

《面向对象程序设计》朋辈课程 · 第二辑

动态存储

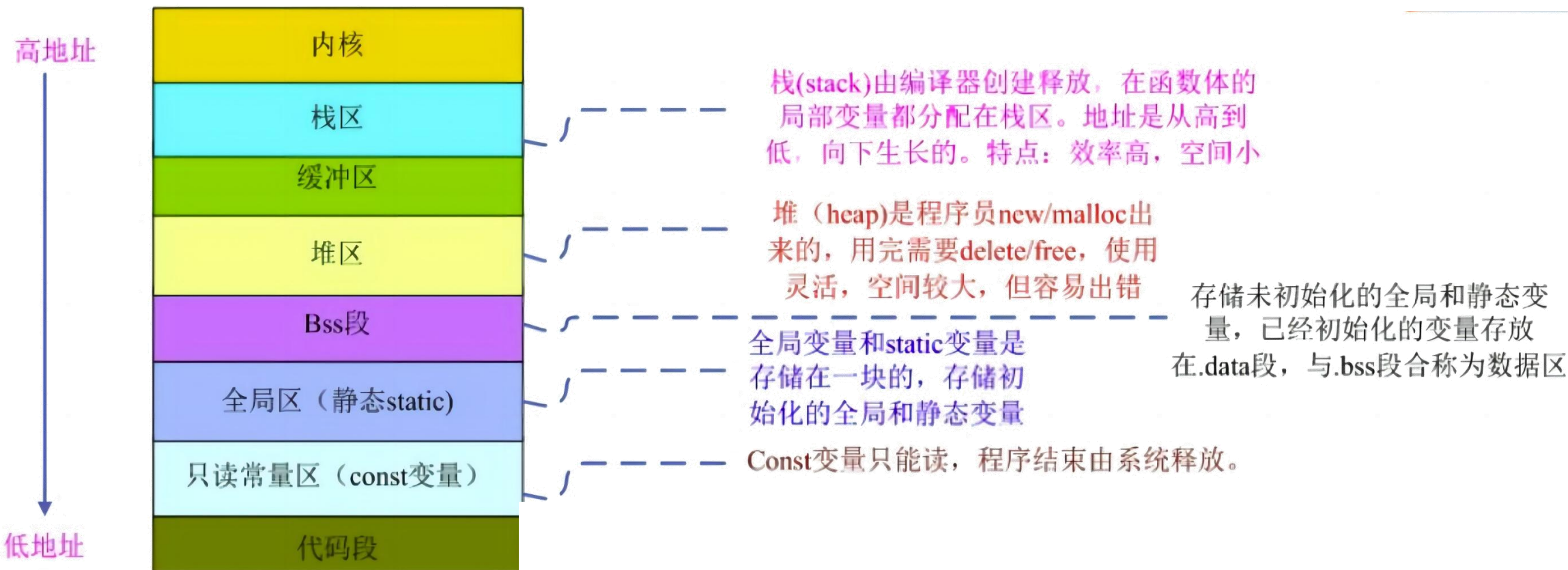
信息学院 赵家宇

The background features a gradient from dark blue on the left to light green on the right. There are three overlapping circles: a large dark green one on the left, a large light green one in the center, and a smaller purple one in the bottom left corner.

PART 01

动态变量与数组

计算机内存分布



不同的内存分配

- 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：在编译时，程序暂不确定所需内存空间的大小；在程序运行阶段，再确定存储空间大小。

单个变量的动态管理：new和delete

- 一般通过new申请一个变量内存空间，其使用方法为：数据类型 指针变量 = new 数据类型；功能包括：
 - 计算指定数据类型需要的内存空间大小；
 - 返回正确的指针类型；
 - 将申请的内存地址赋给一个指针，以后所有操作都通过该指针来间接操作。
- 在new时可以初始化内存的值，使用方法：指针变量 = new 数据类型(初值)；

```
int *p = new int(9);  
delete *p;
```

