

Slides-2

Saurav

# C++ Programming Basics Procedural Aspects

Saurav Samantaray<sup>1</sup>

<sup>1</sup>Department of Mathematics IIT Madras

August 12, 2024



### The Very First C++ Code

Slides-2

Saurav

```
Let the computer greet you.
#include < iostream >
using namespace std;
```

```
// every program has a main
int main()
{
    // print hello world and shift to
    // the next line
    cout << ``Hello World'' << endl;
    return 0;
  }</pre>
```

Save the above into a file "hello.cpp".



### Compiling a C++ Code

#### Slides-2

- ∎ g++ -c hello.cpp.
- This only compiles the code and checks if all the syntaxes make sense or not.
- How do we run this?
- ∎ g++ -o hello.exe hello.cpp
- ./hello.exe.



### Program To Illustrate Basic Features of C++

Slides-2

Saurav

**Task** Write a program that takes in two integers and as input and prints the sum of all integers between them.

- It should be able to take in two integers, lets say "a" and "b".
- It should print the final sum.
- It should have a way to understand a > b or vice-versa.



### Variable Declaration

Slides-2

- int a, b;
  - Explicitly tell the computer which type of variable you want to use.
  - Moreover, computer creates and allocates memory for this.
  - Basic Numerical Variables:
    - int
    - double
  - Operation which can be performed on numerical variables:

∎ a = a + b;	a += b;
∎ a = a - b;	a -= b;
∎ a = a * b;	a *= b;
∎ a = a / b;	a /= b;
∎ a = a % b;	a %= b;
∎ a = a + 1;	a++;
∎ a = a - 1;	a;



# The "if " statement

··· }

Slides-2

if	(a>b)	)				
		{				
		cout	<<''sinc	e a > b w between	e need to them'';	swap
1	It is us Contro	sed to co	ontrol the floons are:	w of the prog	ram.	
	{					
	} e	lse				
	{	100				



Slides-2

```
nested if's;
  if (x > z)
    if (p > q)
      // Both conditions have to be met
      y = 10.0;
multiple if's;
  if (i > 100)
    y = 2.0;
  else if (i < 0)
    y = 10.0
  else
    y = 5.0; \}
```



#### Loops

Slides-2

#### Saurav

# for (int i = a; i <= b; i++) {</pre>

- Executes a collection of statements certain number of times.
- int i = a; this both declares and initialises "i".
- i < = b; checks for the validity until when the loop has to run.
- i++ increments the loop counter.



### Other loops

Slides-2

```
The while loop:
while (x > 1.0)
{
    x * = 0.5;
}
The do while loop:
    do
    {
    x *= 0.5 ;
} while (x > 1.0)
```



#### Arrays

Slides-2

Saurav

- For a type T, T[n] is the type "one-dimensional array of *n* elements of type T", where *n* is a positive integer.
- the elements are indexed from 0 to *n* − 1 and are stored contiguously one after another in memory, e.g.



#### Arrays

- Slides-2
- Saurav

- the first two statements declare vec and sg to be one-dimensional arrays with 3 and 30 elements of type float and textttint, respectively
- a for loop is often used to access all elements of a 1D array.
- a one-dimensional array can be used to store elements of a vector



### 2D-Arrays

Slides-2

Saurav

- Two-dimensional arrays having m rows and n columns (looking like a matrix) can be declared as T[m][n], for elements of type T
- the row index changes from 0 to m-1 and the column index from 0 to n-1



#### Structures

Slides-2

Saurav

Unlike an array that takes values of the same type for all elements, a struct can contain values of different types, e.g.

struct point2d { // a structure of 2D point
 char nm; // name of the point
 float x; // x-coordinate of point
 float y; // y-coordinate of point
};

• This defines a new data type called point2d.

