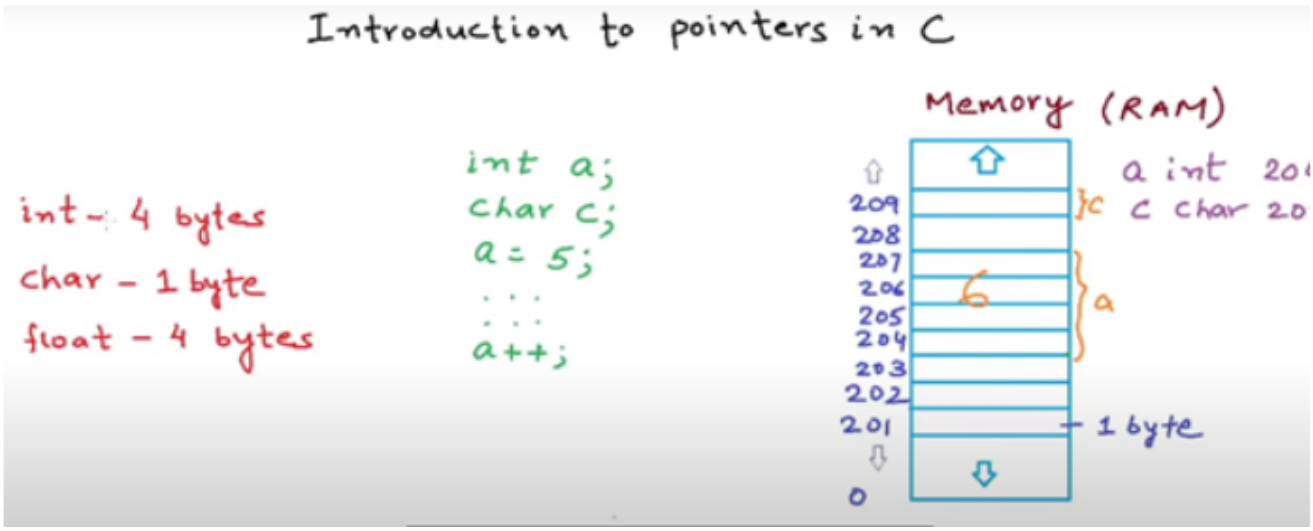
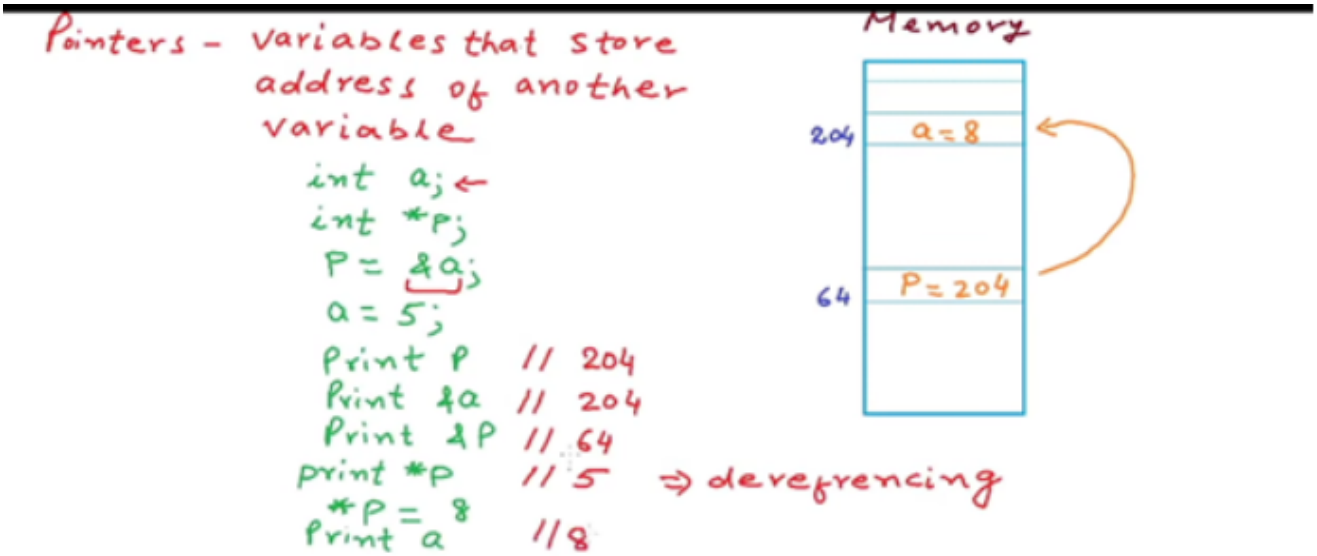
C/C++指针

