文档说明与参考来源

一、文档说明

本文档分为两个部分的内容：

① 编程基础

② 结构体合集

这两个部分的内容参考来源与说明如下：

① 编程基础：参考于 “零壹计算机考研 B 站免费公开课”、“C 语言中文网”

- 编程基础部分的内容剔除了非必要的内容，并且在文档中标注出了 408 考研常考的形式。如果有同学在看这份笔记的时候有些许的困难，你可以继续参看零壹计算机考研 B 站视频讲解课与 C 语言中文网的内容，在那里讲解得会更加的详细。（链接如下）
- 《3 小时速通 C 语言 - 数据结构前置课 - 计算机考研 408 必学课程》：

  https://www.bilibili.com/video/BV1Hc411S7JZ/?spm_id_from=333.1007.top_right_bar_window_custom_collection.content.click&vd_source=694aa77ce59cf66e6a79630f28cf9a95
- 《C 语言入门》：https://c.biancheng.net/c/

有关于零壹计算机考研更多的信息和内容，可以关注：

- B 站：零壹计算机考研
- 知乎：遇见
- Q 群：590091089

② 结构体合集：参考于 “蓝蓝知识星球”

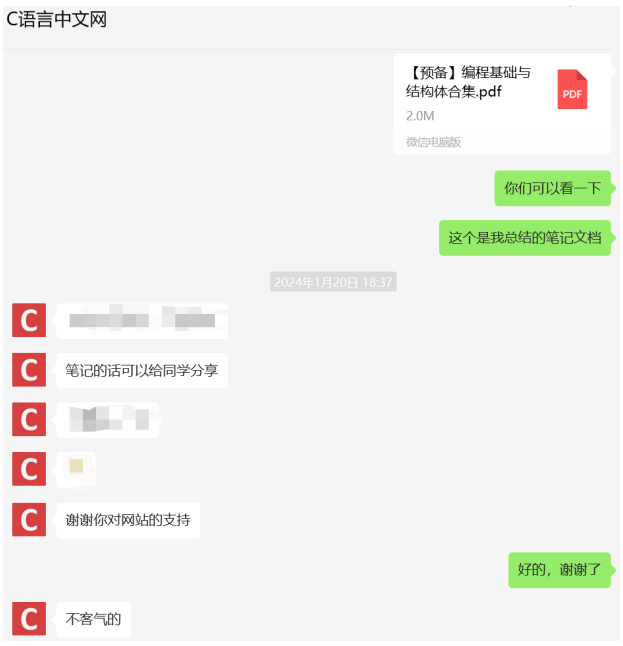
- 这里的内容是关于各类数据结构的定义。这也是是最基础的一步，希望大家可以通过这份文档，掌握各类数据结构的定义。

有关于蓝蓝知识星球的更多的信息和内容，可以关注：

- 这是蓝蓝4000➕的计算机考研圈子（这个星球圈子不错的，里面有用的信息很多，可以进去看看）

二、免责声明

我方已经获得了零壹计算机考研、C 语言中文网、蓝蓝知识星球的内容分享许可。部分聊天截图如下：



对于本文档，我方在此声明，对于以下事项，不承担任何法律责任：

a. 因不可抗力因素（如自然灾害、战争、官方行为活动等）导致的任何损失；

b. 因文档使用者自身过失或错误操作引起的任何损失；

c. 因第三方侵权行为导致的任何损失。

三、忠告

拿着这份文档学习的同学，我故意没有加上页眉页脚和水印，其目的就是为了你们在使用的过程中打印方便，好好利用，好好学习！不要动歪心思！

这里我最后忠告一下大家，不要拿这份笔记去卖！决不允许任何人拿这份笔记在淘宝、咸鱼、抖音、小红书或其他电商平台售卖，也不允许通过其他私底下的任何的交易方式进行买卖。总结和制作都非常辛苦，这都是免费分享的，如果你看到有人在售卖这份笔记，请举报他；如果你选择拿这份笔记去卖，机构的法务会出手，到时候你作为一个大学生，进去蹲几天，留下an底再出来，你刚刚好不容易活了 20 年，剩下的人生就毁了！到时候找你上门了，别怪我们没提前告过你。

编程基础

这个文档旨在助你学习考研 C 语言知识，本文档剔除了非必要的内容，并专注于经常出现在考试中的关键主题。

注意：在数据结构的算法题中，不涉及输入与输出。

第一章　变量与输入输出

1.1　输入输出简介

1. 何为输入输出?