单个变量的动态管理：new和delete

- 一般通过delete释放一个变量内存空间，其使用方法为：`delete 指针变量`；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。
- new创建的，需使用delete来释放；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。

```
int *p = new int(9);  
delete *p;
```

连续变量的动态管理：new[]和delete[]

- 在new时可以申请连续的内存空间，使用方法为：

指针变量 = new 数据类型[元素个数];

- 相应的，需要使用delete[]去释放连续空间，使用方法为：

delete[] 指针变量;

- 需要注意，new[]和delete[]必须配对使用。

```
int *p = new int[100];  
char *q = new char[10];  
delete[] p;  
delete[] q;
```

对比: new/delete和malloc/free

相同点:

- 创建空间后地址保存在指针变量中, 通过该指针间接访问该空间。
- 创建后需要对应手动释放, 将堆内存归还给系统, 否则会造成内存泄漏。

区别:

- malloc()和free()是函数, new和delete是关键字;
- malloc()的返回值是void*, 需要强制类型转换, 且这个转换过程不可逆。

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

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


数据的存储方式

- 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
```

数据的存储方式

- 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，通过`new[]`和`delete[]`可以用来创建和撤销动态数组。
- 如何合理地使用空间，即满足功能需要，又不会浪费？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，创建一个较大的动态数组，将原来数组中的数据复制过来，再撤销原来数组并继续后续操作。

数据的存储方式

- 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；由于存储空间的特质是连续的，因此访问模式可以参考数组。但动态数组没有数组名，只能通过地址间接访问。

```
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
```



PART 02

标准库类型

C++引入了标准库类型

- C++中的数据类型：基本数据类型(内置数据类型)、构造数据类型、抽象数据类型(程序员自定义数据类型)
- C++引入的标准库类型，其并非内置在语言中，而是定义成类；这其中，string和vector是两种最基本的标准库类型：string定义了长度可变的字符串，vector定义了大小(长度)可变的集合。
- 掌握C++的第一步是学习语言的基本知识和标准库。

字符串类型string

- 与其他标准库类型一样，用户程序要使用string类型对象，必须包含相关头文件：

```
#include <string>
using std::string;
```

- string类型支持长度可变的字符串。使用string类将字符串定义为对象，然后利用string类提供的赋值、连接等字符串操作功能，方便地实现对字符串的各种处理。

string对象的定义和初始化

- 定义在形式上与定义变量类似，例如： `string movieTitle;`
- 可以通过赋值运算符初始化string类对象，例如： `string Title="2024";` 也可采用“函数调用表示法”，如下表所示。

<code>string s1;</code>	默认构造函数, s1为空串
<code>string s2(s1);</code>	复制构造函数, 将s2初始化为s1的一个副本(s1为字符数组或string)
<code>string s3("value");</code>	将s3初始化为一个字符串字面值副本
<code>string s4(n, 'c');</code>	将s4初始化为字符'c'的n个副本

string类对象的读写

- 使用标准输入输出函数符来读写string对象。

- 例如