Introduction to pointers



pointers-- variables that store address of another variable

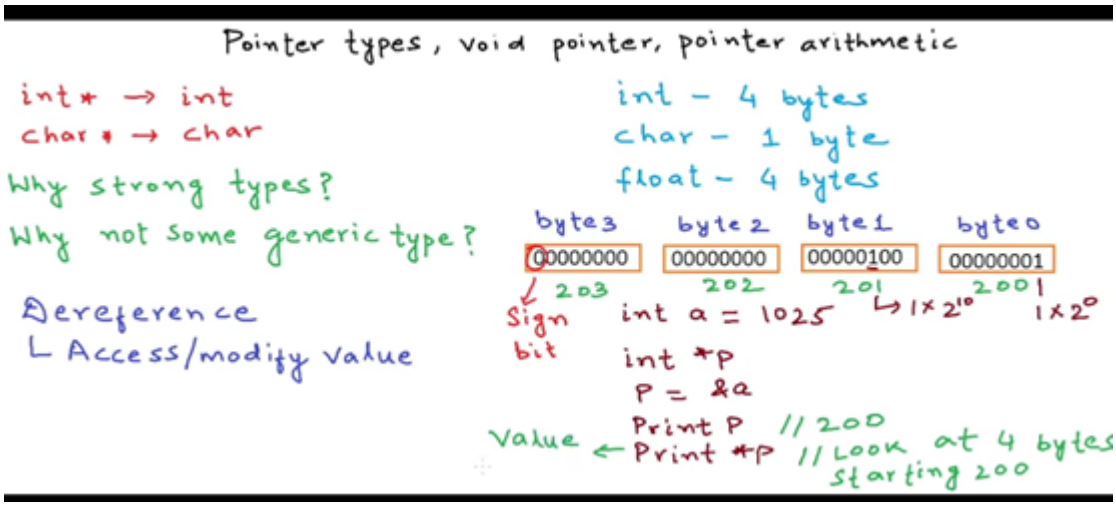


指针算数运算

```
//指针的算数运算
int a= 10;
int *p= &a;
cout<<p;  // assume p is 2002
cout<<p+1;  // p+1 is 2006
```

指针是有类型的

为什么指针有类型？因为我们要access和modify




```
//指针类型eexample
int main()
{
    int a=1025;
    int *p= &a;
    cout<<sizeof(int)<<endl;       //4byte
    cout<<p<<" "<<*p<<endl;        // p为a的地址，  *p为1025
    cout<<p+1<<" "<<*(p+1)<<endl; //*p+1为垃圾
    char * p0;
    p0= (char*)p;   // typecasting
    cout<<sizeof(char)<<endl;    //1byete
    cout<<p0<<" "<<*p0<<endl;    // *p0 为1 ，  二进制
    cout<<p0+1<<" "<< *(p0+1)<<endl; // p0+1 比p0多一个byte，*(p0+1)为4

}
```

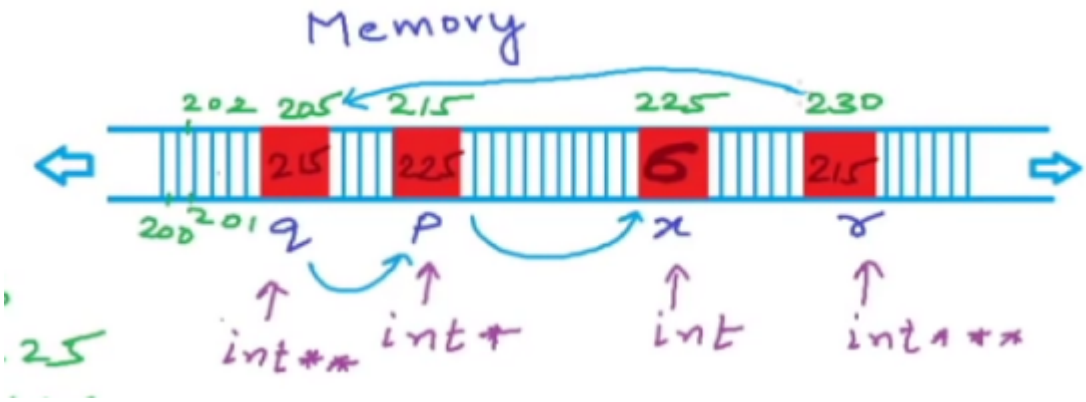
强制类型转换

void为genric pointer

```
int main()
{
    int a=1025;
    int *p =&a;
    void *p0;
    p0= p;
    cout<<p0<<endl;
    cout<<P<<endl;    //p0和p的地址相同，都是指向a
    *p0 // 空指针不能解引用
}
```