- note the semicolon after the right brace
- this is one of the very few places where a semicolon is needed following a right brace



#### Structures

Slides-2

Saurav

Structure members are accessed by the . (dot) operator, e.g.

point2d pt; // declare pt of type point2d
pt.nm = 'f'; // assign 'f' to its field nm
pt.x = 3.14; // assign 3.14 to its field x
pt.y = -3.14; // assign -3.14 to its field y

double a = pt.x; // accessing member x of pt
char c = pt.nm; // accessing member nm of pt



#### Structures

Slides-2

Saurav

 A variable of a struct represents a single object and can be initialised by and assigned to another variable (consequently, all members are copied)

point2d pt2 = pt; // initialise pt2 by pt, pt3 = pt2; // assign pt2 to pt3, membervise

A structure can also be initialised in a way similar to arrays: point2d pt3 = 'F', 2.17, -7.8; // OK, initialisation



### Derived Types

Slides-2

Saurav

#### Basic Data Types

- int
- char
- double, etc.

#### **Derived Data Types**

- Arrays;
- Structures;
- enumeration types: for representing a specific set of values
- unions for storing elements of different types when only one of them is present at a time
- pointers for manipulating addresses or locations of variables
- and so on...



### Enumerations

Slides-2

Saurav

The enumeration type enum is for holding a set of integer values specified by the user:

#### enum

blue,yellow,pink=20,black,red=pink+5,green=20;
is equivalent to

const int blue = 0, yellow = 1, pink = 20, black = 21, red = 25, green = 20;

- by default, the first member (enumerator) in an enum takes value 0 and each succeeding enumerator has the next integer value, unless other integer values are explicitly set
- the constant pink would take value 2 if it were not explicitly defined to be 20 in the definition
- the member black has value 21 since the preceding member pink has value 20
- note that the members may not have to take on different values



#### Enumerations

Slides-2

Saurav

Enumeration types are usually defined to make code more self-documenting; i.e easier for humans to understand
here are a few more typical examples:

```
enum bctype {Dirichlet, Neumann, Robin};
enum vars {DN, VX, VY, VZ, PR};
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT}
enum Color {RED, ORANGE, YELLOW, GREEN,
BLUE, VIOLET};
enum Suit{CLUBS, DIAMONDS, HEARTS, SPADES};
enum Roman {I=1, V=5, X=10, L=50, C=100,
D=500, M=1000};
```



#### Unions

Slides-2

- Unions, like structures, contain members whose individual data types may differ from one another
- however, the members within a union all share the same storage area within the computers memory, whereas each member within a structure is assigned its own unique storage area
- thus, unions are used to conserve memory
- they are useful for applications involving multiple members, where values need not be assigned to all of the members at any one time
- all members take up only as much space as its largest member



#### Unions

Slides-2

Saurav

```
union value {//i,d,c cannot be used at same time
int i;
double d; // d is largest member in storage
char c;
};
```

the union value has three members: i, d, and c

- only one of which can exist at a time
- thus, sizeof(double) bytes of memory are enough for storing an object of value
- members of a union are also accessed by the . (dot) operator; it can be used as the following:

```
int n;
cin >> n; // n is taken at run-time
value x; // x is a variable of type value
if (n == 1) x.i = 5;
else if (n == 2) x.d = 3.14;
else x.c = 'A';
double v = sin(x.d) //error! x.d may not exist at this time
```



#### Unions

Slides-2

```
Suppose that triangle and rectangle are two structures and a figure can be
either a triangle or a rectangle but not both; then a structure for figure can
be declared as struct figure2d {
    char name;
    bool type; // 1 for triangle, 0 for rectangle
    union { // an unnamed union
        triangle tria;
```

```
rectangle rect;
```

```
};
};
```

- If fig is a variable of type figure2d, its members can be accessed as fig.name, fig.type, fig.tria, or fig.rect
- since a figure can not be a rectangle and a triangle at the same time, using a union can save memory space by not storing triangle and rectangle at the same time
- the member fig.type is used to indicate if a triangle or rectangle is being stored in an object fig (e.g. fig.rect is defined when fig.type is 0).



Slides-2

Saurav

For a type T, T\* is the pointer to T. A variable of type T\* can hold the address or location in memory of an object of type T.

