Slides-2

Saurav

C++ Programming Basics
Procedural Aspects

Saurav Samantaray [1]

[1] Department of Mathematics
IIT Madras

August 7, 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
  texttint, 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

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

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

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.

```
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

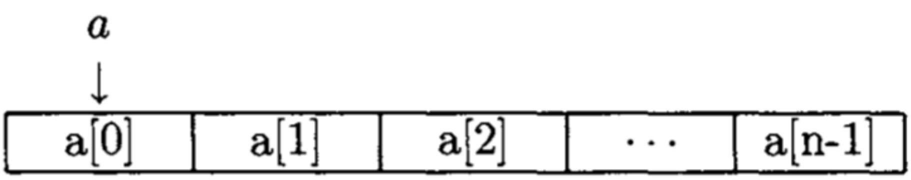
```
int n = 100; // n can also be computed at run−time
double∗ a; // declare a to be a pointer to double
a = new double [n]; // allocate space for n double obje
// a points to the initial object
```

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:



Pointers

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

Pointers

- 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

Pointers

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

Two-dimensional arrays and matrices can be achieved through double pointers (a pointer to a pointer is called a double pointer)

```
int** mx; //double pointer:  a pointer to a pointer
mx = new int* [n]; //new space to hold n pointers to int
              //mx points to initial element mx[0]
for (int i = 0; i < n ; = i ++) mx[i] = new int [m];
//create m objects for each of the n pointers
//mx[i] points to initial element mx[i][0]
```

- 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

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);

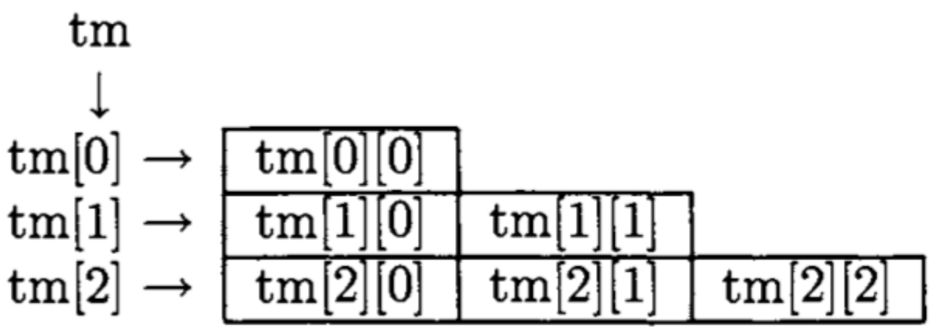
for (int i = 0; i < n; i++) delete[] tm[i];
delete [] tm; // after using it, delete space
```

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]

Slides-2

Saurav

Multiple Pointers



- Note that arrays can only store rectangular matrices
- using rectangular matrices to store triangular matrices or symmetric matrices would waste space

Constant Pointers

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.

Constant Pointers

- 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:

Constant Pointers

```
int m = 1, n = 5;
const int *p = &m; // p points to m, so *p becomes 1
int i = *p; // *p = m = 1, so i = 1
m = 3; // m=3, so *p becomes 3
int j = *p; // *p = 3, so j = 3
p = &n; // ok, p itself can change, *p = 5
int k = *p; // *p = n = 5, so k = 5
```

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

Constant Pointers

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.