空指针不能解引用

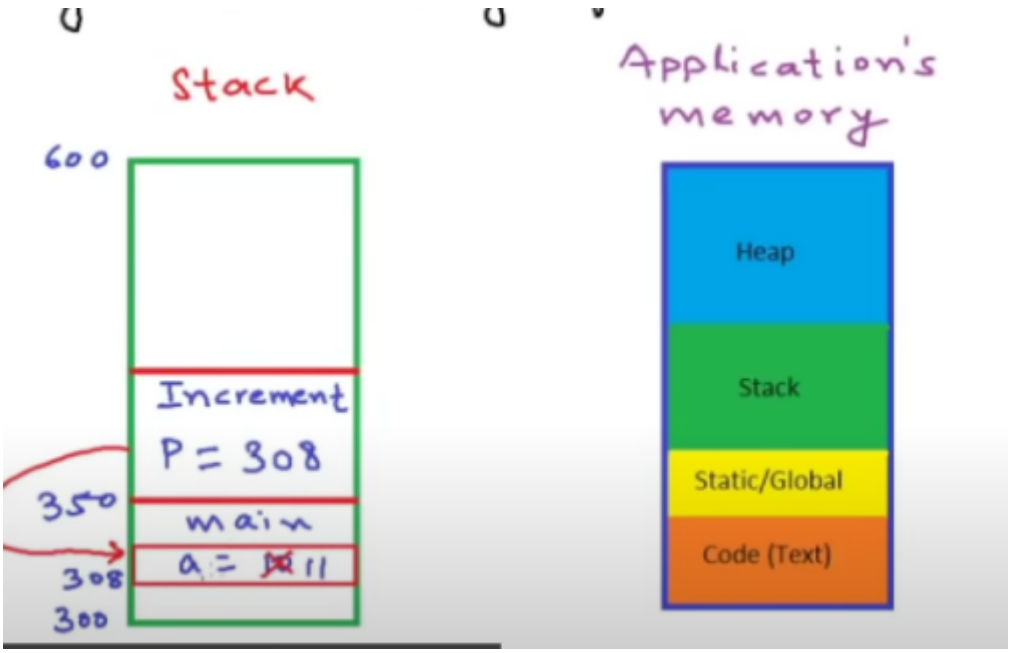
pointer pointer




```
int main()
{
    int x=5;
    int *p =&x;
    *p=6;       //x修改为6
    int **q = &p;      //q是指向p的指针 即指向int的指针的指针
    int ***r = &q;     //r是指向q的指针
    cout<<*p<<endl;  //*p=6
    cout<<*q<<endl;  //*q=225
    cout<<*(*q)<<endl; // *(*q)=6
    cout<<*(*r)<<endl;  //*(*r)=225
    cout<<*(*(*r))<<endl;  //6
}
```

对指针操作最好前边把括号加上，一是*运算符优先级不是很高，另外增加可读性， *p++结果不对

内存分布情况

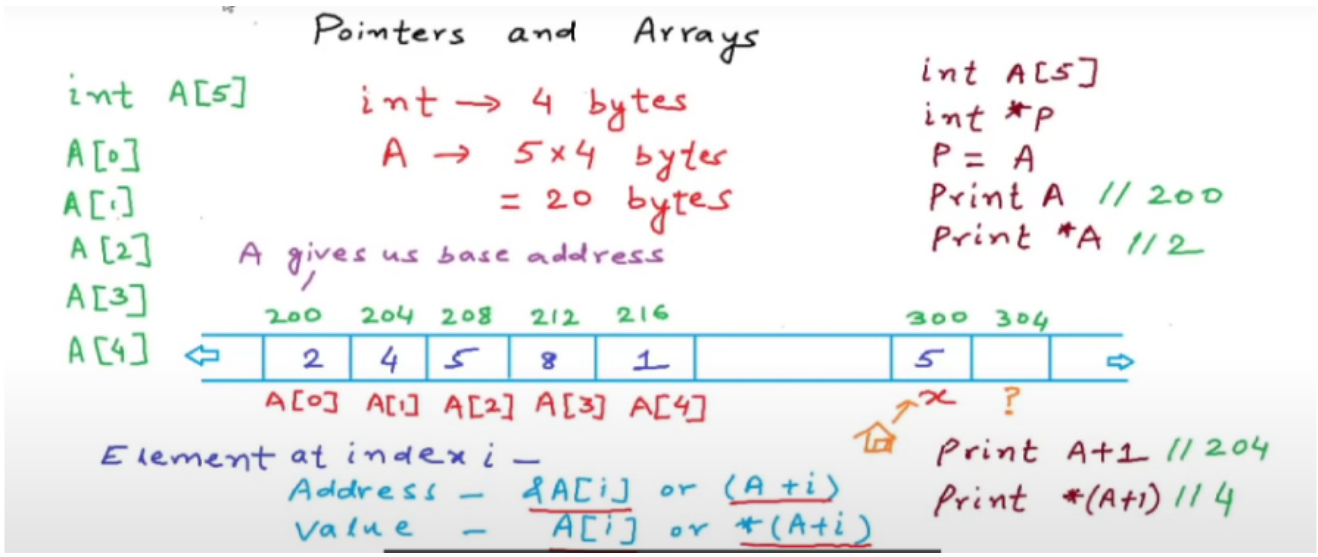


- 代码区存放写的代码指令
- static/global区存放全局和静态变量，周期为程序运行结束
- stack区为栈区，存放局部变量，(栈就是数据结构的栈)，不能grow，如果递归没出口可能溢出，stack overflow
- heap区，动态分配内存，可以grow，如果heap区开辟的数据没有及时delete/free,可能会导致内存泄露 memory leak，java和c#自动回收内存，不会导致内存泄露

  如果函数中传递的是指针，是pass by reference，节省空间，不用copy数据，只需要copy指针即可，但是copy的指针和原来的指针指向一个东西

pointers and array

数组



数组名就是数组第一个元素的地址，数组名不能修改A++ 错误

```
 int A[]{0,0,0}; int *p=A;
A++; //错误的
p++; // 可以的
```

