第 2 章 Chapter 2

指针基础

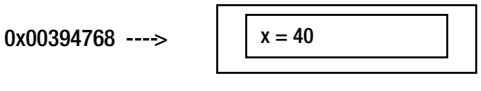
如同其他变量，你需要首先了解指针变量的基础知识，包括：声明、定义和用法。本章介绍指针变量的概念。重点关注指针用法，通过图文结合的方式让其概念直观。本章还介绍有关内存分配与内存释放的知识，以及如何操作指针变量等内部细节。

根据定义，指针是用于存储数据或函数的内存地址的变量，不同于其他数据类型变量，它仅存储值。与一般变量一样，指针也占用内存空间。我们将首先介绍引用 / 解引用变量的概念，这将有助于理解指针如何工作。

2.1 变量的地址

考虑下面的内容：

```
int x = 40;
```



上图表示如何用整型变量 x 存储值 40。对于某个程序，变量 x 什么也不是，仅仅是某些内存地址的存储位置。在上述情况下，将值 40 存储在 0x00394768 位置，这个位置被变量 x 引用。这也意味着，程序中能使用某些变量引用某些地址。如果你还记得第 1 章，每个程序都有代码段。函数也共享部分内存，在代码段其他部分加载函数本身。

在上述情况下，我们试图存储整数值，但请注意，内存地址也是数字或值。如果我们要在其他变量中存储这个数字，那将是什么？如果我们以某个变量存储或访问某个内存地址（如 0x00394768），就必须利

用指针这种特殊变量。

2.2 地址操作符

你可能想知道得到程序中所使用变量地址的方法。“取址”操作符（&）返回操作数的内存地址。地址操作符是一元操作符，适应于变量。下面的实例演示了该操作符如何获取赋值变量的地址。

源代码 . Ptr1.c

```
int main()
{
int var_int ;
printf("Insert data\n");
scanf("%d", &var_int);
return 0;
}
```

上例中，函数 scanf 使用“取址”操作符（&）得到存储用户输入值变量 var_int 的地址，scanf 函数需要知道输入值应该存储的地址。

得到变量的地址

如前所述，数据存储在内存单元中。以下程序演示了如何得到内存单元的地址或存储数据变量的地址。

源代码 . Ptr2.c

```
int main()
{
        int var_int = 40;
        printf ("Address of variable \"var_int\": %p\n", &var_int);
}
```

输出：

```
Address of variable "var_int": 00394768
```

上例中，我们使用 & 操作符得到变量的地址。

如果我们将地址的概念扩展到某个结构变量，该结构变量本身又包含许多其他变量，那么我们可以借助“取址”操作符得到其地址。

源代码 Ptr3.c

```
struct node{
int a;
int b;
};

int  main()
{
        struct node p;
        printf("Address of node = %p\n",&p);
        printf("Address of member variable a = %p\n", &(p.a));
        printf("Address of member variable b = %p\n", &(p.b));
        return 0;
}
```

输出：

```
Address of node = 003AFB00
Address of member variable a = 003AFB00
Address of member variable b = 003AFB04
```

注意上面的输出中数据结构的第一个成员的地址与第二个成员的地址非常相近。这意味着对任意数量的结构体内的成员字段，地址按顺序或依据其大小就近分配。

2.3 指针声明

现在你知道如何通过“取址”操作符获取地址。接下来让我们使用一个变量来存储这个地址。这个特定变量能存储和操作变量地址，称为指针变量。接下来的内容从指针变量声明开始。以下是声明指针变量的一般形式：

```
数据类型 *   变量名 ;
```

实例 1：一个指针变量能指向并存储原始数据类型的地址。

```
int* intptr, char* charptr
```

指针变量声明涉及一个名为解引用操作符 (*) 的特殊操作符，用于

帮助编译器识别它是一个指针变量。关联数据类型通知编译器它持有变量的数据类型地址的类型。解引用和“取址”操作符本质上都是一元的。

实例 2：声明指针聚合数据类型（结构体）

```
struct inner_node {
        int in_a;
        int in_b;
};

struct node{
        int *a;
        int *b;
        struct inner_node* in_node;
};
```