```
string name;  
cout<<"What is your name?";  
cin>>name;  
cout<<name<<"， 你好" <<endl;
```

- string类型的输入操作符，特性：读取并忽略开头所有的空白字符（空格、换行、制表符）；读取字符直至再次遇到空白字符，读取终止。

string类对象的读写

- 用getline读取整行文本：getline(cin, str); 其中str是string对象。
- 从输入流的下一行读取，并保存读取的内容到string中，但不包括换行符（将换行符转为'\0'）；若行开头出现换行符，getline并不忽略而是停止读入；

string类对象的操作

- string类对象的操作,即实现对字符串进行赋值、连接、复制、查找和交换等功能。基本形式: 对象名.成员函数 s对string对象

s.empty()	如果s为空串,则返回true,否则返回false	
s.size()	返回s中字符个数	
s[n]	返回s中位置为n的字符,位置从0开始计数	
s1+s2	s1,s2连接成一个新的字符串	strcat()
s1=s2	把s1内容替换s2的副本	strcpy()
v1==v2	比较v1和v2内容	strcmp()
!=,<,<=,>,>=	保持这些操作符惯有含义	

string类对象的操作

- 字符数组存储的字符串，无法直接使用+；而string可以使用+或+=（连接）；
- 使用+时左右操作数必须至少有一个是string类型，从而实现+运算符的重载。

string s4="hello"+"," 错误

string s5=s1+"","+"world"; 正确

string s6="hello"+","+s2; 错误

- size()是string类型的重要函数，的返回值类型size_type；size()成员函数返回的是string::size_type类型（size_type属于配套类型）；因此不要把size()返回值赋给一个int变量。

容器模版vector

- 与其他标准库类型一样，用户程序要使用vector类型对象，必须包含相关头文件：`#include <vector>`
- vector（容器）类模板是一种更加健壮，且有许多附加功能的数组，其附加功能包括提供下标越界检查、提供数组用相等运算和大小比较、提供数组间赋值等运算。
- vector可以用来存放不同类型的对象。容器中每个对象都有一个对应的整数索引值。在数组生存期内，数组的大小是不会改变的，vector容器则可在运行中动态地增长或缩小；

容器模版vector

- vector不是一种数据类型，而只是一个类模板，可用来定义任意多种数据类型。vector<int>是一种数据类型。因此，使用vector时必须说明vector中保存的对象的类型，通过将类型放在类模板名称后面的<>来指定类型。例如：

```
vector<int> ivec;    //内置类型信息
```

```
vector<Sales_item> Sales_vec; //类类型
```

vector对象的定义和初始化

- vector类提供4种构造函数，用来定义且初始化vector对象。若用T表示数据类型，对象名为v1和v2，则有：

<code>vector<T> v1;</code>	定义容器对象v1,保存类型为T.默认构造函数v1为空.
<code>vector<T> v1(length);</code>	定义length个元素，元素初始值由T来决定.若int,初始为0.
<code>vector<T> v1(length,a);</code>	定义length个元素,元素初始化为a,a的类型为T.
<code>vector<T> v2(v1);</code>	使用已定义的容器构造新容器.要求v2和v1中必须保存同一种元素类型.

vector对象的定义和初始化

- 可以规定元素个数和元素值实现对vector对象的初始化，例如

```
vector<int> ivec4(10,-1); vector<string> svec(10,"Hi!");
```

- vector对象（以及其他标准库容器对象）的重要属性在于可以在运行时高效地添加元素；使用vector时，可以先初始化一个空vector对象，然后再动态地增加元素；
- 向vector添加元素使用函数push_back()。

vector对象的操作

- vector类对象操作的基本形式：对象名.成员函数

<code>s.empty()</code>	如果s为空，则返回true，否则返回false	
<code>s.size()</code>	返回s中元素个数	
<code>s[n]</code>	返回s中位置为n的元素	
<code>s.push_back(t)</code>	在s的末尾增加一个值为t的元素	
<code>s1=s2</code>	把s1内容替换s2的副本	这在之前数组的操作 (整体操作)中是无法实现的！
<code>v1==v2</code>	比较v1和v2内容	
<code>!=,<,<=,>,>=</code>	保持这些操作符惯有含义	

vector对象的操作

- 可以使用下标来访问vector容器中的元素，其仅能对确知已存在的元素进行操作；下标只能用于表达已确定存在的元素；通过下标操作进行赋值时，不会添加任何元素；若事先定义vector容器为空或是较小的长度，则必须通过push_back()来实现元素的添加(实现长度的动态扩展)；
- 赋值运算符=是运算符重载的结果，vector定义的赋值运算符“=”允许同类型的vector对象相互赋值，而不管它们的长度如何；它可以改变赋值目标的大小，使它的元素数目与赋值源的元素数目相同。

vector对象与迭代器iterator

- vector对象的元素除了使用下标来访问，还可以通过迭代器iterator访问元素；
- 迭代器(遍历器)是一种检查容器内元素并遍历元素的数据类型；标准库为每一种标准容器（包括vector）定义了相应的迭代器类型；迭代器类型提供了比下标操作更通用化的方法。
- 迭代器是配套类型访问vector中的元素，可理解为面向对象版本的指针，可作解引用操作来访问元素。现代C++更倾向于使用迭代器而不是下标操作访问容器元素。

迭代器的操作

- `begin`和`end`操作是迭代器的基本操作之一：`begin()`返回迭代器(指针)，如果容器不为空则指向第一个元素；`end()`返回迭代器(指针)，指向末端元素的下一个（实际指向一个不存在的元素）；
- 如果`vector`为空，那么`begin()`和`end()`返回的迭代器相同；如果`vector`不为空，元素存在的范围是半开区间`[begin, end)`；哨兵的作用，表示我们已处理完了`vector`中所有元素。
- 迭代器`*`表示解引用，访问(表示)迭代器所指向的元素，例如`*iter=0`中`iter`是迭代器(指针)变量。

迭代器的操作

- 迭代器的算术操作可理解成指针变量的算术运算，其运算结果仍为迭代器，取值范围： `[begin(),end())`;
- 例如：

`iter+n(iter-n);`//相当于下标

`iter1-iter2` //用于计算两个迭代器对象的距离