数组长度问题

- 正确调用方法

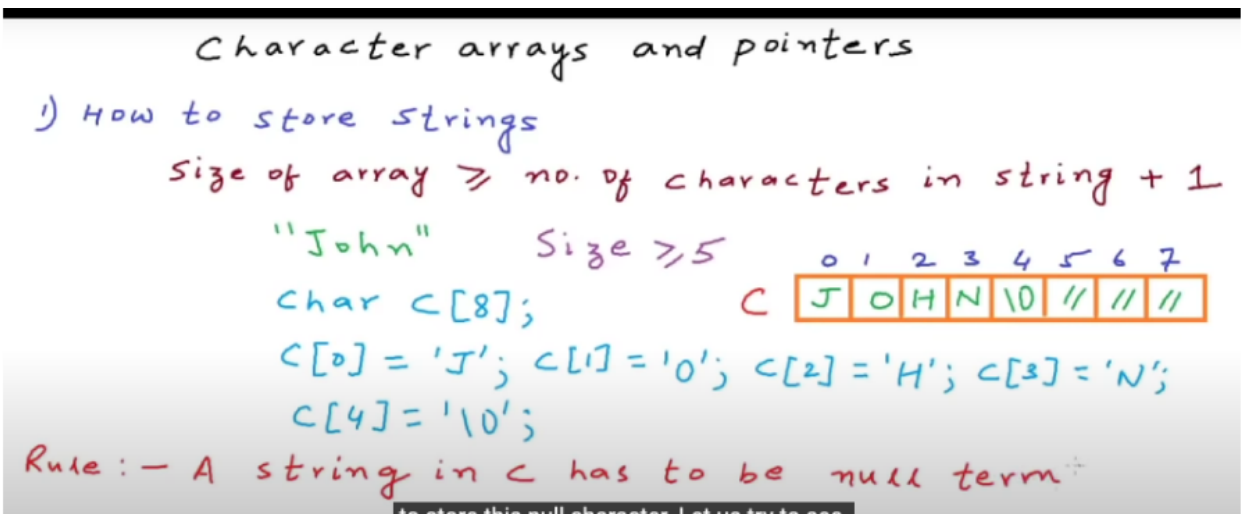
```
int main()
{
    int arr[] ={1,2,3,4,5};
    int total = sizeof(arr)/sizeof(arr[0]); //5
    test(arr,total)//  把total当做参数在调用端传递
}
```

- 错误调用方法

```
void test(int arr[])
{
    int total = sizeof(arr)/sizeof(arr[0]); // 1或者2
}
```

此时传进来的arr是一个指针，一个指针4个byte/8byte，而不是数组的长度

C风格字符



两种字符格式

- 格式1

```c
int main()
{
    char c[20];
    char[0] = 'J';
    char[1] = 'O';
    char[2] = 'H';
    char[3] = 'N';
    char[4] = '\0';  //c风格字符以\0结尾
}
```

