2nd chapter

Memory Allocation

— Address Typecasting

Q) why don't we use like this,

Pointer P = &i; To declare a Pointer, why we had to use 'int*', 'char*' etc ??

Sol^n Lets Assume Pointer P = &i;
creates a Pointer

P ⟶ [Address] {P Stores address}

Now, Lets use *P (dereference operator)

*P ⟶ value of P ⟶ Now, we dont Know what to Print out, int, char etc

Actully, we don't know 2 things now,

1) Number of bytes to Read
2) How to interPret bytes as, weather as integer or character.

CODE ⟶

```
int main() {
    int i = 65;
    char c = i;
    cout << c << endl;
```

output: A
(Refer ASCII Codes)

This is Typecasting (Implicit)

i.e, Storing 'int' type data into 'char' type

— Type Casting is storing different types of data into other different type.

Types of Typecasting

- Implicit Typecasting
- Explicit Typecasting

→ Lets see through a Ex

```
int* p = &i;
char* pc = p;
```

output: error: {Cannot initialize a variable of type 'char*' with an lvalue of type 'int*'}

In such case, we can explicitely tell the Compiler to Just change into 'char*', like this

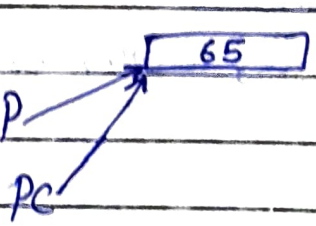
CODE

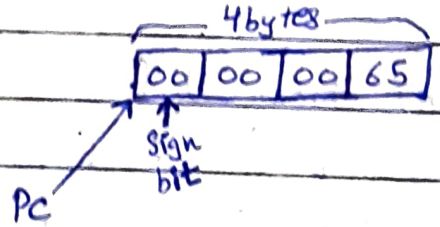
```
int* p = &i;
char* pc = (char*) p;
```

} explicit Typecasting

→ Lets see according to Commands what would be happen (Theoritically)

*p ⟹ 65





accordingly,

*Pc will be Equal to 00

&

*(Pc+3) will be Equal to (65 or A)

→ Lets See through Code

CODE

```
Cout << *p << endl;
Cout << *Pc << endl;
Cout << *(Pc+1) << endl;
Cout << *( Pc+2) << endl;
Cout << *( Pc+3) << endl;
```

outPut: 65
A
— (null)
— (null)
— (null)

i.e, 65 is stored like 

→ The Reason is Little Endian & big Endian System.

Q what will this Print out now?

```
Cout << P << endl;
Cout << Pc << endl;
```

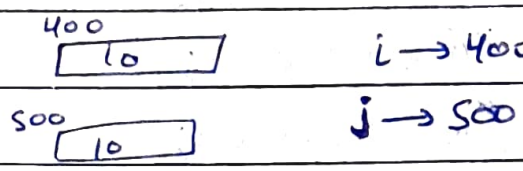
Sol^n Common Answer : both will give out Addresses ⊗
{ char will give out value not address }

outPut : Address of 'P'
A

Reference & Pass by Reference

Reference Variables

→ `int i = 10;`
`int j = i;`
`i++;`
`Cout << j << endl;`



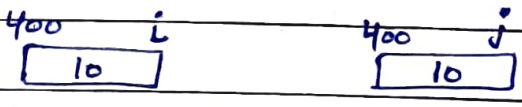
i.e, increment in 'i' won't effect 'j'

→ Actually, There is a way to create such changes in 'i' leads to change in 'j' itself, Lets see

The way is by declaring 'j' as a Reference Variable (&j)

CODE
```
int i = 10;
int& j = i;
i++;
Cout << i<< endl;
Cout << j<< endl;
```

output: 11
11



i.e, Now 'i' & 'j' are identical
changes in 'i' Leads changes in 'j' & vise-verse.

```
int K = 100;
j = K;
Cout << i << end;
```

output: 100

NOTE we Can do,

```
int i;
i = 100;
```

but we Cannot do,

```
int& j;
j = i;
```

error: declaration of reference variable 'j' requires an initializer

→ Now the question arises, what is the need to use such things, like when 'i' is available what is Point to create 'j' out there??

