

Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture No. 02
Recap of C/1

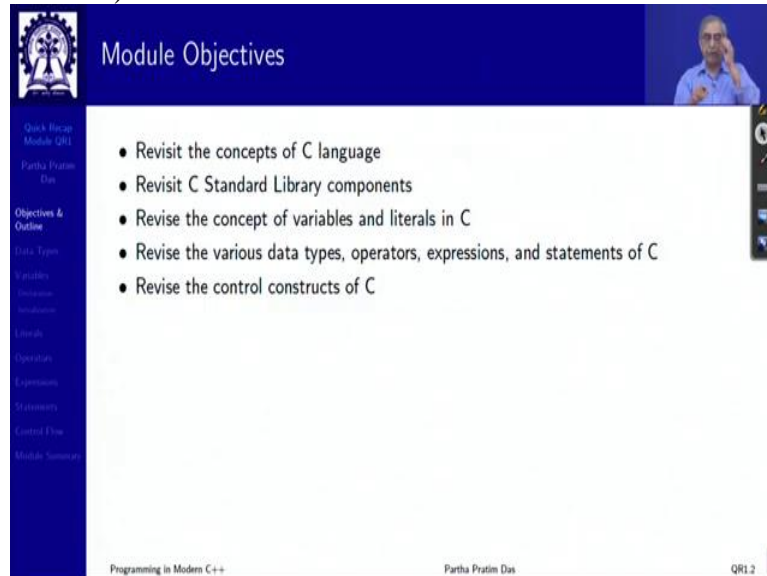
Welcome to Programming in Modern C++. We are going to discuss some preliminary content, which will be useful for you. So, we are sharing this as a week 0 content. So, I mean week 0 is not where the course formerly has started, but you are seeing this video to get prepared rightly for the course. So, there will be couple of videos on the week 0 which will enhance your, requirement of, I mean meeting the requirements for the prerequisites.

So, the first step that I will discuss is quick recap module, is called a quick recap or QR module where I will run through the C language in brief. Now, you might think as to why are we discussing C language, while you have come here to learn programming in modern C++. And what we expect is you are already familiar with the C language and preferably, you have already done quite a, quite some programming in C.

So, if you have done that, this module is not for you, you can just may not waste your time, just skip it. But if you have some, some brown areas, gray areas that well, what happens here in C and so on, then it is good to go through this module, this will at least tell you what are the things that you must be aware and good at when you come to the first week lecture and start discussing C++.

Because in the entire of this course, I am going to assume that you know C, I am going to make innumerable references to C because C++ has built on top of that, it has the legacy of that it has the power of that, it has the compatibility issues with that, it has mixing issues with that. So, knowing C is very, very important.

(Refer Slide Time: 02:38)

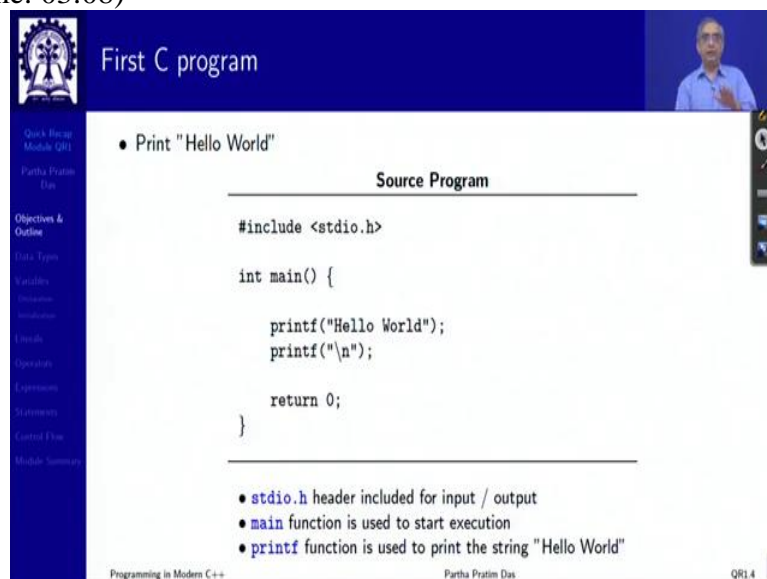


The slide is titled "Module Objectives" and features a list of five bullet points. On the left side, there is a vertical navigation menu with items like "Quick Recap", "Module QR1", "Partha Pratim Das", "Objectives & Outline", "Data Types", "Variables", "Expressions", "Literals", "Operators", "Expressions", "Statements", "Control Flow", and "Module Summary". At the bottom, it says "Programming in Modern C++", "Partha Pratim Das", and "QR1.2".

