Dt. 31st Dec 2022
Pg.

Module - Dynamic Memory Allocation

Topic - Address Typecasting

```
int i=10;
int *p= &i;
```
→ how many bytes to read & how to interpret the data

pointer p = &i; → because to interpret i.e., how to interpret the data stored at memory location

pointer p

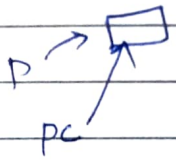
01101011 - - - - -

(3)

```
int i=65;
char c=i; → A
```

ASCII	
A → 2 → 65-92	
a → 2 → 97-122	
32 → space	
9 → tab	
10 → Enter	

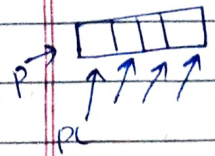
```
int *p= &i;
char *pc= p;
```

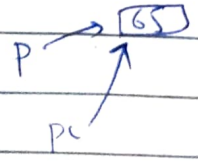


#

Code

```
int i=65;
char c=i; → Implicit Typecasting done by system.
cout<<c<<endl; → A
int *p= &i;
char *pc= p; → X
char *pc= (char*) p;
cout<<*p<<endl; → 65
cout<<*pc<<endl; → A
cout<<*(pc+1)<<" "<<*(pc+2)<<" "<<*(pc+3)<<endl;
```





WOW!

Dt________
Pg________

Integer is kept in reverse order

[65 | _ | _ | 1] → signbit

1) Little Endian
2) Big Endian

cout<<p<<endl; → address

cout<<pc<<endl; → goes to location & starts reading till encounters '\0'

[65 | 0 | |]
↑ p ↑ pc

(3) ch = 'A' → ch ['A'] 850 (d)

pt 850 [890]

610 % 255

255 × 2 = 510

255) 610 (2
510
100

Topic - Reference Variables & Pass by Reference

```
int i=10;        i 400 [ 10 ]        i → 400
int j = i;       j 500 [ 10 ]        j → 500
i++;
cout<<j<<endl; → 10
cout<<i<<endl; → 11
```

To same memory location same, use & variable so that changes are reflected while doing change through any of the variables.

int &j = i; → j is reference to existing memory

```
int i=10;          j, i 400 [ 10 ]
int &j=i;
i++; → 11
cout<<j<<endl;
j++; → 12
cout<<i<<endl;
int k=100;
j=k;
cout<<j<<endl;
```

```
int i=10;
int &j=i;
int k=50;        i → 400
j=k;             j → 400
```

Not allowed

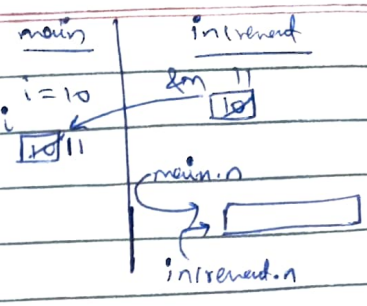
int &j; ✗

j = i; ✗

Tell at the time of declaration of reference variables to which memory it is pointing.

Dt
Pg

```
void increment(int &n){
    n++;
}
```

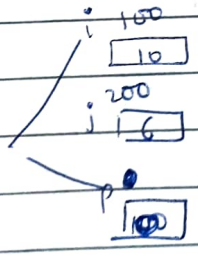


①
```
int f(int n){
    int a = n;
    return a; }
int &k = f(i);   → Always see to use // Bad practice
```

②
```
int *f2(){
    int i = 10;
    return &i; }
int *p = f2();   → // Bad practice
```

In functions & main function
if one changes then changes get reflected in both
No extra memory creation as we are using the existing memory.

①



10 7 10

{{{Dynamic Memory Allocation}}}

→ fixed size problem

```
int arr[100];   → clear at compile time
int n;
cin>>n;
int arr[n] → ×  not allowed
```

WOW!