只有以\0结尾的字符数组才叫字符串！

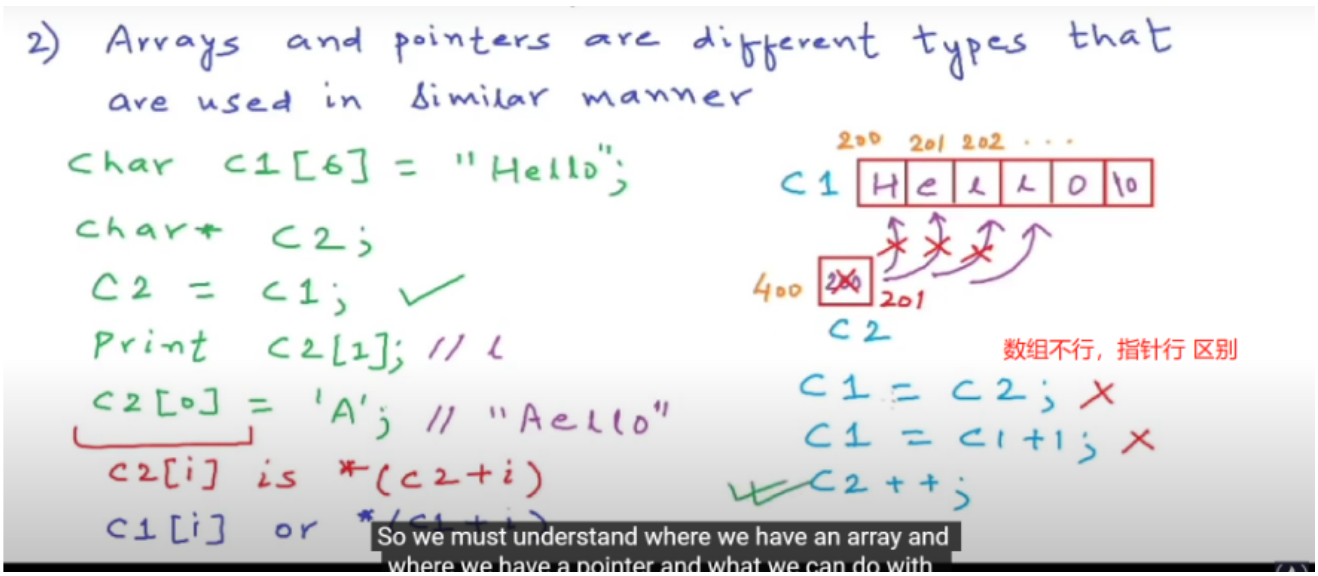
注意数组的长度一定是字符串的长度+1，多一个\0

- 格式2

```c
int main()
{
    char c[] ="JOHN";
}
```

这种字符串的形式隐含包括了\0

arrays and pointers are diffent types that used in similar manner



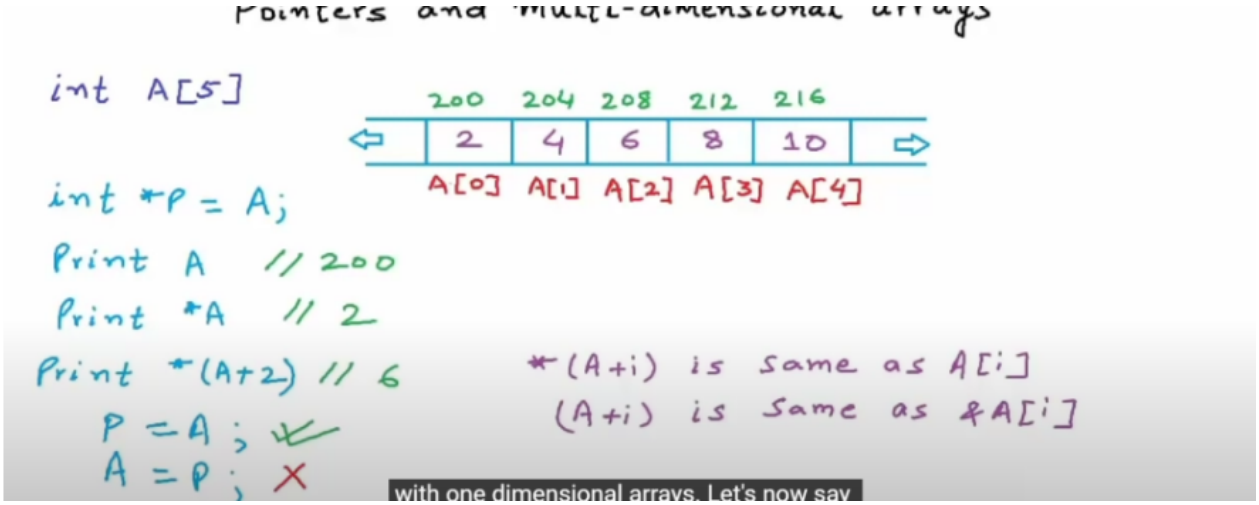
指针可以进行算数运算，数组名不可以

指针定义字符串

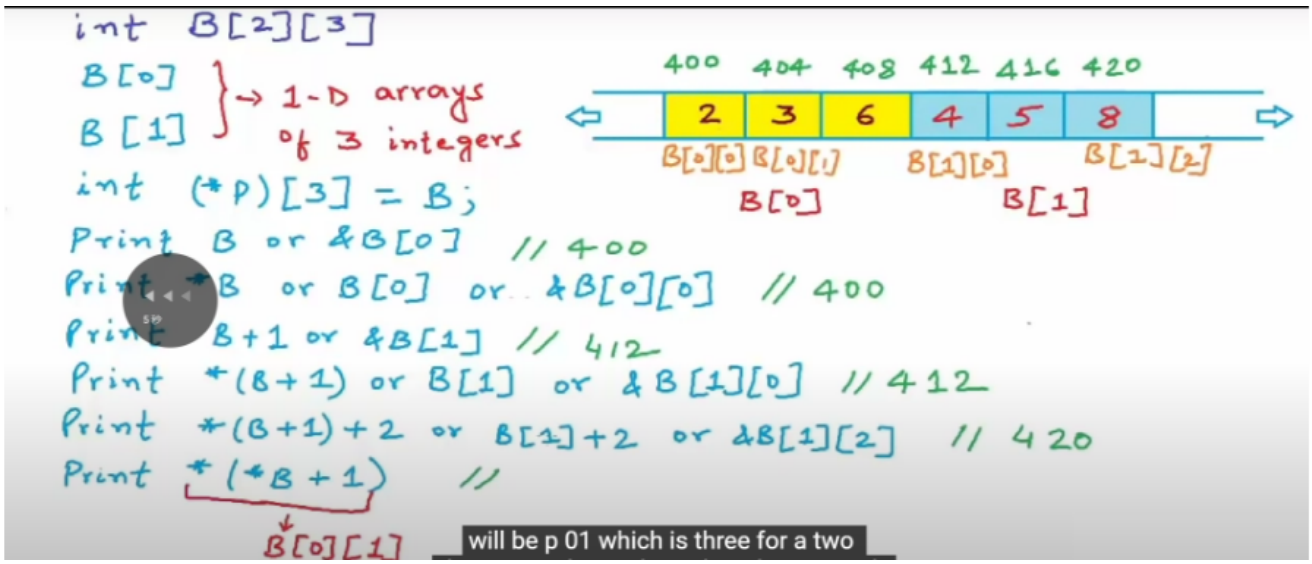
```
int main()
{
    const char* C = "HELLO"  //string gets stored as compile time constant
}