- Revisit the concepts of C language
- Revisit C Standard Library components
- Revise the concept of variables and literals in C
- Revise the various data types, operators, expressions, and statements of C
- Revise the control constructs of C

So, quickly, we will revisit some of the concepts. And just I mean, this module also is I mean, if I want to talk about C as a language, it will take probably 40 modules, 50 modules, this is not that. This is more like pointers, that these are the things just check you know, if you have issues, go back, check the text, check the C books and so on.

(Refer Slide Time: 03:08)



The slide is titled "First C program" and shows a code block for a "Hello World" program. Below the code, there are three bullet points explaining the components. On the left side, there is a vertical navigation menu with items like "Quick Recap", "Module QR1", "Partha Pratim Das", "Objectives & Outline", "Data Types", "Variables", "Expressions", "Literals", "Operators", "Expressions", "Statements", "Control Flow", and "Module Summary". At the bottom, it says "Programming in Modern C++", "Partha Pratim Das", and "QR1.4".

- Print "Hello World"

```
Source Program
#include <stdio.h>

int main() {
    printf("Hello World");
    printf("\n");

    return 0;
}
```

- `stdio.h` header included for input / output
- `main` function is used to start execution
- `printf` function is used to print the string "Hello World"

So, this is everybody starts here, the Hello World program, which includes `stdio.h` for being able to print something and it is included as an input for input and output. And you must have a main function, which by the current standard must return `int`, do not write main function with returning `void` that, that is not, that convention is over, that is become dated now.

And then you return 0 at the end, return 0 means that whatever your program had to do, it has done successfully. Otherwise, we will see later on that you will return minus 1, minus 2

different other values, typically negative values which tell that you have some error. So, when you, you will see me during the different modules through the course you will, many times you will see that I have not, I am not writing return 0 that is just to keep the program text short. But return 0 should always be written, even if you do not write return 0, main is a special function where the compiler ensures that unless you are returning anything else, it will return a 0.

(Refer Slide Time: 04:35)

Data Types

Data types in C are used for declaring variables and deciding on storage and computations

- **Built-in / Basic** data types are used to define raw data
 - `char`
 - `int`
 - `float`
 - `double`

Additionally, C89 defines:

- `_Bool`

All data items of a given type has the same size (in bytes). The size is *implementation-defined*

- **Enumerated Type** data are internally of `int` type and operates on a select subset.

Programming in Modern C++ Partha Pratim Das QR1.5

So, you know there are different data types it is very important to understand the data types of the language. The basic ones are char, int, double and float. There are a lot of variations of that, which are, which are called integral types like char, int. These are integral types so is enum, and there are floating types or floating-point types which are float and double.

Additionally, C89 standard, if you are not familiar with the standards in C, do not worry, I am going to talk about the different standards of the C language. Because the language, C language itself is also changing. We will wonder that it is changing, almost as fast as C++ is changing.

So, if you are using something, some C language which is, which is expected that you are using something which is C89 Or later C95 Or C11. Then you have another type which is defined as underscore bool in a particular header, you will, you will hear me discussing this in the first week of the course. So, enumerated type is also another subset.

(Refer Slide Time: 05:54)

Data Types

Data types in C further include:

- **void**: The type specifier `void` indicates *no type*
- **Derived** data types include:
 - *Array*
 - *Structure* – `struct` & `union`
 - *Pointer*
 - *Function*
 - *String* – C-Strings are really not a type; but can be made to behave as such using functions from `<string.h>` in standard library
- **Type modifiers** include:
 - `short`
 - `long`
 - `signed`
 - `unsigned`

Programming in Modern C++ | Partha Pratim Das | QRI.6

Besides this, you have a special type known as void. Understand void very well, void is not a type, but it can be placed at different places where a type is expected but you do not know what the type is. But not everywhere, like you cannot have a void as a function, a function parameter has a type void and passing. The most two most common use of void in C and subsequently in C++ also in C is.