```
for (vector<int>::iterator iter=ivec.begin(); iter!=ivec.end(); ++iter)
    *iter=0;
```

常迭代器const_iterator

- 使用const_iterator类型定义的迭代器，(迭代器)自身值是可以改变的，但是不能改变其所指向的元素的值，只能用于读取容器内元素；
- const_iterator对象与const的iterator对象比较：const_iterator对象，它的侧重点在于描述对所指的元素只具有读，而不能具有修改；定义一个const的迭代器时，必须初始化迭代器；且在运行过程中无法改变(迭代器本身)值；而对于它所指向的元素是否修改没有限制。

常迭代器const_iterator

- 使用const_iterator类型定义的迭代器，其自身指向是可以改变的，但是不能改变其所指向的元素的值，只能用于读取容器内元素；
- const_iterator对象与const的iterator对象不同，区别在于const_iterator对象描述对所指的元素只具有读，而不能具有修改；定义const的迭代器时必须初始化迭代器；且在运行过程中无法改变(迭代器本身)值；而对于它所指向的元素是否修改没有限制。例如：

```
vector<int> nums(10); const vector<int>::iterator cit=nums.begin();  
  
*cit=1;           //OK,对指向元素的修改  
  
++cit;            //Error.cit定义时有const修饰符
```

string和vector的搭配使用

- 当只有一个字符串的时候直接使用string即可；而当有多个字符串的时候，使用 `vector<string>`；
- 当不确定长度的时候可以用vector，因为不需要定义长度，有新元素时通过插入就可增加长度；并且可以利用库中的几个函数进行操作，如查找，插入操作等通过函数和迭代器就可以解决。

The background features a gradient from dark blue on the left to light green on the right. There are three overlapping circles: a large dark green one on the left, a large light green one in the center, and a smaller purple one in the bottom left corner.

PART 03

链表

线性表与链表

- 线性表是最简单，最常用的一种数据结构，其逻辑结构是 n 个数据元素的有限序列 (a_1, a_2, \dots, a_n) 。
- 线性表的物理结构包括顺序表和链表。顺序表（数组或动态数组）都是占用存储空间都是连续的，添加或删除元素，需要移动大量数据；
- 在生成顺序表时，往往无法准确地知道数据量，因而难免会有空间的浪费；链表能够真正地实现按需申请空间。

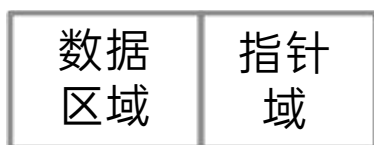
链表的优势

- 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；
- 与动态数组相比，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，或是元素或成员数目不定的情景。

链表的生成

- 第一步骤：结点的分析与形成

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。



用结构体