#### int\* p; // p is a pointer to int

declares the variable p to be a pointer to int; it can be used to store the address in memory of integer variables

- If v is an object, &v gives the address of v (the address-of operator &)
- if p is a pointer variable, \*p gives the value of the object pointed to by p
- we also informally say that \*p is the value pointed to by p
- the operator \* is called the dereferencing or indirection operator



Slides-2

Saurav

The second statement above declares *pi* to be a variable of type: pointer to int, and initialises *pi* with the address of object *i* 

- another way of saying that pointer pi holds the address of object *i* is to say that pointer pi points to object i
- the third statement assigns \*pi, the value of the object pointed to by pi, to j
- the fourth statement is illegal since the address of a variable of one type can not be assigned to a pointer to a different type



Slides-2

Saurav

For a pointer variable p, the value \*p of the object that it points to can change; so can the pointer p itself, e.g.

- Since p is assigned to hold the address of d2 in the statement p = &d2, then \*p can also be used to change the value of object d2 as in the statement \*p = 5.5
- when p points to d2, \*p refers to the value of object d2 and assignment \*p = 5.5 causes d2 to equal 5.5



### Pointers As Arrays

Slides-2

Saurav

 A sequence of objects can be created by the operator new and the address of the initial object can be assigned to a pointer

then this sequence can be used as an array of elements

int n = 100; // n can also be computed at run-time double\* a; // declare a to be a pointer to double a = new double [n]; // allocate space for n double obje // a points to the initial object

the last two statements can also be combined into a more
efficient and compact declaration with an initialisation:
double\* a = new double [n];
// allocate space of n objects



### Pointers As Arrays

Slides-2

- In allocating space for new objects, the keyword new is followed by a type name, which is followed by a positive integer in brackets representing the number of objects to be created
- the positive integer together with the brackets can be omitted when it is 1.
- this statement obtains a piece of memory from the system adequate to store *n* objects of type double and assigns the address of the first object to the pointer *a*.
- these objects can be accessed using the array subscripting operator [ ], with index starting from 0 ending at n-1
- pictorial representation:





Slides-2

Saurav

After their use, these objects can be destroyed by using the operator delete :

delete [ ] a ; // free space pointed to by a

- The system will automatically find the number of objects pointed to by a (actually a only points to the initial object) and free them
- then the space previously occupied by these objects can be reused by the system to create other objects
- since the operator new creates objects at run-time, this is called dynamic memory allocation
- the number of objects to be created by new can be either known at compile-time or computed at run-time, which is preferred over the built-in arrays in many situations



Slides-2

- In contrast, creation of objects at compile-time is called static memory allocation
- thus there are two advantages of dynamic memory allocation: objects no longer in use can be deleted from memory to make room to create other objects, and the number of objects to be created can be computed at run-time
- automatic variables represent objects that exist only in their scopes
- in contrast, an object created by operator new exists independently of the scope in which it is created
- such objects are said to be on the dynamic memory (the heap or the free store)
- they exist until being destroyed by operator delete or to the end of the programme



Slides-2

Saurav

An object can also be initialised at the time of creation using new with the initialised value in parentheses, e.g. double\* y = new double (3.14); // \*y = 3.14 int i = 5: int\* j = new int (i); // \*j = 5, but j does not point to i Declarations of forms T \* a; and T \* a; are equivalent, as in int\* ip; //these declarations are equivalent int \*ip; However, the following two declarations are not equivalent int\* i, j; //i and j are pointers int \*i, j; //i is a pointer to int but j is an int An array of pointers and a pointer to an array can also be defined: int\* ap[10]; //ap is an array of 10 pointers to int int (\*vp)[10]; //vp is a pointer to an array of 10 int Notice that parentheses are needed for the second statement above, which declares vp to be a pointer to an array of 10 integers. The first statement declares ap to be an array of 10 pointers, each of which points to an int



### **Multiple Pointers**

Slides-2

Saurav

- The first statement above declares mx to be a pointer to a pointer, called a double pointer
- the second statement allocates n objects of type int\* and assigns the address of the initial element to mx
- it happens that these n objects are pointers to int
- now, mx has value &mx[0]



