中山大學
SUN YAT-SEN UNIVERSITY

指针

中山大学计算机学院

讲课人：万海

目录

CONTENTS

背景

任何程序数据载入内存后，在内存都有他们的地址。
电脑维修师傅眼中的内存是这样的：内存在物理上是由一组DRAM芯片组成的。

背景

作为一个程序员，我们不需要了解内存的物理结构。操作系统将RAM等硬件和软件结合起来，给程序员提供的一种对内存使用的抽象。

所以程序员眼中的内存，是这样子的：

0	1	2	3	4	...	...	268435454	268435455

可以理解为，内存是一个线性的字节数组。每一个字节都是固定的大小，由8位二进制组成。每一个字节都有一个唯一的编号，编号从0开始，一直到最后一个字节。上图是一个256M的内存，他一共有256x1024x1024 = 268435456个字节，那么它的地址范围就是 0 ~268435455 。

由于内存中的每一个字节都有一个唯一的编号，因此，在程序中使用的变量，常量，甚至函数等数据，当他们被载入到内存中后，都有自己唯一的一个编号，这个编号就是这个数据的地址。指针就是这样形成的。

指针的概念

在C语言中，每一个变量都有一个内存位置，每一个内存位置都定义了可使用 & 运算符访问的地址，它表示了在内存中的地址。

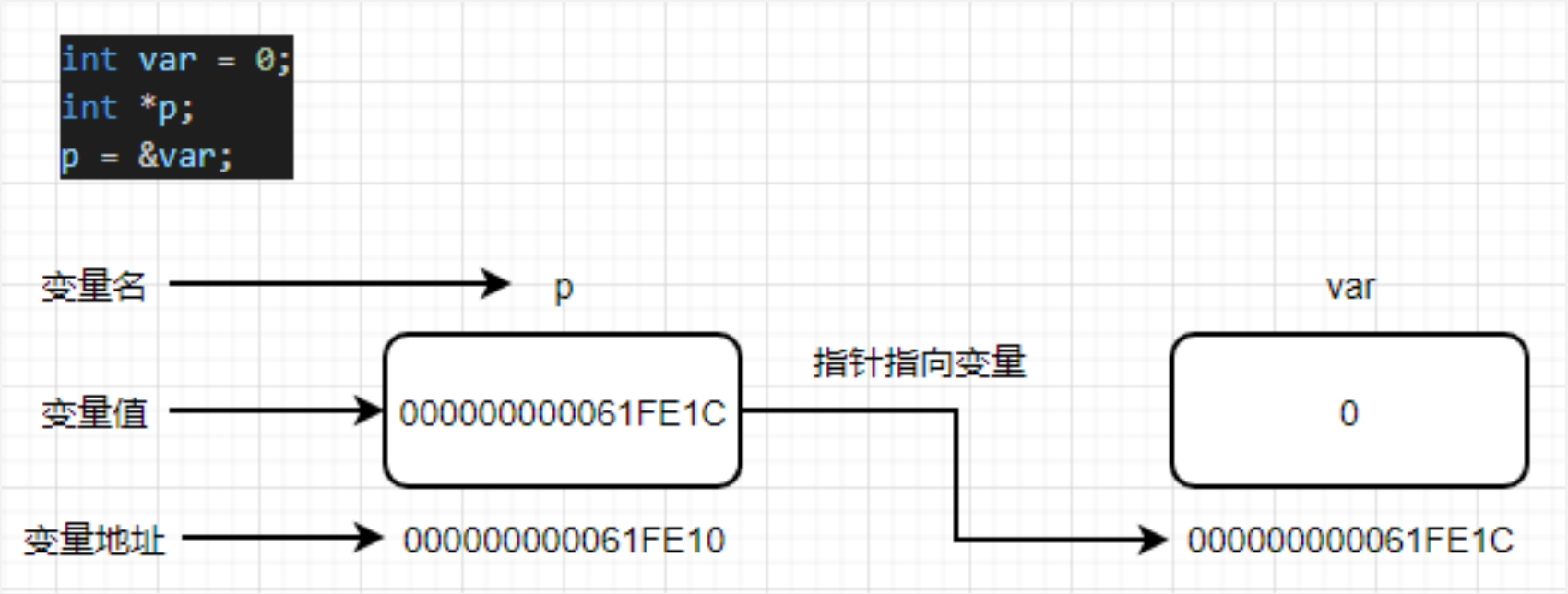
```c
#include <stdio.h>

int main() {
    int var = 0;
    int *p;
    p = &var;
    printf("var变量的值: %d\n", var);
    printf("var变量的地址: %p", p);
    return 0;
}
```

