

C/C++指针

Introduction to pointers

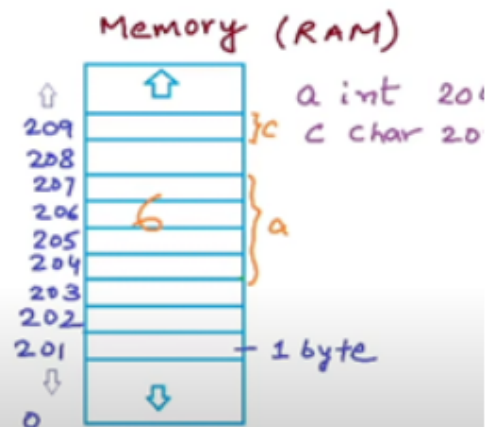
Introduction to pointers in C

int - 4 bytes

char - 1 byte

float - 4 bytes

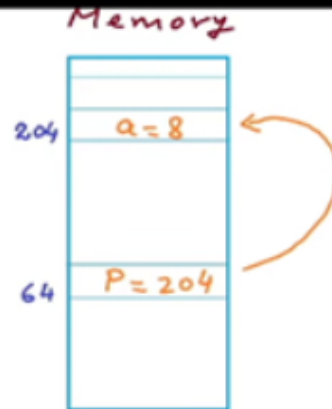
```
int a;  
char c;  
a = 5;  
...  
a++;
```



pointers-- variables that store address of another variable

Pointers - variables that store address of another variable

```
int a; ←  
int *p;  
p = &a;  
a = 5;  
Print p // 204  
Print &a // 204  
Print &p // 64  
print *p // 5 ⇒ dereferencing  
*p = 8  
Print a // 8
```



指针算数运算

```
1 //指针的算数运算  
2 int a = 10;  
3 int *p = &a;  
4 cout << p; // assume p is 2002  
5 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

Pointer types, void pointer, pointer arithmetic

`int*` → `int`
`char*` → `char`

Why strong types?
Why not some generic type?

Dereference
↳ Access/modify value

int - 4 bytes
char - 1 byte
float - 4 bytes

byte3 byte2 byte1 byte0
00000000 00000000 00000100 00000001
↓ 203 202 201 200
Sign bit int a = 1025 1×2^{10} 1×2^0

int *p
p = &a
Print P // 200
Print *p // Look at 4 bytes starting 200

```
1 //指针类型example
2 int main()
3 {
4     int a=1025;
5     int *p= &a;
6     cout<<sizeof(int)<<endl;           //4byte
7     cout<<p<<" "<<*p<<endl;           // p为a的地址, *p为1025
8     cout<<p+1<<" "<<*(p+1)<<endl; // *p+1为垃圾
9     char * p0;
10    p0= (char*)p; // typecasting
11    cout<<sizeof(char)<<endl;           //1byte
12    cout<<p0<<" "<<*p0<<endl;           // *p0 为1, 二进制
13    cout<<p0+1<<" "<<*(p0+1)<<endl; // p0+1 比p0多一个byte, *(p0+1)为4
14
15 }
```

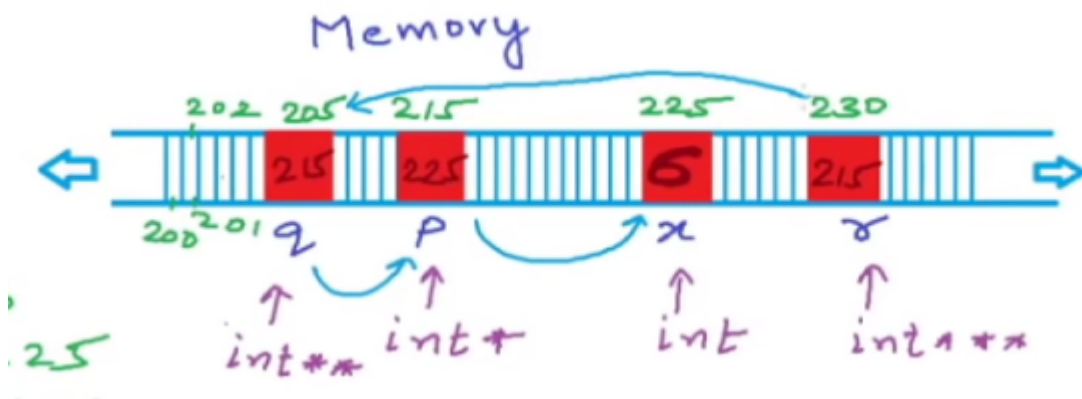
强制类型转换

void为generic pointer