在上例中，struct inner_node * in_node 为指针变量，其中 struct inner_node 为数据类型，指针变量名为 in_node。如之前所述，我们也可将指针变量作为结构体的数据成员。

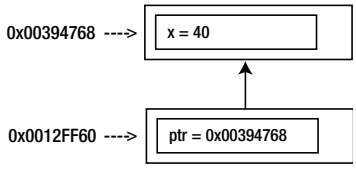
2.4 指针赋值

如同其他变量，指针变量声明时没有指向。程序员必须在解引用它之前让其指向有效的内存地址。我们很快就会讨论解引用的意义。

使用两种方式实现指针变量指向特定内存地址。

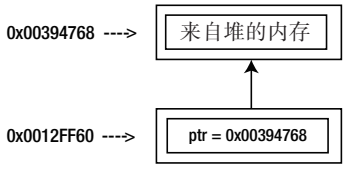
1. 利用指针地址（&）分配变量的地址。

```
int x = 40;
int *ptr;
ptr = &x;  // 使用取址操作符获取变量 x 的地址
```



2. 让指针变量指向来自堆的动态分配内存。

```
int * ptr;
ptr = ( int *) malloc(sizeof(int) * count );
```



情形 1，程序运行时根据变量范围分配内存存储变量 x 的值 40。回想下第 1 章中的内存排列部分。

情形 2，存储变量值的内存显然是通过调用 malloc 函数从堆中返回内存来创建的。

程序员切记任何指针变量的操作都必须在它指向一个有效内存地址的前提下完成，否则会引起分段错误。如果发生分段错误，会导致程序崩溃并最终停止运行。

2.5 指针变量大小

对于程序员来说变量大小是另一个关键问题。程序员应该知道变量使用时消耗多少。指针变量的大小可以是 32 位，也可以是 64 位的，这取决于开发平台。32 位平台下，指针变量大小为（int *、char *、float * 和 void *）4 个字节。事实上，存储聚合数据类型地址的指针变量大小（如数组和结构体）也为 4 个字节。显然指针变量的内存地址大小为 32 位。

下面的源代码给出不同类型指针变量（char *、int * 等）占用的内存大小。

源代码 . Ptr4.c

```
#include <stdio.h>
#include <conio.h>
int main()
{
        char c_var;
        int i_var;
```

```
        double d_var;
        char *char_ptr;
        int *int_ptr;
        double *double_ptr;
        char_ptr = &c_var;
        int_ptr = &i_var;
        double_ptr = &d_var;
        printf("Size of char pointer = %d value = %u\n", sizeof(char_ptr), char_ptr);
        printf("Size of integer pointer = %d value = %u\n", sizeof(int_ptr), int_ptr);
        printf("Size of double pointer = %d value = %u\n", sizeof(double_ptr),double_ptr);
        getch();
}
```

输出：

```
Size of char pointer = 4 value = 4061659
Size of integer pointer = 4 value = 4061644
Size of double pointer = 4 value = 4061628
```

利用指针变量验证其指向结构体变量所占内存大小是很有趣的。下面的代码说明了这点。

源代码 . Ptr5.c

```
#include <stdio.h>
#include <conio.h>
struct inner_node
{
        int in_a;
        int in_b;
};

struct node{
        int *a;
        int *b;
        struct inner_node* in_node;
};

int main()
{
        struct node *p;
        int *arrptr;
        int arr[10];
        arrptr = arr;
        printf("Size of pointer variable (struct node*) = %d\n",sizeof(struct node*));
        printf("Size of pointer variable pointing to int array = %d\n", sizeof(arrptr));
        return 0;
}
```

输出：

```
Size of pointer variable (struct node*) = 4
Size of pointer variable pointing to int array = 4
```