### **Multiple Pointers**

Slides-2

Saurav

Using pointers an n by n lower triangular or symmetric matrix can be defined very conveniently; to save memory, zero or symmetric elements above the main diagonal are not stored

```
double** tm = new double* [n];
```

```
for (int i = 0; i < n; i++) tm[i] = new double [i+1]; 
// allocate (i+1) elements for row i
```

```
for (int i = 0; i < n; i++) // access its elements
  for (int j = 0; j <= i ; j++)
    tm[i][j] = 2.1 / (i + j + 1);</pre>
```

for (int i = 0; i < n; i++) delete [] tm[i]; delete [] tm; // after using it, delete space

tm is created to store an n by n lower triangular matrix. Since the lower triangular part of a matrix contains i + 1 elements in row i for i = 0, 1, ..., n - 1, only i + 1 doubles are allocated for tm[i]



### **Multiple Pointers**

Slides-2

Saurav



Note that arrays can only store rectangular matrices

 using rectangular matrices to store triangular matrices or symmetric matrices would waste space



Slides-2

Saurav

A constant pointer is a pointer that can not be redefined to point to another object; that is, the pointer itself is a constant. It can be declared and used as

```
int m = 1, n = 5;
int* const q = &m; // q is a const pointer,
    // points to m
```

 $q=\&n;\ //$  error , constant q can not change  $*q=n;\ //$  ok , value that q points to is now n int  $k=m;\ //$  k=5

Although q is a constant pointer that can only point to object m, the value of the object that q points to can be changed to the value of n, which is 5; thus, k is initialised to 5.



Slides-2

Saurav

- A related concept is a pointer that points to a constant object, i.e. if p is such a pointer, then the value of the object pointed to by p can not be changed
- it only says that \*p can not be changed explicitly by using it as value
- however, the pointer p itself can be changed to hold the address of another object. It can be declared and used as

int m = 1, n = 5; const int \* p = &m; // p points to constant object \*p = n; // error, \*p can not change explicitly p = &n; // ok, pointer itself can change

There is some subtlety involved here; look at the example:



Slides-2

#### Saurav

Since p points to m at first, the assignment m = 3 changes \*p to 3. Then the assignment p = &n changes \*p to the value of n, which is 5. In other words, \*p has been changed implicitly



Slides-2

Saurav

To avoid the subtlety above, a const pointer that points to a const object can be declared:

int m = 1, n = 5; const int\* const r = &m; // r is a const pointer that points to a const value int i = \*r; // i = 1, since \*r = m = 1 r = &n; // error, r is const pointer \*r = n; // error, r points to const value m =3; //this is the only way to change \*r int j = \*r; // j = 3

Since r is a const pointer that points to a const value m, it can not be redefined to point to other objects, and \*r can not be assigned to other values. The only way to change \*r now is through changing m.



### Void and Null Pointers

Slides-2

- The void pointer (void\*) points to an object of unknown type
- a pointer of any type can be assigned to a variable of type void\*, and two variables of type void\* can be compared for equality and inequality

- its primary use is to define functions that take arguments of arbitrary types or return an untyped object
- a typical example is the C quicksort function qsort(), declared in (stdlib.h), that can sort an array of elements of any type
- see the book of Stroustrup for more details



### Void and Null Pointers

Slides-2

Saurav

- The null pointer points to no object at all. It is a typed pointer (unlike the void pointer) and has the integer value 0
- if at the time of declaration, a pointer can not be assigned to a meaningful initial value, it can be initialised to be the null pointer

double\* dp = 0; // dp initialised to 0, dp is null pointer

- this statement is an initialisation, which declares dp to be a pointer to double and initialises dp with the value 0
- since no object is located at address 0, dp does not point to any object
- later, dp can be assigned to hold the address of some object of type double:

double d = 55; dp = &d; // \*dp=55.0\*dp = 0; // it causes d = 0.0, dp still points to d



### Pointers to Structures

Slides-2

Saurav

- A pointer can point to a structure
- in this case, its members are accessed using the -> operator
- space for structure objects can be allocated by the operator new and freed by delete

point2d ab = {'F', 3, -5}; // ab is of type point2d point2d\* p = &ab; // let p point to ab char name = p->nm; // assign nm field of p to name double xpos = p->x; // assign x field of p to xpos double ypos = p->y; // assign y field of p to ypos