```
1 int main()
2 {
3     int a=1025;
4     int *p =&a;
5     void *p0;
6     p0= p;
7     cout<<p0<<endl;
8     cout<<p<<endl; //p0和p的地址相同, 都是指向a
9     *p0 // 空指针不能解引用
10 }
```

空指针不能解引用

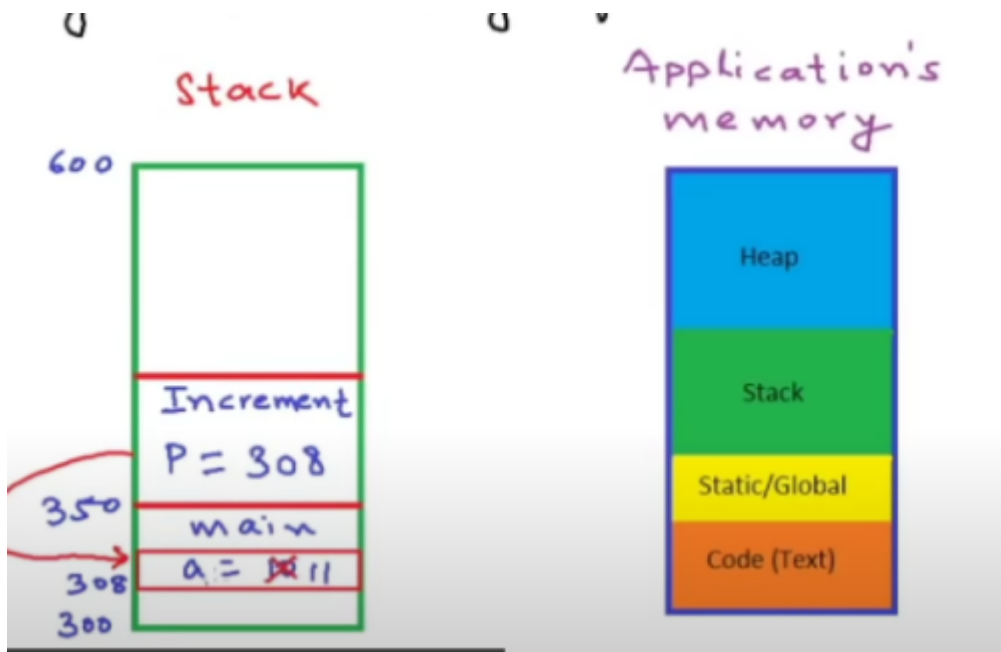
pointer pointer



```
1 int main()
2 {
3     int x=5;
4     int *p = &x;
5     *p=6;      //x修改为6
6     int **q = &p;    //q是指向p的指针 即指向int的指针的指针
7     int ***r = &q;    //r是指向q的指针
8     cout<<*p<<endl;  //p=6
9     cout<<*q<<endl;  //q=225
10    cout<<*(q)<<endl; // *(q)=6
11    cout<<*(r)<<endl;  //*(r)=225
12    cout<<*(*r))<<endl; //6
13 }
```

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

Pointers and Arrays

```

int A[5]
A[0]
A[1]
A[2]
A[3]
A[4]
  
```

int → 4 bytes
 A → 5 × 4 bytes = 20 bytes
 A gives us base address

	200	204	208	212	216	300	304
A[0]	2	4	5	8	1	5	

Element at index i –
 Address – $\&A[i]$ or $(A+i)$
 Value – $A[i]$ or $*(A+i)$

```

int A[5]
int *p
p = A
Print A // 200
Print *A // 2

Print A+1 // 204
Print *(A+1) // 4
  
```

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

```

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

```

数组长度问题

- 正确调用方法

```

1  int main()
2  {
3      int arr[] = {1,2,3,4,5};
4      int total = sizeof(arr)/sizeof(arr[0]); //5
5      test(arr,total)// 把total当做参数在调用端传递
6  }

```

- 错误调用方法

