Q Given an x, n, m find (x^n) % m

```
{ int res = 1
  while (n > 0)
  { if (n & 1)          // means typecast to long long
    { // odd
      res = (1LL * (res) * (x) % m) % m
    }
    x = (1LL * (x) % m * (x) % m) % m;
    n = n >> 1;          // means Divide by 2.
  }
// bit oppn are less costly so we use them
  return res;
}
```

— x — x — x — x — x —

Lecture 25

Pointers in C++

(1) `int num = 5;` → memory block [5] num

(2) memory only has address.
`Cout << num` // get me whatever written in address of num

(3) memory has symbol table which map variable name with memory address.
So num variable refers to address in which 5 is stored.

(4) address of operator (&)
↓ in Hexadecimal
`cout << &num;` // give address of num

(5) Need of pointer
It helps in storing address
`int *ptr = &num;`
`Cout << ptr;` // 0x61fe6b
`cout << *ptr;` // 5
↓ value at pointer
(* is de-reference operator)

(6) always initialise the pointer with any value i.e 0 if nothing available

(7) `int *p = &num`
`int *p` → creating a pointer named p to a int data type
`&num` → address of num

(8) data type of LHS & RHS should be same. `char ch = 'L';`
`char *p = &ch`
`double d = 4.2;`
`double *ptr = &d`

(9) if we do `num++`
`Cout << *ptr;` // 6
ptr is reference to num = 5
So all changes will be reflected

(10) size of pointer is either 4 or 8 always, does not matter the data type it stores. as pointer always stores the address.

(11)

```
int n1 = 5;
int *p3 = &n1;
int a = n1;
n1++;
cout << n1;    // 5
cout << *p3;   // 6
cout << a;     // 6
(*p3)++;
cout << n1;    // 7
cout << *p3;   // 7
```

(12) Pointer does not make any copy.

(13) Copying pointer

```
int *q = p;
```

(value) *q will be same as *p (value)

(address) q will be same as p (address)

```
(*p)++;
```

now value of p and q both increment by 1;

(14)

```
(*p)++;   // move value + 1
p = p + 1;   // add 4 byte in address if data type is int and so on
(*p)--;   // value - 1
```

(15) Void pointer => which is of type void and can be typecasted to any data type.

wild pointer => which is not initialised and point to any garbage value.

Lecture 26

(Pointers advanced)

(1) int arr[10];

4 * 10 = 40 byte memory allocated

arr type — no. of blocks

address of first ind = arr (same)

add of first ind = &arr[0]

*arr == arr[0] (same)

*arr + 1 = 1 increment to value of *arr

*(arr+1) gives value of second index

*(arr+2) gives value of third index

(2) arr[i] = *(arr+i)

or

i[arr] = *(i+arr)

(3) Difference b/w Array and pointers

(1) int arr[10]; // 40 byte memory allocated

sizeof(arr); // 40

where arr point to first index

```
int *p = &arr[0];
// point p to address of first location
```

*p will give us value at first index

sizeof(p); // 8

② int a[20] = {1, 2, 3, 5}

```
cout << &a[0]; // same
cout << &a;    address
cout << a;     in all 3
```

```
int *p = &a[0];
cout << p      // 0x61fe18
cout << *p     // 1 (first element)
cout << &p     // 0x61fe14
```

&a[0] and &p both are different

③ Symbol table content cannot be changed.

int arr[10];

Symbol table has mapping of arr with some address

arr = arr + 1; // error

int *p = &arr[0]

p = p + 1 // move to next memory block, no error

— x — x — x — x — x —

④ Char Array in pointers

int arr[5] = {1, 2, 3, 4, 5}

char ch[6] = "abcde"

cout << arr; // 0x61fe0c

cout << ch; // abcde

→ prints only address in int array

→ prints whole array in char array

Implementation of cout is different for both char array & int array.

```
char *ptr = &ch[0];
cout << ptr;  // abcde
cout << *ptr; // a
```

cout behaves differently for character array.