```

pointers and multi-dimension 数组

- 一维数组

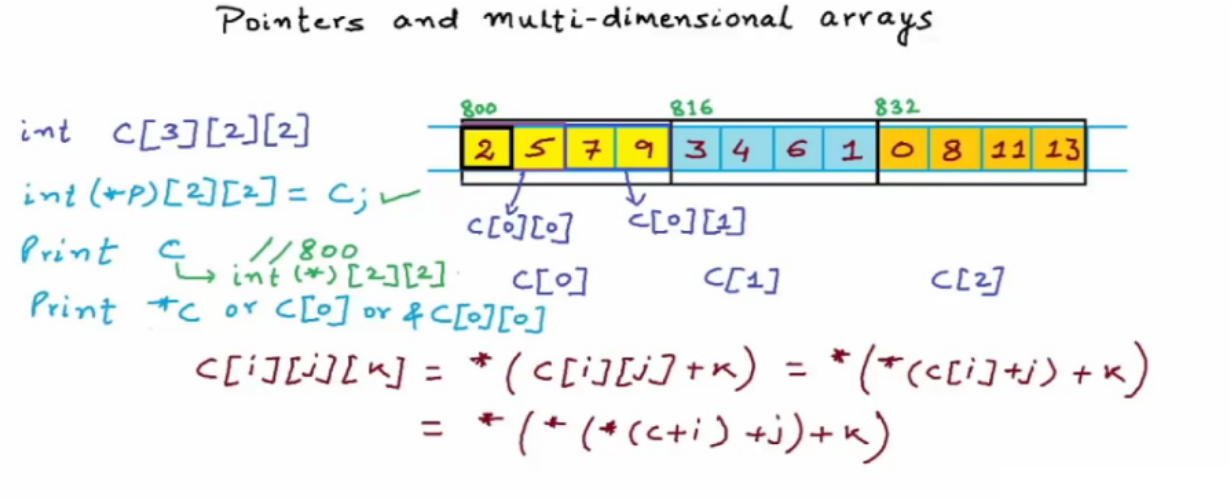


- 二维数组

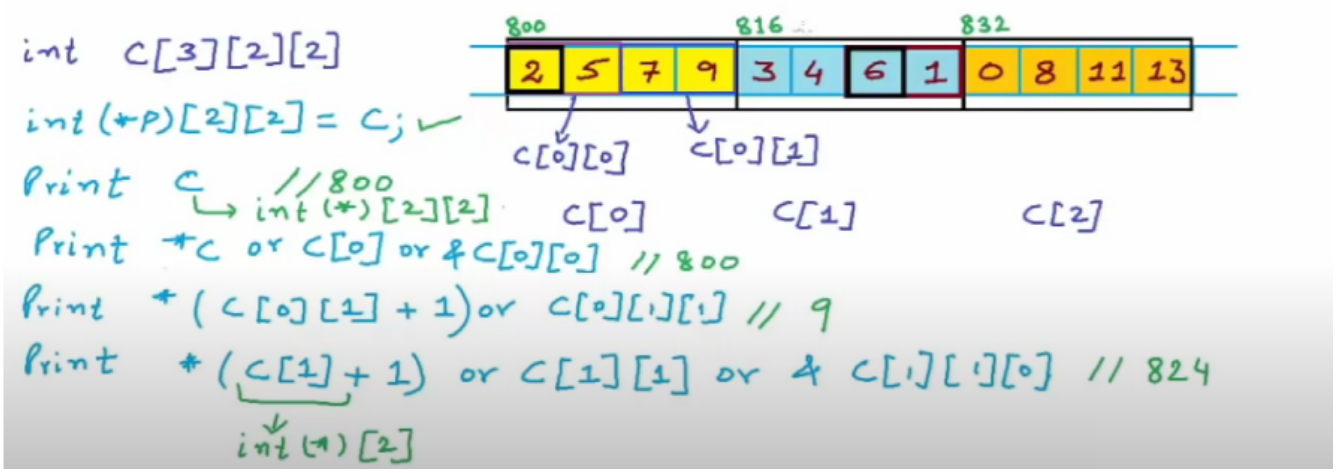


1. 如果想取值，几维数组就得有几个**和[]**
2. **对于二维数组 Bij = *(B[i]+j)* =(*(B+i)+j)**
3. int (*p)[3] =B ; ;对的， p是一个指针数组，数组里边都是指针， 如果 int *P=B 就不对了

