变量的作用域

- 所有变量都有自己的作用域，即该变量的有效区域。
- 按照变量的作用域，可分为：
  - 局部变量(内部变量)
  - 全局变量(外部变量)

局部变量

- 指在一个函数内部定义的变量，它只在本函数的范围内有效， 在此函数之外不能使用这些变量
- 说明:
  - main函数中定义的变量也是局部变量，只在main函数中有效
  - 不同函数中同名变量，占不同内存单元，互不干扰
  - 函数的形式参数也是局部变量
  - 可在复合语句中定义变量，它们只在复合语句的内部有效
  - 变量的定义必须在可执行语句之前，即进入{ }后，首先要定义变量

全局变量

- 在所有函数之外定义的变量
- 它的有效范围从定义变量的位置开始到本源文件结束，一直占内存
- 如在定义时没初始化，系统自动初始化为0

局部变量和全局变量示例

```
#include <stdio.h>
int p=1 , q=5 ;
float  f1( int  a )
{  float  r ;
       :
}
char  s;
int  f2( int  a , int  b )
{   int  sum ;
       :
}
float m , n ;
int  main( )
{   float  x , y , a ;
       :
}
```

a,r等为局部变量

全局变量p和q的有效范围

全局变量s的有效范围

全局变量m和n的有效范围

全局变量的使用

- 全局变量的使用增加了函数间数据联系的渠道，由于在同一文件中的所有函数都能使用全局变量，所以可以利用全局变量从函数中得到一个以上的返回值，而使用return只能返回一个值。
- 例: 求某班成绩的平均分，最高分和最低分。

```
#include <stdio.h>
float  max=0, min=100 ;
float  average( int  n );
int main(void)
{  int  m;     float  ave2 ;
   scanf("%d", &m);
   ave2 = average(m);
   printf("%f,%f,%f\n", ave2, max, min);
   return 0;
}
```

```
float  average( int  n )
{   int  i;    float  s, ave1, sum=0;
    for(i=1; i<=n ; i++)
    {    scanf("%f",&s);
         if (s>max)  max = s;
         if (s<min)  min = s;
         sum=sum+s;
    }
  ave1=sum/n;
  return  (ave1);
}
```

- 建议不要过多的使用全局变量
  - 全局变量在程序的执行过程中一直占用存储单元
  - 它使函数的通用性降低
  - 它会降低程序的清晰性
- 若全局变量与局部变量同名，则全局变量被屏蔽

```
#include <stdio.h>
int a=3, b=5;
max(int  a, int b)
{   int c;
    c = a>b ? a : b;
    return  c;
}
int main(void)
{   int a = 8;
   printf("max=%d",max(a, b));
   return  0;
}
```

运行结果：max=8

```
#include <stdio.h>
int  x = 10;
void  f( )
{  int  x = 1;
   x = x + 1;
   printf("x=%d\n", x);
}
int main(void)
{  x = x + 1;
   printf("x=%d\n", x);
   f( );
   return  0;
}
```

运行结果：
x=11
x=2

变量的存储方式

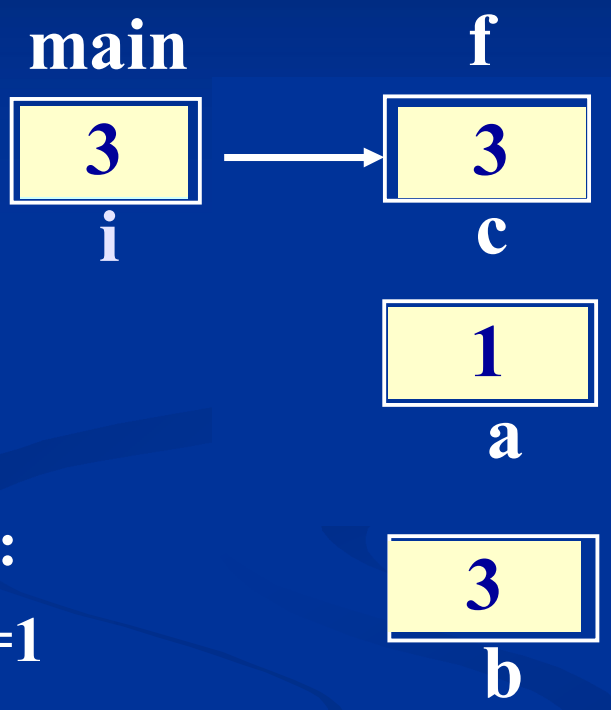
- 从变量的作用域（即从空间）角度来分，可分为局部变量、全局变量。
- 从变量存在的时间 (即变量生存期)角度来分，可分为：动态存储变量、静态存储变量。
- 动态存储变量：用动态存储方式存储的变量。
  - 特点是函数开始调用时为变量分配存储空间，函数结束时释放这些空间。
- 静态存储变量：用静态存储方式存储的变量。
  - 特点是在静态存储区分配存储单元，整个程序运行期间都不释放，在程序结束时才释放空间。