→ Lets See How it is usefull.

— In functions & Pointers, we find that on increment in some Cases changes do not Reflect in main memory, it Can be easily Solved using this.

CODE

```
Void increment(int& i) {
    i++;
}
int main() {
    int i = 10;
    increment(i);
    Cout << i << endl;
```

outPut: 11

```
Void increment(int i) {
    i++;
}
int main() {
    int i = 10;
    increment(i);
    Cout << i << endl;
```

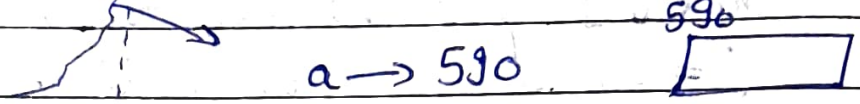
outPut: 10

→ Lets See Something intresting via CODE.

CODE
```
int& f(int n) {
    int a = n;
    return a;
}

int main() {
    int& K = f(i);
    Cout << K << endl;
}
```

Now, there is Huge Problem in this Code, Lets see it



K → 590

Here, 'a' has its Scope of 'int&' inside function only, when Code Leaves the function, it has Right to Remove 'a' or change or store it some where Else.

→ when we Run this Code, System itself generate a warning: reference to local Variable 'a' returned

→ Same warning will be there if we use 'int*' & then use return by reference.

i.e.
```
int* f2() {
    int i = 10;
    return &i;
}
int main() {
    int* P = f2();
}
```

warning: Address of Stack memory associated with local Variable 'i' returned

Dynamic Memory Allocation

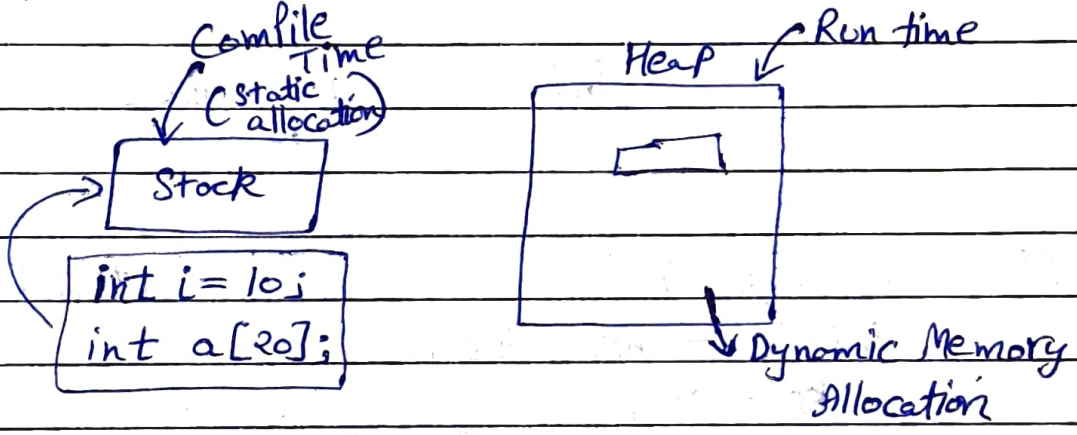
int a[ ]; → when we start with arrays, we need to give them a size. i.e, a[20], b[30] etc

If we do,

```
int n;
Cin >> n;
int a[n];
```

taking size of Array from user, but this might work sometime & sometimes not work. Lets see why so?

→ whenever we allocate memory, there are two types of memory. Stack & Heap. where stack is smaller than Heap.