输出结果:
var变量的值: 0
var变量的地址: 000000000061FE1C

什么是指针？

指针也就是内存地址，指针变量是用来存放内存地址的变量。

```
int var = 0;
int *p;
p = &var;
```



指针变量的定义

指针变量声明的一般形式为：

变量类型 *变量名

如：

```
int             *p1;    /* 整形指针 */
double          *p2;    /* 双精度浮点型指针 */
char            *p3;    /* 字符型指针*/
```

注意：

1. 星号(*)是用来指定一个变量是指针。
2. 所有实际数据类型，对应指针的值的类型都是一样的，都是一个代表内存地址的长的十六进制数。
3. 不同数据类型的指针之间唯一的不同是，指针所指向的变量或常量的数据类型不同。

如何使用指针？

指针常用的操作：
1. 定义一个指针变量　2. 把变量地址赋值给指针
3. 访问指针变量中可用地址的值。
这些是通过使用一元运算符 * 来返回位于操作数所指定地址的变量的值。

```c
#include <stdio.h>

int main() {
    int var = 20;
    int *p;          /*定义一个指针变量*/
    p = &var;        /*把变量地址赋值给指针*/
    printf("var变量的值: %d\n", var);
    printf("var变量的地址: %p\n", p);
    /*使用指针访问变量的值*/
    printf("指针p指向地址中的值: %d", *p);
    return 0;
}
```

var变量的值: 20
var变量的地址: 000000000061FE14
指针p指向地址中的值：20

如何使用指针？

取地址操作：
既然有了指针变量，那就得让他保存其它变量的地址，使用& 运算符取得一个变量的地址。

```c
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int main() {
    int arr[3] = {1, 2, 3};
    int *p = arr;
    int (*fun_p)(int, int) = add;
    const char* str_p = "Hello world";
    return 0;
}
```

特殊的情况，他们并不一定需要使用&取地址：
1. 数组名的值就是这个数组的第一个元素的地址。
2. 函数名的值就是这个函数的地址。
3. 字符串字面值常量作为右值时，就是这个字符串对应的字符数组的名称,也就是这个字符串在内存中的地址。

NULL指针

在变量声明的时候，如果没有确切的地址可以赋值，为指针变量赋一个 NULL 值。NULL 指针是一个定义在标准库中的值为零的常量。

```c
#include <stdio.h>

int main () {
    int *p = NULL;
    printf("p 的地址是 %p\n", p);
    return 0;
}
```

p 的地址是 0000000000000000

void * 指针

由于void是空类型，因此void*类型的指针只保存了指针的值，而丢失了类型信息，我们不知道他指向的数据是什么类型的，只指定这个数据在内存中的起始地址，如果想要完整的提取指向的数据，程序员就必须对这个指针做出正确的类型转换，然后再解指针。因为，编译器不允许直接对void*类型的指针做解指针操作。

```c
#include <stdio.h>

int main() {
    void* void_p1;        //无类型指针
    int* int_p;
    void_p1 = int_p;      //任何指针都可以赋值给void指针
                          //只获得变量/对象地址而不获得大小
    void* void_p2;
    char* char_p = (char*)void_p2;
                          //void指针赋值给其他类型的指针时都要进行转换
}
```

注意：
1. void指针不能复引用
2. void指针不能参与指针运算，除非进行转换

指针的算术运算

一般来说，可以对指针进行四种算术运算：++、--、+、-。

```
1   #include <stdio.h>
2
3   const int MAXN = 3;
4
5   int main () {
6       int arr[MAXN] = {1, 2, 3};
7       int *p = &arr[0];
8       printf("p = %p\n", p);
9       printf("*p = %d\n", *p);
10      int *q = ++p;
11      printf("q = %p\n", q);
12      printf("*q = %d\n", *q);
13      return 0;
14  }
```

```
p = 000000000061FE04
*p = 1
q = 000000000061FE08
*q = 2
```

在执行一次++p后，q指向了下一个整形的位置，即q指向的地址在p指向的地址的基础上，增加了一个int的字节数。自增运算会在不影响内存位置中实际值的情况下，移动指针到下一个内存位置。

指针的比较

指针可以用关系运算符进行比较，如 ==、< 和 >。如果 p1 和 p2 指向两个相关的变量，比如同一个数组中的不同元素，则可对 p1 和 p2 进行大小比较。

```c
#include <stdio.h>

const int MAXN = 3;

int main () {
    int arr[MAXN] = {1, 2, 3};
    int *p = &arr[0];
    int i = 0;
    for (int *p = &arr[0]; p <= & arr[MAXN-1]; p++, i++)
        printf("*p[ %d ] = %d\n", i, *p);
    return 0;
}
```

```
*p[ 0 ] = 1
*p[ 1 ] = 2
*p[ 2 ] = 3
```