变量的存储类型

- **auto** (自动的)：如果局部变量不作存储类型说明，均为动态存储变量
- **register**(寄存器的)：可以提高“存取”速度，因为从寄存器存取数据比从内存存取数据快
- **static** (静态的)：它的存储空间在程序的运行期间都固定不变。该类变量在其函数调用结束后仍然保留着变量的值， 下次调用该函数，静态局部变量中仍保留上次调用结束时的值。
- **extern**(外部的)：可以扩展全局变量的作用域，或引用其它文件的全局变量

静态变量和动态变量的对比

```
#include <stdio.h>
void  f(int  c)
{
    int a = 0;
    static int b = 0;
    a++;
    b++;
    printf("%d: a=%d, b=%d\n", c, a, b);
}
int main(void )
{
   int  i;
   for (i=1; i<=3; i++)
        f( i );
   return  0;
}
```



输出结果:

1: a=1, b=1

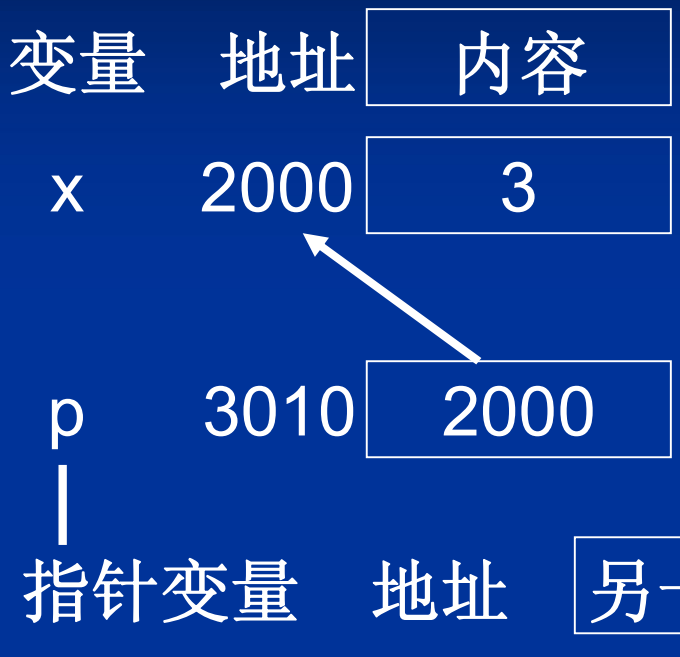
2: a=1, b=2

3: a=1, b=3

指针概述

- 地址
  - 在计算机中，把内存区划分为一个一个的存储单元，每个单元为一个字节（8 位），它们都有一个编号，这个编号就是内存单元的地址。
- 变量的地址
  - 每个变量在内存中都占有一定字节数的存储单元。程序编译时，根据程序中定义的变量类型，在内存中为其分配了相应字节数的存储空间。
  - 变量在内存中所占存储空间的首地址，称变量的地址。该地址存储的内容，就是变量的值(变量的内容)。