One is when you are when a function does not have anything to return, it just returns a control but no value to return, you put void as a return type. This is one use which is which is very important. The second use is using a pointer to void, `void*`. So, whenever you are, you have a pointer to a memory location, but you do not know what kind of data type is residing there, what data of which kind of data type is residing in that memory you call that pointer as `void*` and it can be then cast to any other `int*`, `float*`, `double*` this kind of.

Or many of the common library functions like `malloc` uses the `void*` pointer as a return. So, these are the two primary usages, there are a few more shades, which I will discuss when you actually go through the course. Naturally, there are different derived types, all of which you must be good at, array, structure which includes `struct` and `union`, pointer, function and string is not exactly a type.

But with the `string.h` header, you can deal with a string as if it is a type but it is not an, I mean do not get me wrong, it is not a part of the language. And therefore, it is not a data type in that way, C does not allow you to define data types of your own. Then there are certain type modifiers, which say I have `int`, then I say I have `short int`, so it is a modifier. So, what we are saying that it will behave like the `int`, except that it is shorter than `int`.

Maybe my int by default is four bytes. Short int will be say two bytes. Your, your system manual will define that there is no, it is not defined in the C standard. Short int may also be abbreviated or written in short as just short. If we just say short x, it means short index. Similarly, you can do the other thing, you can widen a data type up to the extent supported that is by using long. And you can treat a data type as signed or unsigned, whether it is int or it is particularly used with int and char.

(Refer Slide Time: 08:50)

Variables

- A **variable** is a name given to a *storage area*
- **Declaration of Variables**
 - Each *variable* in C has a *specific type*, which determines the size and layout of the storage (memory) for the variable
 - The *name of a variable* can be composed of *letters, digits, and the underscore character*. It *must begin* with either a *letter* or an *underscore*

```
int    i, noOfData;
char   c, endOfSession;
float  f, velocity;
double d, dist_in_light_years;
unsigned int i, nPeople;
short int i, nCount;
unsigned char c, ascii_char;
int    a[10]; ;
```

Programming in Modern C++ Partha Pratim Das QR1.7

You know the variables, the name given to a storage area. Every variable will need in C as well as in C++, every variable needs a declaration. So, when you put the data type and the name of the variable and that must happen, that statement, declaration statement must be there before any use of that variable is possible. So, that is a declaration which is must for a variable.

You cannot use variables without declaration in C or C++. Variable names as can have alpha, all alphabetic characters, lowercase, uppercase, and all ten numeric digits. It can also use underscore. But it must start with an alphabet or a letter or the underscore character, it cannot start with a numerical. Like 10i is not a valid variable. You have done all this, just to you know recap into.

(Refer Slide Time: 09:56)

The image shows two identical screenshots of a presentation slide titled "Variables". The slide is part of a course by Partha Pratim Das, covering topics like Quick Recap, Module Q01, and various programming concepts. The slide content is as follows:

- Initialization of Variables**
 - Initialization is setting an initial value to a variable at its declaration
 - C variables declared can be initialized with the help of operator '='
 - Multiple variables can be initialized in a single statement by single value

```
int    i = 10, j = 20, numberOfWorkDays = 22;
char   c = 'x';
float  weight = 4.5;
double density = 0.0;
const int nElements = 100; // const must always be initialized
char*  name[] = {"Partha", "Pratim", "Das"}; // Array size is 3
```

- Definition of Variables**
 - A variable is defined when a value is written to it using
 - assignment operator '='
 - pointer aliasing

```
int i = 10; // Array size is 3
int* p = &i; // Address of i set to p
i = 20; // Assignment
*p = 30; // Pointer aliasing
```

The second screenshot is identical to the first but includes handwritten annotations: a checkmark next to the assignment operator '=' in the definition list, and curly braces grouping the pointer aliasing examples in the code block.

Then variables, once you have the variables declared, you can also initialize the variable. Initializing it is while you are declaring, at that point itself, you can set what is the value it should start with. So, these are different examples of initialization, it looks like an assignment, but it is not an assignment. Because you are just defining this, what is an assignment?

Assignment is when you have a value and you are putting another value into that. But here, the variable does not exist till it is declared. So, you are saying that I am declaring density as a double type variable to start with the value of 0.0. That is the, there is the initialization. And then you can define different values for the variable by doing assignment operator or by analyzing pointers and so on you, you must have had used this. So, these were more, these were initializations were these are assignments that you get. So, know your variables and their declaration, initialization and definition and assignment will.