```

1  void test(int arr[])
2  {
3      int total = sizeof(arr)/sizeof(arr[0]); // 1或者2
4  }

```

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

Character arrays and pointers

1) How to store strings

Size of array \geq no. of characters in string + 1

"John" Size ≥ 5

char C[8];

C[0] = 'J'; C[1] = 'O'; C[2] = 'H'; C[3] = 'N'; C[4] = '\0';

Rule: - A string in C has to be null term

0	1	2	3	4	5	6	7
J	O	H	N	\0	//	//	//

两种字符格式

- 格式1

```

1 int main()
2 {
3     char C[20];
4     char[0] = 'J';
5     char[1] = 'O';
6     char[2] = 'H';
7     char[3] = 'N';
8     char[4] = '\0'; //c风格字符以\0结尾
9 }

```

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

```

1 int main()
2 {
3     char C[] = "JOHN";
4 }

```

这种字符串的形式隐含包括了\0

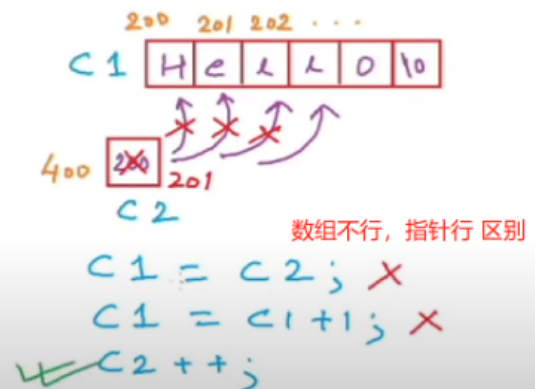
arrays and pointers are different types that used in similar manner

2) Arrays and pointers are different types that are used in similar manner

```

char C1[6] = "Hello";
char* C2;
C2 = C1; ✓
Print C2[1]; // L
C2[0] = 'A'; // "Aello"
C2[i] is *(C2+i)
C1[i] or *(C1+i)

```



So we must understand where we have an array and where we have a pointer and what we can do with

指针可以进行算数运算, 数组名不可以

指针定义字符串

```

1 int main()
2 {
3     const char* C = "HELLO" //string gets stored as compile time constant
4 }
5

```

pointers and multi-dimension 数组

- 一维数组

Pointers and multi-dimensional arrays

```

int A[5]
int *P = A;
Print A // 200
Print *A // 2
Print *(A+2) // 6
P = A; ✓
A = P; ✗

```

$*(A+i)$ is same as $A[i]$
 $(A+i)$ is same as $\&A[i]$

with one dimensional arrays. Let's now say

- 二维数组

```

int B[2][3]
B[0] } → 1-D arrays
B[1] } of 3 integers
int (*P)[3] = B;
Print B or &B[0] // 400
Print *B or B[0] or &B[0][0] // 400
Print B+1 or &B[1] // 412
Print *(B+1) or B[1] or &B[1][0] // 412
Print *(B+1)+2 or B[1]+2 or &B[1][2] // 420
Print *(B+1) //

```

$B[0][1]$ will be p 01 which is three for a two

- 如果想取值，几维数组就得有几个**和[]**
- 对于二维数组 $B_{ij} = (B[i+j]) = (*(B+i)+j)$
- `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

Pointers and multi-dimensional arrays

