[自定义数据类型（构造数据类型）](#header-n2)  
 [1 结构体 struct](#header-n3)  
 [1.1 结构体定义的语法](#header-n4)  
 [1.2 结构体的使用](#header-n14)  
 [1.3 typedef](#header-n25)  
 [1.4 结构体定义时不允许出现的成员](#header-n39)  
 [2 联合体 union](#header-n52)   
 [3 枚举 enum](#header-n64)  
 [4. 动态内存分配](#header-n83)  
 [5. 链表](#header-n111)

### 自定义数据类型（构造数据类型）

#### 1 结构体 struct

##### 1.1 结构体定义的语法

struct 结构体标识符  
{  
 类型1 结构体成员1;  
 类型2 结构体成员2;  
 ....   
};

// 学生结构体类型  
struct student   
{  
 char name[20]; // 学生姓名  
 char id[20]; // 学生学号  
 int age; // 学生年龄  
 int c; // 学生C语言成绩  
};

* struct 是结构体的关键字
* student 是表示结构体类型的标识符
* **struct student 是定义的新的结构体类型名称**

##### 1.2 结构体的使用

* 定义结构体变量时，使用 struct + 结构体标识符 共同组成结构体类型名称。
* 访问结构体变量成员，需要使用 **成员访问运算符 （.运算符）**
* **结构体指针**访问结构体变量成员，需要通过 **结构体指针访问成员运算符**（->运算符）
* 结构体变量允许直接赋值，实际上就是内存字节的拷贝，把赋值号右侧结构体变量的每个字节拷贝到赋值号左侧的结构体变量内存中。

int main(void)   
{  
 struct student lizhe;  
 struct student \* lixiaozhe = &lizhe;  
  
 lizhe.c = 59;  
 lizhe.age = 18;  
 **strcpy(lizhe.name, "lizhe");** strcpy(lizhe.id, "2019011250");  
 printf("%s %s %d %d\n", lizhe.name, lizhe.id, lizhe.age, lizhe.c);  
  
 (\*lixiaozhe).c = 60;  
 printf("%s %s %d %d\n", lizhe.name, lizhe.id, lizhe.age, lizhe.c);  
  
 lixiaozhe->c = 59;  
 printf("%s %s %d %d\n", lizhe.name, lizhe.id, lizhe.age, lizhe.c);  
  
 return 0;  
}

##### 1.3 typedef

* 给一个已知的变量，起别名

typedef 已知类型名称 新名称;

* 结构体类型定义完成后起别名

struct student   
{  
 char name[20]; // 学生姓名  
 char id[20]; // 学生学号  
 int age; // 学生年龄  
 int c; // 学生C语言成绩  
};  
  
typedef struct student Student;

* **结构体类型定义同时起别名**

typedef struct student   
{  
 char name[20]; // 学生姓名  
 char id[20]; // 学生学号  
 int age; // 学生年龄  
 int c; // 学生C语言成绩  
} Student;

int main(void)   
{  
 struct student lizhe;  
 Student huchaoran; // Student 与 struct student 同一种类型  
 return 0;  
}

##### 1.4 结构体定义时不允许出现的成员

1. C语言不允许结构体中出现**静态生存周期**的成员；
2. 不允许结构体中出现**当前结构体类型**的成员，不允许直接或简介类型定义嵌套。因为编译器无法确定结构体内存大小。
3. 不允许出现**当前结构体别名的指针**，因为定义结构体类型时别名还未起作用。

typedef struct student   
{  
 char name[20]; // 学生姓名  
 char id[20]; // 学生学号  
 int age; // 学生年龄  
 int c; // 学生C语言成绩  
 // static int year; // 不允许出现静态生存周期成员  
 // struct student tongzhuo; // 不允许出现当前类型成员  
 struct student \* tongzhuo; // 允许定义当前结构体类型指针成员  
 // 因**指针大小恒定 32位4字节 64位8字节** // Student \* tongzhuo2; // 不允许出现使用当前结构体类型别名的指针  
} Student;