(Refer Slide Time: 11:08)

The slide is titled "Literals" and features a blue header with a logo on the left and a small video feed of a speaker on the right. The main content is a list of bullet points and code examples. The first bullet point states that literals refer to fixed values of a built-in type. The second bullet point states that literals can be of any of the basic data types. Below these are two lists of code examples, each with a comment explaining the literal's type. The first list shows literals for int, double, char, and char*. The second list shows the same literals with the (const) keyword, indicating they are constant values. A sidebar on the left contains a navigation menu with items like "Check Back", "Module QRI", "Partha Pratim Das", "Objectives & Prerequisites", "Data Types", "Operators", "Expressions", "Statements", "Control Flow", and "Module Summary". The footer of the slide includes "Programming in Modern C++", "Partha Pratim Das", and "QRI.9".

- *Literals* refer to *fixed values* of a *built-in type*
- *Literals* can be of any of the basic data types

```
212 // (int) Decimal literal
0173 // (int) Octal literal
0b1010 // (int) Binary literal
0xF2 // (int) Hexadecimal literal
3.14 // (double) Floating-point literal
'x' // (char) Character literal
"Hello" // (char *) String literal
```

- In C*9, literals are constant values having *const* types as:

```
212 // (const int) Decimal literal
0173 // (const int) Octal literal
0b1010 // (const int) Binary literal
0xF2 // (const int) Hexadecimal literal
3.14 // (const double) Floating-point literal
'x' // (const char) Character literal
"Hello" // (const char *) String literal
```

Note that, every data type that you have in the language. We call them the built-in type has certain format in which the constants of that type can be written, these are called literals. So, if I write 212, it means a decimal literal. We are writing it in the base 10 number system, it is an integer in the base 10 number system. So, it is int and the decimal literal. If I put a 0 in front, you will think okay, it how does it matter?

It does not matter? No, it does matter in C, if you put a 0 in front, it is not taken as 173 in decimal, it is taken as the number 173 in octal literals. Where the base is basically 8. So, it is 3 into 8 to the power 0 plus 7 into 8 to the power 1 plus 1 into 8 to the power 2. Similarly, you can have a binary literal by starting with 0b and then writing the 10 digits.

You can have a hexadecimal, that is based 12 literals, written with starting with 0x so no. And remaining are, I am, I am sure you are more familiar with how to write a floating-point literal, a character and a string. Now C89 onwards, you have something called, a something called values being constant, it is called constant. So, these are literals are always of the constant type.

When we when I talk about constants in the course, you will be able to relate to that. But just if you are not familiar with constant C, because it is a later addition and maybe not so, frequently used in C. Then you may want to remember, that all literals in C and C++ are constants, that is they cannot be changed, literal has a value and that is.

(Refer Slide Time: 13:35)

Operators

- An **operator** denotes a *specific operation*. C has the following types of operators:
 - **Arithmetic Operators:** + - * / % ++ --
 - **Relational Operators:** == != > < >= <= ✓
 - **Logical Operators:** && || ! ✓
 - **Bit-wise Operators:** & | ~ << >> ✓
 - **Assignment Operators:** = += -= *= /= ... ✓
 - **Miscellaneous Operators:** . sizeof & * ?: ✓
- **Arity of Operators:** Number of operand(s) for an operator
 - +, -, *, & operators can be *unary* (1 operand) or *binary* (2 operands)
 - ==, !=, >, <, >=, <=, &&, ||, +=, -=, *=, /=, &, |, <<, >> can work only as *binary* (2 operands) operators
 - sizeof ! ~ ++ -- can work only as *unary* (1 operand) operators
 - ?: works as *ternary* (3 operands) operator. The condition is the first operand and the if true logic and if false logic corresponds to the other two operands.

Programming in Modern C++ Partha Pratim Das QRI.10

Naturally there are several operators. And most of the operators are binary, some are unary and there is one which is ternary. And these operators are for example, these are as you know these are the most common is a binary. These are unary and occurs in two versions, prefix version and postfix version. These are comparison operators; these are logical operators and or and the third one is unary which is negation.

