进阶C语言

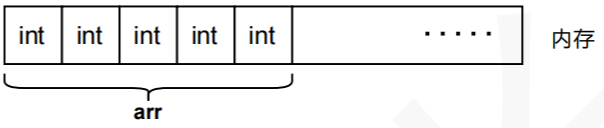
数组

数组是由n个相同数据类型的数据元素构成的有限序列，被存储在一块**连续的**内存空间中。

其基本操作如下。

- **定义数组**：你可以定义一个数组：int arr[5]; 这将创建一个可以存储5个**整数**的数组arr。

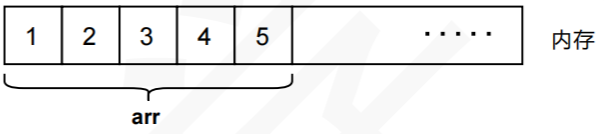
```
int arr[5];
```



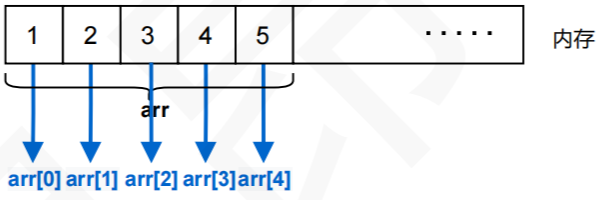
- **初始化数组**： 在定义数组的同时，可以初始化数组：int arr[5] = {1, 2, 3, 4, 5};

  这将创建一个包含5个元素的数组，初始值分别为1, 2, 3, 4, 5。

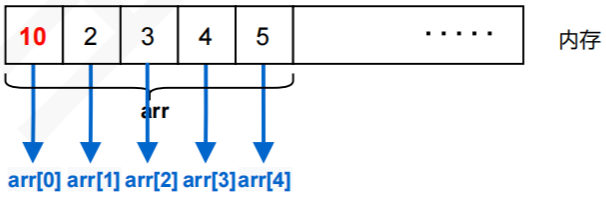
```
int arr[5] = {1, 2, 3, 4, 5};
```



- **访问数组元素**：你可以通过索引（下标）来访问数组元素，**数组的索引从0开始**。例如，arr[0]表示数组的第一个元素。

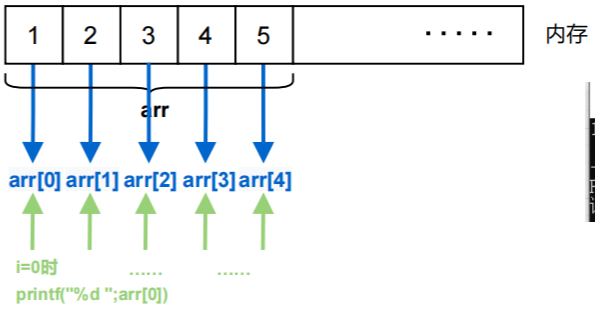


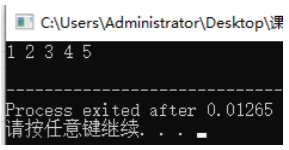
- **修改数组元素**：你可以通过索引来修改数组元素。例如，arr[0] = 10;将把数组的第一个元素修改为10。



- **遍历数组**：　你可以使用循环来**遍历**数组。例如：

```
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    for(int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```





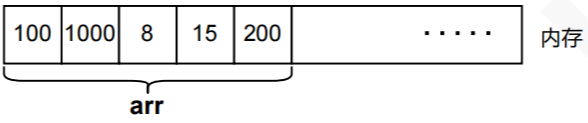
该代码运行结果如上

练习

```
#include<stdio.h>
int main(){
    int arr[5]={100,1000,8,15,200};
    for(int i=0;i<5;i++){
        printf("a[%d]=%5d,其内存地址为%d\n",i,arr[i],&arr[i]);
    }
    return 0;
}
```

int arr[5]={100,1000,8,15,200};

定义了一个由int类型变量构成的数组arr，其中有5个数组元素，分别为arr[0] arr[1] arr[2] arr[3] arr[4]
数组元素的值分别是arr[0]=100,arr[1]=1000,arr[2]=8,arr[3]=15,arr[4]=200



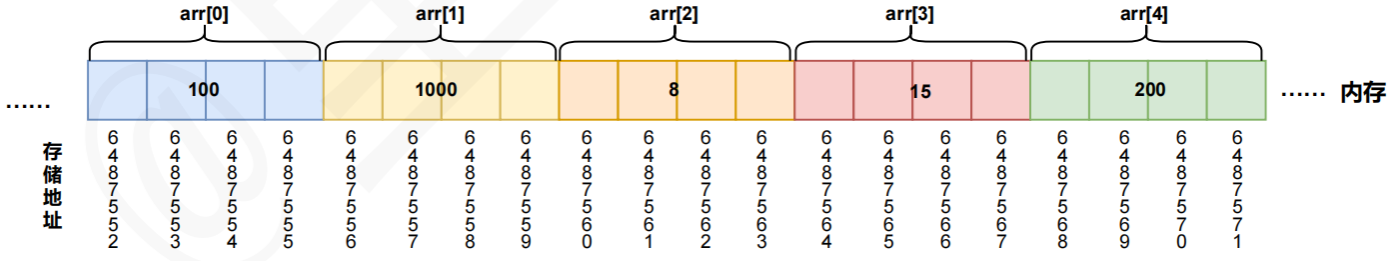
for(int i=0;i<5;i++){}

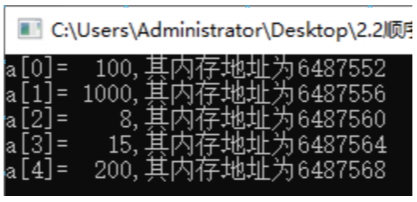
引入一个for循环遍历数组arr，其中：
定义了一个int类型的变量i，并给i赋值为0（int i=0）；
循环的条件为:i<5;
每次满足条件后修改i的值，使得i+1（i++）；