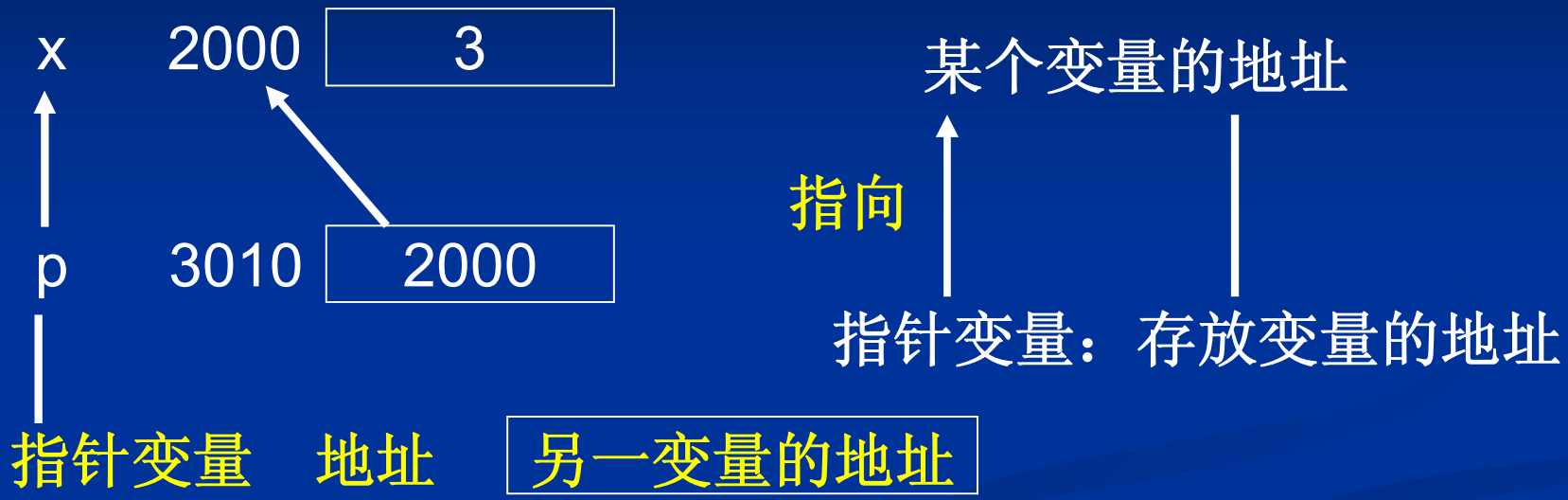
变量的访问方式




```
int x;
x=3;
printf("x=%d\n", x);
```

- **直接访问**：通过变量名或变量名所对应的地址访问变量的存储区，存取其值。
- **间接访问**：将一个变量的地址存放在另一个变量中，使用时先找到后者的地址，再从中取出前者的地址。

指针和指针变量的概念



- 指针的概念
  - 由于通过地址能找到所需的变量单元，地址象一根针一样“指向”该变量单元，所以将地址形象的称为：“指针”。
- 指针变量的概念
  - 指针变量就是存放变量地址的变量，它用来指向另一个变量。

指针变量的定义

- 格式：类型名 *指针变量名；
- 例： int *p; p是整型指针，指向整型变量

  float *q; q是指向实型变量的指针变量

  char *r;
- 说明
  - 类型名是指针变量指向的变量的数据类型
  - 在变量定义时，* 号表示该变量是指针变量，不可省
  - 注意：指针变量名是p，而不是*p

指针变量使用

- 定义指针变量时进行初始化，或使用赋值语句。
- 两个运算符
  - &：取地址运算符。 可作用于一般变量或指针变量
  - *：指针运算符。 只能作用于指针变量
  - 它们优化级相同，都是单目运算符，满足右结合。

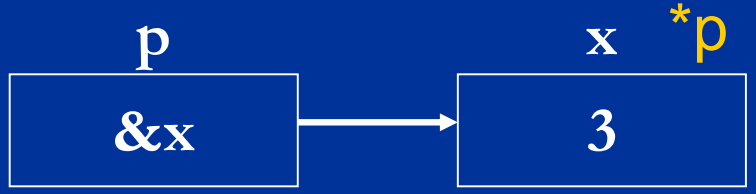
例：int *p, x;

x = 3;

p = &x; 把x的地址赋给p， 即p指向x

或：

int x, *p=&x;