These operations are bitwise and in two-bit patterns or whoring two-bit patterns or negative a bit pattern or shifting a bit pattern left or right. Varieties of assignment operators, other operators like sizeof, address of dereference, question mark colon the ternary operator and so on. So, be very clear about the arity of the operator and depending on the context some operators have different arity, though they look the same.

For example, star occurring between x and y, two integer variables will be treated as a multiplication. But if x is of type integer, or p is of type integer, let us say then star p just occurring before that is a dereferencing operator, which is going to take the, take the address that p carries and go there and find out, take the value. So, I mean for several please, remind yourself that the context really is important.

Like plus is binary for addition and plus is unary if you just want to say this a positive value. By default, every value is every integer or floating-point value is positive, but you can still write plus 5 to mean that it is, I really mean plus 5. Similarly, four minus, ampersand also has that it is, it is a bitwise ending operator as well as it is an address of operator and so on. So, here I have kind of summarize very quickly and in a very short space as to what the arity of

the operators are including, that some of them are used in more than one way. And you must be very clear and familiar with that.

(Refer Slide Time: 16:07)

Operators

*2 + 3 * 4*

- **Operator Precedence:** Determines which operator will be performed first in a chain of different operators
 - The precedence of all operators are defined in the following order: (left to right - Highest to lowest precedence)
 - `()`, `[]`, `++`, `--`, `+` (unary), `-` (unary), `!`, `~`, `*`, `&`, `sizeof`, `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `==`, `!=`, `*`, `=`, `/`, `&`, `|`, `&&`, `||`, `?:`, `=`, `+=`, `-=`, `*=`, `/=`, `<<=`, `>>=`
- **Operator Associativity** Indicates in what order operators of equal precedence in an expression are applied
- Consider the expression $a @ b @ c$. If the operator `@` has left associativity, this expression would be interpreted as $(a @ b) @ c$. If the operator has right associativity, the expression would be interpreted as $a @ (b @ c)$
 - *Right-to-Left:* `?:`, `=`, `+=`, `-=`, `*=`, `/=`, `<<=`, `>>=`, `-`, `+-`, `!`, `~`, `*`, `&`, `sizeof`
 - *Left-to-Right:* `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `==`, `!=`, `*`, `=`, `/=`, `&`, `|`, `&&`, `||`

Programming in Modern C++ Partha Pratim Das QRI.11

Operators

*2 + 3 * 4*

- **Operator Precedence:** Determines which operator will be performed first in a chain of different operators
 - The precedence of all operators are defined in the following order: (left to right - Highest to lowest precedence)
 - `()`, `[]`, `++`, `--`, `+` (unary), `-` (unary), `!`, `~`, `*`, `&`, `sizeof`, `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `==`, `!=`, `*`, `=`, `/=`, `&`, `|`, `&&`, `||`, `?:`, `=`, `+=`, `-=`, `*=`, `/=`, `<<=`, `>>=`
- **Operator Associativity** Indicates in what order operators of equal precedence in an expression are applied
- Consider the expression $a @ b @ c$. If the operator `@` has left associativity, this expression would be interpreted as $(a @ b) @ c$. If the operator has right associativity, the expression would be interpreted as $a @ (b @ c)$
 - *Right-to-Left:* `?:`, `=`, `+=`, `-=`, `*=`, `/=`, `<<=`, `>>=`, `-`, `+-`, `!`, `~`, `*`, `&`, `sizeof`
 - *Left-to-Right:* `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `==`, `!=`, `*`, `=`, `/=`, `&`, `|`, `&&`, `||`

Programming in Modern C++ Partha Pratim Das QRI.11

Operators

- **Operator Precedence:** Determines which operator will be performed first in a chain of different operators
 - The precedence of all operators are defined in the following order: (left to right - Highest to lowest precedence)
 - `()`, `[]`, `++`, `--`, `+` (unary), `-` (unary), `!`, `~`, `*`, `&`, `sizeof`, `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `==`, `!=`, `*`, `=`, `/`, `=`, `&`, `|`, `&&`, `||`, `?:`, `=`, `+`, `-`, `*`, `/`, `<<=`, `>>=`
- **Operator Associativity** Indicates in what order operators of equal precedence in an expression are applied
 - $a = b = c$ (handwritten)
 - $2 + 3 + 5$ (handwritten)
 - $2 - 3 - 5$ (handwritten)
- Consider the expression $a @ b @ c$. If the operator `@` has left associativity, this expression would be interpreted as $(a @ b) @ c$. If the operator has right associativity, the expression would be interpreted as $a @ (b @ c)$
 - **Right-to-Left:** `?:`, `=`, `+`, `-`, `*`, `/`, `<<=`, `>>=`, `-`, `+-`, `!`, `~`, `*`, `&`, `sizeof`
 - **Left-to-Right:** `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `==`, `!=`, `*`, `=`, `/`, `=`, `&`, `|`, `&&`, `||`

