


# C++ Pointers, Memory



# More Pointers

- A way to allocate/manage memory on the heap
  - A way to rapidly iterate over arrays
  - For C use malloc and free
  - For C++ use new and delete
- 
- For C
    - You have to use pointers
  - **For C++ ... Caution**
    - Pointers are the source of many, many bugs
    - Use Standard Library instead, it allocates and manages heap memory for you
  - To manage heap memory using pointers...Read on

# Pointers – Correct (ish)

Throw an exception and things go poorly

```
const int MY_SIZE = 10;

bool dynamic_good() {
    int *p = new int[MY_SIZE];

    //do some work

    //free if allocated
    if (p)
    {
        delete[] p;
        p = 0;
    }
}

int main()
{
    dynamic_good();
    return 0;
}
```



# Pointers – Correct (ish)

Throw an exception and things go poorly

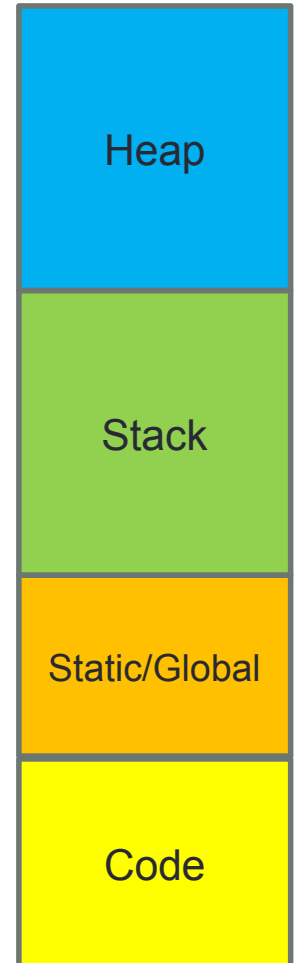
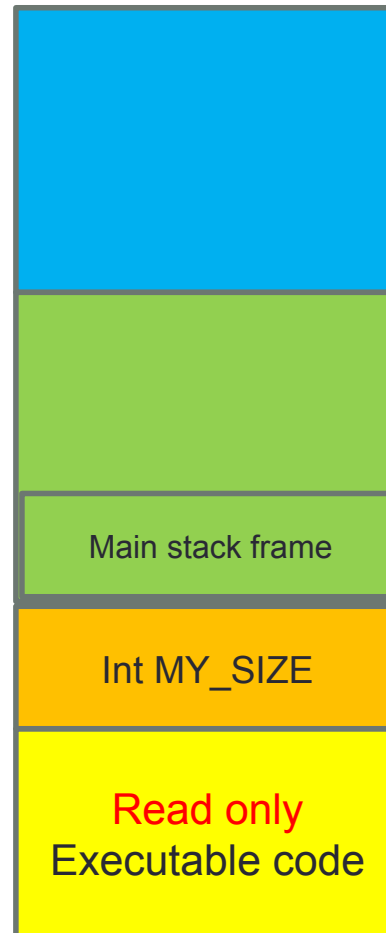
```
const int MY_SIZE = 10;

bool dynamic_good() {
    int *p = new int[MY_SIZE];

    //do some work

    //free if allocated
    if (p)
    {
        delete[] p;
        p = 0;
    }
}

int main()
{
    dynamic_good();
    return 0;
}
```