通过p取x值：printf(“%d”, *p)， *p表示p所指向的变量x的值。

NULL指针

- 标准定义了NULL指针，它作为一个特殊的指针变量，表示不指向任何东西。
- NULL包含在stdio.h中，被定义成符号常量，与整数0对应。
- 例：int *p = NULL; 此时p为空指针。
- 对指针变量进行显示的初始化是种好做法，如果你暂时不知道指针将指向哪里，就把它先初始化为NULL。
- 对一个NULL指针进行间接访问操作是非法的，所以在对指针变量进行间接访问之前，要确保它并非NULL指针。
- NULL可以赋值给指向任何类型的指针变量。

使用指针变量需注意的问题(1)

- 只能用同类型变量的地址给指针变量赋值
  - 例：int x, *p;　　p=&x;

    int x;　float *p;　p=&x;　错
- 使用指针变量，一定先初始化，使其指向一个确定的地址
  - 例：int *p;　*p = 10;　错
- 指针可以进行加减整数的运算，但不同于一般的算术运算。
  - 例：int a[5], *p=a;　p=p+2;　p++;
  - 说明：p增1，是指向下一个变量，而不是简单的理解为地址值加1。一般而言，指针作为操作数加上或减去一个整数n，其指针值的变化为加上或减去n*sizeof(基类型)个字节。

使用指针变量需注意的问题(2)

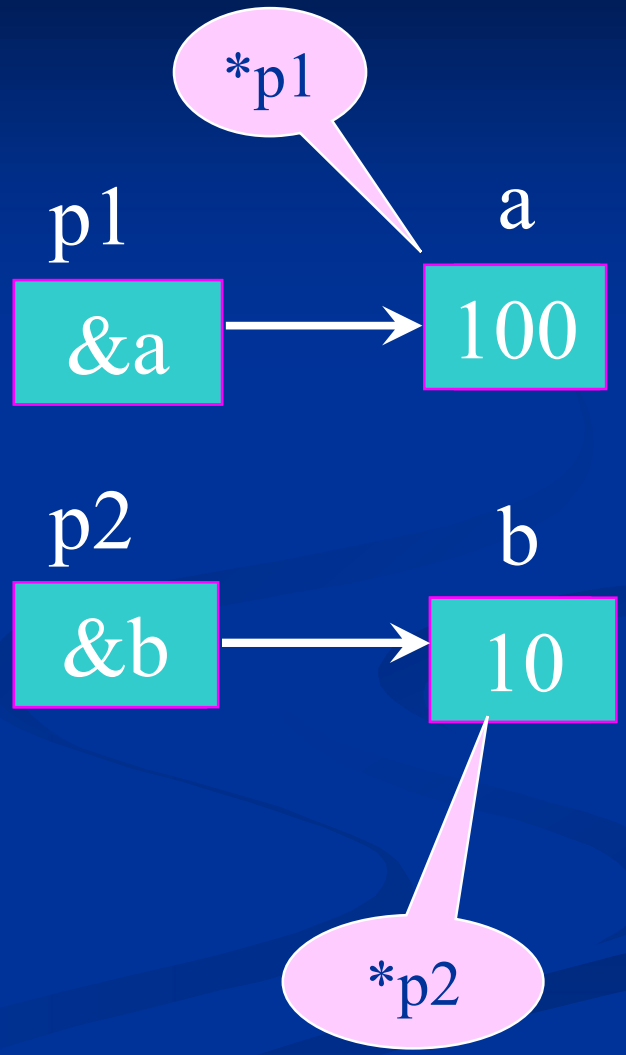
- p与*p不同：
  - p是指针变量，p的值是p所指向的变量的地址
  - *p 是p 所指向的变量， *p的值是p所指向的变量的值
- 引用指针变量时的 * 与 定义指针变量时的 * 不同，定义变量时的 * 只是表示其后的变量是指针变量
- 当p = &x ; *p与x相同;