在上例中，struct node* 数据类型大小为 4 字节，这与内存地址大小始终为 4 个字节的事实相符。

2.6 指针解引用

你能存储和获取某个变量的地址，并将其成功存储到指针变量中，让我们来想想你能做什么。指针变量存储地址；使用“取值”操作符（*）（更准确些）访问该地址存储的值。这种特殊技术称为指针解引用。某些文献中也称为间接引用。后面的章节中你会看到使用指针变量的优势。

变量用来存储一个值，这条规则同样适用于指针变量。指针变量的值为某些内存单元的地址。一旦指针变量存储了内存地址，我们就能获得在该存储单元的值。让我们看看利用指针引用如何实现这个过程。

我们通过解引用操作符（*）得到某些内存单元的存储值。该操作符也称为“取值”操作符。分析下面的代码：

```
int x = 10; /* 某个内存单元存储值 10 */
int *ptr = &x; /* 现在指针变量 "ptr" 正指向内存单元 x = 10 */
printf("Address of variable \"x\" = %p\n", &x); /* 打印内存单元 x 的地址 */
printf("Address of variable \"x\" = %p\n", ptr); /* 使用 "ptr" 变量，打印内存单元 x
的地址，它的值在内存单元 "x"*/
printf("Value of variable \"x\" = %d\n", x); /* 打印变量 x 的值 */
printf("Value stored at address ptr = %p is %d\n", ptr, *ptr); /* 使用取值操作符 (*ptr)
打印内存单元 x 的值 */
```

变量 ptr 的值和表达式（&x）的值本质上是等价的，为变量 x 的内存位置，因为 ptr 现指向 x。

利用解引用操作符（*）得到某些内存单元的值。因此，表达式 *ptr、*(&x) 和 x 的值都为 10。

注意 如图 2-1 所示，在解引用任意指针变量前，必须确保它指向一个有效的内存地址，否则引发错误。图 2-1 中实例 B 的错误原因就是访问了一个无效的内存地址。

在实例 B 的第一行，试图在一个内存地址赋值 10，这是无效的，因为变量 ptr 指向一个无效的内存单元。

为使程序正确运行，我们需将 ptr 变量指向一个有效的内存单元。下面的代码说明如何采用适当方法来实现该目标：

```
int count = 1; //"count" 变量将用于申请一个整型大小的内存单元
int *ptr = (int *) malloc ( sizeof(int) * count );
```

现在 ptr 变量指向一个有效内存单元。

```
*ptr = 10; // 此时我们给 "ptr" 指向的内存单元赋值
free(ptr) ; //  此时我们释放变量 "ptr" 指向的内存
*ptr = 20; //  此时程序又将抛出一个分段错误，因为我们试图访问已被释放的内存
```

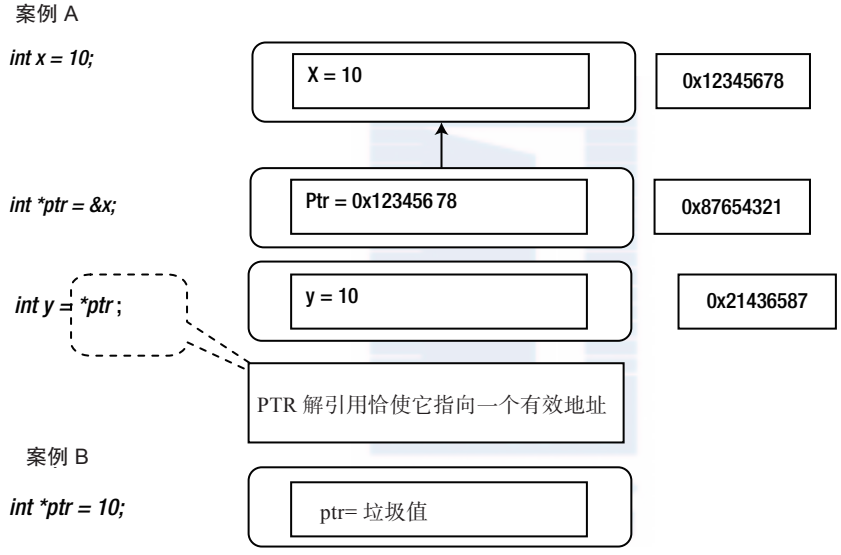


图 2-1　指针变量分别指向一个有效内存地址和一个无效内存地址

2.7　指针的基本用法

你已学会如何声明和初始化指针。现在我们将学习指针的最基本用法，更确切地说是使用指针的优势。函数和参数齐头并进。通过指针变量间接引用有很多操作内存值的方法。学习本节的内容有助于理解刷新生命周期、变量范围和第 1 章中的栈段等内容。