```
int C[3][2][2]
int (*P)[2][2] = C; ✓
Print C // 800
Print *C or C[0] or &C[0][0]
```

$$C[i][j][k] = *(C[i][j] + k) = *(*(C[i] + j) + k)$$

$$= *(* (C + i) + j) + k$$

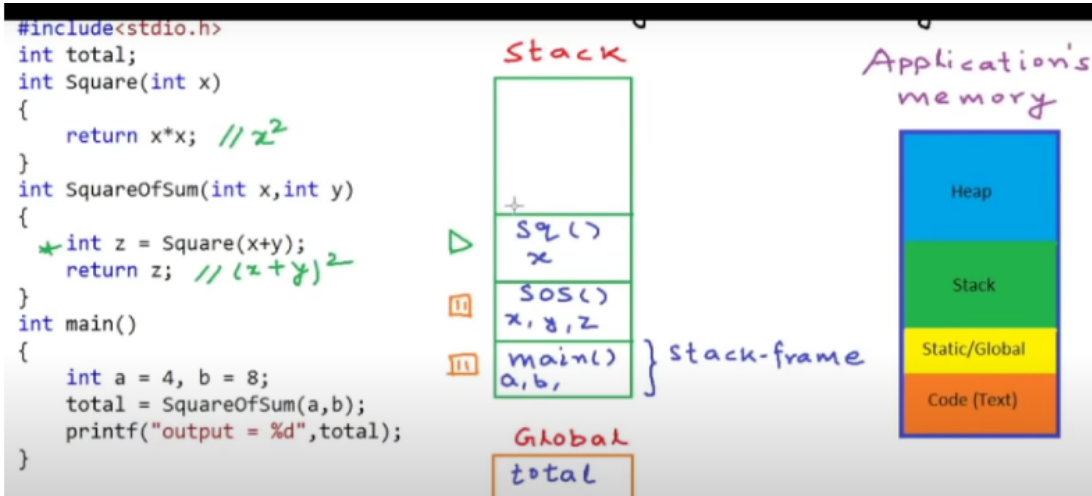
可以理解为有三个 2*2 的数组

```
int C[3][2][2]
int (*P)[2][2] = C; ✓
Print C // 800
Print *C or C[0] or &C[0][0] // 800
Print *(C[0][1] + 1) or C[0][1][1] // 9
Print *(C[1] + 1) or C[1][1] or &C[1][1][0] // 824
      ↓
    int (*)[2]
```

- 多维数组的函数传参方法
 - void Func (int A[] [3]) ; //传递二维
 - void Func (int (*A)[2] [2]); //传递三维
 - void Func (int ***A); //传递三维

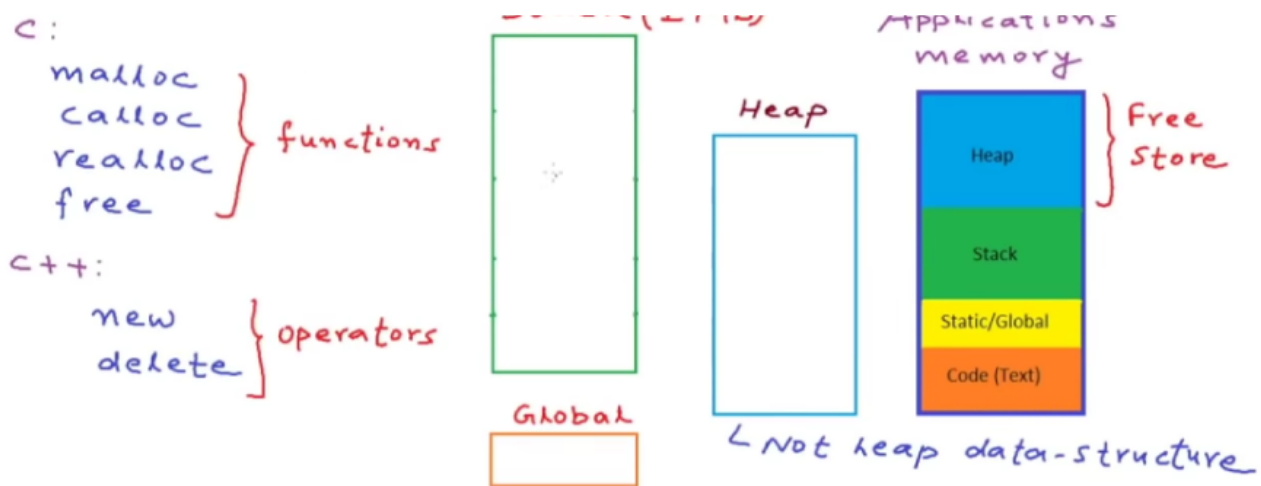
pointers and dynamic memory

- 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

```
1 int main()
2 {
3     int a; int *p;
4     p = (int*)malloc(sizeof(int)); // malloc 返回空指针，要强制类型转换
5     free(p); //释放内存
6     p = (int*)malloc(20 * sizeof(int)); //malloc分配一个数组内存
7 }
```

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. realloc，使用自己之前开辟的内存