 $p \rightarrow x = 15.0$ ; // ab.x = 15  $p \rightarrow y = 26.0$ ; // ab.y = 26  $p \rightarrow nm = 'h'$ ; // ab.nm = 'h'

point2d\* q = new point2d; // allocate space for q q->x = 5; // assign value to its member delete q; // deallocate space



### Pointers to Char

Slides-2

Saurav

- By a convention in C, a string constant is terminated by the null character '\0', with value 0
- $\blacksquare$  thus, the size of "hello" is 6 and its type is const char[6] with the last element equal to  $'\backslash 0'$
- due to its compatibility to C, C++ allows one to assign a string constant to char\* directly

char\* str = "hello"; // assign string constant to char\*
str[4] = 'o'; // error, string constant can not change

char str2 [] = "hello"; // array of 6 char, sizeof (str str2[4] = 'o'; // OK, now str2 = "hello"

char\* str3 = new char [5]; str3 [4] = 'o'; // OK delete [] str3;



### Pointers and Arrays

Slides-2

Saurav

- As in C, pointers and arrays are closely related
- the name of an array can be used as a const pointer to its initial element

int v[5] = {6,9,4,5,7}; int\* q = &v[0]; // point to initial element. \*q = 6int\* p = v; // point to initial element. \*p = 6

int\* s = v + 5; // pointer arithmetic , \*s is undefined

It is legal to declare a pointer to the last-plus-one element of an array. Since it does not point to any element of the array, the value that such a pointer points to is undefined.



#### Blocks

Slides-2

Saurav

- A block is any piece of code between curly brackets.
- A variable, when declared inside a block, may be used throughout that block,
- but only within that block

```
{
    int i;
    i = 5;
    {
        int j;
        i = 10;
        j = 10;
    }
    j = 5; // incorrect j is not declared here
}
```

j is said to be *out of scope* 



#### Functions

Slides-2

- there is serious limitations to being restricted to writing codes that may be placed just inside curly brackets after the initial line of code "int main ()"
- if we were to apply the same operations in different places when writing code
- we would have to repeat the lines of the code that performed these operations everywhere in the code where they were required.
- It would be much more convenient if we could write a function that we could call whenever we wanted to perform these operations.



# Simple Functions, Declaration and Call

Slides-2 Saurav

```
A simple program that writes and uses a function to determine
the minimum value of two double floating point variables x and
y and stores it in the double precision variable minimum
// Declaration of the function
double CalculateMinimum (double a, double b);
int main ()
ł
    double x = 4.0, y = -8.0;
    double minimum_value = CalculateMinimum (x, y);
    cout << '' The minimum of '' << x << '' and '' << y
            << '' is '' << minimum_value << endl;
    return 0;
}
```

The function must be declared before it can be called anywhere in the code.



# Simple Functions, Definition

Slides-2

Saurav

```
the definition can be made anywhere in the code as long
    as the declaration was done earlier before the first call
  once declared the programme looks for a valid definition
    which has the same prototype as of the declaration
   definition of the function
double CalculateMinimum (double a, double b)
    double minimum:
    if (a < b)
        minimum = a;
    else //a > b
        minimum = b;
```

return minimum;



### Passing Pointers as Function Arguments

Slides-2

- any changes to a variable made inside a function will have no effect outside the function.
- avoids unintentional alteration and keeps it local (advantage)
- however, there are occasions where we do want changes to a variable inside a function to have an effect outside a function.
- if a complex number is given in polar form  $z = re^{i\theta}$
- we may wish to get the real part (x) and imaginary part (y) return back from the function
- but a function can return only one variable
- It would be useful to include variables x and y in the function call.
- this would not work either, as values assigned inside the function would not have any effect outside.
- Fortunately pointers provide with one way around this problem.
- instead of sending the variables x and y to the function we send address of theses variables



### Passing Pointers as Function Arguments

Slides-2

- the third and fourth arguments are pointers to-that is, the addresses of the real part and imaginary part
- we send the address of these variables to the function,
- behind the scenes a copy of these addresses is made, and these copies are used in the function
- these copies refer to the same memory as the original variables
- so it is this memory that the results are stored in



