Exercise: write a program to test if a given number n is prime.

- How will you do this manually?
  - Eratosthenes' sieve
  - We are required to "remember" all the primes determined till n.
  - So far we have no way of doing this
- Can we do something less efficient, but without requiring us to remember too many things?
  - Check if any of the numbers from 2 to n-1 divide n.

So there is an algorithm that you may or may not remember or may or may not know called the Sieve Of Eratosthenes. So, how does that work? Well very quickly we start with 2 and we keep on checking whether a certain number a certain number divides n okay and this we make it a little bit more efficient by saying look if we want to check whether a given number i divides n we first have to check whether it is a prime number itself or not.

So that is the Sieve of Eratosthenes, we are not going to follow that method so it does not matter if you did not understand what Eratosthenes' Sieve is okay. We are going to do something slightly different, we are going to do which is something slightly less efficient, but it will do things without requiring us to remember too many things okay. Here is what we will do, we will start with 2 and go all the way till n minus 1 and check if any of these numbers divides n.

If one of these numbers divide n, then clearly n is composite. If none of these numbers divides n, then n is a prime okay so that is going to be our algorithm. So start with 2 as a candidate divisor and check if it divides n, otherwise go on to the next candidate, next candidate until n minus 1 and check if any of those candidate divisors divides n okay. And if we find any such numbers we will we will conclude that in fact that number is composite, otherwise that number will be prime.

(Refer Time Slide: 11:44)

## Solution

```
#include <simplecpp>

main_program{
  int n, divisor=2; cin >> n;
  bool divisorFound = false;  // no divisor found for n so far
  // check if divisor divides n as it varies from 2 to n-1
  // if divisor divides n, set divisorFound = true
  repeat(n-2){
    if(n % divisor == 0) divisorFound = true;
    divisor = divisor + 1;
  }
  if(!divisorFound) cout <<"Prime.\n";
  else cout <<"Composite.\n";
}
```

Okay so let us write the program we will include simplecpp and will have the main program and n is the number that we want that we want to decide whether it is prime or composite and divisor is our candidate divisor which we are going to start of at 2. So the first thing that we do is, we read into read n so we expect the user to type in whatever number he or she wants to check whether it is prime or not. Then we are going to have a Boolean variable okay so remember we said we are going to go over a candidate divisor starting at 2 and this is our first candidate divisor over here. So I should have perhaps used a bigger name candidate divisor but that is what is mean okay. So this is our candidate divisor and we are going to start with this and go up till n minus 1. And if ever I find that candidate divisor divides n, then I will make divisor found to be true. So, so far I have not found any divisor for n okay so I will make it false, I will set it to be as false. So we will check if divisor divides n as it varies from 2 to n minus 1 okay and then thereby set divisor found. So if the divisor divides n then said divisor found equal to true okay and how do we do this? Well we have to check for all number between 2 and n minus one so there are n minus 2 such numbers. So we are going to have a repeat loop of n minus 2 steps n minus 2 iterations and we are going to check if n mod divisor is 0 or in other words is n perfectly divided by divisor. If it is, then we have found a divisor and so we will say divisor found to be true. And then we want to check the next divisor and therefore, we will set divisor equal to divisor plus 1 and that is it okay. After this if divisor found is true then we know that the number is composite, so we will say if not divisor found then cout prime. We could have written if divisor found then print composite,

but we just wrote it in a different order, else we are going to print that this number is composite okay that is it.

So, what has happened over here? We have used 'divisorfound' a Boolean variable okay to accumulate whether or not we have found a divisor as we go through the loop. Okay so this is one use we can put Boolean variables to. So we are remembering the previous whether or not divisors were found previously and that is a Boolean value whether a divisor was found is either true or false. So that information can be accumulated, can be put in Boolean variable and so we have given it an appropriate name and inside that we are putting this information and at the end if divisor found is false that is we did not find any divisor at all then we are going to print out prime otherwise we will print out composite.

(Refer Time Slide: 15:27)

## Exercise

Execute the program mentally for n = 100.

- What answer does it produce?
- Are you happy with how the program executes?

So I would like you to execute this program mentally for n equal to n equal to 100 and I would like you to verify that it produces the correct answer, but I would also like you to think about whether you think the program is a good program or whether it is perhaps doing too much work. And if it is and you will indeed see that it is doing unnecessary work we will find ways of remedying that a little bit later in the course.

(Refer Time Slide: 16:09)

## What we discussed

- Switch statement: Recommend do not use.
- Logical Data: Can remember results of condition evaluation

Next: Lecture summary

Alright, so what have we discussed? We have discussed the switch statement and we have recommended that it should not be used. We discussed logical data and said that it can be used to remember results of conditional evaluation. In the next brief segment, we are just going to summarize this lecture sequence, but before that let us take a short break.

(Refer Time Slide: 17:05)

## Summary

- Conditional execution makes life interesting.
- 3 forms of if.
  - You can nest if statements inside each other: some pitfalls in this are discussed in the book.
- Compound conditions
- Logical data

Try the exercises at the end of the book.

Welcome back in the last segment we discussed the switch statement and logical data in this segment we are nearly going to summarize the lecture. So first of all I should observe that conditional execution makes life interesting, we noted at the beginning that the tax calculation program cannot be solved, cannot be written without something like conditional

execution. Something like an if statement. Then we discussed 3 forms of if, plain if, if condition consequent, if then else, if condition, consequent else alternate and then the most general with several consequents and a single alternate. And I should point out that you can nest if statements inside each other okay so the consequent can be a block and that block can contain if statements. But there are some pitfalls in it and those pitfalls are discussed in the book so please read the book. We also looked at compound conditions so conditions which are made by saying look I want this condition to be true and this condition to be true or may be this condition to be true or this condition to be true or this condition should be false. Then we also talked about logical data we did talk about the switch statement but I am not mentioning it over here because I really do not want you to use it. And before I finish this lecture as usual I will ask you to try the exercise not at the end of the book but at the end of the chapter. So please do that that is absolutely necessary and I hope you will like those exercises, so that is it that is it for this lecture sequence. Thank you.