Programming in Modern C++ | Parth Pratim Das | QR1.11

So, now along with the operators come the notion of precedence. That is if I write say x plus y into z or let me make it, let me make it simpler. Say if I write 2 plus 3 into 4, you know that even though I go through the expression from left to right, 3 into 4 will be done first into 12. And then 2 will be, it will be added to 2. It is not like you will add 2 plus 3 to make it 5 and multiply it with by 4.

Because this has a higher precedence in school itself, we learned about BODMAS rule and so on. So, over the entire range of operators, that precedence you will have to know. So, the list is given but do not get, do not stay scared. I will, I always follow a very, conservative principle that, some of course, I know like BODMAS rule, nobody can forget. Or the fact that in an expression if there is a function called that is going to happen prior to anything else.

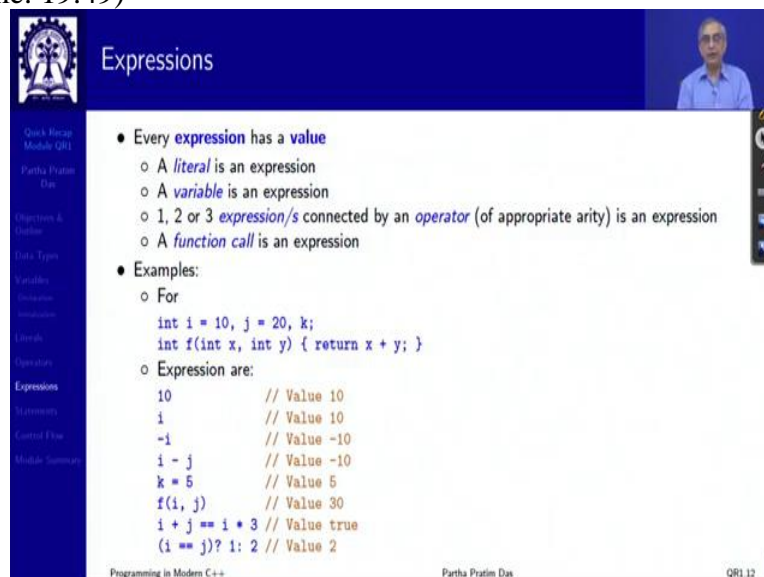
These who cannot forget, but there is the list is long. So, if you are confused about what is the precedence, it is better to use parenthesis, to mean exactly what you meant. So, use this safety in C as well as in C++. Now, operators do have associativity, that is if there are multiple occurrences of the operator in this, in the same expression, $2 + 3 + 5$, then the question obviously is that which one will be done first.

In plus it does not matter. But the moment I write it as minus, it matters as to whether I do $2 - 3$ first or $3 - 5$ first. If I if we do the left one first, we say it is the left associative operator. If we do the right one first, we say it is a right associative operator. And say in terms of right associativity in usual, algebra or arithmetic that we learned, we get more often the left associative operators. Because that is, that is algebraic.

And right associative operators are primarily as we will see are computational. For example, if there are three variables a b and c of type integer, in C I can write this. So, in C++, so, what does it mean that I take the value of c put it to b, and the value that have put in I take that again and put it to a. Now, naturally this cannot start happening from left, it has no meaning, it has to happen from right.

So, assignment operator is right associative. So, if assignment operator is right associative, then naturally its different variants like +=, -=, they are also right associative there are a few others which are right associative as well. So, remember the associativity. And again, if you are confused, if you do not, remember, use parenthesis to make it clear as to what you are saying. But the native behavior of the operators will follow the precedence and associativity along with its arity from the context.