2.7.1 传值

函数能从调用者接收信息并将结果返给调用者。该技术就是函数中信息传递的最基本形式。

函数原型

```
int function_name( int param1, int param2, int param3.....);
```

上述函数声明中，int param1、int param2 和 int param3 称为输入参数。该函数声明的返回类型为整型，其功能是说明此函数将整型值返回给调用者。

这项特定技术仅将值传递给被调用函数。值传递后将复制到被调用相应的栈中。同样，重复返回调用函数值这一过程。

```
void calling_function(void)
{
int t1, t2, t3;
t1 = 10;
t2 = 20;
t3 = called_function(t1, t2);
}
```

Local copy of the calling_function

t1 = 10

t2 = 20

t3 = 30

```
int called_function(int x, int y)
{
int t1, t2, t3;
t1 = x;
t2 = y;
t3 = t1 + t2;
return t3;
}
```

Local copy of the called_function

t1 = 10

t2 = 20

t3 = 30

2.7.2 引用传递

函数之间传递信息的另一种技术为通过引用传递来传递变量的内存地址而不是变量值本身。

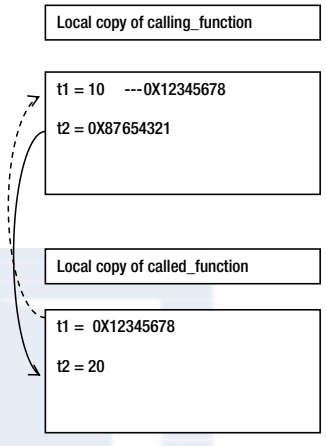
函数原型

```
int* function_name( int* param);
```

在上述函数声明中，输入参数为 int * param，也就是希望从调用者那里接收一个整型变量的地址，而功能是将一个整型变量的地址返回给调用者。

```
void calling_function(void)
{
int t1;
int *t2;
t1 = 10;
t2 = called_function(&t1);
}

int* called_function(int* x)
{
int t2;
int *t1;
int *t3;
t1 = x;
t2 = 10;
t3 = (int*)malloc(sizeof(int));
t3 = *t1 + t2;
return &t3;
}
```



在上述情形中仅有一个变量的地址被传递给被调用函数，并将其复制到栈中。相比前一项技术，该技术有两个优点。

1. **数据量被复制**：在复制内存地址和复制信息量总是 4 字节的情形下复制参数，但和前一个例子中传递的信息量是相同的。

情形 1：传值

```
struct data
{
int x;
int y;
};
void func(struct data v1)
{
struct data v2 = v1;
}
int main()
{
struct data var;
var.x = 10;
var.y = 20;
func( var );
return 0;
}
```

在上例中，struct data 类型变量的大小为 8 字节。由于使用按值传递，8 字节被复制到被调用函数 func 的栈中。

情形 2：引用传递

```
struct data
{
int x;
int y;
};
void func(struct data* v1)
{
struct data *v2 = v1;
}
int main()
{
struct data var;
var->x = 10;
var->y = 20;
func(& var );
return 0;
}
```

在上例中，通过指针传递给结构体变量参数的大小为 4 字节。

2. **辅助变量：**通过引用技术有可能在不同函数中对某个函数中的局部变量进行操作。

2.8 指针和常量

你可能听说过 const 关键字，也在编程中使用过它。普通的 const 型变量的含义就是初始化时的赋值在其生存期内不能修改。同时使用指针和常量会有不同效果。

2.8.1 常量指针变量

常量指针是一个仅指向唯一内存地址的指针变量。因此，指针变量的值不能修改。

常量指针声明：< 指针类型 *> const < 变量名 >

实例：int* const ptr1, char* const ptr2;