- 以计算器计算为例。我们用键盘或手动输入 $1+1$，那么计算机就应该要对我们的输入进行处理。通过计算之后，得到我们想要的正确的结果 2，然后输出到屏幕上。

输入与输出

2. 输出

- 先来看下面 C 语言代码输出的例子：

```
printf("%d\n", n);

printf("格式控制字符串\n", 输出列表);
```

解释几个点：

① 这里的 `printf` 就是 C 语言的输出函数，他的作用是输出中 `" "` 的东西。

② `\n` 会被自动识别为换行符，这一整行构成了一条输出语句。`;` 是语句结束的标志。

- 再看一个更丰富的输出例子：

```
int main() {
    printf("hello world!\n");
    printf("%d\n", 1);
    printf("%f\n", 3.1415);
    printf("%d %d\n", 1, 8);//我们也可以把多个数据一起进行输出
    printf("衬衫的价格为 %d 镑 %d 便士", 9, 15);//加上对应的提示，进行输出
    return 0;
}
```

解释几个点：

① 这里的 `"%d"`，`"%f"` 是格式转换符，用于指定输入和输出的**值的格式。**

② `printf` 可以输出空格。

1.2　变量

- 定义：程序运行过程中可以改变的量，也是程序运行时临时存储数据的地方。
- C 语言中，在内存中找一块区域：

```
int a;
```

解释几个点：

① `int`是 Integer 的简写，意思是整数。`a` 是我们给这块区域起的名字；当然也可以叫其他名字，例如 abc、mn123 等。

这个语句的意思是：在内存中找一块区域，命名为 `a`，用它来存放整数。

不过 `int a;` 仅仅是在内存中找了一块可以保存整数的区域，那么如何将 123、100、999 这样的数字放进去呢?

- C 语言中，这样向内存中存放整数：

```
a = 123;
```

解释几个点：

① `=` 在数学中叫“等于号”，例如 1+2=3，但在 C 语言中，这个过程叫做**赋值。赋值是指把数据放到内存的过程。**

- 总结：`int a;` 创造了一个变量 `a`，我们把这个过程叫做**变量定义。** `a=123;` 把 123 交给了变量 `a`，我们把这个过程叫做**给变量赋值**；又因为是第一次赋值，也称**变量的初始化**，或者**赋初值。**

1.3 用变量进行输入与输出

1. 输出

- 来看一个用变量进行输入的例子：

```c
int score;
score = 95;
printf("My score is %d!\n", score);
```

解释几个点：

① 用 `int` 类型进行定义了 `score` 这个变量。

② 给 `score` 赋值为了 `95` 。

③ 之所以用与 `"My score is %d!"` 相同的名字“score”，是为了表示：**变量名，即实际含义** 。

2. 输入

> 终于到了输入的环节！程序是人机交互的媒介，有输出必然也有输入。

- 将上面的 C 语言代码，改为输入代码的例子：

```c
int main() {
    int score;
    scanf("%d", &score);
    printf("My score is %d!\n", score);
    return 0;
}
```

解释几个点：

① 用 `scanf` 来读取键盘输入的值，然后将**读取的值**存储在变量 `score` 中。

- 同理看下面 C 语言代码输出的例子：

```c
scanf("%d", &n);

scanf("格式控制字符串", &输入列表);
```

解释几个点：

① `scanf` 是 `scan format` 的缩写，意思是格式化扫描，也就是从键盘获得用户输入，和 `printf` 的功能正好相反。

② 在输入完毕后，需要按下键盘上的回车键。这是因为，输入，是以**行**为单位进行的，**行**的结束标志就是回车键。

用户每次按下回车键，程序就会认为完成了一次输入操作，`scanf()` 就开始读取用户输入的内容，并根据格式控制字符串从中提取有效数据，只要用户输入的内容和格式控制字符串匹配，就能够正确提取。

本质上讲，用户输入的内容都是字符串，`scanf()` 完成的是从字符串中提取有效数据的过程。

③ `scanf` 的变量前要带一个 `&` 符号。`&` 称为取地址符，也就是获取变量在内存中的地址。

第二章　C 语言 - 数据类型

数据是放在内存中的，变量是给这块内存起的名字，有了变量就可以找到并使用这份数据。但问题是，该如何使用呢？

我们知道，诸如数字、文字、符号、图形、音频、视频等数据都是以二进制形式存储在内存中的，它们并没有本质上的区别，那么，00010000 该理解为数字 16 呢，还是图像中某个像素的颜色呢，还是要发出某个声音呢？如果没有特别指明，我们并不知道。

也就是说，内存中的数据有多种解释方式，使用之前必须要确定；上面的 `int a;` 就表明，这份数据是整数，不能理解为像素、声音等。`int` 有一个专业的称呼，叫做**数据类型**（Data Type）。