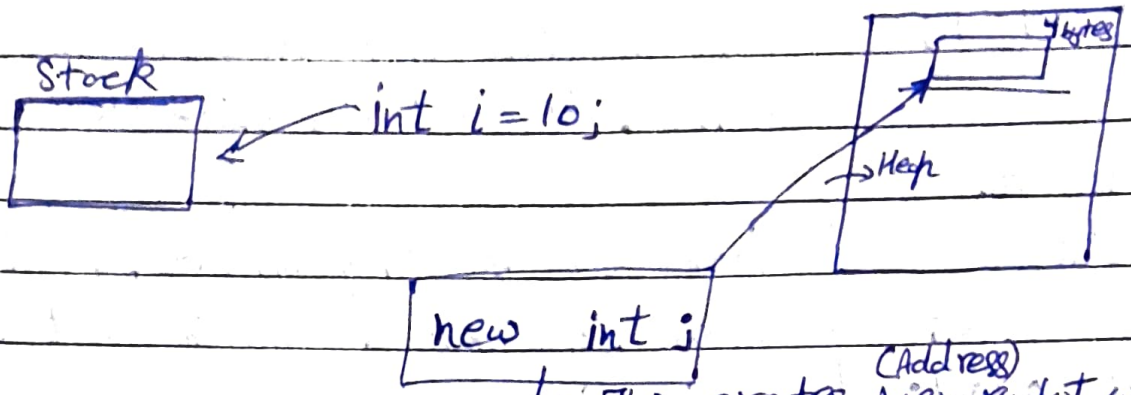


→ Now, If we create int a[20000]; Then the stack that Program creates during Compile time is big enough so there will be no issues.

→ but when we take input from user, at Run time stack cannot be changed. i.e, if 'n = 20000' it might be out of memory in stack

→ Now, in such case solution is to create this memory in Heap, but yet now whatever we learned is on stack only, so now we need to know about 'dynamic Memory allocation', i.e, allocating memory in heap.

→ Now, Lets See How to Allocate memory dynamically.



↳ This creates memory (Address) but we Cannot access it, we need to store it then only we Can access it.

ie,

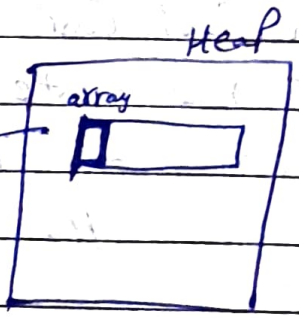
```
int * p = new int;
*p = 10;
```

CODE

```
int main() {
  int *p = new int;
   *p = 10;
  Cout << *p << endl;
// Similarly
  double* Pd = new double;
  Char* C = new char;
```

output: 10

→ Now, Lets See the Case of Array.




```
int* p = new int[50];
```

CODE
```
int* Pa = new int[50];
int n;
cin >>n;
int* Pa2 = new int[n];
```

→ This Solves two Problems,

1) we take a large value for array 10000 or big, but if user use only 10 or less then a lot of memory get wasted out.
2) array can be alloted according to user & smooth functioning of machine.

→ Pa2 is our array now

Pa2[0] = 10; → *(Pa2+0)

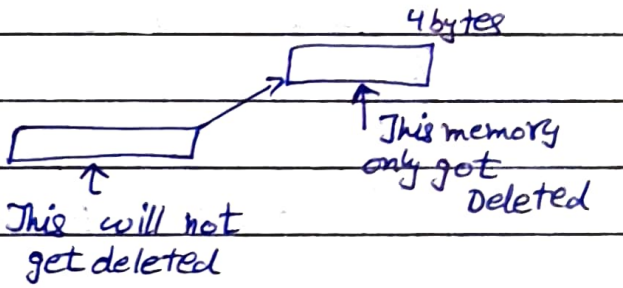
Pa2[i] → *(Pa2+i)

→ An Important difference b/w Stack & dynamic memory Allocation,

Stack	Dynamic
- auto Release of memory on the basis of Scope of Variable.	- manual Release is Required using "delete"

CODE NOTE

```
int *P = new int;
delete P;
```




```
P = new int;
delete P;
```
} In Case of Single element

i.e we Can Still use Pointer "P" & reassign it

```
P = new int[100];
delete [] P;
```
} In Case of arrays

Pointer will get deleted only when Code moves out of Scope.

