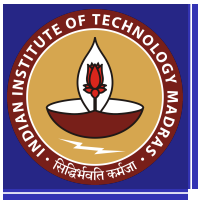
C++ Programming Basics
Procedural Aspects

Saurav Samantaray [1]

[1]Department of Mathematics
IIT Madras

August 6, 2024



The Very First C++ Code

- 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;
    }
```

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

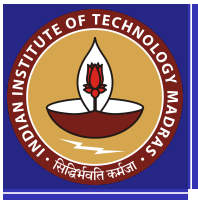
Compiling a C++ Code

- `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++

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

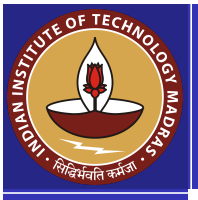
- It should be able to take in two integers, lets say "a" and "b".
- It should print the final sum.
- It should have a way to understand $a > b$ or vice-versa.



Variable Declaration


```
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


```
if (a>b)
        {
        cout <<''since a > b we need to swap
                        between them'' ;
        ....
```

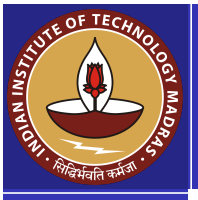
- It is used to control the flow of the program.
- Control options are:
  - ```
    if (??)
    {
    ...
    }
    else
    {
    ...
    }
    ```
    ```

- nested if's;

```
if (x > z)
{
  if (p > q)
  {
    // Both conditions have to be met
    y = 10.0;
  }
}
```

- multiple if's;

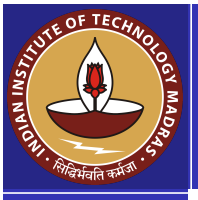
```
if (i > 100)
{
  y = 2.0;
else if (i < 0)
{
  y = 10.0
}
else
{
  y = 5.0; }
```



Loops


```
for (int i = a; i <= b; i++)
        {
```

- 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

- The `while` loop:

```
while (x > 1.0)
{
  x * = 0.5;
}
```

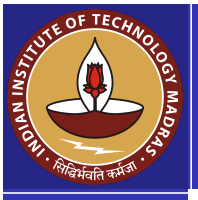
- The `do while` loop:

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

Arrays

- 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.

```
float vec[3]; // array of 3 floats : vec[0]
                    // vec[1] ,vec[2]
int sg[30]; // array of 30 ints: sg[0],
                   // ..., sg[29]
vec[0] = 1.0; // accessing element 0 of vec
vec[1] = 2.0; // accessing element 1 of vec
for(int i = 0; i < 30; i++) sg[i] = i*i + 7;
int j =sg[29]; // accessing the last
                       // element of sg
```



Arrays

- 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

- 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 $n-1$

```
double mt[2][5]; // 2D array of 2 rows
                                // and 5  columns


for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 5; j++) {
        mt[i][j] = i + j;
    }
}
```

Structures

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 points
  char nm;  // name of the point
  float x;    // x-coordinate of point
  float y;    // y-coordinate of point
};
```

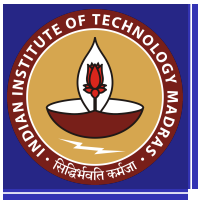
- This defines a new data type called `point2d`.
- note the semicolon after the right brace
- this is one of the very few places where a semicolon is needed following a right brace

Structures

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
```



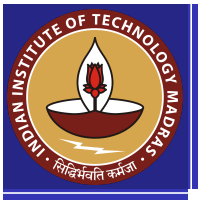
Structures

- 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
```



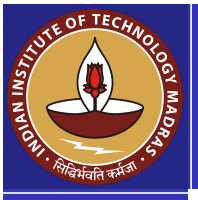
Derived Types

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

- 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

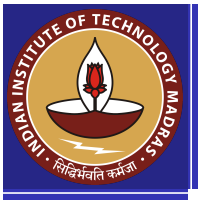
```
const int blue = 0, yellow = 1, pink = 20,
black = 21, red = 25, green = 20;
```

- 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

- 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

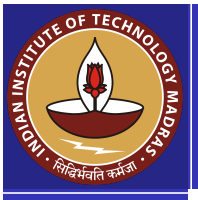
- Unions, like structures, contain members whose individual data types may differ from one another
- however, the members within a union all share the same storage area within the computers memory, whereas each member within a structure is assigned its own unique storage area
- thus, unions are used 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


```
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
```

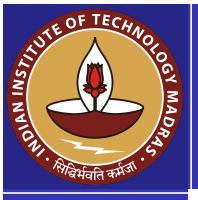


Unions

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).



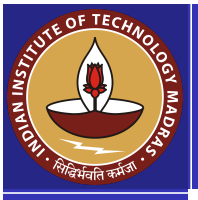
Pointers

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

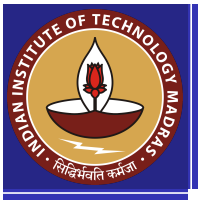
- If v is an object, $\&v$ gives the address of v (the address-of operator &)
- if p is a pointer variable, $*p$ gives the value of the object pointed to by p
- we also informally say that $*p$ is the value pointed to by p
- the operator $*$ is called the dereferencing or indirection operator



Pointers


```
int i = 5; // i is int, value of object i is 5
int* pi = &i; //pi is a pointer to int
              // and assign address of i to pi
int j = *pi; //value of object pointed to by pi
              //is assigned to j, so j=5
double* d = &j; // illegal
```

- 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



Pointers

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.

```
double d1 = 2.7, d2 = 3.1;
double* p = &d1;  // p points to d1,
                  //now *p = 2.7
double a = *p; // a = 2.7
p = &d2; // p now points to d2, *p = 3.1
double b = *p; // b = 3.1
*p = 5.5; // value p points to is now 5.5
double c = *p; // c = 5.5
double d = d2; // d = 5.5, since *p=5.5
```

- 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

- 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

- 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 subsripting operator [ ], with index starting from 0 ending at $n-1$
- pictorial representation: