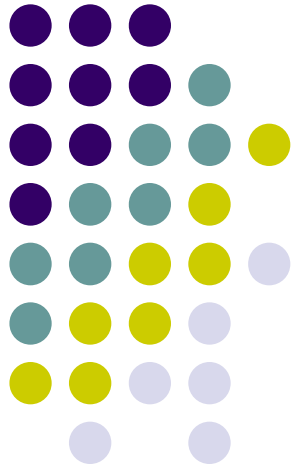
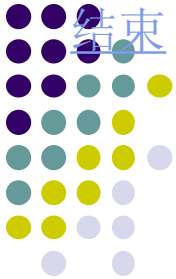
动态数组和链表

吴清锋

2023年春

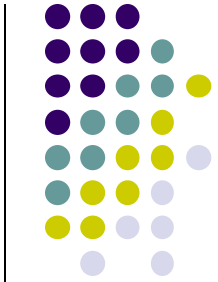


提纲



- 数组回顾
- 动态内存分配：概述
- 动态变量
- 动态数组
- 链表

回顾：数组概述

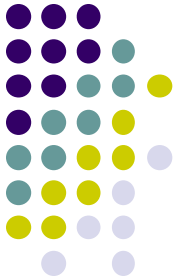


- **数组**是相同类型数据的集合
  即：数组的基本功能是用来**存储**数据
- 数组的难点：数组以循环为依托，实现**各类算法,即：**
  1 数组和循环是相伴的
  2 算法的核心要掌握
  （1）排序算法
  （2）查找算法

回顾：数组分类1

- 一维数组
  - 一维数组的基本操作——与循环语句联合使用
  - 一维数组存储的数据具有“线性特征”
  - 基本操作：在子函数中处理数组数据
    如：void Mysort(int a[],int n);
- 二维数组
  - 思考：问题域中有哪些需要用二维数组来表示？
    例如：有相同类型数据的二维表、线性代数中的矩阵、图像处理中的二维图像等。
- **值得提醒的是：**需要依据问题域中数据的特征来选择，选择在代码中的表示：究竟是一维数组还是二维数组

回顾：数组分类2

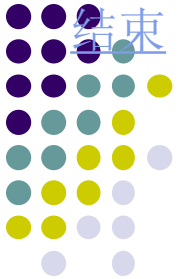


- 字符数组
  - 字符数组中存放字符信息
    char str[10]={‘0’,’1’,’2’,’3’,’4’,’5’,’6’,’7’,’8’,’9’};
  - 字符数组存储字符串（PK：指针变量处理字符串）

	字符数组	指针变量
是否有具体存储字符串的空间	有 char str[87]; gets(str); //cin>>str;	无 char *pstr; gets(pstr); **//Error**
初始化	str=“Hello!” **//Error**	pstr=“Hello!”;

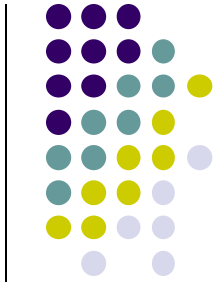
- C没有提供字符串变量类型，而C++有string
- **可以用功能更加强大的string来表示字符串**

提纲



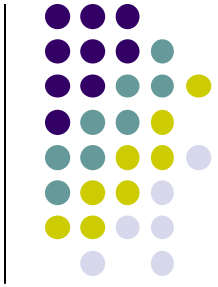
- 数组回顾
- 动态内存分配：概述
- 动态变量
- 动态数组
- 链表

静态内存分配与动态内存分配



- **静态内存分配**：在编译时，程序根据变量的类型确定所需内存空间的大小，从而系统在适当的时候为它们分配确定的存储空间，并会自动收回。
- **动态内存分配**：在程序运行阶段，再确定存储空间大小。

计算机内存空间分布



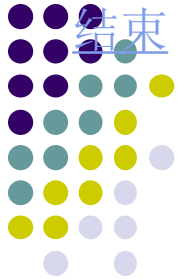
代码区：存放程序代码

全局与静态变量区：存放全局与静态变量

局部变量区（栈区）：存放局部变量

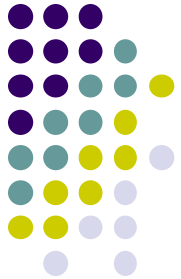
自由存储区（堆区）：存放动态申请的空间

提纲



- 数组回顾
- 动态内存分配：概述
- **动态变量**
- 动态数组
- 链表

C++中内存管理：单个变量



- new，“新建”（房子）
  - **格式：**数据类型 指针变量 = **new** 数据类型;
  - **例子：** int * p = new int;
  - **功能：**
    1、计算指定数据类型需要的内存空间大小；
    2、返回正确的指针类型；
    3、**程序员的职责**是将该地址赋给一个指针，以后的所有操作都是通过该指针来间接操作的（*p）。
- delete，“删除”（拆房子）
  - **格式：**delete 指针变量;
  - **例子：**delete p;
  - **功能：** delete 将释放指定指针所指向的内存空间，在使用完内存后，能够将其归还给内存池。

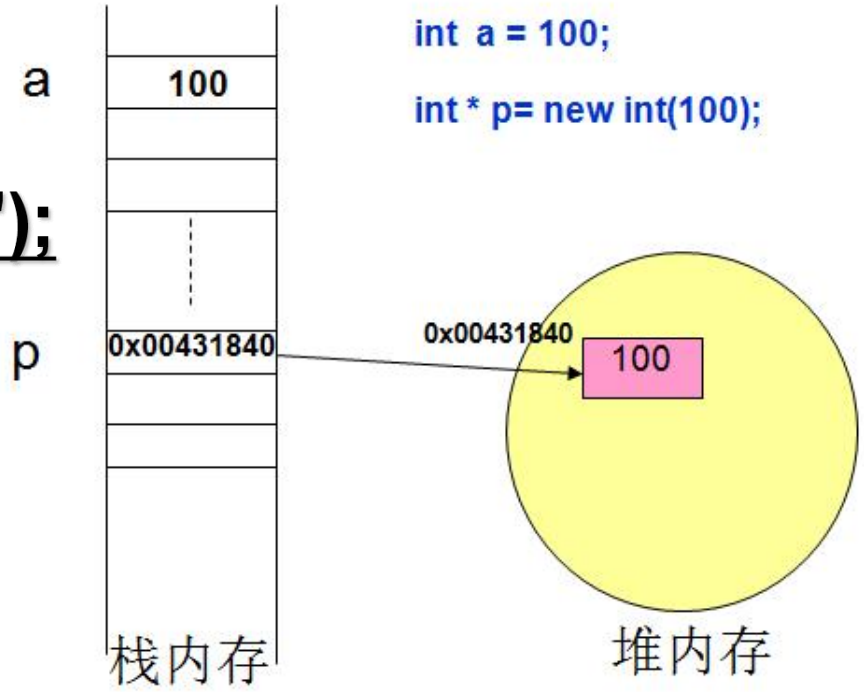
在new 时初始化内存的值

- 语法：指针变量 = new 数据类型(初值);

int* p = new int(100);

cout << *p << endl;

char* pchar = new char('A');

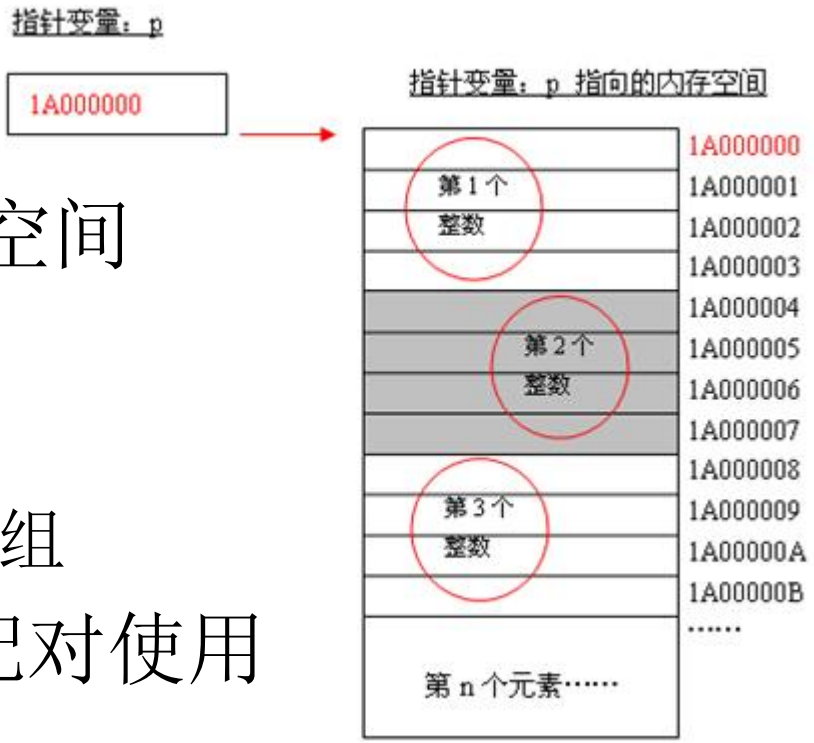


- 区分不同的存储区域
  - 栈区和堆区
- new和delete配对使用
  - new无法和free使用；
  - 不是new来创建的空间，也不能单独使用delete

```
int jugs=5; int *pi=&jugs;
delete pi;
```

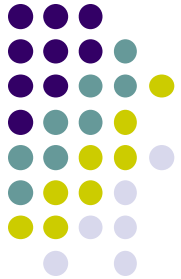
C++中内存管理：连续多个变量

- new[ ]，“新建” 连续空间
  - 格式：数据类型 指针变量 = **new** 数据类型**[元素个数]**；
  - 例子： int* p = new int[20];



- delete [ ]，“删除” 连续空间
  - 格式：delete []指针变量;
  - 例子：delete [] p;
  - 功能： delete [ ]释放整个数组
- 强调：new [ ]和delete [ ]配对使用
- 特性与应用场景：
  - 使用new，在程序的运行阶段，根据用户提供的信息（数组的长度），动态创建数组

new 和 delete 的关系



- 总的“土地”有限！不用的房子要收回！
- new和delete的“唇齿相依”关系
  1、new创建的，需使用delete来释放
  2、new创建的，若没有delete，会造成“内存泄漏”
  不类似栈区或静态区，变量是由生命周期来界定的
- 例子：

```
int* p;
p = new int;
*p = 100;
cout << *p << endl;
delete p;
```

分析：

1、delete p执行后，p指针还在吗？

delete释放p指向的内存，但不会删除指针p本身,p可以重新指向新的内存块.

2、p在，里面的值是多少？

p中的值还保留着原先的地址值。

比较：new、delete与malloc、free

- 函　数：malloc()和free()
- 运算符：new和delete

运算符：

```
int *p;
p=new int;
*p=1;  //delete p;
```

函数：

```
int *p;
p=(int *)malloc(int);
*p=1;  //free p;
```

共性：

1 都是创建一个空间后，返回值保存在某个指针变量中，通过该变量来访问该空间。

2 创建后都需要对应地手动释放，将堆内存归还给系统，否则会造成内存泄漏。

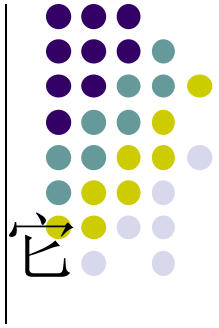
但是：在malloc()的返回值是void*，需要强制类型转换；且两者之间无法跨越。

关于存储，你应该知道的事

- 数据的三种存储方式
  - **自动存储：对应栈区stack，如**函数调用、局部变量
  - **静态存储：对应全局与静态区，如**全局变量、静态变量
  - **动态存储：对应堆区heap，**程序执行时，动态申请的内存

```
int * pt ;
    //声明了一个pt指针，四个字节，放在栈里面的
pt = new int;
   // 新建一个int形的数据空间放在堆里面，再把这个数据的地址赋给pt。
delete pt;
  //把pt指向的地址所占的内存释放掉。
```

有限的堆区（自由存储区）



- 用于动态分配的内存空间又称为自由存储区，它通常由操作系统进行管理且数量是有限的
- 若堆内存耗尽，系统无法再对new提出的堆内存分配请求给予满足，动态存储分配操作失败。因此在执行动态存储分配后要检查new返回的指针是否为空。

```
int *pia=new int[1024];
if (pia==NULL) {
    //堆内存耗尽，动态存储分配失败，退出程序
    cout<<“Cannot allocate more M.\n”;
    exit(1);
 }
```

内存泄漏与悬浮指针

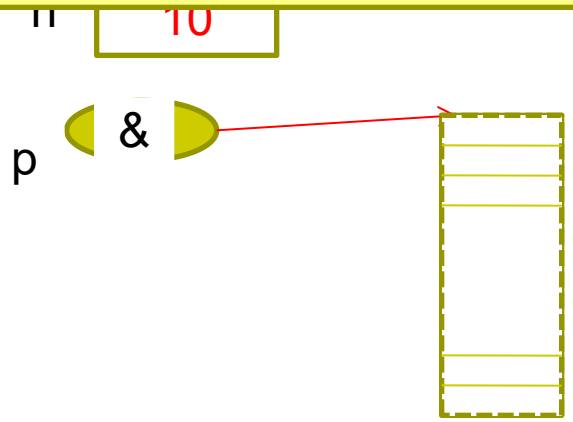
- 内存泄漏【失控的卫星】

解决的思路：不忘记使用delete将空间释放

- 悬浮指针【乱指的遥控器】

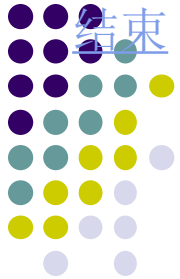
解决的思路： 指针变量delete之后，设置为NULL

```
int *p;
int n; n=10;
p=new int[n];
delete []p;
p[0]=10; //Error
```



在数组撤销后，指针变量p中还存储的动态数组的地址，但是实际这段空间已经归还给系统，可能已经另有用。

提纲



- 数组回顾
- 动态内存分配：概述
- 动态变量
- **动态数组：侧重在应用**
- 链表

数组长度的困惑

- 数组的使用时，程序员需要在评估数组大小之后，预先定义数组【静态内存分配】
  即： **int a[n]; //n是变量** **Error**
- 伪进步：

```
#define Maxlenth 100
int a[Maxlenth];  //预先申请大量空间
int i, n; cin>>n; //n需要小于100
for (i=0;i<n;i++)
    { cin>>a[i]; … }
```

- 期望，能够**动态获取一组**空间
  1 动态获取：在程序运行过程中，创建空间（不是依赖于预先定义）
  2 一组空间:各个个体之间不一定要“紧挨着”，即要有某种方式（显示或隐式）将变量空间“串”起来

动态数组创建

创建和撤销某段连续空间

```
int n,*p;
cin>>n;
p=new int[n] //int [n]
delete [] p;
```

- 动态数组：动态+数组
  - 用户程序执行过程**动态**创建空间
  - 创建的区域是**连续占据**一段空间（**类似数组**）
    与数组相比：可以在程序运行中指定空间长度
- new []和delete[ ]可以用来创建和撤销动态数组
- **问题：**如何合理地使用空间，即满足功能需要，又不会浪费？
- 一种**常用**的方法
  首先创建一个较小的动态数组用于存储数据，当这个数组放不下处理的数据时：
  1 创建一个较大的动态数组
  2 将原来数组中的数据复制过来
  3 再撤销原来数组
  4 继续后续操作

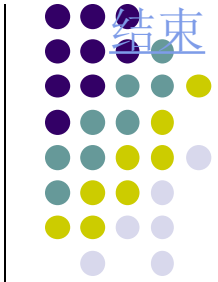
动态数组的访问

返回

- 动态数组的访问
  - 动态数组的优势在于，能在程序的运行过程中动态地确定数组的长度。
  - 特质：存储空间的特质是连续的，因此**访问模式可以参考数组（访问模式）**
- 动态数组没有数组名，**只能通过地址间接访问**(循环)

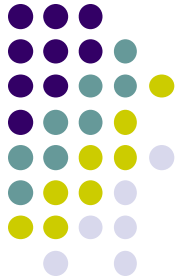
```
int n=get_size();  //get_size()泛指某种获取长度的方式
int *p=new int[n];
for (int *q=p;q!=p+n;++q)
    process() ;       //process()泛指动态数组的处理
```

提纲



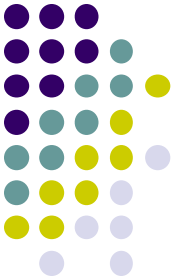
- 数组回顾
- 动态内存分配：概述
- 动态变量
- 动态数组
- **链表概述**

链表与链表的基本操作



- 线性表是最简单，最常用的一种数据结构。
- 线性表的逻辑结构是n个数据元素的有限序列（a1, a2, …, an）。
- 线性表的物理结构包括：顺序表，链表 。

线式存储的不足和链式存储的出现



- **原因：**
  1数组（传统或动态数组）都是连续占用存储空间，添加或删除元素，需要移动大量数据；
  2 往往无法准确地知道数据量，难免会有空间的浪费。没有真正地实现“按需”申请空间。

- **链表形成步骤：**
  1 某单个空间的形成，即结点
  2 如何让空间之间“连接”起来
  思考：数组中是如何连接的？

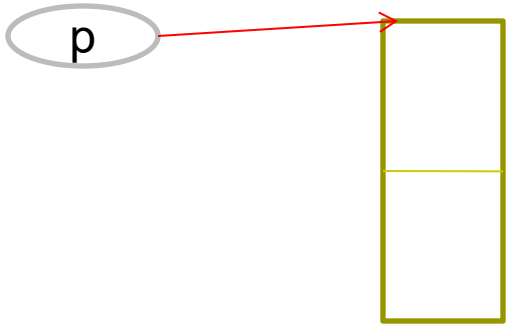
数组中空间的连接是依赖彼此空间之间的连续性

第一步骤：结点的分析与形成

- 结点的构成分析【**设计图纸**】
  除了要存储数据外，还需要额外存储与其他元素之间的逻辑关系.由于下个空间是按需生成，位置上不一定相邻，用指针来指向，因此，还需要有“指针域”