顾名思义，数据类型用来说明数据的类型，确定了数据的解释方式，让计算机和程序员不会产生歧义。

数据类型可以分为两大类：基本数据类型和复合数据类型。

2.1　基本数据类型

> 很多地方也叫 “原子类型”，是不可再分的类型。也叫 “基类型”。

- 定义：像 int、float、char 等是由 C 语言本身提供的数据类型，不能再进行拆分，我们称之为基本数据类型。

数据类型		长度（32位）	表示范围（32位）	格式控制
字符类型	char	1	-2^{7}~2^{7}-1	%c
整数类型 短整型	short	2	-2^{15}~2^{15}-1	%d
整数类型 整型	int	4	-2^{31}~2^{31}-1	%d
整数类型 长整型	long	4	-2^{63}~2^{63}-1	%ld
浮点数类型 单精度浮点型	float	4	精度可达10^{-38}	%f
浮点数类型 双精度浮点型	double	8	精度可达10^{-308}	%lf
布尔类型	bool	1	1 or 0	%d
无类型	void			

- 数据长度：所谓数据长度（Length），是指数据占用多少个**字节**。占用的字节越多，能存储的数据就越多，对于数字来说，值就会更大，反之能存储的数据就有限。

  所以，在定义变量时还要指明数据的长度。而这恰恰是数据类型的另外一个作用。数据类型除了指明数据的解释方式，还指明了数据的长度。因为在 C 语言中，每一种数据类型所占用的字节数都是固定的，知道了数据类型，也就知道了数据的长度。

- 注意：在 C 语言中，没有直接支持布尔（boolean）类型。通常情况下，我们使用 `int` 或 `char` 来表示真值（true）和假值（false）。在 C++ 中有布尔类型。

  ① 使用 `int` 类型变量作为布尔类型的标志位：

```c
#include <stdio.h>

int main() {
    int flag; // 定义一个int类型的变量flag

    printf("请输入一个布尔值(0/1): ");
    scanf("%d", &flag); // 从键盘获取一个整数并存储到flag变量中

    if (flag == 1) {
        printf("这是真\n");
    } else {
        printf("这是假\n");
    }

    return 0;
}
```

② 使用 `char` 类型变量作为布尔类型的标志位：

```c
#include <stdio.h>

int main() {
    char flag; // 定义一个char类型的变量flag

    printf("请输入一个布尔值('Y'/'N'): ");
    scanf(" %c", &flag); // 从键盘获取一个字符并存储到flag变量中

    if (flag == 'Y') {
        printf("这是真\n");
    } else {
        printf("这是假\n");
    }
```

```
14
15      return 0;
16  }
```

2.2 复合数据类型

> 复合数据类型有结构体、数组、共用体，枚举类型、类。考研初试只让你会结构体和数组。

1. 基本方式

- 定义：由若干个基本数据类型或复合数据类型组成，是可以再分解的。
- 结构体定义：一批不同类型的数据组合而成的结构型数据。组成结构型数据的每个数据称为结构型数据的 **“成员”**。

```c
//结构体定义：
struct 结构体名 {
    结构体成员表
};

//结构体举例：
struct Student {
    char name;
    int id;
    int score;
};

//声明一个结构体变量
struct Student stu1;

//访问结构体变量并初始化赋值
stu1.name = 'A';
stu1.id = 1001;

//利用输入给结构体中的元素赋值
scanf("%d", &stu.score);

//输出结构体中的元素
printf("%c %d %d", stu.name, stu.id, stu.score);
```

解释几个点：

① 结构体可以包含多个基本类型的数据，也可以包含其他的结构体。

② 结构体是一种数据类型！

③ `struct` 表明是结构体类型，`Student` 是结构体名字，`stu1` 才是结构体变量。

④ 使用 `struct Student` 来定义变量的时候，这两个词一个都不能少！

⑤ 理论上讲结构体的各个成员在内存中是连续存储的，和数组非常类似。

name	id	score

⑥ 结构体和数组类似，数组使用下标 `[ ]` 获取单个元素，结构体使用点号 `.` 获取单个成员。

⑦ 需要注意的是，**结构体**是一种自定义的数据类型，是创建变量的模板，**不占用内存空间**；**结构体变量**才包含了实实在在的数据，**需要内存空间来存储。**

2. 考研常用方式

- 你也可以在定义结构体的同时定义结构体变量：

```
struct Student {
    char name;
    int id;
    int score;
} stu1, stu2;
```

将变量放在结构体定义的最后即可。

- 如果只需要 stu1、stu2 两个变量，后面不需要再使用结构体名定义其它变量，那么在定义时也可以不给出结构体名：

```
struct {
    char name;
    int id;
    int score;
} stu1, stu2;
```

这样做书写简单，但是因为没有结构体名，后面就没法用该结构体定义新的变量。

- 接下来看一个自定义类型：（其实就是给原来的类型起一个别名，仅此而已）

```
typedef 原类型名 新类型名;

typedef int id;

id id;
id = 32;
scanf("%d", &id);
printf("%d", id);
```

在这里，我们用 `typedef` 给 `int` 起了一个别名 `id`。在这之后，你若想要声明一个 `int` 类型的整数变量，就可以使用 `id` 进行声明。

- **自定义结构体类型：**（考研用这个）

```
typedef struct {
    char name;
    int id;
    int score;
} Student;

Student stu;
```

利用 `typedef` 给结构体起别名为 `Student`，那么，在使用 `Student` 来声明结构体变量时，可以省略前面的 `struct`，同时在表示学生这个类型时，也更加简洁、形象了。

- 小拓展：利用 C++ 特性（知道有个么回事就行了，考研用上面的“自定义结构体类型”）

```
struct Student {
    char name;
    int id;
    int score;
};

Student stu;
```

若使用 C++ 特性，在这里，可以直接使用 `Student` 来声明结构体变量，不需要加上 `struct` ，也不用 `typedef` 起别名。（其实就是因为当初人们也嫌 C 语言那么来回自定义，搞的麻烦，就在 C++ 中优化了这一情况。你结构体不是本来就可以拥有一个名字么，那直接用你结构体的名字不就行了。）

第三章　C 语言 - 运算符

C 语言的规则和数学的规则是一样的。

3.1　赋值运算符与算数运算符

1. 赋值运算符

赋值运算符 “=”，常与变量和表达式搭配使用。

- 变量 = 表达式

```c
int a = 1;
int b = a;
int c = b+2;
```

2. 算数运算符

- 四则运算的运算符：`+`、`-`、`*`、`/`
- 取余：`%`（或称作 “模运算”，就是取余数）

```c
int divResult = 7/3;    //向下取整，在这里结果应为 2
int remainder = 7%3;    //进行取余操作，在这里结果应为 1
```

- 自增：`++`
- 自减：`--`

```c
a++;    //在执行之后再把变量的值自增+1

int main(){
    int a = 3;
    printf("a++ = %d\n", a++);
}

int main(){
    int a = 3;
    printf("a++ = %d\n", a);
    a = a + 1;
}

++a;    //在执行之前把变量的值自增+1

int main(){
    int a = 3;
    printf("++a = %d\n", ++a);
}

int main(){
    int a = 3;
    a = a + 1;
    printf("++a = %d\n", a);
}
```

- `+`、`-`、`*`、`/`、`=` 是双目运算符；
- `++`、`--` 是单目运算符；
- `? :` 是三目运算符

3.2　关系运算符

> 关系运算符在使用时，它的的两边都会有一个表达式，比如变量、数值、加减乘除运算等，关系运算符的作用就是判明这两个表达式的大小关系。注意，是判明大小关系，不是其他关系。

- C 语言提供了关系运算符：`>`、`>=`、`<`、`<=`、`==`、`!=` 关系运算符的执行结果为 bool 类型。
- C 语言中，判断表达式是否相等的符号为 `==`，判断表达式不等的符号为 `!=` 。

3.3 逻辑运算符

- C 语言提供了三种逻辑运算符：`&&`、`||`、`!`，分别表示“与”、“或”、“非”
- `&&` 的左右两边的表达式的结果都为真时，结果才为真，否则为假。

  `||` 的左右两边的表达式的结果只要有一个为真，结果就为真。

  `!` 后面的表达式结果为真，结果就为假。

看一个复杂的例子：

- ```
      从右往左算
  <---------------------

  4 > 3 || 5 == 4 + 1 && 0 > 1

      等价于

  true || true && false

  表达式结果为：真

  bool a = (4 > 3 || 5 == 4 + 1) && 0 > 1; //false
  bool b = 4 > 3 || (5 == 4 + 1 && 0 > 1); //true
  ```
  ```

第四章　选择结构与循环结构

4.1　程序的顺序执行结构

- 定义：程序从主函数 (main) 开始，从上到下逐行的进行，直到 `return 0` 结束。

> `return 0` 可以省略不写。若省略掉，程序默认执行主函数最后一行代码，然后结束。
>
> 为了保持良好的编程习惯，一定要加上 `return 0` 。

4.2　选择结构

1. 单分支 if 语句

- 单分支 if 语句可以不加 `{}` 。但是为了规范性，还是加上好。

```
if(条件) {
    条件为真时，需要实现的功能
}
```

2. 多分支 if 语句

- if 语句还可以与 else 结合，达到多分支的效果

```
if(条件) {
    条件为真时，需要实现的功能
} else {
    条件为假时，需要实现的功能
}
```

```
if(条件1) {
    条件1成立时，需要实现的功能
} else if(条件2) {
    条件2成立时，需要实现的功能
} else if(条件3) {
    条件3成立时，需要实现的功能
} else {
    所有条件都不成立时的操作
}
```

3. switch 语句

- 若 if 语句和 switch 语句都成立的话，优先使用 switch 语句。

