原 C语言 | 10 结构体与共同体

2019年03月18日 16:00:03 岁月静好，负重前行 阅读数：19 更多

10.1 用typedef 说明一个新类型

typedef为C语言的关键字，作用是为一种数据类型(基本类型或自定义数据类型)定义一个新名字，不能创建新类型。

1. 与#define不同，typedef仅限于数据类型，而不是能是表达式或具体的值
2. #define发生在预处理，typedef发生在编译阶段

```
#include <stdio.h>

typedef int INT;
typedef char BYTE;
typedef BYTE T_BYTE;
typedef unsigned char UBYTE;

typedef struct type
{
UBYTE a;
INT b;
T_BYTE c;
}TYPE, *PTYPE;

int main()
{
TYPE t;
t.a = 254;
t.b = 10;
t.c = 'c';

PTYPE p = &t;
printf("%u, %d, %c\n", p->a, p->b, p->c);

return 0;
}
```

10.2 结构体类型数据的定义和成员的引用

10.2.1 概述

数组：描述一组具有相同类型数据的有序集合，用于处理大量相同类型的数据运算。

有时我们需要将不同类型的数据组合成一个有机的整体，如：一个学生有学号/姓名/性别/年龄/地址等属性。显然单独定义以上变量比较繁琐，数据不

C语言中给出了另一种构造数据类型——结构体。

10.2.2 结构体变量的定义和初始化

定义结构体变量的方式：

1. 先声明结构体类型再定义变量名
2. 在声明类型的同时定义变量
3. 直接定义结构体类型变量（无类型名）

结构体类型和结构体变量关系：

1. 结构体类型：指定了一个结构体类型，它相当于一个模型，但其中并无具体数据，系统对之也不分配实际内存单元。
2. 结构体变量：系统根据结构体类型（内部成员状况）为之分配空间。

```
//结构体类型的定义
struct stu
{
char name[50];
int age;
};

//先定义类型，再定义变量（常用）
struct stu s1 = { "mike", 18 };


//定义类型同时定义变量
struct stu2
{
char name[50];
int age;
}s2 = { "lily", 22 };

struct
```

```
{
char name[50];
int age;
}s3 = { "yuri", 25 };
```

10.2.3 结构体成员的使用

```
#include<stdio.h>
#include<string.h>

//结构体类型的定义
struct stu
{
char name[50];
int age;
};

int main()
{
struct stu s1;

//如果是普通变量，通过点运算符操作结构体成员
strcpy(s1.name, "abc");
s1.age = 18;
printf("s1.name = %s, s1.age = %d\n", s1.name, s1.age);

//如果是指针变量，通过->操作结构体成员
strcpy((&s1)->name, "test");
(&s1)->age = 22;
printf("(&s1)->name = %s, (&s1)->age = %d\n", (&s1)->name, (&s1)->age);

return 0;
}
```

10.2.4 结构体数组

```
#include <stdio.h>

//统计学生成绩
```

```
struct stu
{
int num;
char name[20];
char sex;
float score;
};

int main()
{
//定义一个含有5个元素的结构体数组并将其初始化
struct stu boy[5] = {
{ 101, "Li ping", 'M', 45 },
{ 102, "Zhang ping", 'M', 62.5 },
{ 103, "He fang", 'F', 92.5 },
{ 104, "Cheng ling", 'F', 87 },
{ 105, "Wang ming", 'M', 58 }};

int i = 0;
int c = 0;
float ave, s = 0;
for (i = 0; i < 5; i++)
{
s += boy[i].score; //计算总分
if (boy[i].score < 60)
{
c += 1; //统计不及格人的分数
}
}

printf("s=%f\n", s);//打印总分数
ave = s / 5; //计算平均分数
printf("average=%f\ncount=%d\n\n", ave, c); //打印平均分与不及格人数


for (i = 0; i < 5; i++)
{
```

```
printf(" name=%s,  score=%f\n", boy[i].name, boy[i].score);
        // printf(" name=%s,  score=%f\n", (boy+i)->name, (boy+i)->score);

}

return 0;
}
```