```
int i=0;
while（i<5）{
    i++;
}
```

printf("a[%d]=%5d,其内存地址为%d\n",i,arr[i],&arr[i]);

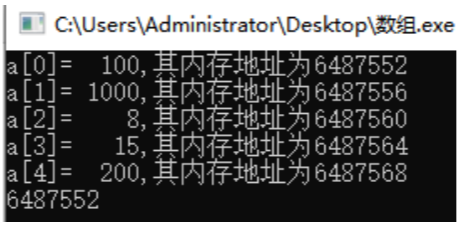
for 循环中执行体的内容为printf()，回忆printf()函数的用法，输出除占位符外" "内的所有内容，
即输入“a[ ]= ,其内存地址为 （换行）”
占位符%d，被变量i的值所取代后输出；
占位符%5d，被变量arr[i]的值所取代后输出；
占位符%d，被变量&arr[i]的值所替代后输出；
&为地址符，即求arr[i]的存储地址（房间号），&arr[i]即数组元素i所在内存中的房间号





对于任何一个变量来说，如果其包含了多个小房间，则**认为变量地址为第一个小房间的地址，也就是首地址。**
对于数组arr[]来说，**arr除了代表整个数组变量名以外，还代表了整个数组变量的首地址，也就是第一个小房间的地址。**

```
#include<stdio.h>
int main(){
	int arr[5]={100,1000,8,15,200};
	for(int i=0;i<5;i++){
		printf("a[%d]=%5d,其内存地址为%d\n",i,arr[i],&arr[i]);
	}
	printf("%d",arr);
	return 0;
}
```



学了那么多，我们来做道题。

题目：编写一个C语言程序，找出数组中的最大元素和它的索引。

要求：

- 定义一个包含10个元素的整数数组，并初始化为任意值。
- 找出数组中的最大元素以及它的索引。
- 打印出最大元素和它的索引。

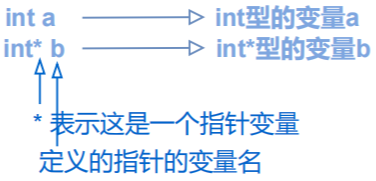
答案请参考课程代码文件。

指针

指针本质上也是一类变量类型，只不过其存储的不是具体的数据，而是某个内存单元的地址。

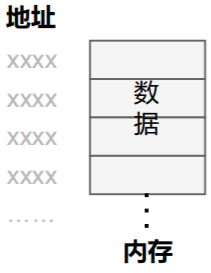
作为一种变量类型，指针和普通变量本质上没有区别，可以赋值，也可以运算，基本格式为变量类型 * 变量名

也就是说，能用int 定义一个int类型的变量，也就可以用int* 定义一个int*型的变量



由于指针是存储**地址**的变量类型

所以先回忆一下地址和数据的概念，接下来的内容必须建立在对地址和数据认识透彻的基础上！



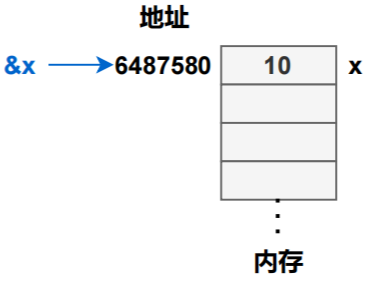
内存中存取数据，
地址是数据在内存中存放的位置
回忆之前的类比，内存=宾馆，数据=入住的人，地址是房间号，
地址的作用是来帮助人找到正确的房间（帮助找到数据进行读取操作）