```
struct Node {  
    int content; //结点数据  
    Node *next; //下一个结点的地址  
};  
Node *p = new Node; //生成一个节点  
p->content = 'a'; //为节点赋值
```

链表的生成

- 第二步骤：链表的形成

需要几个基本的指针：head指针（可选）、tail指针（可选），最后结点的指针域必须为NULL。

链表的形成过程就是添加结点的过程，分为**头插法**（新添加的结点在表头）和**尾插法**（新添加的结点在表尾）。

```
while(...){  
    Node *q = new Node;  
    //生成一个节点  
    p->next = q;  
    q->content = 'B';  
    p = q;  
}
```

链表的基本操作

链表的基本操作举例：

- 随机访问某个结点
 - 创建一个空链表
 - 在链表中插入结点
 - 在链表中删除结点
 - 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）
- 链表本质与数组一样，是一种用来存储数据的不连续的空间；
 - 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。

The background features a gradient from dark blue on the left to light green on the right. There are three overlapping circles: a large dark green one on the left, a large light green one in the center, and a smaller purple one at the bottom left. The text is positioned to the right of the green circles.

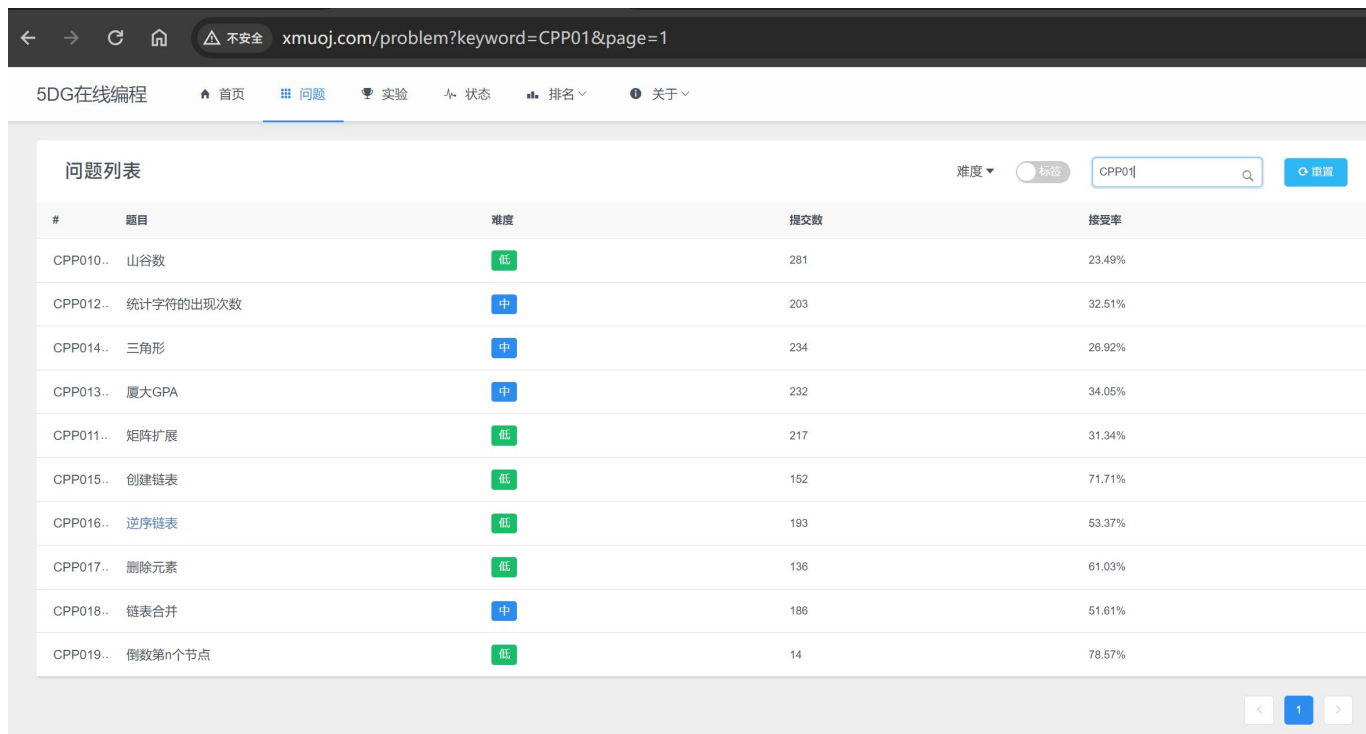
PART 04

期中复习实战

链表实操内容

- 建议完成xmuoj.com平台上的如下题目，增强对C++链表的掌握：

- **CPP015** 创建链表
- CPP016 逆序链表
- CPP017 删除元素
- **CPP018** 链表合并
- CPP019 倒数第n个节点
- CPP020 后半链表
- CPP021 合并子链表
- **CPP022** 链表重排



The screenshot shows the '问题列表' (Problem List) on the xmuoj.com website. The search bar contains 'CPP01'. The table lists 10 problems with their IDs, titles, difficulty levels, submission counts, and acceptance rates.

#	题目	难度	提交数	接受率
CPP010...	山谷数	低	281	23.49%
CPP012...	统计字符的出现次数	中	203	32.51%
CPP014...	三角形	中	234	26.92%
CPP013...	厦大GPA	中	232	34.05%
CPP011...	矩阵扩展	低	217	31.34%
CPP015...	创建链表	低	152	71.71%
CPP016...	逆序链表	低	193	53.37%
CPP017...	删除元素	低	136	61.03%
CPP018...	链表合并	中	186	51.61%
CPP019...	倒数第n个节点	低	14	78.57%

实操1：生成链表（CPP015）

- 当要输入数的个数不确定时，可以用链表来存储这些数。我们首先要做的是创建一个链表。
- **输入：**输入一串正整数和-1，两个数之间用空格隔开，以-1作为结束标记。在输入的数中，只有正整数和-1，不会出现其他数，且-1一定出现在最后。
- **输出：**按输入时的顺序，输出所有的正整数，且每个正整数后输出一个空格。如果没有输入任何正整数，则直接输出-1。

Input Example:

5 3 2 6 100 9 12 30232 -1

Output Example:

5 3 2 6 100 9 12 30232

Input Example:

-1

Output Example:

-1

实操1：生成链表（CPP015）