10.2.5 结构体套结构体

```
#include <stdio.h>

struct person
{
char name[20];
char sex;
};

struct stu
{
int id;
struct person info;
};

int main()
{
struct stu s[2] = { 1, "lily", 'F', 2, "yuri", 'M' };

int i = 0;
for (i = 0; i < 2; i++)
{
printf("id = %d\tinfo.name=%s\tinfo.sex=%c\n", s[i].id, s[i].info.name, s[i].info.sex);
}

return 0;
}
```

10.2.6 结构体赋值

```
#include<stdio.h>
#include<string.h>

//结构体类型的定义
struct stu
{
char name[50];
int age;
};

int main()
{
struct stu s1;

//如果是普通变量，通过点运算符操作结构体成员
strcpy(s1.name, "abc");
s1.age = 18;
printf("s1.name = %s, s1.age = %d\n", s1.name, s1.age);

//相同类型的两个结构体变量，可以相互赋值
//把s1成员变量的值拷贝给s2成员变量的内存
//s1和s2只是成员变量的值一样而已，它们还是没有关系的两个变量
struct stu s2 = s1;
//memcpy(&s2, &s1, sizeof(s1));
printf("s2.name = %s, s2.age = %d\n", s2.name, s2.age);

return 0;
}
```

10.3 共用体(联合体)

1. 联合union是一个能在同一个存储空间存储不同类型数据的类型；
2. 联合体所占的内存长度等于其最长成员的长度倍数，也有叫做共用体；
3. 同一内存段可以用来存放几种不同类型的成员，但每一瞬时只有一种起作用；
4. 共用体变量中起作用的成员是最后一次存放的成员，在存入一个新的成员后原有的成员的值会被覆盖；
5. 共用体变量的地址和它的各成员的地址都是同一地址。

```
#include <stdio.h>

//共用体也叫联合体
```

```
union Test
{
unsigned char a;
unsigned int b;
unsigned short c;
};

int main()
{
//定义共用体变量
union Test tmp;

//1、所有成员的首地址是一样的
printf("%p, %p, %p\n", &(tmp.a), &(tmp.b), &(tmp.c));

//2、共用体大小为最大成员类型的大小
printf("%lu\n", sizeof(union Test));

//3、一个成员赋值，会影响另外的成员
//左边是高位，右边是低位
//低位放低地址，高位放高地址
tmp.b = 0x44332211;

printf("%x\n", tmp.a); //11
printf("%x\n", tmp.c); //2211

tmp.a = 0x00;
printf("short: %x\n", tmp.c); //2200
printf("int: %x\n", tmp.b); //44332200

return 0;
}
```

10.4 通过结构体构成链表,单向链表的建立,结点数据的输出、删除与插入

10.4.1 什么是链表

1. 链表是一种常用的数据结构，它通过指针将一些列数据结点，连接成一个数据链。相对于数组，链表具有更好的动态性（非顺序存储）链式存储.
2. 数据域用来存储数据，指针域用于建立与下一个结点的联系。
3. 建立链表时无需预先知道数据总量的，可以随机的分配空间，可以高效的在链表中的任意位置实时插入或删除数据。
4. 链表的开销，主要是访问顺序性和组织链的空间损失。

数组和链表的区别：

```
数组：一次性分配一块连续的存储区域。
优点：随机访问元素效率高
缺点：1) 需要分配一块连续的存储区域（很大区域，有可能分配失败）
      2) 删除和插入某个元素效率低
链表：无需一次性分配一块连续的存储区域，只需分配n块节点存储区域，通过指针建立关系。
优点：1) 不需要一块连续的存储区域
  2) 删除和插入某个元素效率高
缺点：随机访问元素效率低
```