```
#include<stdio.h>
int main(){
    int x=10;
    int* y=&x;
printf("x=%d\n",x);
printf("x的地址=%d\n",&x);
printf("y=%d\n",y);
printf("根据y保存的地址取值=&d\n",*y);
printf("y的地址=%d\n",&y);
return 0;
}
```

int x=10;

定义了一个int类型的变量，变量名为x

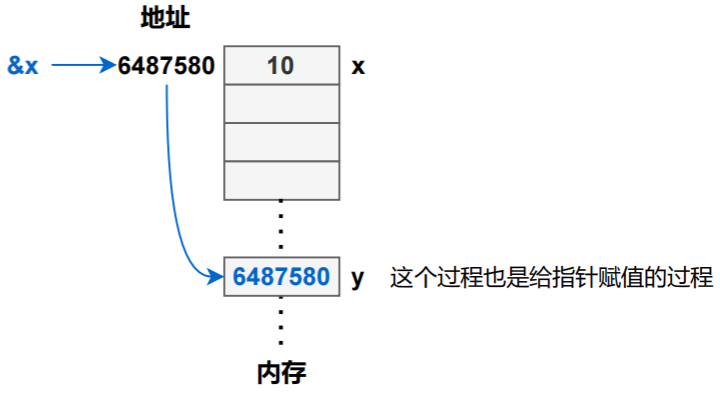
&为地址符，&x表示获取int型变量x的内存地址



int *y=&x;

定义了一个int*类型的变量，变量名为y;

y的变量值为&x，即变量x的内存地址（6487580）



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

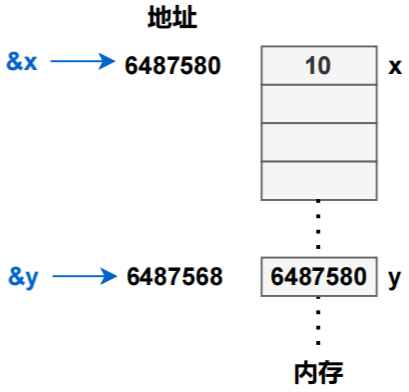
打印x的值　　x=10

printf("x的地址=%d\n",&x);

打印x所在地址的值　　x=6487580

printf("y=%d\n",y);

打印y的值　　y=6487580



printf("根据y保存的地址取值=%d\n",*y);

打印*y的值

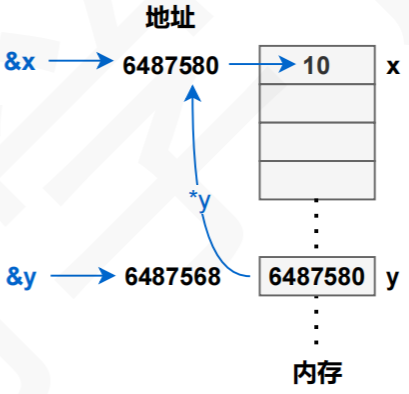
*结合变量名的时候可以进行取值，也就是说*y在找到x所在的地址后

能将地址内的值打印出来

也就是*y=10

printf("y的地址=%d\n",&y);

打印y所在地址的值　　y=6487568




```cpp
#include<stdio.h>
int main(){
    int x=10;
    int* y=&x;
    printf("x=%d\n",x);
    printf("x的地址=%d\n",&x);
    printf("y=%d\n",y);
    printf("根据y保存的地址取值=%d\n",*y);
    printf("y的地址=%d\n",&y);
    return 0;
}
```

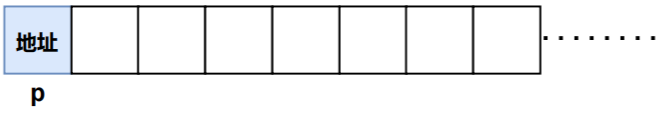
```
x=10
x的地址=6487580
y=6487580
根据y保存的地址取值=10
y的地址=6487568

--------------------------------
Process exited after 0.138 seconds with return value 0
请按任意键继续. . .
```

malloc函数

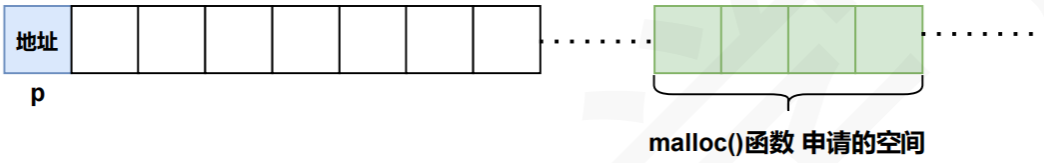
malloc函数会帮我们在内存中申请一块空间，并返回这块空间的地址。

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int* p;
p=(int*)malloc(4);
printf("%d ",p);
return 0;
}
```