```
switch(表达式) {
    case 值1:
        ...;    //表达式结果 = 值1时执行的代码
        break;
    case 值2:
        ...;    //表达式结果 = 值2时执行的代码
        break;
    default:
        ...;    //如果 switch 后跟的值与所有分支的值都不相等，则走 default 分支
        break;
}
```

看一个例子：

```
int main() {
    int number;
    scanf("%d", &number);
    switch(number) {
        case 1: // 如果 number == 1
            printf("Monday");
            break;
        case 2: // 如果 number == 2
            printf("Tuesday");
            break;
        case 3: // 如果 number == 3
            printf("Wednesday");
            break;
        case 4: // 如果 number == 4
            printf("Thursday");
            break;
        case 5: // 如果 number == 5
            printf("Friday");
            break;
        case 6: // 如果 number == 6
            printf("Saturday");
            break;
        case 7: // 如果 number == 7
            printf("Sunday");
            break;
        default:
            printf("invalid number"); // 不是 1 - 7 的数字时，给出无效输入的提示
            break;
    }
}
```

使用 switch 代码的逻辑会明显的清晰很多，也不会有大串的 if - else 的看起来累赘的重复代码，在分支数较多时，建议使用 switch。

注意：在 C 语言中，switch 和 case 后面，都只能跟**整形表达式。**在写分支时，不能漏了 break，否则，在满足某一个分支后，会连着后面的分支一起执行。

4.3 循环结构

所谓循环（Loop），就是重复地执行同一段代码，例如要计算 1+2+3+.....+99+100 的值，就要重复进行 99 次加法运算。

1. while 循环

- 例：编写程序，依次输出 1-10，使用 while 循环来实现。

i	1	2	…	10	11
执行	√	√	√	√	×

当 i 从 1 到 10 的时候，**满足进入循环的条件**，代码块中的代码会被执行，直到 i 变为 11 ，**不满足进入循环的条件**，循环体中的代码不会被执行。

```c
int main() {
    int i=1;
    while (i<=10) { //进入while的条件是i<=10
        printf("%d ", i); //打印i的值（写代码的时候可以先写循环体的代码，再去写循环条件）
        i=i+1; //给i的值+1
    }
}
```

2. for 循环

- 例：编写程序，依次输出 1-10，使用 for 循环来实现。

i	1	2	…	10	11
执行	√	√	√	√	×

当 i 从 1 到 10 的时候，**满足进入循环的条件**，代码块中的代码会被执行，直到 i 变为 11 ，**不满足进入循环的条件**，跳出循环。

```c
int main() {
    int i;
    for (i=1; i<=10; i++) { //这里i++的作用就是单纯的让i的值+1
        printf("%d ", i);
    }
}
```

3. 多重 for 循环

- **考研的第一种二重循环的用法**：直接打印 i 和 j 的值。（这也是最简单的二重循环的用法）

```c
int main() {
    int i; //用于外层循环的变量
    int j; //用于内层循环的变量
    int n; //循环的次数，需要赋值！

    for(i=1; i<=n; i++) { //第一个循环体，也叫外层循环
        for(j=1; j<=n; j++) { //第二个循环体，也叫内层循环
            printf("%d%d ", i, j); //这句话本来是在第二个循环体内部的，本来这一句话只执行n次，但是外面又套了一层循环，其目的是让第二个循环体再执行n次。那么这句话总共会执行n×n次。
        }
        printf("\n");
    }
}
```

i	1					
j	1	2	3	…	n	n+1
执行	√	√	√	√	√	×
i	2					
j	1	2	3	…	n	n+1
执行	√	√	√	√	√	×
…						
i	n					
j	1	2	3	…	n	n+1
执行	√	√	√	√	√	×

以上循环形式的输出结果为：

```
令 n=5;

结果为：
11 12 13 14 15
21 22 23 24 25
31 32 33 34 35
41 42 43 44 45
51 52 53 54 55
```

- **考研的第二种二重循环的用法**：逐次递增 j 的值。

```c
int main() {
    int i; //用于外层循环的变量
    int j; //用于内层循环的变量
    int n; //循环的次数，需要赋值！

    for(i=1; i<=n; i++) {
        for(j=1; j<=i; j++) { //只改了这里，把n改为了i。第二个for循环每次只能执行i次，由第一个循环体的循环变量决定其循环次数。
            printf("%d ", i); //算这句话的执行次数需要把每一行列出来，然后按照递增数列求和来算。
        }
        printf("\n");
    }
}
```

i	1					
j	1	2				
执行	√	√				
i	2					
j	1	2	3			
执行	√	√	√			
…						
i	n					
j	1	2	3	…	n	n+1
执行	√	√	√	√	√	×

for 循环可以与 while 循环转换（了解）