10.4.2 创建链表

使用结构体定义节点类型：

```
typedef struct _LINKNODE
{
int id; //数据域
struct _LINKNODE* next; //指针域
}link_node;
```

编写函数：link_node* init_linklist()

建立带有头结点的单向链表，循环创建结点，结点数据域中的数值从键盘输入，以 -1 作为输入结束标志，链表的头结点地址由函数值返回.

```
typedef struct _LINKNODE{
int data;
struct _LINKNODE* next;
}link_node;

link_node* init_linklist(){

//创建头结点指针
link_node* head = NULL;
//给头结点分配内存
head = (link_node*)malloc(sizeof(link_node));
if (head == NULL){
return NULL;
}
head->data = -1;
head->next = NULL;

//保存当前节点
link_node* p_current = head;
int data = -1;
//循环向链表中插入节点
while (1){

printf("please input data:\n");
scanf("%d",&data);
```

```
//如果输入-1，则退出循环
if (data == -1){
break;
}

//给新节点分配内存
link_node* newnode = (link_node*)malloc(sizeof(link_node));
if (newnode == NULL){
break;
}

//给节点赋值
newnode->data = data;
newnode->next = NULL;

//新节点入链表，也就是将节点插入到最后一个节点的下一个位置
p_current->next = newnode;
//更新辅助指针p_current
p_current = newnode;
}

return head;
}
```

10.4.3 遍历链表

编写函数：void foreach_linklist(link_node* head)

顺序输出单向链表各项结点数据域中的内容：

```
//遍历链表
void foreach_linklist(link_node* head){
if (head == NULL){
return;
}

//赋值指针变量
link_node* p_current = head->next;
while (p_current != NULL){
printf("%d ",p_current->data);
p_current = p_current->next;
}
printf("\n");
}
```

10.4.4 插入节点

编写函数: void insert_linklist(link_node* head,int val,int data).

在指定值后面插入数据data,如果值val不存在，则在尾部插入。

```
//在值val前插入节点
void insert_linklist(link_node* head, int val, int data){

if (head == NULL){
return;
}

//两个辅助指针
link_node* p_prev = head;
link_node* p_current = p_prev->next;
while (p_current != NULL){
if (p_current->data == val){
break;
}
p_prev = p_current;
p_current = p_prev->next;
}
```

```
//如果p_current为NULL，说明不存在值为val的节点
if (p_current == NULL){
printf("不存在值为%d的节点!\n",val);
return;
}

//创建新的节点
link_node* newnode = (link_node*)malloc(sizeof(link_node));
newnode->data = data;
newnode->next = NULL;

//新节点入链表
newnode->next = p_current;
p_prev->next = newnode;
}
```

10.4.5 删除节点

编写函数: void remove_linklist(link_node* head,int val)

删除第一个值为val的结点.

```
//删除值为val的节点
void remove_linklist(link_node* head,int val){
if (head == NULL){
return;
}

//辅助指针
link_node* p_prev = head;
link_node* p_current = p_prev->next;

//查找值为val的节点
while (p_current != NULL){
if (p_current->data == val){
break;
}
p_prev = p_current;
p_current = p_prev->next;
}
//如果p_current为NULL，表示没有找到
if (p_current == NULL){
return;
}

//删除当前节点： 重新建立待删除节点(p_current)的前驱后继节点关系
p_prev->next = p_current->next;
//释放待删除节点的内存
free(p_current);
}
```

10.4.6 销毁链表

编写函数: void destroy_linklist(link_node* head)

销毁链表，释放所有节点的空间.

```
//销毁链表
void destroy_linklist(link_node* head){
if (head == NULL){
return;
}
//赋值指针
link_node* p_current = head;
while (p_current != NULL){
//缓存当前节点下一个节点
link_node* p_next = p_current->next;
```

```
free(p_current);
p_current = p_next;
}
}
```

想对作者说点什么