(Refer Slide Time: 19:49)



Expressions

- Every expression has a value
 - A *literal* is an expression
 - A *variable* is an expression
 - 1, 2 or 3 *expression/s* connected by an *operator* (of appropriate arity) is an expression
 - A *function call* is an expression
- Examples:
 - For

```
int i = 10, j = 20, k;  
int f(int x, int y) { return x + y; }
```
 - Expression are:

10	// Value 10
i	// Value 10
-i	// Value -10
i - j	// Value -10
k = 5	// Value 5
f(i, j)	// Value 30
i + j == i * 3	// Value true
(i == j)? 1: 2	// Value 2

Programming in Modern C++ Partha Pratim Das QRI.12

An expression is what we need in the computation and expression has a value. This is what you must remember always. The difference between you, I mean I often come across programmers, when students were confused between what is an expression and what is a statement. There is no confusion, an expression must have a value. That is the basic thing. So, a literal or a constant is an expression, a variable is an expression.

And any one, two or three expressions, connected by operators we have just discussed, is an expression. Because each one of them will have a value, a function call is an expression, if that function is returning some value. If it is a function, its return type is void, then the function call is not an expression. Because it does not represent a value. And, of course, there are near list of examples of expressions. I will not go through each one of them. I am sure

you will know that, I just wanted to highlight the fact that you understand always get it deep into your mind that whenever we talk about expression, it means it must have a value.

(Refer Slide Time: 20:55)

Statement

- A **statement** is a command for a specific action. It has *no value*
 - A ; (semicolon) is a (null) statement
 - An *expression terminated by a ; (semicolon)* is a statement
 - A list of *one or more statements* enclosed within a pair of curly braces { and } or block is a *compound statement*
 - *Control constructs* like if, if-else, switch, for, while, do-while, goto, continue, break, return are statements
- Example: **Expression statements**

Expressions	Statements
$i + j$	$i + j;$
$k = i + j$	$k = i + j;$
$\text{funct}(i, j)$	$\text{funct}(i, j);$
$k = \text{funct}(i, j)$	$k = \text{funct}(i, j);$

- Example: **Compound statements**

```
{
    int i = 2, j = 3, t;

    t = i;
    i = j;
    j = t;
}
```

Handwritten annotations on the slide include:
- $b = a = 3;$ with a wavy line above it.
- $a = 3$ written twice.
- A diagram showing a memory box labeled 'a' containing the value 3, with an arrow pointing from the expression $a = 3$ to the box.

In contrast, a statement is an action. It is a specific action, so it has no value. So when, when you keep on taking, I mean literals, variables, operators, expressions, more expressions, keep on building up building up building up, you always have a value. And then if you want to say that, well, I am done with it, you can make it into a statement that now this expression, value I am done with, so make it into an action.

An action might actually have some side effects. For example, I will just give you a simple example. an assign 3 is an expression because assignment is an expression because it has a value. Because unless it had a value, you cannot, you could not have done, you could not have assigned it to b. So, it has a value. But it has a side effect also, the side effect is there is a memory called a.

Where you have, you are putting this value 3. What is it is an expression? an assign 3 gives you a value 3. But the side effect is you are putting this value 3 into the memory location of a, which is an action. So, you can see that, I mean the language is kind of, playing around with this thing. So, it is not only mathematics, mathematically, an assign 3 is as a value 3. But programmatically an assign 3 has a side effect.

So that is action, this side effect is action. So, I will say an assign 3, a is an expression, an assign 3, I would say, I am done with it, that the action. I put a semicolon, and I said this is a statement, I am done with that. b assigned an assigned 3 is an expression, I make it into an

action, because there are two side effects. One is an assign 3 has put 3 in a b is that 3 assigned to b as put 3 in the memory location of b.

So, this is now a statement. So, that is that is the basic difference you must always carry in mind. In addition, there are several statements, which are basically for controlling, the control I mean controlling the flow in the program as to what you do. So, we call them as control constructs. And for convenience of the language expression, we say that if we are doing multiple statements, then we can put them as a block by using curly braces. And think as if it is one statement. This is a, this is a state textual convenience that we, that we use.