* 结构体数组示例

#include <stdio.h>  
  
// 从命令行依次输入5个学生的姓名和C语言成绩  
// 要求将5个学生的信息根据C语言成绩进行递增排序  
// 排序完成后输出  
  
typedef struct student   
{  
 char name[20];  
 int c;  
} Student;  
  
int main(void)  
{  
 Student arr[5];  
 int i, j;  
 for (i=0; i<5; i++) {  
 scanf("%s %d", arr[i].name, &arr[i].c);  
 }  
 for (i=0; i<5-1; i++) {  
 for (j=0; j<5-1-i; j++) {  
 if (arr[j].c > arr[j+1].c) {  
 **Student temp = arr[j];** arr[j] = arr[j+1];  
 arr[j+1] = temp;  
 }  
 }  
 }  
 for (i=0; i<5; i++) {  
 printf("%s %d\n", arr[i].name, arr[i].c);  
 }  
  
 return 0;  
}

#### 2 联合体 union

* 又叫做 共用体 或 联合
* 定义语法跟结构体一直，只不过关键字由 struct 变为 union
* 联合体中的所有成员共同使用同一块内存。
* 联合体在使用时，只有最后赋值的成员才会起作用，它不能保证之前赋值的成员数据还是有效数据。

#include <stdio.h>  
  
union test  
{  
 char a;  
 int b;  
};  
  
int main(void)   
{  
 union test t1;  
 t1.a = 'A';  
 t1.b = 10000;  
 printf("%c\n", t1.a);  
 printf("%d\n", t1.b);  
  
 t1.a = 'B';  
 printf("%c\n", t1.a);  
 printf("%d\n", t1.b);  
  
 return 0;  
}

#### 3 枚举 enum

* 枚举：将可能的取值一一列举出来。
* 枚举变量的值只能是列举出来的值中的一个。
* 定义语法：
* enum 枚举标识符 {有限集合元素列表}:
* enum weekday {sun, mon, tues, wednes, thurs, fri, satur};
* enum sex {boy, girl};
* 枚举类型在定义时，大括号中给出的是所有取值可能，并且用逗号隔开
* 枚举类型可以增加程序的可读性和安全性。
* 枚举在内存中保存的实际上就是一个正整数。
* 枚举本质上是整数，但是不能将整数赋值给枚举类型变量。

#### 4. 动态内存分配

* C语言允许开发者根据自己对内存的实际需求，来手动的申请内存及释放内存。
* 如果使用动态内存分配，需要引入头文件 malloc.h。

// 向系统申请size大小内存，返回申请到的内存首地址  
// 申请失败返回NULL。  
void \* malloc (unsigned int size);  
  
// 向系统申请 size\*count 大小的内存，返回申请到的内存首地址  
// 申请失败返回NULL。  
void \* calloc (unsigned int count, unsigned int size);  
  
// 将申请到的内存归还给系统  
void free(void \* p);

* 动态内存分配申请到的内存都是在内存的**堆区**。
  + 栈区：所有的动态生存周期的变量，局部变量。
  + 堆区：开发者动态申请到的内存。
  + 只读区：字符串字面值。
  + 静态全局区：静态生存周期的变量，全局变量和static局部变量。
  + 代码区：执行代码。
* malloc 函数不会对申请到的内存做任何初始化操作。
* calloc 函数会将申请到的内存初始化为0。
* free 函数会将你之前申请到的内存归还给系统。

#include <stdio.h>  
#include <malloc.h>  
  
int main(void)  
{  
 // 命令行输入n（1~10000），表示接下来会输入n个整数，  
 // 对n个整数从小到大排序后输出。  
  
 int n, i, j;  
 int \* p = NULL;  
  
 // 1. 读取n的值  
 scanf("%d", &n);  
 // 2. 根据n的值动态在堆区申请 n\*sizeof(int) 大小的内存空间  
 p = (int \*) malloc (n\*sizeof(int));   
 // 3. 命令行读取n个整数  
 for (i=0; i<n; i++) {  
 scanf("%d", p+i); // &p[i] <==> &\*(p+i) <==>p+i  
 }  
 // 4. 对n个整数进行从小到大冒泡排序  
 for (i=0; i<n-1; i++) {  
 for (j=0; j<n-1-i; j++) {  
 if (p[j] > p[j+1]) {  
 int temp = p[j];  
 p[j] = p[j+1];  
 p[j+1] = temp;  
 }  
 }  
 }  
 // 5. 将排序好的n个整数进行输出  
 for (i=0; i<n; i++) {  
 printf("%d\n", p[i]);  
 }  
 // 6. 释放申请到的内存空间  
 free(p);  
  
 return 0;  
}

#### 5. 链表

* 顺序存储（数组）在数据操作时的缺陷。
  + 扩容不方便
  + 数据的插入和删除操作代价过高
* 链式存储结构解决了顺序存储结构上存在的问题。
* 下面代码帮助大家理解链表结构。

struct node   
{  
 int data; // 数据域  
 struct node \* next; // 指针域  
};  
  
int main(void)  
{  
 struct node \* head = NULL, \* temp = NULL;  
 struct node n1, n2, n3;  
 n1.data = 11; n1.next = &n2;  
 n2.data = 22; n2.next = &n3;  
 n3.data = 33; n3.next = NULL;  
 head = &n1;  
 for (temp=head; temp!=NULL; temp=temp->next) {  
 printf("%d\n", temp->data);  
 }  
  
 return 0;  
}

* 链表的基本操作

#include <stdio.h>  
#include <malloc.h>  
  
typedef struct node  
{  
 int data;  
 struct node \* next;  
} Node, \* List;  
  