### Sending Arrays to Functions

Slides-2

- when sending arrays to a function, whether memory being dynamically allocated or not it is the address of the first element of the array that is being sent to the function
- changes to this address will not have an effect in the code from which the function is called:
- however, the contents of this address- that is, the contents of the array- may be changed
- any changes made to an array inside a function will have an effect when that variable is used subsequently outside the function

- we donot have to specify the size of the first index of an array in the function prototype
- the size is computed by the compiler, it is ignored even if included during compilation
- however the size of every subsequent indices are crucial



### **Reference Variables**

Slides-2

- an alternate to using pointers to allow changes made to a variable within a function to have an effect outside the function is to use *reference variables*
- these are variables that are used inside a function that are a different name for the same variable as that sent to a function
- while using reference variables any changes inside the the function will have an effect outside the function
- these are much easier to use than pointers: all that has to be done is the inclusion of the symbol & before the variable name in the declaration of the function and the prototype
- this indicates that the variable is reference variable



### **Reference Variables**

Slides-2

Saurav

}

- the references behave like pointers behind the scene
- but without having to convert to an address with & on the function call
- they provide a layer of syntatic sugar to ease the programmer's burden



### Argument Passing

Slides-2

- pass by value does not change the value of the arguments and thus is safe
- on the other hand, pass by reference or pass by value for pointers usually implies that the values of the arguments are going to be changed through the function call, unless they are explicitly told not to, using the keyword const

```
int g (int val, const int& ref) {
// ref is not supposed to be changed
val ++;
return ref + val;
}
```

```
because of the const specifier, the compiler will give a warning if ref is to
be changed inside the function, e.g. it is illegal to write
void w ( const int& ref) {
  ref = 5; // WRONG, ref is not writable
}
```



## Default Values for Function Arguments

Slides-2

- If we are writing a function to implement an iterative technique, auch as the Newton-Raphson technique for finding a root of a nonlinear equation
- we will be content if the solution is accurate upto a tolerance (may be 10<sup>-6</sup>!!)
- rarely we would want to change the tolerance
- one may want to restrict the function evaluation to some extent
- if its implemented via a while loop numerical rouding off errors may hinder the solution to cut through the tolerance to stop the process
- is therefore quite prudent to write a function for implementing the Newton-Raphson, that sets a default tolerance for the solution



Slides-2

- and a default maximum iterations
- we would be able to call this function without specifying these default values
- however if we did want to call this function with different values then we would be able to do this as well (see new-rap.cpp)



### Function Overloading

Slides-2

- suppose we want to write a function to multiply a vector by a scalar and another function to multiply a matrix by a scalar
- it would be desirable to call both these functions Multiply.
- this is alloweed in C++
- we write different function prototypes and functions for both of these operations;
- the compiler then chooses the correct function based n the input arguments
- this is know as function overloading

```
void Multiply (double scaler, double* u, double* v, int n);
void Multiply (double scaler, double** A, double** B, int n)
// vector multiplication
Multiply (s, u, v, n);
// matrix multiplication
Multiply (t, A, B, n);
```



### Function Pointers

Slides-2

- suppose we want to write a function to implement the solution of the non-linear equation f(x) = 0 using Newton-Raphson
- where f is user specified
- may want to call this function more than once for solving non-linear equations
- and for diffferent user-specified function during the same execution
- to achieve this we need to specify the appropriate non-linear function each time the function is called
- this may be done using *function pointers*.



Slides-2

Saurav

- we specify two functions
- we declare a function pointer \*p\_function
- this declaration specifies that the function that this pointer refers to must

accept one input argument which is double precision
 return one double precision floating point variable

```
double (*p_function) (double x);
```

```
p_function = &myFunction;
cout << p_function (2.0) << endl;</pre>
```

```
p_function = &myotherFunction;
cout << p_function (2.0) << endl;</pre>
```



### **Recursive Functions**

Slides-2

Saurav

- in some applications, we may wish to call a function from within the same function:
- this is known as recursion
- and is possible in C++
- the simplest application is calculation of factorial of a positive integer n, denoted by fact (n)

$$fact(n) = n \times fact(n-1), \quad n > 1$$
  
 $fact(n) = 1, \qquad n = 1$ 

return (n \* CalculateFactorial (n - 1));