Static Memory Allocation

Memory

int i=10;

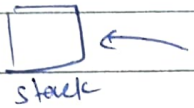
int arr[20];

int arr[1000];

Dynamic memory Allocation

Heap Memory

How to create memory at Heap Memory at Runtime.



int i=10;

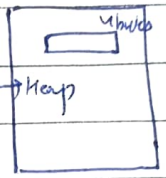
int *a = new int;

keyword

On Stack

Address of pointer

*a = 10;



double *pd = new double;

*pd = 20;

Code

1) int *p = new int;

*p = 10;

cout << *p << endl;

double *pd = new double;

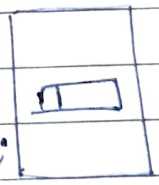
*pd = 20;

cout << *pd << endl;

char *pc = new char;

*pc = 'A';

cout << *pc << endl;



int *pd = new int[50];

2) On stack ← int *pa2 = new int[50]; → 200 on heap → Based on user requirement

pa2[0] = 10;

int arr[1000]; ← compiler, more

int n;

cin >> n;

for (i<n) {

cin >> arr[i]; }

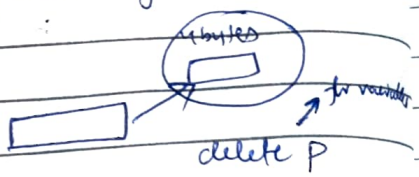
arr[i] → *(arr+i)

WOW!

Dt
Pg

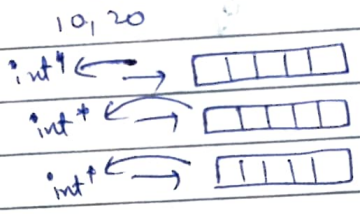
In Static Allocation, memory is based on scope → automatic release of

In Dynamic Allocation, memory is deallocated using delete keyword.
↓ manual release



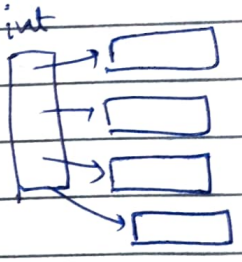
delete [] arr;
↓
for arrays

Dynamic Allocation of 2D Arrays



[int*, int*]

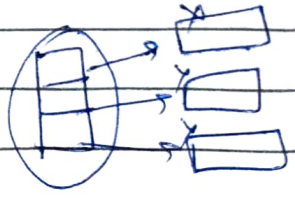
```
int **arr = new int*[20];
arr[0] = new int[25];

p[4] =
```



Code

```
int **p = new int*[10];
for(int i=0; i<10; i++){
    p[i] = new int[20];
    for(int j=0; j<20; j++){
        cin>>p[i][j]; }
}
for(int i=0; i<10; i++){
    delete [] p[i]; }
delete [] p;
```

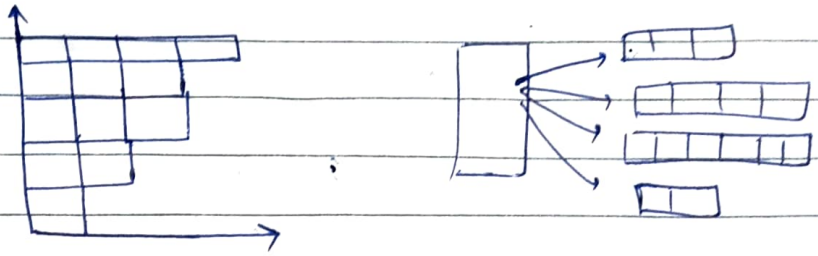
10 × 1000 = 1000
+ 80
1080



WOW!

Dt_____
Pg_____

→ Jagged Arrays → An array of arrays whose elements are arrays of variables sized columns.



Special keywords for better code practice & help much faster

** Macros & Global Variables **

#define

pi → 3.14

area → 3.14 * r * r

① It will be hard to change

②

Preprocessor directive before compilation

#define PI 3.14 → before compilation

```
main(){ int r; cin>>r;
    return 3.14 * r * r;
                ↓
                PI
```

② Global Variables → which are accessible throughout the program.