```c
int main() {
    int i; //用于外层循环的变量
    int j; //用于内层循环的变量
    int n; //循环的次数，需要赋值！

    for(i=1; i<=n; i++) {
        j=1;
        while(j<=i){  //while语句其实就是把for循环的三个语句拆开依次放在头、中间、尾部。
            printf("%d", i);
            j++;
        }
        printf("\n");
    }
}
```

以上两种循环形式的输出结果为：

```
令 n=5;

结果为：
1
22
333
4444
55555
```

4.4 continue 和 break

1. break

- 举个例子，尝试计算 1+2+3+...+10，当累加的值第一次大于 38 时，输出当前所加的值i和累加值 sum 并跳出循环：

```c
int main() {
    int sum = 0;
    int i;
    for (i=1; i<=10; i++) {
        sum = sum + i;
        if (sum>38) {
            printf("i = %d, sum = %d", i, sum);
            break;
        }
    }
}
```

2. continue

使用 continue 可以立刻结束本次循环，开始下一次循环。

- 例：把 1-10 按顺序输出，唯独不能输出 5 。

```c
int main() {
    int i = 1;
    while(i<=10) {
        if(i==5) { //意思就是当i=5的时候，这次的循环就不执行了。也就是题目中的不输出5。
            i++;
            continue;
        }
        printf("%d ", i++);
    }
}
```

第五章　数组

5.1　数组的定义与声明

- 定义：数组可以存储一批 **同类型数据。**它所包含的每一个数据叫做 **数组元素**（Element），所包含的数据的个数称为 **数组长度**（Length）。

```
int score[10]; //类型 数组名称[元素数量];
```

在这里，我们声明了一个数组，名字叫 `score`，数组 `score` 里面的元素是 int 类型，数组 `score` 里面可以存放 10 个整型变量。通过声明 `score` 数组，我们就轻松的声明了 10 个 int 类型的数据，可以存下 10 个同学的分数了。

5.2　数组的初始化与访问

1. 数组的初始化

- 如何初始化数组中的元素?

  可以直接把数组元素的初值依次放进一对大括号中，元素之间用逗号隔开。

```
int score[3] = {1, 2, 3};
```

- 若声明了大小为 5 的数组空间，却只初始化了一部分数组元素，那么剩余的数组元素就会被初始化为 0 。

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

数组元素	score[0]	score[1]	score[2]	score[3]	score[4]
元素值	1	2	3	0	0

2. 数组的访问

- 如何访问数组中的元素?

  使用变量名加上中括号，在中括号中填上数字下标访问对应的元素。（**下标从 0 开始!**）

```
int score[9];
```

数组元素	score[0]	score[1]	score[2]	score[3]	score[4]	score[5]	score[6]	score[7]	score[8]
元素下标	0	1	2	3	4	5	6	7	8

数组是一个整体，它的内存是连续的；也就是说，数组元素之间是相互挨着的，彼此之间没有一点点缝隙。

- 可以利用循环结构访问数组并赋值。

```
int main() {
    int score[10];
    int i;
    for(i=0; i<10; i++) {
        score[i]=i;
    }
}
```

5.3 数组实战

例：声明一个空间大小为 10 的数组，当数组下标为偶数时，数组元素赋值为 0，下标为奇数时，数组元素赋值为当前的下标值。

```c
int main() {
    int num[10];
    for(int i=0; i<10; i++) { //这个循环的作用是给num数组赋值，考试的时候这个就是关键代码，只需要写这个。
        if(i%2 == 0) {
            num[i] = 0;
        } else {
            num[i] = i;
        }
    }
    for(int i=0; i<10; i++) { //这个循环是输出num数组每个元素的值，考试的时候不需要写这个。
        printf("%d ", num[i]);
    }
}
```

5.4 二维数组

上节讲解的数组可以看作是一行连续的数据，只有一个下标，称为**一维数组**。在实际问题中有很多数据是二维的或多维的，因此 C 语言允许构造多维数组。多维数组元素有多个下标，以确定它在数组中的位置。本节只介绍二维数组，多维数组可由二维数组类推而得到。

- 我们可以将二维数组看做一个 Excel 表格，有行有列，length1 表示行数，length2 表示列数，要在二维数组中定位某个元素，必须同时指明行和列。

```c
类型 数组名称[length1][length2];

int a[3][4];
```

前面的中括号代表**一维数组的数量**，后面的中括号代表**一维数组的长度**。

	col 0	col 1	col 2	col 3
row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- 例：把二维数组看作一个矩阵，把矩阵每一行的元素都赋值为当前行数的值。(如下表所示)

	col 0	col 1	col 2
row 0	0	0	0
row 1	1	1	1
row 2	2	2	2