- 三维数组



可以理解为有三个 2*2的数组

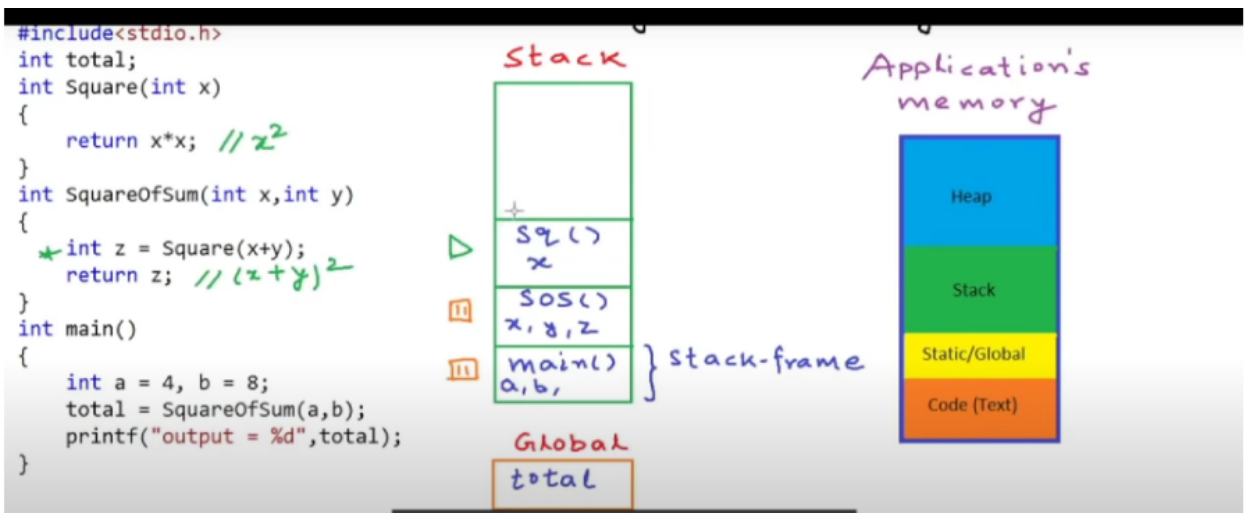


- 多维数组的函数传参方法

  1. void Func（int A[] [3]）; //传递二维
  2. void Func (int (*A)[2] [2]); //传递三维
  3. void Func (int ***A); //传递三维

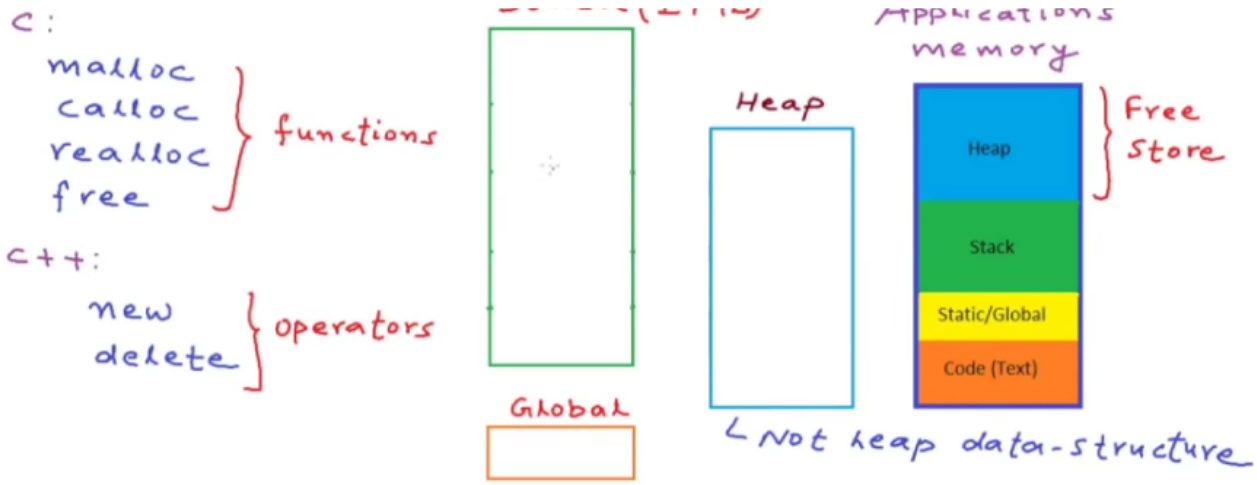
pointers and dynamic memory

- 压栈过程



堆区的内存在代码编译前就需要确定了，如果你想开辟数组，必须告诉确定的数组长度！！

数据结构的里边的堆和内存中的堆区没有任何关系



1. malloc 分配内存

```
int main()
{
    int a; int *p;
    p = (int*)malloc(sizeof(int));   // malloc 返回空指针，要强制类型转换
    free(p) ;  //释放内存
    p =(int*)malloc(20 * sizeof(int));  //malloc分配一个数组内存
}
```

c++的new和delete不用强制类型转换

2. calloc分配的内存自己初始化为0，而malloc初始化为垃圾值

   calloc -- void * calloc(size_t num, size_t size);

3. relloc，使用自己之前开辟的内存

```
int main()
{
    int *A =(int*)malloc(5*sizeof(int));
    for(int i=0; i<5;i++)
       A[i] = i+1;
    int *B = (int*)realloc(A,2*5*sizeof(int));

}
```