```
struct node {
    int val; node* next;
};
node* createList() {
    //将这个函数补充完整
}
int main() {
    node* head = createList();
    if (head == NULL) cout << "-1 ";
    else node* p = head;
    while (p != NULL){
        cout << p->val << " ";
        p = p->next;}
}
```

```
node* createList() {
    node *head = NULL, *tail = NULL;
    int val;
    cin >> val;
    if (val == -1)
        return NULL;
    head = new node;
    tail = head;
    tail->val = val;
    cin >> val;
    while (val != -1) {
        tail->next = new node;
        tail = tail->next;
        tail->val = val;
        cin >> val;
    }
    return head;
}
```

实操2：链表合并（CPP018）

- 两个链表存的数都是升序的，对两个链表做合并，合并后的链表仍然是升序的。
- 输入：输入数据包括两行。每行均输入一串正整数和-1，两个数之间用空格隔开，以-1作为结束标记；只有正整数和-1，不会出现其他数，且-1一定出现在最后；输入的正数都是升序的。
- 输出：依次输出合并后的链表中元素。如果链表为空，则直接输出-1。

Input Example:

1 2 3 5 -1

2 4 9 -1

Output Example:

1 2 2 3 4 5 9

Input Example:

-1

-1

Output Example:

-1

实操2：链表合并 (CPP018)