```c
int main() {
    int a[3][3];
    for (int i=0; i<3; i++) {
        for (int j=0; j<3; j++){
            a[i][j] = i;
        }
    }
}
```

- 我们也可以直接对二维数组中的**某一个元素**进行操作。

  例：给第一个一维数组的第一个元素赋值为 10 。

	col 0	col 1	col 2
row 0	10	0	0
row 1	1	1	1
row 2	2	2	2

```
score[0][0]=10;
```

- 我们也可以对二维数组中的**某一个元素**进行操作。(也就是对某一个一维数组进行操作)

  例：给第一个一维数组全部赋值为 10 。

	col 0	col 1	col 2
row 0	10	10	10
row 1	1	1	1
row 2	2	2	2

```
for (int i=0; i<3; i++) {
    a[0][i] = 10;
}
```

二维数组在概念上是二维的，但在内存中是连续存放的；换句话说，二维数组的各个元素是相互挨着的，彼此之间没有缝隙。那么，如何在线性内存中存放二维数组呢？有两种方式：

- 一种是按行排列， 即放完一行之后再放入第二行；
- 另一种是按列排列， 即放完一列之后再放入第二列。

在 C 语言中，二维数组是按行排列的。也就是先存放 `a[0]` 行，再存放 `a[1]` 行，最后存放 `a[2]` 行；每行中的 4 个元素也是依次存放。数组 a 为 int 类型，每个元素占用 4 个字节，整个数组共占用 4×(3×4)=48 个字节。

你可以这样认为，二维数组是由多个长度相同的一维数组构成的。

5.5 字符串与结构体数组

1. 字符串（字符数组）

- 定义：用来存放字符的数组称为**字符数组** 。

```
char a[10];  //一维字符数组
char b[5][10];  //二维字符数组
char c[20]={'c', ' ', 'p', 'r', 'o', 'g', 'r', 'a','m'};  // 给部分数组元素赋值
```

字符数组实际上是一系列字符的集合，也就是**字符串（String）**。在 C 语言中，没有专门的字符串变量，没有 string 类型，通常就用一个字符数组来存放一个字符串。

- C 语言规定，可以将字符串直接赋值给字符数组。

```
char str1[4] = {"a.b"};
char str2[8] = "a.bc";  //这种形式更加简洁，实际开发中常用
```

解释几个点：

① 数组第 0 个元素为 `'a'`，第 1 个元素为 `'.'`，第 2 个元素为 `'b'`，后面的元素以此类推。

② 长度为 4 的字符数组只能存储三个字符元素，这是因为字符数组的末尾固定为 `'\0'` 。

str1	a	.	b	\0

str2	a	.	b	c	\0			

③ 由 `" "` 包围的字符串会自动在末尾添加 `'\0'` 。例如，`"abc123"` 从表面看起来只包含了 6 个字符，其实不然，C 语言会在最后隐式地添加一个 `'\0'` ，这个过程是在后台默默地进行的，所以我们感受不到。

④ 当我们不确定字符串的长度时，我们可以先把字符数组的长度设大一些，长度范围至少要能包括我们想要声明的最长字符串。

- 为了方便，你也可以不指定数组长度，从而写作：

```
char str[] = {"a.b"};
char str[] = "a.bc";  //这种形式更加简洁，实际开发中常用
```

2. 字符串输出和字符串长度

- 字符串的输出使用 `%s`，不推荐直接对字符串输入。

```
int main() {
    char str[] = "a.bc";
    printf("%s", str);
}
```

- 所谓字符串长度，就是字符串包含了多少个字符（不包括最后的结束符 `'\0'`）。例如 `"abc"` 的长度是 3，而不是 4。

  在 C 语言中，我们使用 `string.h` 头文件中的 strlen() 函数来求字符串的长度，它的用法为：

```
length strlen(strname);
```

  strname 是字符串的名字，或者字符数组的名字；length 是使用 strlen() 后得到的字符串长度，是一个整数。

  例如：

```
#include <stdio.h>
#include <string.h>  //记得引入该头文件

int main(){
    char str[] = "a.bc";
    long len = strlen(str);
    printf("The length of the string is %ld.\n", len);

    return 0;
}

输出结果：The length of the string is 4.
```

3. 结构体数组

> 在实际应用中，C 语言结构体数组常被用来表示一个拥有相同数据结构的群体，比如一个班的学生、一个车间的职工等。

- 定义：数组中的每个元素都是一个结构体。

  定义方式一：

```
typedef struct {
    char name[20];
    int id;
    int score;
} Student[20];
```

  定义方式二：

