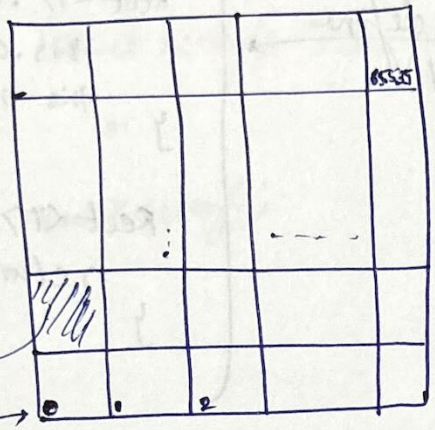
Static & Dynamic Memory Allocation

one (1) segment of Main memory.



smallest unit (Byte) having address

1 Byte = (8 Bit)

* (address)

Total Blocks = 65536

= 64 × 1024 Byte

KB

= 64 KB

(0/1)

8 Bit → 1 Byte → 2^8

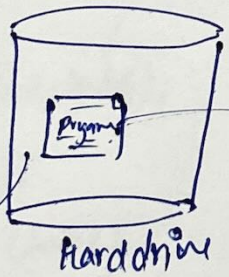
1024 Byte → 1 KB

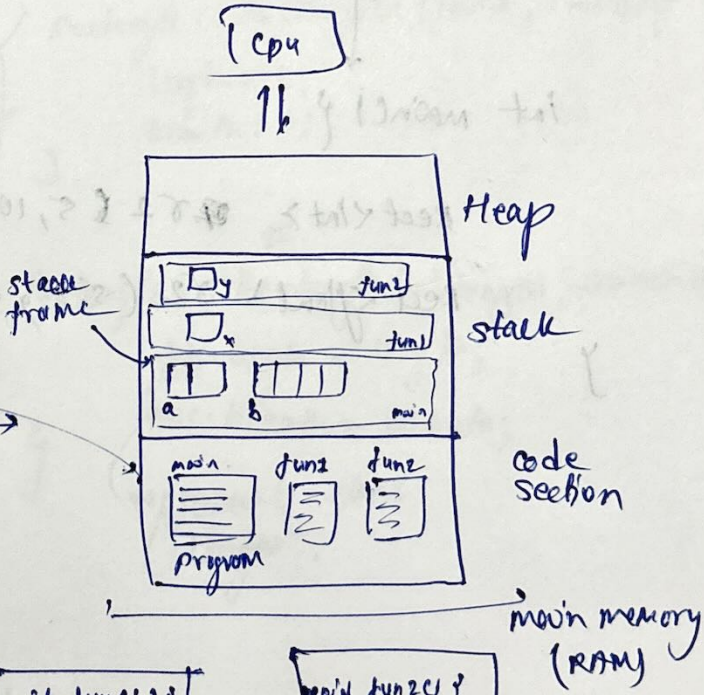
1024 KB → 1 MB

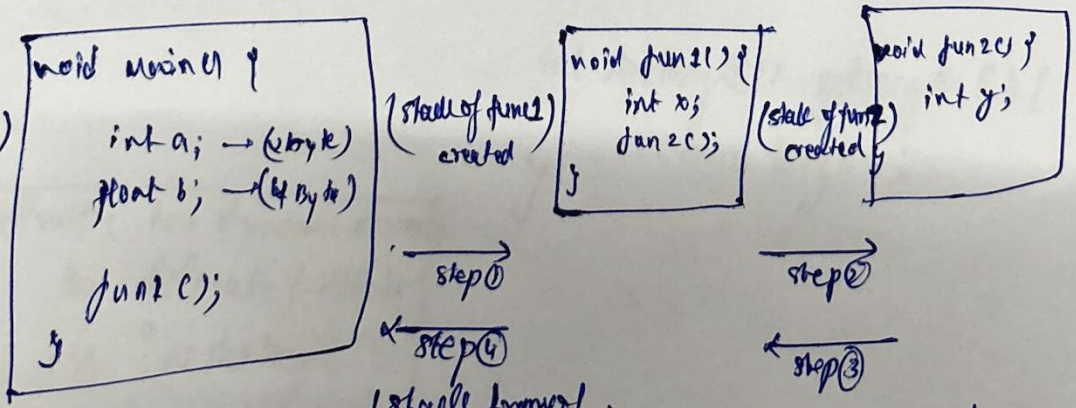
1024 MB → 1 GB

so on

Static memory Allocation







(Stacke of main) created

step ①

step ⑤

(Stacke frame of main) deleted

(Stacke of fun1 created

step ②

step ④

(Stacke frame of fun1 deleted)

(Stacke of fun2 created

step ③

(Stacke frame of fun2 deleted)