```
1 int main()
2 {
3     int *A = (int*)malloc(5*sizeof(int));
4     for(int i=0; i<5; i++)
5         A[i] = i+1;
6     int *B = (int*)realloc(A, 2*5*sizeof(int));
7
8 }
```

- o realloc 现在原有的地方内存扩展，看看够不够
- o 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
- o 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

```

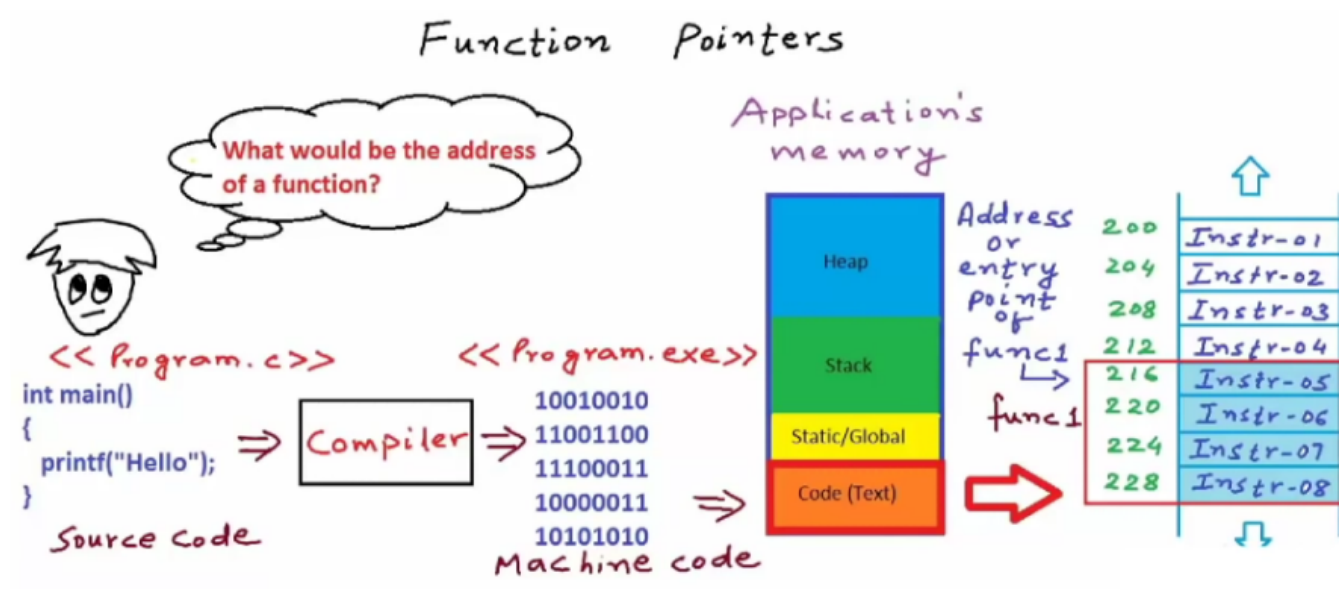
1 void printHelloWorld()
2 {
3     cout<<"helloWorld"<<endl;
4 }
5
6 int *Add(int *a, int *b)
7 {
8     int c =(*a)+(*b);
9     return &c;
10 }
11 int main()
12 {
13     int a=2, b=4;
14     int *ptr = Add(&a,&b);
15     printHelloWorld();
16     cout<<*ptr;           //结果32767
17 }

```

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

```
1 void A()
2 {
3     cout<<"hello";
4 }
5 void B(void(*p)())
6 {
7     p();
8 }
9 int main()
10 {
11     void(*p)() = A;
12     B(p);
13     //或者直接
14     B(A);
15 }
```

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

```
1  int compare(const void*a, const void *b)
2  {
3      int A= *((int*)a);
4      int B= *((int*)b);
5      return A-B;
6  }
7  int main()
8  {
9      int i{0},A[]{15,24,1,8,5,6};
10     qsort(A,6,sizeof(int),compare);
11     for(i=0;i<6;i++)cout<<A[i]<<" ";
12 }
```

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](#)