This will print till it gets Null character '\0'

⑤ char ch[6] = "abcde";

Steps of implementation of above statement

① Make temp memory and store "abcde" in it.

② make block named ch and store temp memory content in it

char *c = "abcde"

Steps [RISKY APPROACH]

① make temp memory & store "abcde" in it.

② make a pointer c pointing to a block which has address of 'a'

⑥ Functions & pointers

We can change value of pointer using update fun^n but we cannot change the address as copy of that pointer will be passed in fun^n

```
int func^n(int arr[])
{
    This does not mean whole array is being passed it only passes pointer to first element of array.
}
```

⇕

```
int func^n(int *arr)
{   // this is also correct.
}
```

So we can send part of our array as parameter in a func^n

```
func^n(arr + 3);
```

// will send ~~addr~~ pointer pointing to third element of array as our parameter to func^n

— x — x — x — x — x —

Lecture 27

(Pointer Part 3)

① ~~Pointer of~~ int *ptr = &value ~~we can ots~~ we use this format as:

1. it tells us which data type we're referring to
2. How many bytes we need to consider

② Double pointer

Pointer to a pointer

```
int **ptr2 = &ptr;
int i = 5;
int *ptr = &i;
int **ptr1 = &ptr
int ***ptr2 = &ptr1
```

```
int i = 5   710
        : 5
int *ptr = &i   820
        710
        ptr
int **ptr2 = &ptr   1000
        820
        ptr2
```

Symbol Table

i → 710

ptr → 820

ptr2 → 1000

```
cout << *ptr2; // will give address of ptr
cout << **ptr2; // will give value of *ptr i.e 5
```

③ Double Pointer & function

```
void func(int **ptr)
{   // ptr is double pointer to ptr1
    ptr = ptr + 1;
        // No change as ~~ptr = *ptr + 1~~ copy of it will be formed
    *ptr = *ptr + 1;
    // change will come.
    *ptr has address of ptr 1
    So move it to next block
    0x61feb4 → 0x61febc
    **ptr = **ptr + 1;
    // modify value at ptr1 by 1 i.e i = 5 → 6
}
```

④ Practice Questions

1) int arr[6] = {11, 21, 13};
int *p = arr;
cout << p[2] << endl;
p[2] same as *(p+2)
means 13

2) int arr[] = {11, 21, 31, 41}
int *ptr = arr++;
cout << *ptr;
This is error because arr++ means we are making change in Symbol table which is not possible

3) char arr[] = "abcde"
char *p = &arr[0];
p++;
cout << p; // bcde

4) char str[] = "Lokesh"
char *p = str
cout << str[0]; // L
cout << p[0]; // L

5) int first = 110;
int *p = &first;
int **q = &p;
int second = (**q)++ + 9;
// 111
cout << first; // 111
cout << second; // 119

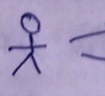
Pointer Codestudio Practice [Done]

— x — x — x — x —

Lecture 28

Dynamic Memory allocation

① Reference Variable



int i = 5; 710

Let say its other name is j

memory location and block is same but names are different.

int i = 5;
Creating a reference variable
int &j = i

② now i++ or j++ will have same effect on value

③ why reference variable is needed ?

we already saw pass by value where a copy is created & changes are not reflected in main func^n. changes are only made inside update func^n. if we pass variable by reference, values will be changed in main also.

```
void call by value (int n)
{
    n++;
}
void call by reference (int &n)
{
    n++;
}
```

In pass by value, new memory or a copy is created
In pass by reference, no new memory is created, no copy is created and changes are reflected in real time.