Dynamic Mem allocation

```
int Main(){
    int * p; → (8 Byte)
```

creating mem in heap

```
    p = (int *) malloc (sizeof(int) * 5);
```

or

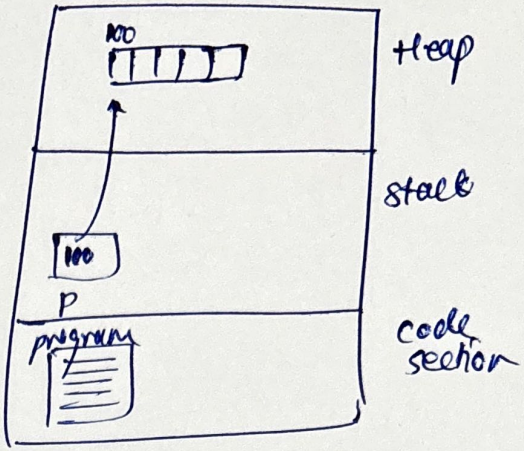
```
    p = new int[5];
```

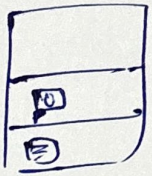
releasing after use

```
    delete [] p;   (cpp)
    free(p);       (C-lang)
```

otherwise memory will be full (memory leak)

```
    p = NULL;
};
```





Pointers

"int" datatype

address = 100

"pointer int" datatype → int *p = &a;

a = 5; → 5 (100)

given address of a (100)

return data type as "pointer int"

print(p) → 100

print(*(address)) → value at address

gives value at given address

Note

Actual address: 0x0abc

```
#include <typeinfo>
int a = 5;
typeid(0x0abc).name() → int
typeid(a).name → int
typeid(&a).name → Pi
```

"pointer int"

you cannot store

int *p; ⊗

p = 0x0

p = (int*) 0x0 ✓

typecast "int" to "pointer int".

1D Array

Array stored in "stack memory".

int Arr[3] = {0, 1, 2}; → How array store data?

Arr: name of variable → is actually address of first element

int *p;

p = Arr

p = &(Arr[0]) } → same thing.

p = &Arr

sizeof(3) represents its length.

Arr: | 0 | 1 | 2 |

100 104 108

(Arr+1) (Arr+2)

= 2 × (number of Bytes)

= 2 × 4

= 8

"int" takes 4 Bytes

print(Arr) → 100

print(&Arr) → 100

print(&(Arr[0])) → 100

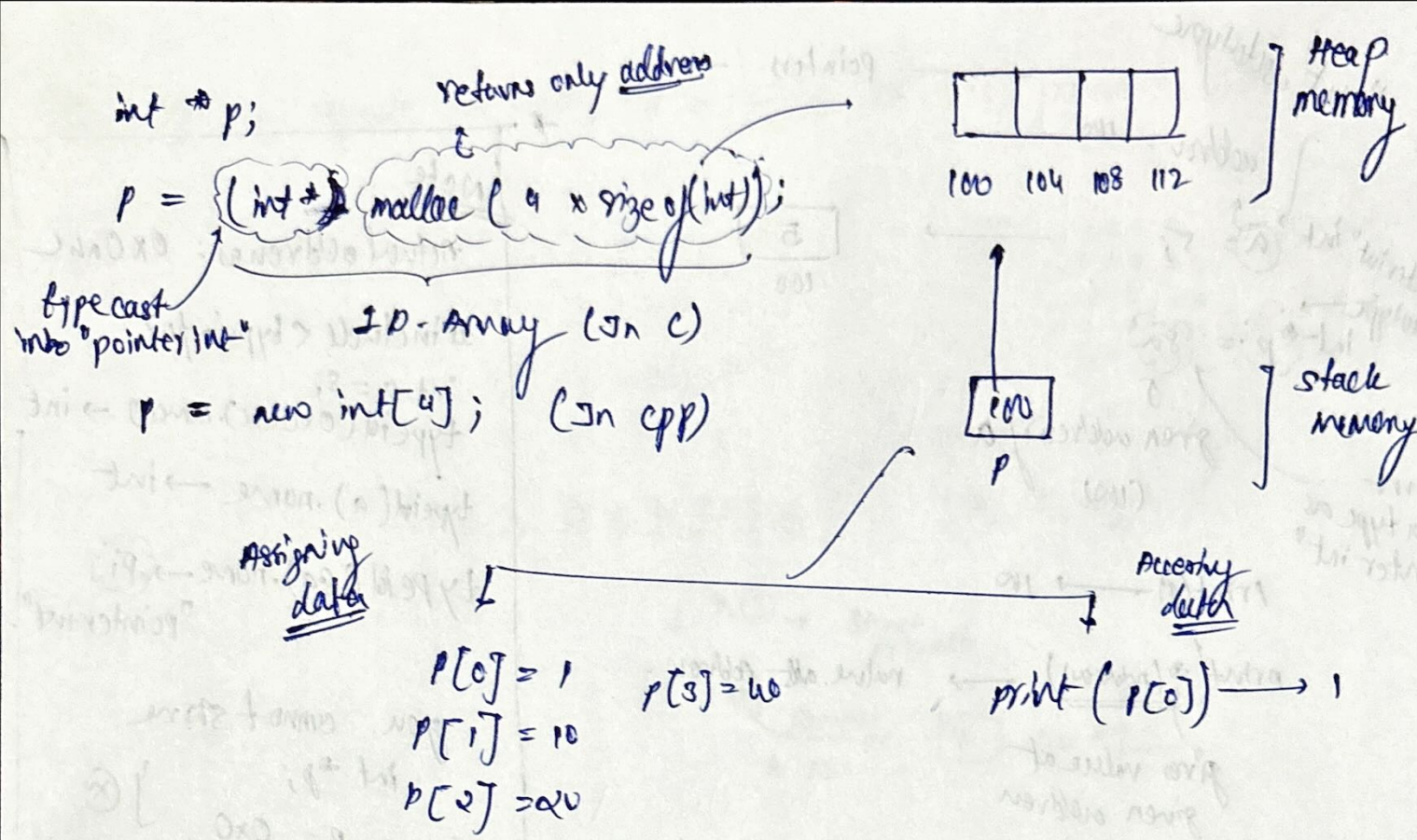
In reality addresses are in hexadecimals (eg 0x0123)