   - relloc 现在原有的地方内存扩展，看看够不够
   - 如果内存不够连续的话，copy原来的数值，去新地方扩展，并且把之前的内存deallocate
   - realloc(A,0)==free A , realloc(NULL,1)=malloc

Pointers and function return

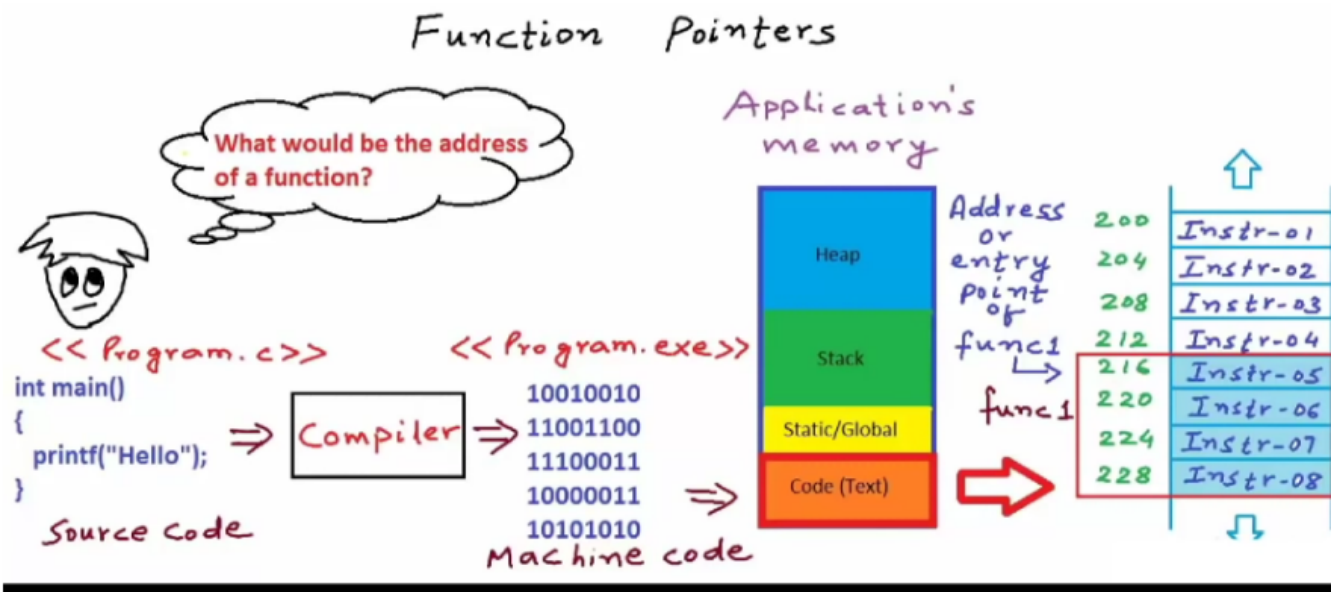
永远不要返回临时对象的指针或者引用用

```cpp
void printHelloWorld()
{
    cout<<"helloworld"<<endl;
}

int *Add(int *a, int *b)
{
    int c =(*a)+(*b);
    return &c;
}
int main()
{
    int a=2, b=4;
    int *ptr = Add(&a,&b);
    printHelloWorld();
    cout<<*ptr;             //结果32767
}
```

如果不打印helloworld，答案恰巧是6对的，因为堆区暂时没有被别的覆盖

Function pointers

函数的地址就是函数指令中第一条按指令的地址，指向函数的入口



函数指针的类型一定非常准确才行！

```
//Function Pointers in C/C++
#include<stdio.h>
int Add(int a,int b)
{
    return a+b;
}
int main()
{
    int c;
    int (*p)(int,int);
    p = &Add;
    c = (*p)(2,3); //de-referencing and executing the function.
```

注意不能写成 int *p(int,int)； 编译器会把它看成函数的声明， 也可以写成 p =Add，不加&*

- typedef 简化函数指针的定义 typedef int(*fp)(int a), fp就是一个新的类型，可以定义传递一个int，返回一个int的函数指针
- 可以定义一个函数指针数组 fp b[]{f1,f2,f3}; 可以通过遍历的方式来访问函数

---

callback

Function can be passed as arguments to functions

```
void A()
{
    cout<<"hello";
}
void B(void(*p)())
{
    p();
}
int main()
{
    void(*p)()= A;
    B(p);
    //或者直接
    B(A);
}
```

函数指针的用途

1. 算法调用，例如排序算法的比较方法传递

```
int compare(const void*a, const void *b)
{
    int A= *((int*)a);
    int B= *((int*)b);
    return A-B;
}
int main()
{
    int i{0},A[]{15,24,1,8,5,6};
    qsort(A,6,sizeof(int),compare);
    for(i=0;i<6;i++)cout<<A[i]<<" ";
}
```

qsort 可以sort任何类型的数组，因为void pointer的设计，只需要改compare

2. event handling 事件处理 ,类似于GUI的效果

常量指针，指针常量

常量指针