指针和数组

1. 数组名作为右值的时候，就是第一个元素的地址。
2. 指向数组元素的指针支持递增和递减运算。
（实质上所有指针都支持递增递减运算 ，但只有在数组中使用才是有意义的）
3. p = p + 1 意思是，让p指向原来指向的内存块的下一个相邻的相同类型的内存块。
4. p[n] == *(p+n)
   p[n][m] == *(*(p+n) + m)
5. 当对数组名使用sizeof时，返回的是整个数组占用的内存字节数。当把数组名赋值给一个指针后，再对指针使用sizeof运算符，返回的是指针的大小。

指针数组

```c
#include <stdio.h>

const int MAXN = 3;

int main () {
    int arr[MAXN] = {1, 2, 3};
    for (int i=0; i<MAXN; i++)
        printf("arr[ %d ] = %d\n",i, arr[i]);
    return 0;
}
```

```
arr[ 0 ] = 1
arr[ 1 ] = 2
arr[ 2 ] = 3
```

指针数组

```c
#include <stdio.h>

const int MAXN = 3;

int main () {
    int arr[MAXN] = {1, 2, 3};
    int *p[MAXN] = {NULL, NULL, NULL};
    for (int i=0; i<MAXN; i++)
        p[i] = &arr[i];
    for (int i=0; i<MAXN; i++)
        printf("*p[ %d ] = %d\n",i, *p[i]);
    return 0;
}
```

```
*p[ 0 ] = 1
*p[ 1 ] = 2
*p[ 2 ] = 3
```

指针数组

```c
#include <stdio.h>

const int MAXN = 3;

int main () {
    const char *str[MAXN] = {"SYSU", "is", "best"};
    for (int i=0; i<MAXN; i++)
        printf("str[ %d ] = %s\n",i, str[i]);
    return 0;
}
```

```
str[ 0 ] = SYSU
str[ 1 ] = is
str[ 2 ] = best
```

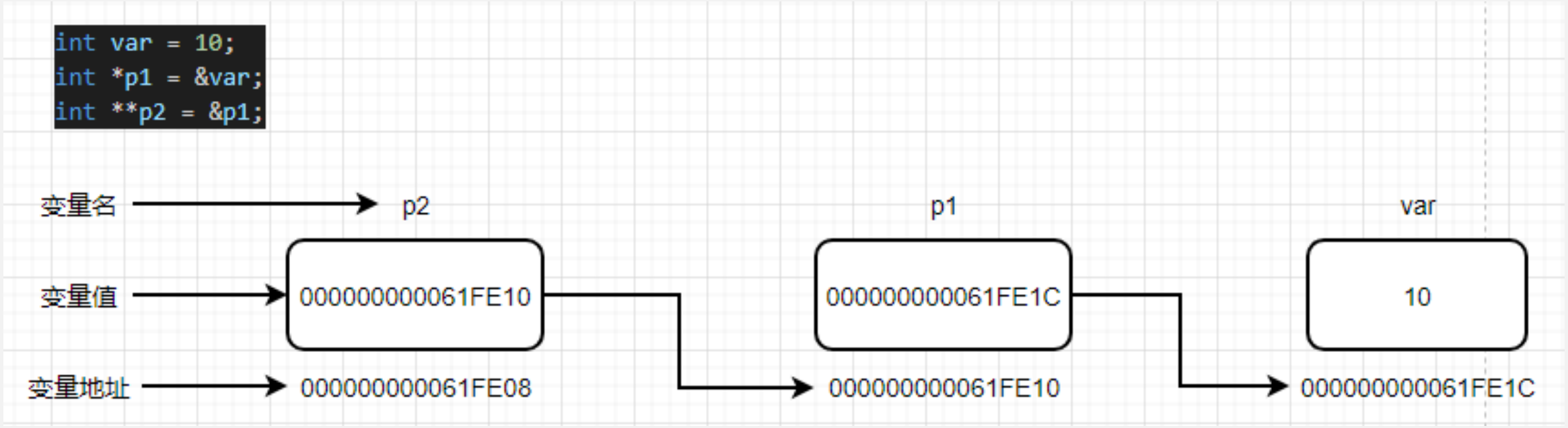
指向指针的指针

```c
#include <stdio.h>

int main () {
    int var = 10;
    int *p1 = &var;
    int **p2 = &p1;
    printf("p1 = %p\n", p1);
    printf("*p1 = %d\n", *p1);
    printf("p2 = %p\n", p2);
    printf("*p2 = %p\n", *p2);
    printf("**p2 = %d\n", **p2);
    return 0;
}
```

```
p1 = 000000000061FE1C
*p1 = 10
p2 = 000000000061FE10
*p2 = 000000000061FE1C
**p2 = 10
```