以下为常量指针使用规则。

1. 常量指针变量声明时必须初始化。

源代码 . Ptr6.c

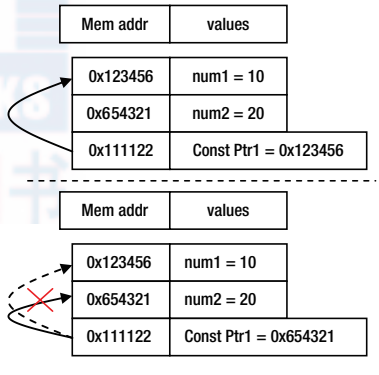
```
int main()
{
int num = 10;
int* const ptr1 = &num;  //Initialization of const ptr
printf("Value stored at pointer = %d\n",*ptr1);
}
```

2. 一旦完成初始化，常量指针不能再指向其他内存地址。

源代码 . Ptr7.c

```
#1. int main()
#2. {
#3. int num1 = 10;
#4. int num2 = 20;
#5. int* const ptr1 = &num1;  //Initialization of const ptr

#6. ptr1 = &num2; // can't do this
#7. printf("Value stored at pointer = %d\n",*ptr1);
#8. }
```



在上述程序中，常量指针变量 ptr1 在第 5 行被初始化，指向变量 int num1 的内存地址。在第 6 行，程序试图让常量指针变量 ptr1 指向变量 int num2 的内存地址。如果编译这部分代码，编译器会显示一个编译错误。

2.8.2 常量指针

常量指针是指某个指针变量的值（即变量的内存地址）不能修改指

定内存地址存放的值。不同指针可指向指定变量。

常量指针声明：const< 指针类型 *> < 变量名 >

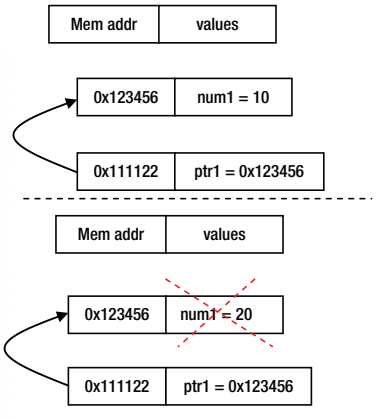
实例：const int* ptr1, const char* ptr2;

源代码 . Ptr8.c

```
#1. int main()
#2. {
#3. int num1 = 10;
#4. const int* ptr1;

#5. int* ptr2;
#6. ptr1 = &num1;


#7. *ptr1 = 20; //can't do this
#8. num1 = 20; //can be done
#9. printf("Value stored at pointer = %d\n",*ptr1);
#10. }
```



当试图编译上述代码时，编译器会显示一个由于第 7 行代码引起的编译错误。

2.8.3 指针常量

指针常量的概念为指针变量是常量。换句话说，指针变量仅指向被初始化的内存地址，指针以后不能指向内存单元。此外，指定指针不能修改指定地址存储的值。总之，不能改变指针变量的值，也不能修改存储在该地址的值。

指针常量声明：const< 指针类型 *> const < 变量名 >

实例：const int*const ptr1, const char* const ptr2;

源代码 . Ptr9.c

```
#1. int main()
#2. {
#3. int num1 = 10;
#4. int num2 = 20;
#5. const int* ptr1 = &num1;
#6. int* ptr2;
#7. *ptr1 = 20; // 不能修改常量指针指向的值
#8. num1 = 20; //can be done
#9. ptr1 = &num2; // 不能修改常量指针的值（即 .– 常量指针一旦被初始化就只能某个内存地址）
point to any other memory address once initialized
#10.printf("Value stored at pointer = %d\n",*ptr1);
#11. }
```