④ We can also pass reference variable in return type

```
int &  update (int a)
  {
    int num = a;
    int &ans = num;
    return ans;
  }
```

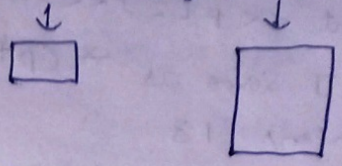
This is a Bad practice because num, ans are local variables and will free up that memory outside update() block

⑤
```
int * fun(int n)
  { int *ptr = &n
    return ptr;
  }
```
This is also a BAD Practice as ptr is local variable whose memory will be freed up outside fun().

⑥
```
int n;
cin >> n;
int arr[n];  // This was BAD
             Practice as size
```
of array is known only at runtime, we should know size of array at compile time.

int arr[10000] would be better than this.

Every program has limited memory to get executed
Program has 2 type of memory
Stack memory, Heap memory



These memories will be alloted based on size which we get during compile time. When we get size of array in runtime, we might not have that much stack space and our program will crash.

⑦ How to tackle this situation?
We will use heap memory to do this size allocation using variable.
To Create anything in heap we need to use 'new' keyboard

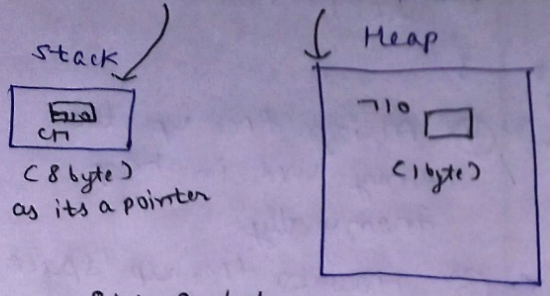
While using stack memory we call it static memory allocation while using heap memory we call it Dynamic memory allocation

```
new char;  // This will
```
give us memory location of a block in heap which we can access using pointer.

```
char *ch = new char;
```

⑧

```
char *ch = new char;
```



8+1 = 9 byte.

⑨ we can create dynamic array like new int [5]; which we store in a pointer.

```
int *arr = new int [5];
```

8 byte as its pointer

give address of first block

5 x 4 = 20 byte

③ in static allocation memory gets freed up automatically.

memory will added in heap 4+4+4..... and at some-time, memory will get crashed

③ in dynamic everything is manual so we need to free up the memory we have used. also using delete keyword.

```
delete i;
or
delete [] arr;
```

⑩ Static VIS Dynamic memory allocation

① `int arr[50];`

50 x 4 = 200 byte in stack

① `int *arr = new int[50];`

200+8 = 208 byte in stack & heap.

② `while (true) { int a = 5; }`

Everytime memory will be allocated to a int in stack and will be destroyed at func^n ends.

② `while (true) { int *ptr = new int; }`

Everytime 8 + 4 = 12 byte will be allocated, 8 in stack, 4 in heap stack memory will be freed up at func^n end but heap memory never destroys

⑪ Addres Typecasting

Type casting means changing one data type to another.

Implicit Typecasting

where compiler itself do type-cast

```
int i = 65;
char c = i;
cout << c;   // 'A'
```

Explicit Typecasting

which is to be done by coder

```
int i = 65;
int *p = &i;
char **pt = (char *) p;
```

we can't do

`char **pt = p;` as we cannot store int pointer in char pointer so we typecast it using (char*) This is called explicit typecasting.

Lecture 29

Dynamic memory allocation in 2D Arrays.

① `int arr[5][5]` // Creates a 2D Array of 5 rows 5 columns

② This will create a 2D array in static memory

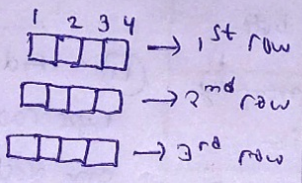
What if we need to create it dynamically ?

③
```
int m;
cin >> m;
int n;
cin >> n;
int arr[m][n]; // BAD Practice
```

`int *arr = new int[n];`

gives us an array in heap memory dynamically.