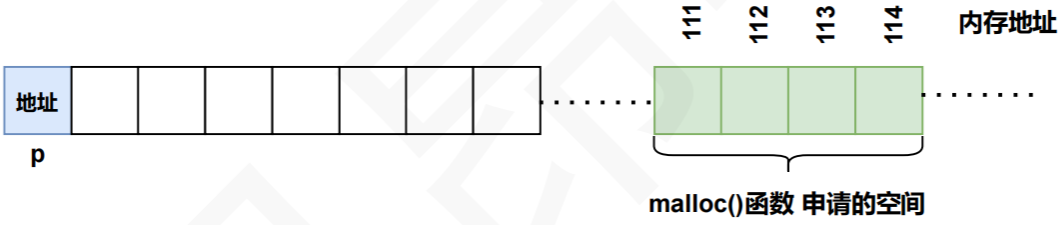
int* p; 定义了一个指针型变量 p



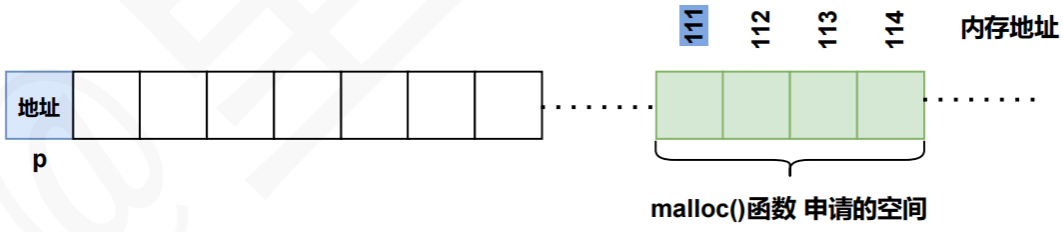
p=(int*)malloc(4);
其中 malloc(4)表示malloc函数在内存中申请了4个小房间（构成一个内存块/存储区域）；



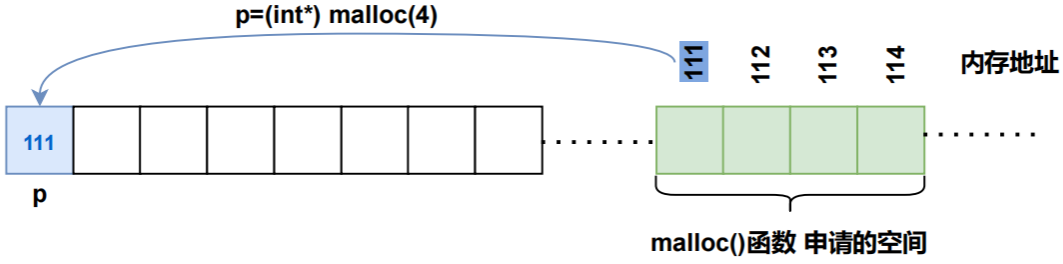
而这四个小房间(内存块)分别有自己的房间号(内存地址)
malloc函数返回的就是这个房间的第一个地址（111）



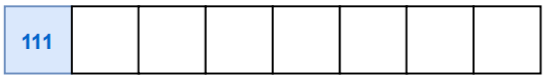
也就是(int*)malloc(4) 等同于(int*) 地址111
而地址由什么存放？——指针
所以(int*)地址111，也就是将一个里面存着地址111的指针转换为int型的指针！



接着将int类型的指针(里面的内存地址为111)储存在指针p中



printf("%d",p);
最终输出的结果为指针p的值



p

p的值为111

结构体（struct）

在C语言中，结构体是一种自定义的复合数据类型，它可以包含多个不同类型的变量和数据成员。

结构体的作用在于将一组相关的变量和数据聚合在一起，更好地组织程序结果或数据。

结构体以strcut关键词开头 →

```
struct student {
    int id;
    int age;
    float height;
};
```

← { }中列出结构体中包含的变量或者成员

这个例子中，student——结构体名
id、age、height分别是结构体中的变量

```
#include <stdio.h>

struct student {
    int id;
    int age;
    float height;
};

int main() {
    struct student student1;
        student1.id=1008;
        student1.age = 30;
        student1.height = 170.5;

    printf("学号：%d\n", student1.id);
    printf("年龄：%d\n", student1.age);
    printf("身高：%.2f\n", student1.height);

    return 0;
}
```

→ student结构体类型的变量

→ .叫做结构体成员运算符，
通过.可以student1的id、age和height

typedef

在C语言中，typedef是一个关键字，作用是为现有的数据类型起别名，目的就是方便！

基本格式： typedef 已有变量类型 别名

```
typedef   int   elemtype;
elemtype a = 10;  // 等同于 int a = 10;
```

此外，typedef在处理结构体和指针时都大有作用，能使代码更清晰易读

```c
#include <stdio.h>

struct student {
    int id;
    int age;
    float height;
};

int main() {
    struct student student1;
        student1.id=1008;
        student1.age = 30;
        student1.height = 170.5;

    printf("学号：%d\n", student1.id);
    printf("年龄：%d\n", student1.age);
    printf("身高：%.2f\n", student1.height);

    return 0;
}
```

使用typedef定义结构体

```c
#include <stdio.h>

struct student {
    int id;
    int age;
    float height;
};
typedef struct student Stu;
int main() {
    Stu student1;
    student1.id=1008;
    student1.age = 30;
    student1.height = 170.5;

    printf("学号：%d\n", student1.id);
    printf("年龄：%d\n", student1.age);
    printf("身高：%.2f\n", student1.height);

    return 0;
}
```