```
struct node { int val; node* next; };
node* createList(){
    //此函数实现链表的创建，实现省略
}
node* merge(node* head1, node* head2){
    //将这个函数补充完整
}
int main() {
    node *head1 = createList();
    node *head2 = createList();
    node *head3 = merge(head1, head2);
    if (head3 == NULL) cout << "-1 ";
    else {
        node *p = head3;
        while (p != NULL) {
            cout<<p->val<<" "; p = p->next;}
    }
}
```

```
node* merge(node* head1, node* head2) {
    node* dum = new ListNode(0), cur = dum;
    node* list1 = head1, list2 = head2;
    while (list1!=NULL && list2!=NULL) {
        if (list1->val < list2->val) {
            cur->next = list1;
            list1 = list1->next;
        }
        else {
            cur->next = list2;
            list2 = list2->next;
        }
        cur = cur->next;
    }
    cur->next = (list1!=NULL)?list1:list2;
    return dum->next;
}
```

实操3：链表重组(CPP022)

- 输入一个链表，进行重新分组排列：所有索引为奇数的节点为第一组，所有索引为偶数的节点为第二组。第一个节点被看作奇数节点，重新排列后的两组内部的相对顺序应该与输入时保持一致。
- **输入：**输入一串正整数和-1，两个数之间用空格隔开，以-1作为结束标记；只有正整数和-1，不会出现其他数，且-1一定出现在最后。
- **输出：**输出重新排列后的链表，每个正整数后面有一个空格。如果输入链表为空，则输出-1。

Input Example:

2 1 3 5 6 4 7 -1

Output Example:

2 3 6 7 1 5 4

Input Example:

-1

Output Example:

-1

实操3：链表重组(CPP022)

```
struct ListNode {
    int val; ListNode* next;
};
ListNode* createList(){
    //此函数实现了链表的创建，实现省略
}
ListNode* oddEvenList(ListNode* head) {
    //这个函数需要你补充完整
}
int main() {
    ListNode* head = createList();
    ListNode* ans = oddEvenList(head);
    if (ans == NULL) cout << "-1";
    else {
        while (ans != NULL) {
            cout << ans->val << " ";
            ans = ans->next;
        }
    }
```

```
ListNode* oddEvenList(ListNode* head) {
    if (head == NULL) return head;
    ListNode* evenHead = head->next;
    ListNode* odd = head;
    ListNode* even = evenHead;
    while (even!=NULL && even->next!=NULL) {
        odd->next = even->next;
        odd = odd->next;
        even->next = odd->next;
        even = even->next;
    }
    odd->next = evenHead;
    return head;
}
```

期中上机拓展内容

- 以下拓展题请在leetcode.cn中进行挑战:

- LCR007 三数之和
- LCR181 字符串中的单词反转
- LCR182 动态口令
- LCR154 复杂链表的复制
- 92 反转链表II
- 328 奇偶链表
- 812 最大三角形面积
- 2130 最大孪生和

拓展1：字符串中的单词反转(LCR181)

- 你在与一位习惯从右往左阅读的朋友发消息，他发出的文字顺序都与正常相反但单词内容正确，为了和他顺利交流你决定写一个转换程序，把他所发的消息 `message` 转换为正常语序。
- 注意：输入字符串 `message` 中可能会存在前导空格、尾随空格或者单词间的多个空格。输入字符串可能在前面、后面或单词中间包含多余的空格。返回的结果字符串中，单词间应当仅用单个空格分隔，且不包含任何额外的空格。

Input Example:

`the sky is blue`

Output Example:

`blue is sky the`

Input Example:

`hello world!`

Output Example:

`world! hello`

拓展1：字符串中的单词反转(LCR181)

```
char* reverseWords (char* s) {
    int len = strlen(s);
    char *ans = (char *)calloc(1, len+1), *m;
    ans += len;
    while (true) {
        while (*s == ' ') ++s;
        if (m = strstr(s, " ")) {
            strncpy(ans-=m-s, s, m-s);
            *-- ans = ' ', s = m;
        }
        else {
            ans -= (len = strlen(s));
            strncpy(ans, s, len);
            return ans + (*ans == ' ');
        }
    }
}
```