当试图编译上述代码时，编译器会显示一个由于第 9 行代码引起的编译错误。

2.9 多级指针

到现在为止，你已学习了一级间接引用过程。可曾想过多级间接引用的可能性。正如前面所学的，指针变量能存储其他变量的内存地址，并有可能进一步扩大该概念。指针变量本身的地址能存储在其他的指针变量中。

存储指针变量地址的变量称为指针变量指针。我们能进行新的扩展，如指针变量指针的指针，等等。

指针变量指针

我们现在来讨论指针间接引用的第二级。分析下面的代码片段：

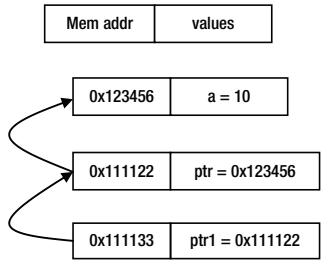
```
int a = 10;
int *ptr = &a;
```

在这段代码中，声明整型变量 a 和指向整型变量的整数指针变量 ptr。现在将看到如何将这个整数指针变量的地址存储到另一个指针变量。为了实现将指针变量的地址存储到另一个变量，这里需要一个不同类型的变量。

声明：< 数据类型 > ** < 变量名 >

星号的数目取决于间接引用的级数。随着星号数目的增加间接引用的级数也会增加。

```
int a = 10;
int *ptr = &a;
int **ptr1 = &ptr;
```



源代码 . Ptr10.c

```
int main()
{
        int num = 10;
        int *ptr = &num;
        int **mptr = &ptr;
        printf("Value of var num = %d\n", num);
        printf("Value of var num = %d\n", *ptr);
        printf("Value of var num = %d\n", **mptr);

        printf("Address of var num = %p\n", &num);
        printf("Address of var num = %p\n", ptr);
        printf("Address of var num = %p\n", *mptr);

        printf("Address of pointer var ptr = %p\n",&ptr);
        printf("Address of pointer var ptr = %p\n",mptr);
        printf("Address of pointer var mptr = %p\n",&mptr);

        return 0;
}
```

2.10 理解神秘的指针表达式

由于指针地址解引用方法的多样性使得理解指针表达式变得很神秘。本节着重介绍通过分离等效表达式更容易地理解这些表达式。你可能会发现本节有点脱节，但它是综合全部内容的一个好方法。

从分析一级指针的间接引用情形开始。在接下来的所有讨论中，假定整型的变量默认初始值为 10 和它的内存存储位置为 0x0001。我们会发现两种情形（引用和解引用）下表达式的等效性。

2.10.1 一级指针引用

```
int val = 10;
```

Varname/Addr	Value
val/0x0001	10

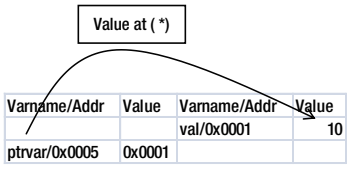
```
int *ptrvar = &val;
```

Varname/Addr	Value	Varname/Addr	Value
		val/0x0001	10
ptrvar/0x0005	0x0001		

因为指针变量 ptrvar 存储变量 val 的地址，同样表达式 &val 也得到相同值，说明两个表达式是等效的。所以，我们说 `ptrvar == &val`。

2.10.2 一级指针解引用

如你所知，解引用使用“取值”(*) 操作符。如果试图用此操作符对指针变量操作，就会得到存储在该内存地址的值。




```
*ptrvar == 10
```

上节中看到 `ptrvar ==&val`。使用“地址”操作符操作表达式也能得到相同的值。

```
*(&val) == 10
```

所以，下面的表达式是等效的：

```
*ptrvar == *(&val) == 10.
```

现在在二级间接引用上做同样的练习。同样使用前面用到的两个变量和一个存储整型指针变量地址的新变量。

a. `int val = 10;`
b. `int *ptrvar = &val;`
c. `int **ptrptrvar = &ptrvar;`

让我们从引用开始。

2.10.3 二级指针引用

对于前两个变量，内存示意图与前面绘制的一级间接引用示意图一样。对于第三个指向整型类型的指针变量的指针请参考下面的示意图。

```
int **ptrptrvar = &ptrvar;
```