为结构体类型变量（变量名为student）取小名

小名为Stu，即struct student = Stu

```c
#include <stdio.h>

typedef  struct student {
    int id;
    int age;
    float height;
}Stu;
int main() {
    Stu student1;
    student1.id=1008;
    student1.age = 30;
    student1.height = 170.5;

    printf("学号：%d\n", student1.id);
    printf("年龄：%d\n", student1.age);
    printf("身高：%.2f\n", student1.height);

    return 0;
}
```

递归

在C语言中，**递归就是在过程或函数里调用自身。**

递归函数通常有两个部分：基本情况（base case）和递归情况（recursive case）。

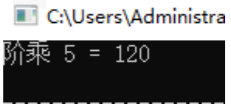
基本情况是函数不再自我调用的情况，而递归情况是函数继续调用自己的情况。

让我们以计算阶乘为例来解释递归。

阶乘函数的定义是：一个数的阶乘等于这个数乘以它前面所有正整数的乘积。

例如，5的阶乘（写作5!）等于5 * 4 * 3 * 2 * 1 = 120。以下是一个使用递归计算阶乘的C语言代码示例：

```c
#include <stdio.h>
// 定义一个递归函数来计算阶乘
int factorial(int n) {
  if (n == 0) {
      return 1;
   }
   else {
      return n * factorial(n - 1);
   }
}
int main() {
   int num = 5;
   printf("阶乘 %d = %d\n", num, factorial(num));
   return 0;
}
```




```c
int main() {
   int num = 5;
   printf("阶乘 %d = %d\n", num, factorial(num));
   return 0;
}
```

定义了一个int型变量num，赋值为5；
输出的结果由阶乘 num = factorial(num)构成；
涉及到factorial()函数，需调用**factorial()函数**

```c
int factorial(int n) {
  if (n == 0) {
      return 1;
   }
   else {
      return n * factorial(n - 1);
   }
}
```

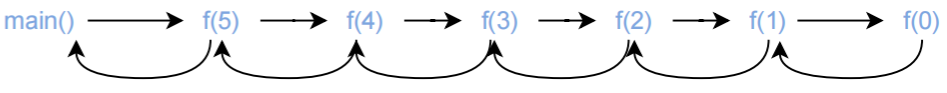
factorial()函数内嵌了一个if else的条件判断语句；
当代入factorial()函数的值为0时，执行if执行体中的内容；

```c
{
    return 1;
}
```

当代入factorial()函数的值不为0时，执行else执行体中的内容；

```c
{
    return n * factorial(n - 1);
}
```

此时又涉及到factorial()函数，这个在函数中调用自身的过程就是递归

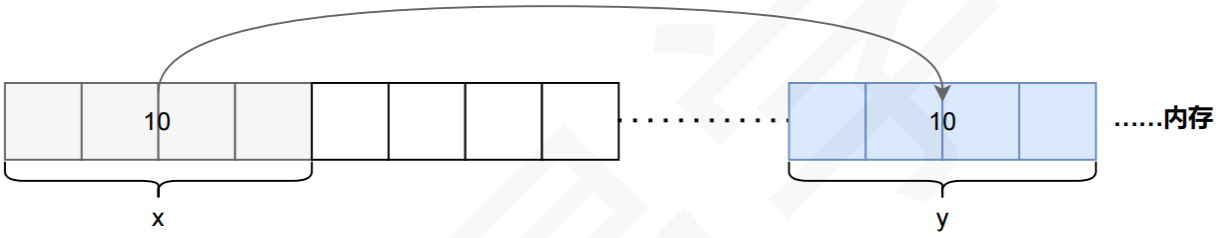


引用

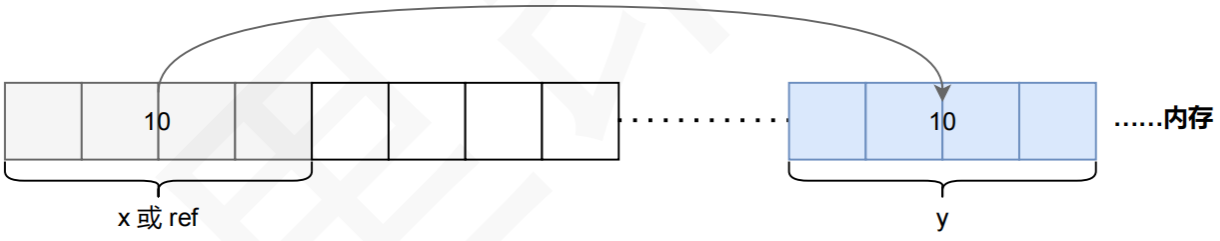
C++是对C语言的继承，引用（reference）就是C++对C语言的扩充。
在C++中，引用就是某个已存在变量的另一个名字。
一旦一个引用被定义并初始化，就不能再让它引用其他变量。

其格式：变量类型 & 引用名称 = 变量名

```
#include <stdio.h>
int main() {
    int x = 10;
    int y=x;
        int& ref = x;
        printf("x的值为：%d\n", x);
        printf("y的值为：%d\n", y);
        printf("ref的值为：%d\n", ref);
    // 通过引用修改变量的值
    ref = 20;
        printf("修改后，x的值为：%d\n", x);
        printf("修改后，y的值为：%d\n", y);
        printf("修改后，ref的值为：%d\n", ref);
    return 0;
}
```