p2就是一个指向指针的指针

指向指针的指针



当一个目标值被一个指针间接指向到另一个指针时，访问这个值需要使用两个星号运算符。

传递指针给函数

C 语言允许您传递指针给函数，只需要简单地声明函数参数为指针类型即可。

```c
#include <stdio.h>

const int MAXN = 3;

void cswap(int *a, int *b) {
    int m = *a;
    *a = *b;
    *b = m;
}

int main () {
    int a = 3, b = 4;
    printf("a = %d, and b = %d\n", a, b);
    cswap(&a, &b);
    printf("a = %d, and b = %d\n", a, b);
}
```

```
a = 3, and b = 4
a = 4, and b = 3
```

指针可以直接操作它指向的数据。

传递指针给函数

同样的，C 语言也支持形参为数组的函数。

```c
#include <stdio.h>

const int MAXN = 3;

void inc(int *arr, int n) {
    for (int i=0; i<n; i++)
        arr[i]++;
}

int main () {
    int arr[MAXN] = {1, 2, 3};
    inc(arr, MAXN);
    for (int i=0; i<MAXN; i++)
        printf("arr[ %d ] = %d\n", i, arr[i]);
}
```

```
arr[ 0 ] = 2
arr[ 1 ] = 3
arr[ 2 ] = 4
```

从函数返回指针

同样的，C 语言也支持指针作为函数的返回值。

```
p[ 0 ] = 0
p[ 1 ] = 1
p[ 2 ] = 2
```

```c
#include <stdio.h>

const int MAXN = 3;

int* build(int n) {
    static int r[MAXN];
    for (int i=0; i<n; i++)
        r[i] = i;
    return r;
}

int main () {
    int *p;
    p = build(MAXN);
    for (int i=0; i<MAXN; i++)
        printf("p[ %d ] = %d\n", i, p[i]);
}
```

C 不支持在调用函数时返回局部变量的地址，除非定义局部变量为 static 变量。

因为局部变量是存储在内存的栈区内，当函数调用结束后，局部变量所占的内存地址便被释放了，因此当其函数执行完毕后，函数内的变量便不再拥有那个内存地址，所以不能返回其指针。

除非将其变量定义为 static 变量，static 变量的值存放在内存中的静态数据区，不会随着函数执行的结束而被清除，故能返回其地址。

函数指针

如果在程序中定义了一个函数，那么在编译时系统就会为这个函数代码分配一段存储空间，这段存储空间的首地址称为这个函数的地址。而且函数名表示的就是这个地址。既然是地址我们就可以定义一个指针变量来存放，这个指针变量就叫作函数指针变量，简称函数指针。

定义方式：
函数返回值类型 (* 指针变量名) (函数参数列表);
如：
int(*p)(int, int);

需要注意的是，指向函数的指针变量没有 ++ 和 -- 运算。

函数指针

```c
#include <stdio.h>

int Max(int a,int b) {
    if (a>b)
        return a;
    else
        return b;
}

int main() {
    int a = 5, b = 6;
    int (*p)(int, int);
    p = Max;
    int c = (*p)(a, b);
    printf("%d is bigger.", c);
    return 0;
}
```

```
6 is bigger.
```

具体的使用如下：

```
int Func(int x);     /*声明一个函数*/
int (*p) (int x);    /*定义一个函数指针*/
p = Func;            /*将Func函数的首地址赋
给指针变量p*/
```

回调函数

```c
#include <stdio.h>
#include <stdlib.h>

const int MAXN = 5;

int cmpfunc (const void * a, const void * b) {
   return ( *(int*)a - *(int*)b );
}

int main() {
    int n = 5;
    int arr[MAXN] = {5, 3, 1, 2, 4};
    qsort(arr, 5, sizeof(int), cmpfunc);
    printf("排序之后的列表: \n");
    for (int i=0; i<n; i++)
        printf("%d ", arr[i]);
}
```

排序之后的列表：
1 2 3 4 5

那么qsort是怎么知道如何排列的呢？这就要靠cmpfunc了。

如果我们想从大到小排列，只需要修改cmpfunc即可。

实际上，cmpfunc就是一个回调函数！

回调函数

```
void qsort(void *base,int nelem,int width,
           int (*fcmp)(const void *,const void *));
int fcmp(const void *,const void *);
```

调用函数有两种方法：
1. 直接调用：在函数A的函数体里通过书写函数B的函数名来调用之，使内存中对应函数B的代码得以执行。
2. 间接调用：在函数A的函数体里并不出现函数B的函数名，而是使用指向函数B的函数指针来使内存中属于函数B的代码片断得以执行。

被间接调用的函数就是回调函数。