// typedef struct node Node;  
// typedef struct node \* List;  
  
// 在链表末尾添加节点  
// List \* 会因为要在当前函数修改main函数中的list的值  
// List \* pList <==> struct node \*\* pList  
void pushBackList(List \* pList, int data)  
{  
 Node \* temp = \*pList;  
 Node \* newNode = (Node \*) malloc(sizeof(Node));  
 newNode->next = NULL;  
 newNode->data = data;  
  
 if (temp == NULL) {  
 // 空链表  
 \*pList = newNode;  
 }  
 else {  
 // 非空链表，有有效节点  
 // 遍历链表，找到最后一个节点（尾节点的next等于NULL）  
 while (temp->next != NULL)  
 temp = temp->next;  
 temp->next = newNode;  
 }  
}  
  
  
// 打印链表所有节点  
void printList(List list)   
{  
 Node \* temp = list;  
 for (; temp!=NULL; temp=temp->next)  
 printf("%d\n", temp->data);  
 puts("===================");  
}  
  
// 求链表节点个数  
int sizeList(List list)   
{  
 int size = 0;  
 Node \* temp = list;  
 for (; temp!=NULL; temp=temp->next)  
 size++;  
 return size;  
}  
  
// 释放链表  
void freeList(List \* pList)   
{  
 Node \* temp = \*pList;  
 Node \* del = NULL;  
 while (temp != NULL) {  
 del = temp;  
 temp = temp->next;  
 free(del);  
 }  
}  
  
// 根据指定数据删除匹配节点  
void deleteList(List \* pList, int data)   
{  
 Node \* temp = \*pList;  
 Node \* del = NULL;  
 while (temp->next != NULL) {  
 if (temp->next->data != data) {  
 temp = temp->next;  
 continue;  
 }  
 // temp->next->data == data  
 del = temp->next;  
 temp->next = del->next;  
 free(del);  
 }  
  
 // 判断首节点是否要删除  
  **temp = \*pList;**  
 if (temp->data == data) {  
 \*pList = temp->next;  
 free(temp);  
 }  
}  
  
// 在指定位置插入节点 return 0 表示插入失败 1 表示成功  
int insertList(List \* pList, int index, int data)   
{  
 Node \* temp = \*pList;  
 Node \* newNode = NULL;  
 int size = sizeList(temp);  
 int i;  
   
 // 判断插入位置是否合法   
 if (index < 0 || index > size)  
 return 0;  
  
 // 创建新节点  
 newNode = (Node \*) malloc (sizeof(Node));  
 newNode->data = data;  
 newNode->next = NULL;  
  
 // 如果在首节点之前插入节点  
 if (index == 0) {  
 **newNode->next = temp;  
 \*pList = newNode;** return 1;  
 }  
  
 // 在非首节点之前插入节点，寻找插入位置的前一个节点  
 for (i=1; i<index; i++)  
 temp = temp->next;  
  
 newNode->next = temp->next;  
 temp->next = newNode;  
 return 1;  
}  
  
int main(void)  
{  
 // 创建一个空链表  
 List list = NULL; // struct node \* list = NULL;  
 pushBackList(&list, 33);  
 pushBackList(&list, 11);  
 pushBackList(&list, 22);  
 pushBackList(&list, 33);  
 pushBackList(&list, 44);  
 pushBackList(&list, 55);  
 pushBackList(&list, 33);  
 pushBackList(&list, 66);  
 pushBackList(&list, 33);  
 printf("size = %d\n", sizeList(list));  
 printList(list);  
  
 deleteList(&list, 33);  
 printf("size = %d\n", sizeList(list));  
 printList(list);  
  
 insertList(&list, 5, 100);  
 insertList(&list, 2, 200);  
 insertList(&list, 0, 300);  
 printf("size = %d\n", sizeList(list));  
 printList(list);  
  
 freeList(&list);  
  
 return 0;  
}