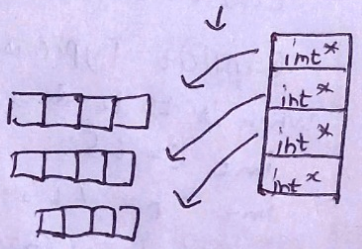
`arr[3][4]` is



★ `new int*[n];`

We know we can create array like `int *arr;` Someway

```
int **arr = new int*[n];
```

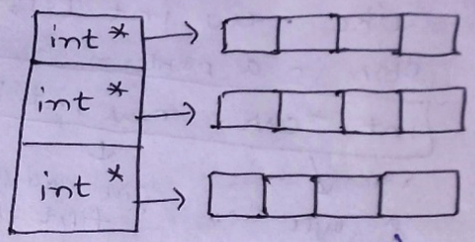


Map these int* with arrays dynamically by mapping each int* with new int[n] Run a loop.

```
for(   )
{ arr[i] = new int[n];
}
```

④ Always free up the memory used in heap manually

⑤ How to free up space of 2D Array created dynamically ?



We need to free up part first using for loop.

Let say no. of rows = n

```
for(i=0 → n)
{ delete[] arr[i];
}
```

now we free up int* vali memory

```
delete[] arr;
```

Lecture 30

Macros, Global Variables, Inline Functions & Default Args

① Macro

We use #define to create a macro.

② Let say we use pie(x) too much in our program.

now if we use $\pi = 3.14$, 1000 times in our code and we need to make $3.14 \rightarrow 3.1$ then we have to make 1000 changes which is not possible or time consuming process.

We can also declare a variable of type double.

double pi = 3.14;

but this will take some memory

Can we optimise it?

We will use macro

#define PI 3.14

A piece of code in program that is replaced by value of macro.

We can update any variable +1 or -1 anytime but we cannot update a macro.

When macro name is encountered by compiler, it replaces name with definition of macro.

macro need not be terminated with ;

#define PI 3.14

(Macro name: PI; value of macro: 3.14)

#define AREA(l,b) (l*b)

(macro name: AREA(l,b); macro value: (l*b))

#define min(a,b)

((a)<(b)) ? (a) : (b))

This is called Function-like macro which finds min of 2 values.

③ Global Variable

When we want to share a variable between functions.

We can do it using reference variable also.

```
void A(int& i)
{   cout<<i;
    b(i);
}
void B(int& i)
{   cout<<i;
}
```

Same can be done using Global variable.

Local Variable: Variable whose life span is just within a block.

```
#include< >
using std;
int score = 15;  // Global Variable

main()
{

}
```

We should never use Global Variable, because any func^n can change its value which in turn will reflect everywhere so always use concept of reference variable for var sharing.

④ Inline functions

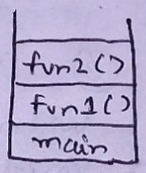
They are used to reduce function call overhead.

when we call a new func^n, it come upon stack again which derease the performance little bit.

Ternary operator

Cond^n ? a : b

a → if true, b → if false



ans = (i > j) ? "i greater" : "j greater";

We put it inside func^n and write inline.

if our func^n body is of 1 line we can make it inline

```
inline int getMax(int &a, int &b)
{ return (a>b)? a : b;
}
```

During compile time all getMax(a,b) will be replaced by (a>b)? a:b

No extra memory usage, no overhead func^n call.

⑤ Default Arguments

To make a argument in a func^n optional i.e value mile toh use krle vrna ek default value use krle

```
Void print(int arr[],
    int n, int start=0)
{   // start is default
        argument
}
main()
{   print(arr, n, 3); // ✓
    print(arr, n); // ✓
}
```

we have to always make rightmost argument as default argument first

```
void f(arr, int n=0, int
        start)
{ // Wrong first
    make start as
    default args
}
```

⑥ Constant Variable

```
Const int i=5; ✓
Const int i;
i = 6;          → ✗
Const int * ptr = &val; ✓
int * Const ptr = &val; ✓
Const int * Const ptr = &val; ✓
```