# Pointers – Correct (ish)

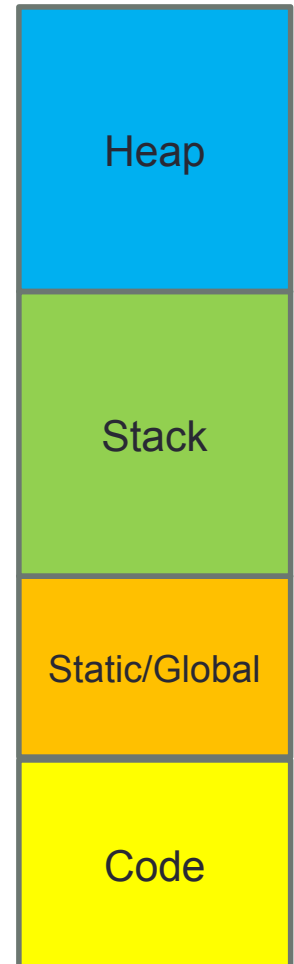
```
const int MY_SIZE = 10;

bool dynamic_good() {
    int *p = new int[MY_SIZE];

    //do some work

    //free if allocated
    if (p)
    {
        delete[] p;
        p = 0;
    }
}

int main()
{
    dynamic_good();
    return 0;
}
```



# Pointers – Correct (ish)

Throw an exception and things go poorly

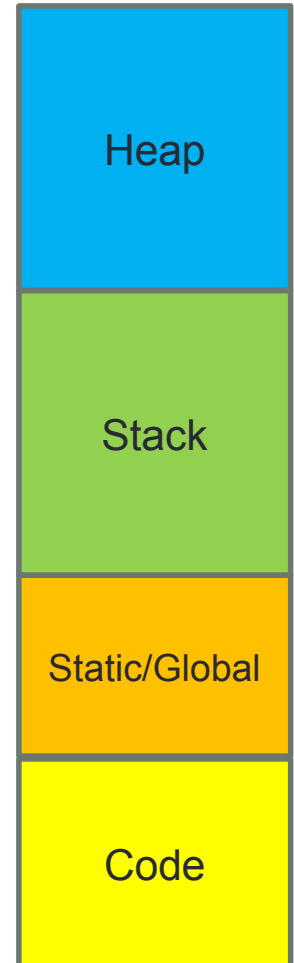
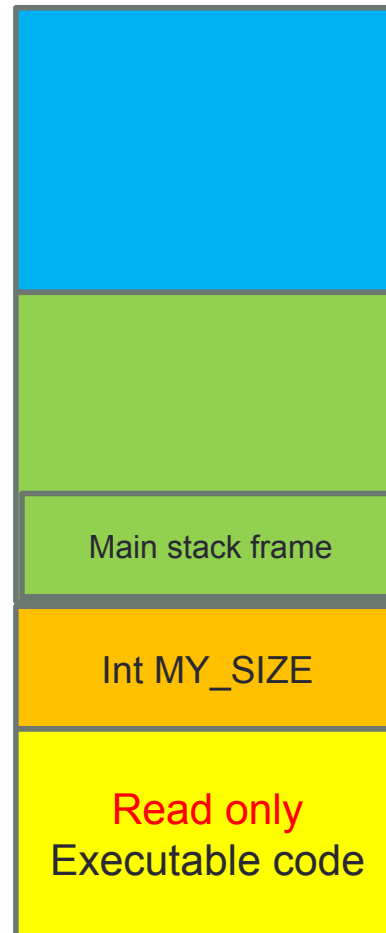
```
const int MY_SIZE = 10;

bool dynamic_good() {
    int *p = new int[MY_SIZE];

    //do some work

    //free if allocated
    if (p)
    {
        delete[] p;
        p = 0;
    }
}

int main()
{
    dynamic_good();
    return 0;
}
```



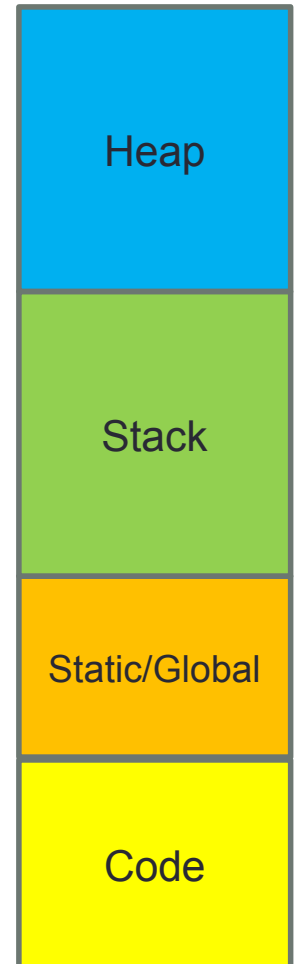
# Pointers - Dereference

```
void pointerDereference(){
    int    *pInt    = 0;
    int    *pInt2   = 0;
    int myInt[5]    = {0,1,2,3,4};

    //2 ways to set pointers
    //to an array
    pInt      = &myInt[0];
    pInt      = myInt;

    pInt2 = pInt;

    for (int i=0;i<5;i++)
    {
        cout<< *(pInt + i) <<" ";
    }
    cout<<std::endl;
}
```