(Refer Slide Time: 24:09)

Control Constructs

- These statements control the flow based on conditions:
 - *Selection-statement*: if, if-else, switch
 - *Labeled-statement*: Statements labeled with identifier, case, or default
 - *Iteration-statement*: for, while, do-while
 - *Jump-statement*: goto, continue, break, return
- Examples:

<pre>if (a < b) { int t; t = a; a = b; b = t; }</pre>	<pre>if (x < 5) x = x + 1; else { x = x + 2; --y; }</pre>	<pre>switch (i) { case 1: x = 5; break; case 3: x = 10; default: x = 15; }</pre>
<pre>int sum = 0; for(i = 0; i < 5; ++i) { int j = i * i; sum += j; }</pre>	<pre>while (n) { sum += n; if (sum > 20) break; --n; }</pre>	<pre>int f(int x, int y) { return x * y; }</pre>

Programming in Modern C++ Partha Pratim Das QR1.14

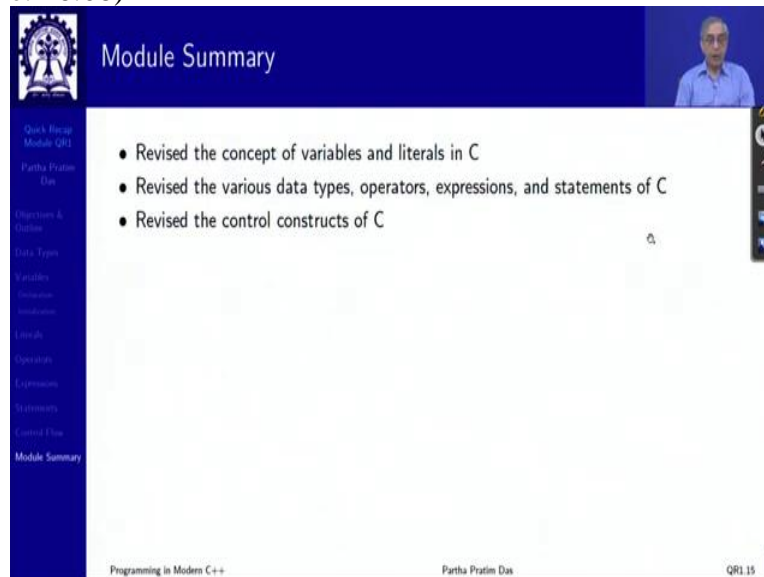
So, there are several control constructs, I will not go through them. Just a reminder, that you have selection statement if, if else and switch three, three different ways. One is, if is one way actually two ways but the else part will just fall through. If else naturally, two-way switch is multi-way. And you have labeled statements, like case default these are labeled statements. You have three iteration constructs in C, carries on to C++ for, while and do while.

You must be aware of all of that. And you have variety of jump statements. One is explicit jump using goto, using a label of, for a for a statement, which as you must be aware is strongly recommended that you do not use. But again, at some point in the course, I will tell you some very rare, but strong context where using a goto is useful. But normally you should keep in your mind that the goto does not exist in your language.

If you want to be a good programmer. Always manage your jumps either by continue or by break and certainly by return when you are coming back from a function. So, this is about the statement. So, there are arithmetic expressions which are converted to statements by

semicolon. So, those are called it arithmetic statements and you have variety of control statements. Besides that, you have statements like declaration statement index and so on. You know, there are some other, management kinds of statements which actually do not compute, but actually give tells, tells enough information to the compiler. So, that the overall variable and memory all these can be managed properly.

(Refer Slide Time: 26:08)



The screenshot shows a video lecture slide titled "Module Summary". The slide content includes a list of bullet points: "Revised the concept of variables and literals in C", "Revised the various data types, operators, expressions, and statements of C", and "Revised the control constructs of C". The slide also features a navigation menu on the left side with items like "Quick Recap", "Module QRI", "Partha Pratim Das", "Objectives & Outcomes", "Data Types", "Variables", "Operators", "Expressions", "Statements", "Control Flow", and "Module Summary". The bottom of the slide displays "Programming in Modern C++", "Partha Pratim Das", and "QRI 15".

So, in this, quick recap, first quick recap module, I have taken you through the revision of the concepts of variables, literals, data types, operators, expressions and statements. And again, I will remind you that it is very important that you understand all of these very well to start making good progress from the beginning of the course. So, if you are, blurry, gray in any of these then do make a good preparation on that, do a recap.

Obviously, when I walked through in the first week of the course, after module one, which is introductory, so in the remaining modules of the first week, I will also show a number of C examples as a reminder, as well as to show that how coming to C++ will keep on improving things. So, thank you very much for your attention. I hope this will assist you get properly ready for the course. And I will have another quick recap module.