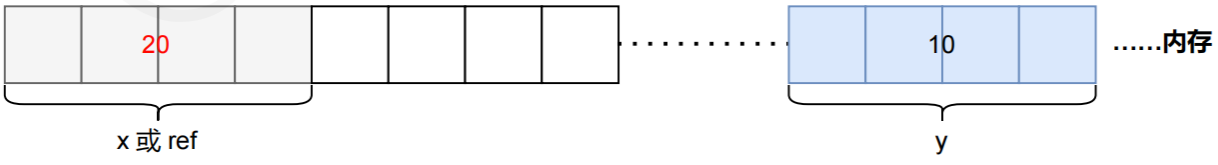


int& ref=x 语句执行后：

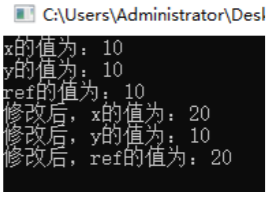


为已存在的int类型的变量x定义一个别名ref；

修改ref的值（执行ref=20）后：

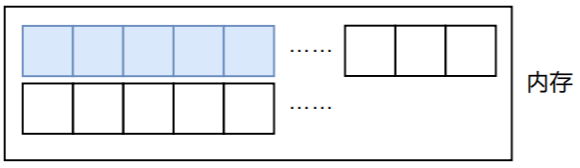


ref和x就是一个东西，对ref或者x任一进行操作，另一者也会跟着操作



链表

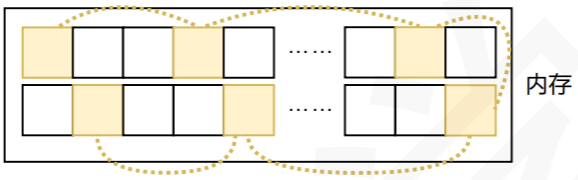
还记得数组吗？数组是由n个相同数据类型的数据元素构成的有限序列，被存储在一块**连续的**内存空间中。



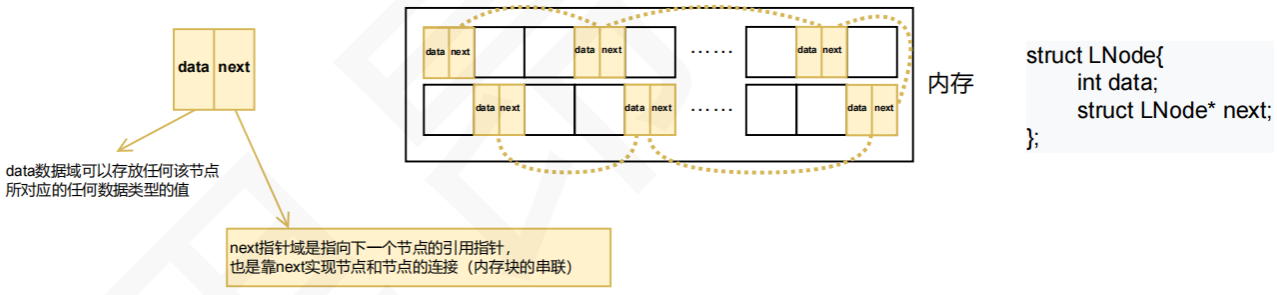
比如，int arr[5]，也就是内存中会创建一个连续的存储空间来存5个int类型的变量

但如果我的数组要存的数据量非常大呢？当arr[100000]时，内存可能不再会为我们分配这么大的连续存储空间。

因此，需要引入链表——**能够将不连续的内存块串联起来从而存储数据，是一种动态的、非顺序的存储结构**

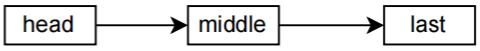


在链表中，每个节点都会有data数据域和next指针域



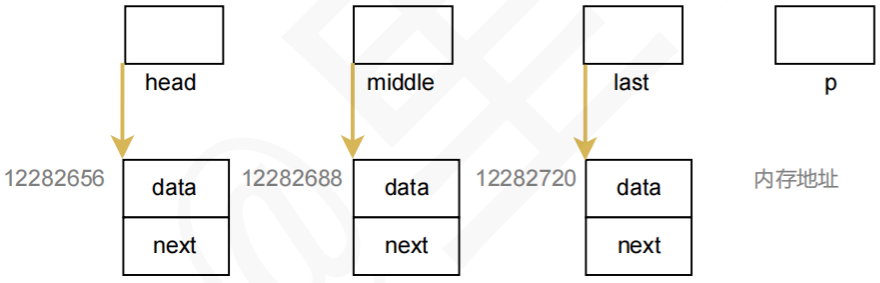
理解：交朋友，每个小朋友就像一个结点，小朋友的名字可以类比为存放在data数据域中的数据，而小朋友手中握着的通向下一个小朋友家的地图就等同于next指针域，指向下一个小朋友的住址，通过每个小朋友手中的地图，帮助小朋友们交到朋友，朋友们之间就形成了一条“链表”

```
//链表
#include<stdio.h>
#include<malloc.h>
struct LNode{
	int data;
	struct LNode* next;
};
int main(){
	struct LNode *head, *middle, *last,*p;

	head=(struct LNode*)malloc(sizeof(struct LNode));
	middle=(struct LNode*)malloc(sizeof(struct LNode));
	last=(struct LNode*)malloc(sizeof(struct LNode));

	head->data=10;
	middle->data=20;
	last->data=30;

	head->next=middle;
	middle->next=last;
	last->next=NULL;

	p=head;
	while(p!=NULL){
		printf("当前链表结点的数据为%d，地址为%d\n",p->data,p);
	p=p->next;
	}
	return 0;
}
```