— Dynamic Allocation of 2-D arrays

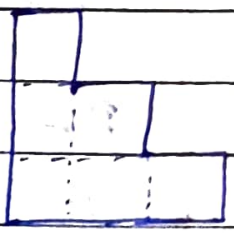
— firstly,

= new int [10][20] ⊗ Total wrong

→ Lets understand it through CODE

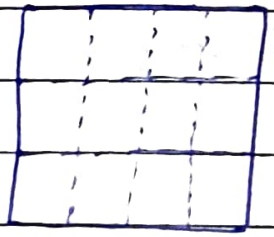
CODE
```
int main() {
    int m, n;
    cin >> m >> n;
    int** P = new int*[m];
    for (int i=0; i<m; i++) {
        P[i] = new int[i+1]; // P[i] = new int[n];
        for (int j=0; j<n; j++) {
            cin >> P[i][j];
        }
    }
    for (int i=0; i<m; i++) {
        delete [] P;
    }
    delete [] P;
}
```

—> The above CODE will create Array looks like

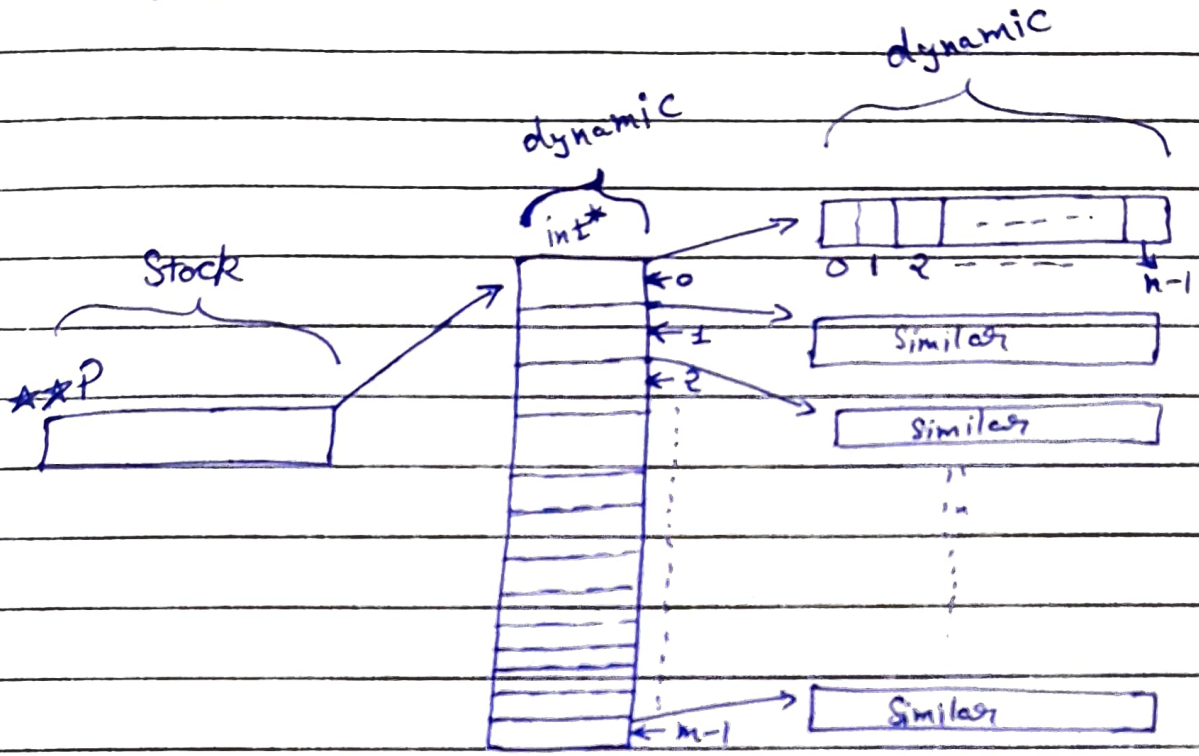


→ If we use int[n]; Then we get this array

Type



→ Working of CODE



→ Let for Ex we had to Access P[3][4] from [5][5] array