```
string reverseWords (string s) {
    int n = s.size();
    string ans = "";
    for (int i = n - 1; i >= 0; i--) {
        if (s[i] != ' ') {
            int right = i;
            while (i >= 0 && s[i] != ' ')
                i--;
            ans+=s.substr(i+1, right-i)+" ";
        }
    }
    return ans.substr(0, ans.size()-1);
}
```

- 思考：若输出的字符串中要求标点符号位置与输入一致，可以怎样修改？

拓展2：最大孪生和（2130）

- 在一个大小为 n 且 n 为偶数的链表中，任意 $0 \leq i \leq n/2-1$ ，第 i 个节点（下标从0开始）的孪生节点为第 $(n-1-i)$ 个节点。例如 $n = 4$ 的链表中节点0是节点3的孪生节点，节点1是节点2的孪生节点。
- 孪生和为一个节点和它孪生节点两者值之和。给你一个长度为偶数的链表的头节点head，请你返回链表的`最大孪生和`。

Input Example:

5,4,2,1

Output Example:

6

Input Example:

1 10000 -1000 10999 -1 1

Output Example:

9999

拓展2：最大孪生和（2130）

```
int pairSum(ListNode* head) {
    ListNode *slow = head, *fast = head->next;
    while (fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }
    ListNode* last = slow->next;
    while (last->next) {
        ListNode* cur = last->next;
        last->next = cur->next;
        cur->next = slow->next;
        slow->next = cur;
    }
    int ans = 0;
    ListNode* x = head, *y = slow->next;
    while (y) {
        ans = max(ans, x->val + y->val);
        x = x->next; y = y->next;
    }
    return ans;
}
```

```
int pairSum(ListNode* head) {
    vector<int> arr;
    while (head) {
        arr.push_back(head->val);
        head = head->next;
    }
    int maxSum = 0, n = arr.size();
    for (int i = 0; i < n / 2; i++)
        maxSum = max(maxSum, arr[i] + arr[n-i-1]);
    return maxSum;
}
```

- 思考：试拓展到n为奇数的情况。

拓展3：反转链表（92）

- 给你单链表的头指针 `head` 和两个整数 `left` 和 `right` ($left \leq right$)，请你反转从位置 `left` 到位置 `right` 的链表节点，返回反转后的链表。

Input Example:

`head = [1,2,3,4,5]`, `left = 2`, `right = 4`

Output Example:

`[1,4,3,2,5]`

Input Example:

`head = [5]`, `left = 1`, `right = 1`

Output Example:

`[5]`

```
struct ListNode {  
    int val;  
    ListNode *next;  
    ListNode(int x) : val(x), next(NULL)  
    {}  
};
```

拓展3：反转链表（92）

```
ListNode* reverseBetween(ListNode* head, int
m, int n) {
    ListNode* dummy=new ListNode(-1);
    ListNode* pre=dummy;
    dummy->next=head;

    for(int i=0;i<m-1;i++)
        pre=pre->next;
    ListNode* cur=pre->next;
    for(int i=m;i<n;i++) {
        ListNode* t=cur->next;
        cur->next=t->next;
        t->next=pre->next;
        pre->next=t;
    }
    return dummy->next;
}
```

```
ListNode* reverseBetween(ListNode* head, int left, int right) {
    ListNode *q = head, *p, *r, *temp_head = NULL, *temp_tail;
    for(int i = 1; i < left; i++) {
        if(i == left - 1)temp_head = q;
        q = q->next;
    }
    temp_tail = q;//temp_tail指向left节点
    p = temp_head;
    r = q->next;
    for(int i = left; i < right; i++) {
        q->next = p;
        p = q; q = r; r = r->next;
    }
    q->next = p;
    if(temp_head)
        temp_head->next = q;
    temp_tail->next = r;
    return temp_head ? head : q;
}
```

感谢倾听

给个好评！