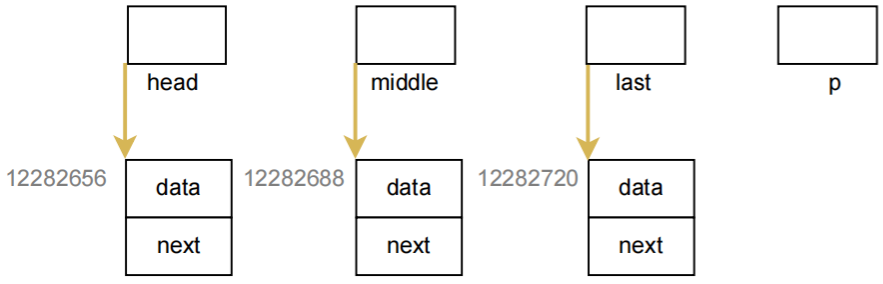


struct LNode *head, *middle, *last,*p; 建立结构体类型的指针head、middle、last、p

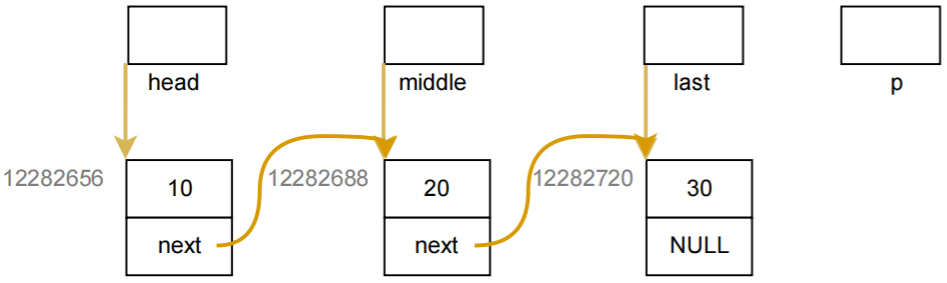
head=(struct LNode*)malloc(sizeof(struct LNode));
middle=(struct LNode*)malloc(sizeof(struct LNode));
last=(struct LNode*)malloc(sizeof(struct LNode));



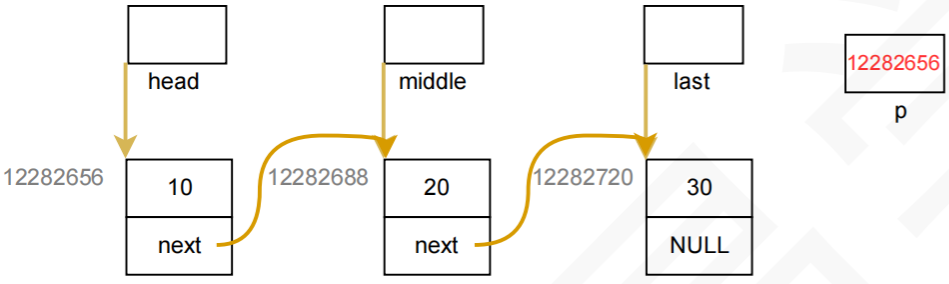

```
head->data=10;
middle->data=20;
last->data=30;
```



```
head->next=middle;
middle->next=last;
last->next=NULL;
```