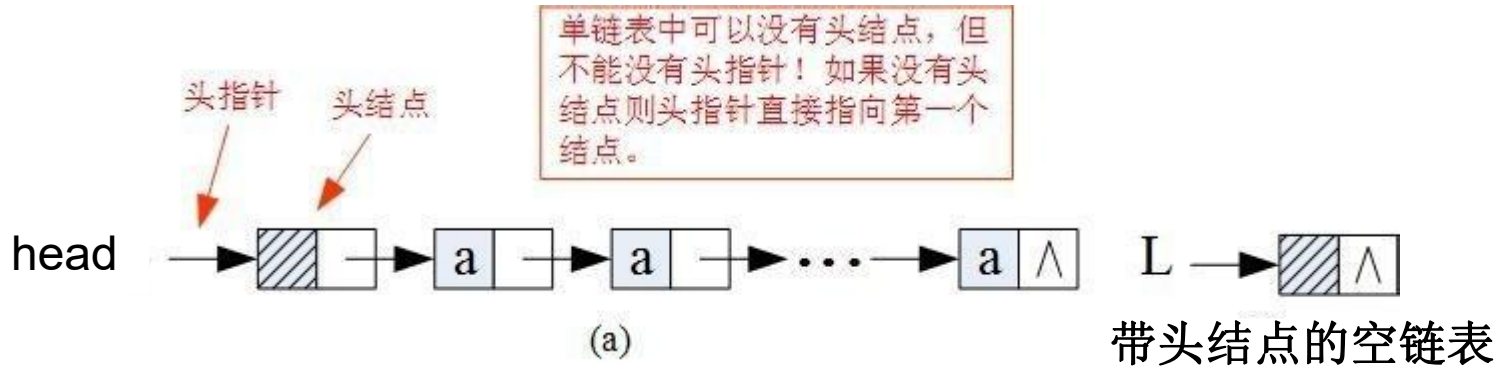

```
struct Node {
   int content; //结点数据
   Node *next; //下一个结点的地址
};
```

- 结点的形成【**施工**】
  Node *p=new Node;
  p->content=‘a’;



第二步骤：链表的形成

- 几个基本的指针
  - head指针
  - 最后结点的指针域：NULL
- 带头结点和不带头结点



- 链表的形成过程就是添加结点的过程，分：
  - 新添加的结点在表头
  - 新添加的结点在表尾

关键定义：链表的基本操作

返回

- 什么叫基本操作？

  链表的本质是存储单元，为了实现应用，类比数组中的a[i]等（视为：基本操作），链表也应有为应用提供支撑的基础操作

- 有哪些基本操作？
  - 随机访问某个结点
  - 创建一个空链表
  - 在链表中插入一个结点
  - 在链表中删除一个结点
  - 在链表中检索某个值
  - 链表的输出
  - 链表的释放（逐一元素的释放）

•操作视为一个个函数，关注实现

•难点：需要与数组存储方式做比较，分析时间性能

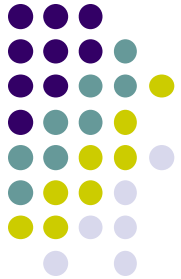
关键定义：链表的应用

返回

- 链表本质与数组一样，是一种用来存储数据的空间（只是这种空间的形成是按需分配产生）
- 在链表的基本操作都已经函数实现，并可直接提供调用的前提下，使用链表来**解决问题**
  如：通讯录

- **要从问题域出发，分析线性存储和链式存储的优缺点，选择合适的存储形式。**

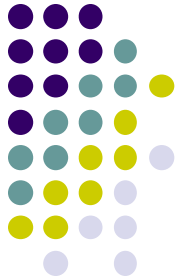
采用链表的意义



- 与定长数组相比，链表能更好地利用内存，按需分配和释放存储空间；
- 在链表中插入或删除一个节点，只需改变某节点“链接”成员的指向，而不需移动其他节点，相对数组元素的插入和删除效率高；

  因此，链表适合于对大线性表频繁插入和删除元素或成员数目不定的情景。

关于链表的代码实现



- 基于C语言
- **基于C++语言（C++、类、模板和STL）**
- 基于Java