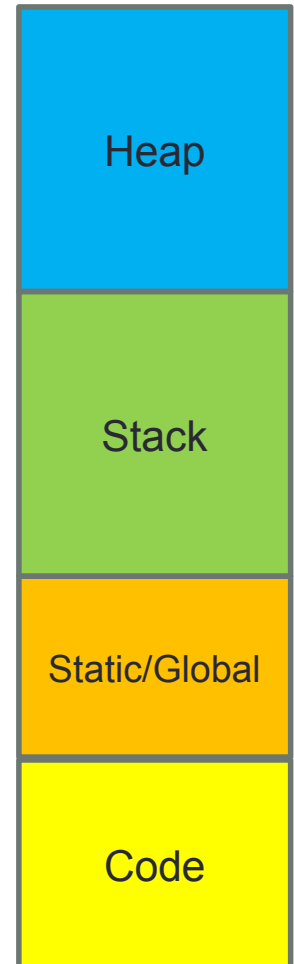
# Pointers – Dangling

```
int *p = new int[MY_SIZE];
if (!p)
    return false;

int *p2 = p;

if (p)
{
    delete [] p;
    p=0;
}

//what about p2?
```

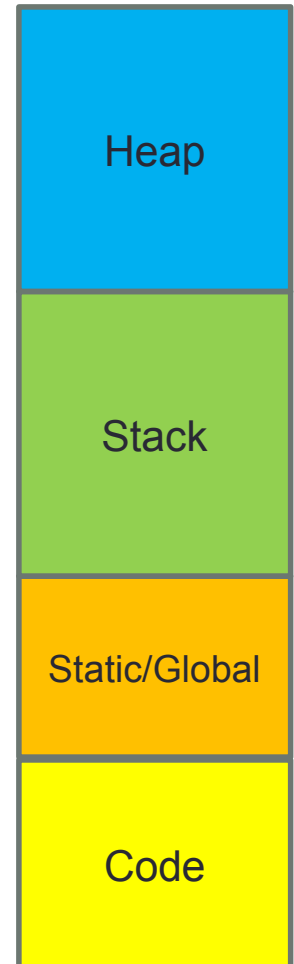


# Pointers – Memory Leak

```
const int MY_SIZE =10;

bool dynamic_memleak() {
    int *p = new int[MY_SIZE];
    if (!p)
        return false;

    return 0;
}
```



# Passing Pointers - review

```
char myString[] = "I am at an alpha low";  
char *pChar = myString;  
  
pointerByValue(pChar);  
pointerByRef(pChar);
```

```
//pointers by value  
void pointerByValue(char *myPointer){
```

```
//pointers by ref  
void pointerByRef(char *&myPointer){
```

# Pointers - different types

- Pointers to different types are different
- Cannot (for the most part) assign 1 to another

```
int *pInt =0;  
double *pdouble = 0;  
pInt = pdouble;
```



# Pointer tip

- If you create something using `new[]`
- You must delete using `delete[]`
- If you create something using `new`
- You must delete using `delete`

- **//Example**

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

```
delete p; //undefined behaviour, sometimes OK sometimes  
//not
```

# Conclusions

- Pointers are dangerous
- Please study this lecture, readings (especially intro one) and the example programs online
- We will see more pointers as we start on objects.