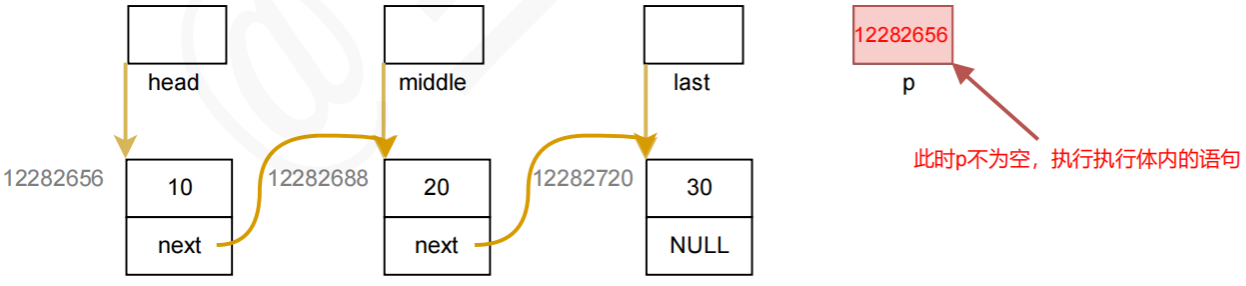


p=head; 由于p为指针型变量，用指针p保存了head的值，也就是p=第一个节点的地址，12282656



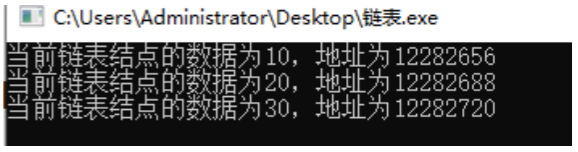

```
while(p!=NULL){
printf("当前链表结点的数据为%d，地址为%d\n",p->data,p);
p=p->next;
}
```

while循环，当p不等于空时，则执行执行体中的两条语句



printf("当前链表结点的数据为%d，地址为%d\n",p->data,p);
其中p->data为10，p为12282656

p=p->next;
即将p赋值为p->next的值，而next的值为下个节点的地址，p=12282688

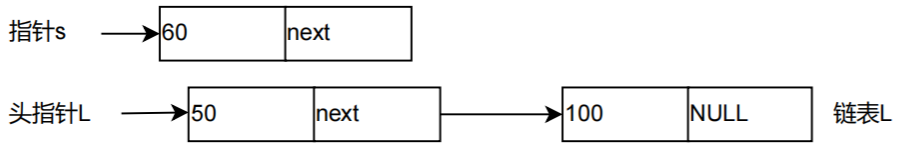


……直至跳出循环

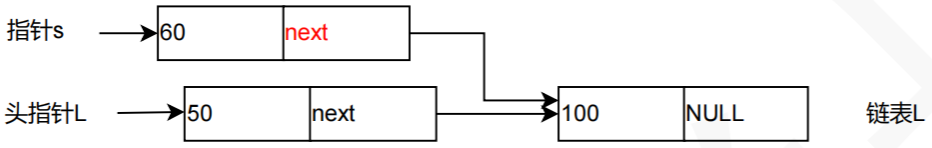
3.链表的基本操作

插入操作

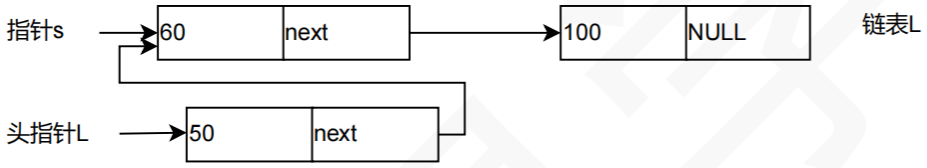
现想将指针s指向的结点插入链表L中，头指针L是记录链表的第一个结点地址的指针变量。



首先将s指向的结点的next域修改，将里面的地址修改成L->next，即s->next=L->next;

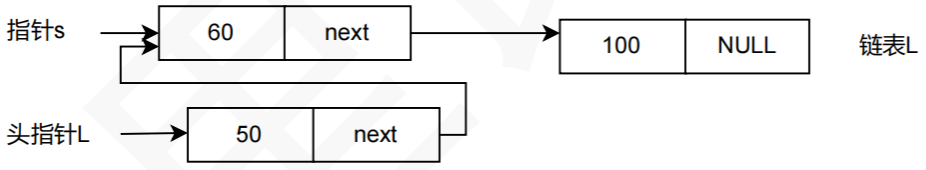


接着将L指向的结点的next域修改，将里面的地址修改成s，即L->next=s;

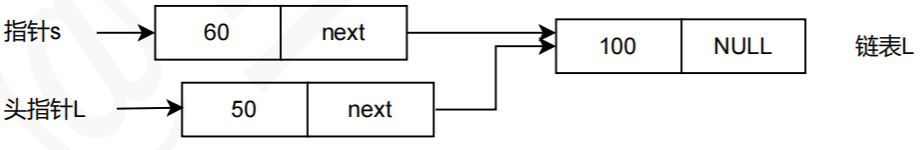


删除操作

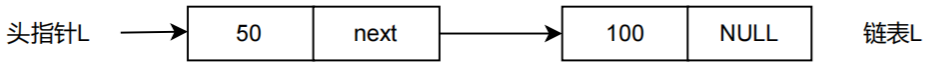
现想将指针s指向的结点从链表L中删除，头指针L是记录链表的第一个结点地址的指针变量。



首先将L指向的结点的next域修改，将里面的地址修改成s->next，即L->next=s->next;



接着将s指向的结点所占用的内存空间释放，即free(s)



[例2] 在一个单链表中，已知 q 所指结点是 p 所指结点的前驱结点，若在 q 和 p 之间插入结点 s, 则执行( )。

A.s->next=p->next;p->next=s;

B.p->next=s->next; s->next=p;

C.q->next=s;s->next=p;

D.p->next=s;s->next=q;