Size allocation of data type is compiler dependent.

(Eg int may 4 Byte)

can use pointer as similar as array

Accessing data	Assigning data
p[0] → 0	p[0] = 10
p[1] → 1	↓
p[2] → 2	p[0] → 10



Note

⇒ Once heap memory is allocated, if you used it then you should delete/release it too. (otherwise it will be present there & waste resources).

⇒ To delete/release for array/subset

delete [] p; (cpp)

free(p); (c)

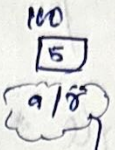
— 2D Array ——

↳ (will be covered in Arrays)

Reference

int a = 5;

⇒



two name for same address

int &r = a;

useful for parameter passing.

r = 10

int b = 20
r = b

print(a) ⟶ 20

print(a) ⟶ 10

struct

struct Rectangle {
int length;
int Breath;
}

→ to use it

int main() {
struct Rectangle r1;
r1.length = 10;
r1.Breath = 20;
}

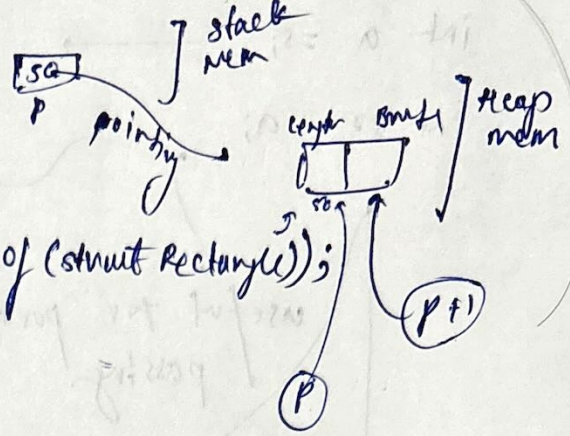
struct Rectangle r2 = {10, 20};

pointer

struct Rectangle *p;
p = &r2;

(*p).length = 35; or p → length = 35

To use it in heap memory

struct Rectangle *p; → 

p = (struct Rectangle *) malloc (sizeof (struct Rectangle));

or

p = new Rectangle;

Template class

```
#include <iostream>
using namespace std;
class Rectangle{
    private:
        int length
        int Breadth
    public:
        Rectangle(int l, int b);
        int getlength();
        void setlength(int l);
};
```

to create logic

```
Rectangle::Rectangle(int l, int b){
    length = l;
    breath = b;
}
```

or

```
Rectangle::Rectangle(int length, int breath){
    this->length = length;
    this->breath = breath;
}
```

represents class "length".

```
int Rectangle::getlength(){
    return length;
}
```

Note

If private has dynamic memory then to delocate/delete it use "~Rectangle();"
(first create it in class also)

```
template <class T>
class Rect {
    private:
        T a;
        T b;
    public:
        Rect(T a, T b);
        T getA()
};
```

T → (int / float / double / etc)

To define fn

```
template <class T>
Rect<T> :: Rect(T a, T b){
    this.a = a;
    this.b = b;
}
template <class T>
T Rect<T> :: getA(){
    return a;
}
```

To use this class

```
int main(){
    Rect<int> r1(5, 10);
    Rect<float> r2(5.2, 10.1);
}
```