则有：**&*p → &(*p) → &x → p　是地址**

***&x → *(&x) → *p → x　是变量**

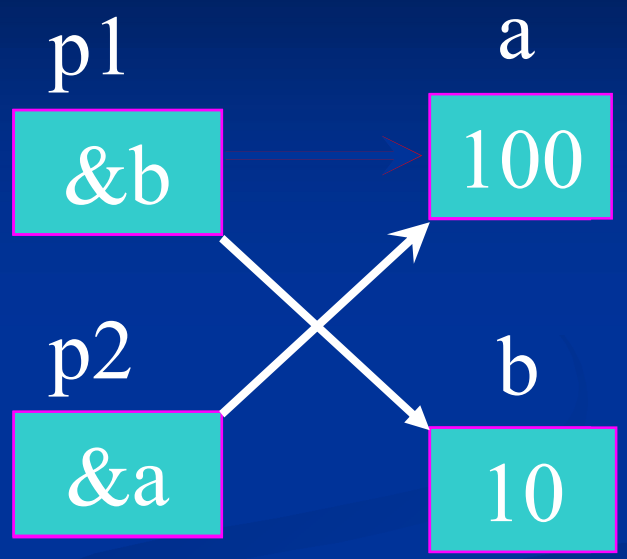
例：指针变量的赋值操作

```
#include <stdio.h>
int main(void)
{
    int   a,  b;
    int   *p1, *p2;
    a=100;  b=10;
    p1 = &a;
    p2 = &b;
    printf("%d,%d\n",  a, b);
    printf("%d,%d\n", *p1,*p2);
    return 0;
}
```



例:

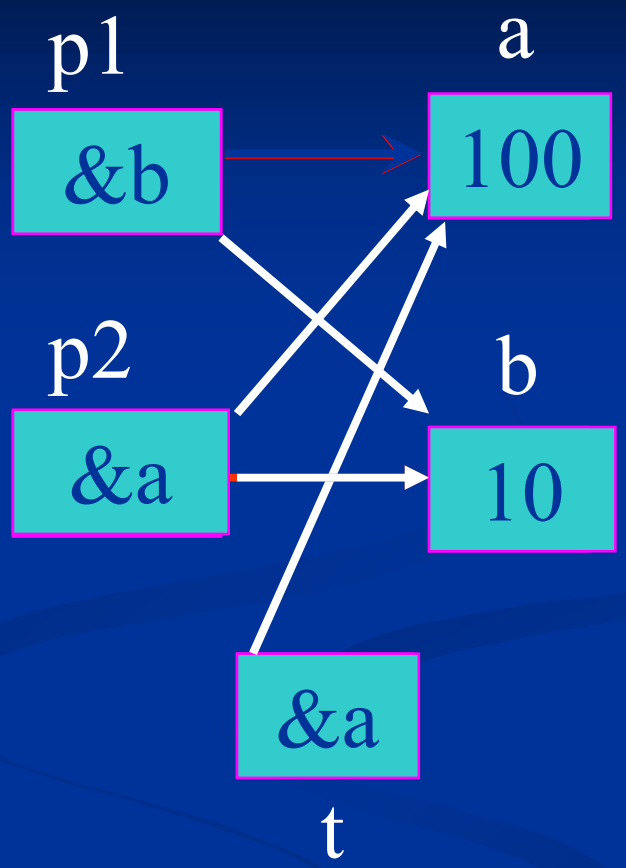
```
#include <stdio.h>
int main(void)
{
    int   a,  b;
    int   *p1, *p2;
    a=100;  b=10;
    p1 = &a;
    p2 = p1;
    p1 = &b;
    printf("%d,%d\n",  a, b);
    printf("%d,%d\n", *p1, *p2);
    return 0;
}
```



输出结果:
100, 10
10, 100

例: 指针变量的初始化操作

```
#include <stdio.h>
int main(void)
{
    int   a=100,  b=10;
    int   *p1=&a, *p2=&b, *t;
    printf("%d,%d\n", *p1, *p2);
    t = p1;
    p1 = p2;
    p2 = t;
    printf("%d,%d\n", *p1,*p2);
    return 0;
}
```



例:

```
#include <stdio.h>
int main(void)
{
    int   a=100,  b=10;
    int   *p1=&a, *p2=&b, t;
    printf("%d,%d\n", *p1, *p2);
    t = *p1;
    *p1 = *p2;
    *p2 = t;
    printf("%d,%d\n", *p1,*p2);
    return 0;
}
```