↓

Not a good practice

```
int a;
void g(){
    a++; → 11
    cout<<a<<endl; → 12
}
void f(){
    cout<<a<<endl; → 10
    a++; → 11
    g();
}
int main(){
    a=10;
    f();
    cout<<a<<endl; →
}
```

10
12
12

WOW!

Dt
Pg

$(36/6)*6 \rightarrow (6*6) \rightarrow 36$

Inline and Default Arguments

①
```
int a,b;
cin>>a>>b;
int c;
if(a>b){
    c=a;}
else{c=b;}
```

② Tertiary opr
```
(a>b)? a:b;
```
Condition T F
```
int ans=(a>b)? a:b;
```

③ Using functions to avoid code repetition.

inline ↓ single line fn

① output file is little
② not depends on compiler

```
inline int max(int a, int b){
    return (a>b)? a:b;
}
main(){
    int a,b; cin>>a>>b;
    int c= max(a,b);
    int x,y;
    x=12;
    y=34;
    int z= max(x,y);
}
```

1) Performance hit Not happen } declare f^n as inline f^n → whole body of the f^n gets copied
2) Code readable

Dt
Pg

Default Arguments

```
int sum(int a[], int size, int start = 0){
    int ans = 0;
    for(int i = start; i < size; i++){
        ans += a[i]; }
    return ans;
}

main(){
    int a[20];
    cout << sum(a, 20) << endl;
}
```

```
int sum2(int a, int b = 0, int c = 0, int d = 0){
    return a+b+c+d;
}
```

sum2(a , — , — , —)

Rightmost first

✗ error

```
int main(){
    int a=1, b=2, c=3, d=4;
    cout << sum2(a, b) << endl;
}
```

Correct

a — b = 0 — c = 0 — d = 0

Rightmost ~~bit~~

✓

Inline f^n's are used to reduce function call overhead. They are expanded in a line when they are invoked.

WOW!

Dt
Pg

Constant Variables

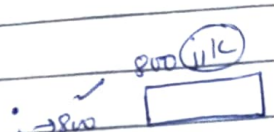
↓

They have fixed value

```
const int i=10;   → Not able to change it further
i=12;             → X error

const int i;
i=10;             → X error
```

} → Correct way

```
const int i=10;

int j=12;
const int &k=j;
k++;              → X
j++;

cout << k << endl;

int const i2=10;
const int i3=10;
```

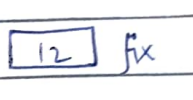


// constant ref to const int.

```
int const j2=12;
int const &k2=j2;
j2++;             → X Not changeable
k2++;             → X
```

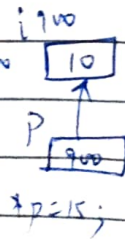
// Reference from a const int

```
int const j3=12;
int &k3=j3;       → X
k3++;             → X
j3++;             → X
```

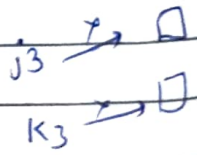


Const pointers

```
int const i=10;
const int *p=&i;
```




```
*p=15;
```



Benefit in F

Constant Poi

①

WOW!

Dt
Pg

```
int j=12;
int const *p2 = &j;
j++;      → ✓
(*p2)++;  → ✗
```

further

Benefit in Function Calling

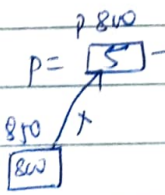
```
void f(int const *p){
    (*p)++; }

void g(int const &a){
    a++; }

main(){
    int j=12;
    int *p=&j;
    f(p);
}
```

Constant Pointers

```
int i=10, j=12;
int const *p = &i;
p = &j;
int * const p2 = &i;
(*p2)++;
p2 = &j;
int const * const p3 = &i;
p3 = &j;
(*p3)++;
```

RW

①



WOW!

Dt
Pg

Reference Variables
& → Address of opr

Heap Memory → memory not allocated in continous manner instead where it gets space & then links all memory blocks together

END OF TOPIC