Varname/Addr	Value	Varname/Addr	Value	Varname/Addr	Value
				val/0x0001	10
		ptrvar/0x0005	0x0001		
ptrptrvar/0x0009	0x0005				

变量 ptrptrvar 是一个指向指针变量的指针，它存储指针变量的地址。因此，需验证表达式 `ptrptrvar == &ptrvar`。

2.10.4 二级指针解引用

在本节中，我们对最上级使用取值操作符来分析表达式的含义如何变化。

```
*ptrptrvar == ptrvar == 0x0001
```

因为 ptrptrvar == &ptrvar，我们利用另一个等效表达式也能得到同样的值。

```
*(&ptrvar) == 0x0001
```

因此，`* ptrptrvar == ptrvar == *(&ptrvar) == 0x0001`。

现在我们使用二级间接引用：`** ptrptrvar`，该表达式赋值为 10，我们可写为 `**ptrptrvar == 10`。

在上述表达式中，如果我们试图替代 *ptrptrvar 部分，能得到相同结果的等效表达式。即 `**ptrptrvar == *(ptrvar) == *(*(&ptrvar)) == 10`。

图 2-1 说明了上述讨论的概念。

图 2-2 给出了关于指针如何指向变量和二级指针如何使用的直观解释（参照上例）。为了间接访问某个实际变量，可使用“取值”操作符（&）或组合使用“取址”操作符（*）和“取值”操作符（&）。

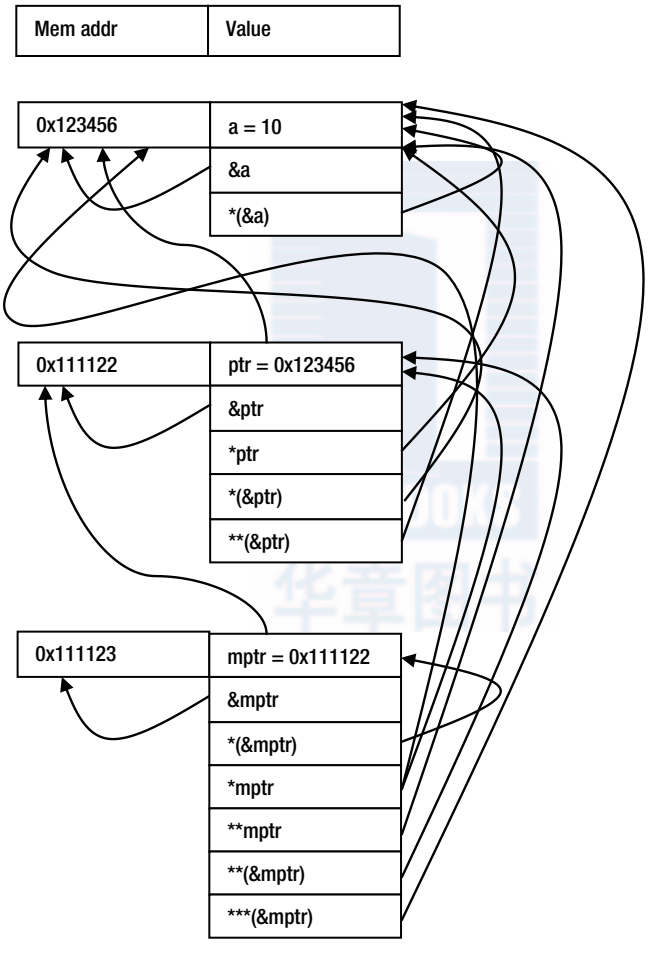


图 2-2　指针表达式

2.11 小结

本章介绍指针的基础知识及使用方法。本章目标让读者理解引用和解引用概念。多级间接引用时更应该关注此概念。结构体变量指针在本章仅做简单介绍，之后有一章会专门介绍指针在结构体类型变量中的使用。

下一章中介绍更高级的指针运算概念，还会介绍数组指针的用法。