```
struct Student {
    char name[20];
    int id;
    int score;
};

struct Student stu[20];
```

  结构体数组的每个**元素**都含有结构体全部的**成员**。

第六章　指针

所谓指针，也就是内存的地址；所谓指针变量，也就是保存了内存地址的变量。不过，人们往往不会区分两者的概念，而是混淆在一起使用，在必要的情况下，大家也要注意区分。

指针是 C 语言中广泛使用的一种数据类型，说指针是 C 语言的灵魂也不为过。利用指针变量可以表示出各种数据结构；方便的使用数组和字符串；能够处理计算机内存地址，编写出精练而高效的程序。正确理解和使用指针，是我们掌握 C 语言基本编程能力的一个标志。

在学习指针这个章节之前，我们需要先了解计算机内存地址的概念。

6.1　数据存储与地址

我们假设计算机内存有 32 个等大的内存单元，可以存放 32 个 int 类型数据。为了方便取用和标识，我们给这 32 个内存单元分别编号为 0 - 31，这些编号就是 **内存的地址。**

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

- 当我们声明一个 int 类型的变量 a 时，编译器就会在内存中取一块空闲的空间分配给变量 a 。

  在这里，我们假设 8 这个地址空间是空闲的，编译器把 8 这个地址空间分配给了变量 a 。

```
1 int a;
```

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

  为了方便查看，我们通常会把图中的地址平铺开：

	a															
地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
数据																	...

- 学习 scanf 时，我们提到，在输入表列中，要在元素前面加上取地址符号 `&`

```
1 scanf("%d", &a);
```

  在这里，就是让编译器去找到 `a` 的地址，**把输入的值存放进地址对应的数据里** 。其中，`&a` 代表的就是变量 `a` 的地址，是变量 `a` 在计算机内存中存储的位置。

一切都是地址

C 语言用变量来存储数据，用函数来定义一段可以重复使用的代码，它们最终都要放到内存中才能供 CPU 使用。

数据和代码都以二进制的形式存储在内存中，计算机无法从格式上区分某块内存到底存储的是数据还是代码。当程序被加载到内存后，操作系统会给不同的内存块指定不同的权限，**拥有读取和执行权限的内存块就是代码，而拥有读取和写入权限（也可能只有读取权限）的内存块就是数据。**

CPU 访问内存时需要的是地址，而不是变量名和函数名！变量名和函数名只是地址的一种助记符，当源文件被编译和链接成可执行程序后，它们都会被替换成地址。编译和链接过程的一项重要任务就是找到这些名称所对应的地址。

假设变量 a、b、c 在内存中的地址分别是 0X1000、0X2000、0X3000，那么加法运算 `c = a + b;` 将会被转换成类似下面的形式：

```
1 0X3000 = (0X1000) + (0X2000);
```

`( )` 表示取值操作，整个表达式的意思是，取出地址 0X1000 和 0X2000 上的值，将它们相加，把相加的结果赋值给地址为 0X3000 的内存

变量名和函数名为我们提供了方便，让我们在编写代码的过程中可以使用易于阅读和理解的英文字符串，不用直接面对二进制地址，那场景简直让人崩溃。

需要注意的是，虽然变量名、函数名、字符串名和数组名在本质上是一样的，它们都是地址的助记符，但在编写代码的过程中，我们认为变量名表示的是数据本身，而函数名、字符串名和数组名表示的是代码块或数据块的首地址。

6.2 指针基本概念与用法

6.2.1 指针基本概念

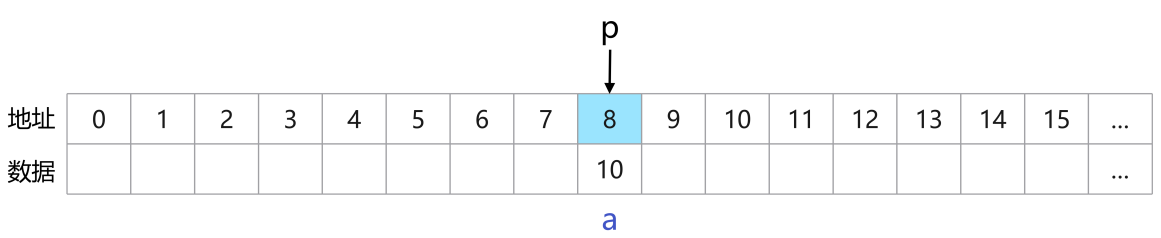
- 指针就是地址，地址就是指针。
- 如果一个变量存储了一份数据的指针，我们就称它为 **指针变量**。
- 在 C 语言中，允许用一个变量来存放指针，这种变量称为 **指针变量**。指针变量的值就是某份数据的地址，这样的一份数据可以是数组、字符串、函数，也可以是另外的一个普通变量或指针变量。

```
int a = 10;
int p = &a;
printf("&a = %d, p = %d, a = %d, *p = %d,", &a, p, a, *p);
```

解释几个点：

① 我们定义了一个变量 `a` ，定义了一个指针变量 `p` ，并让指针变量 `p` 指向了变量 `a` 的地址，那么此时 `p` 和 `&a` 都表示内存中的地址 8 。

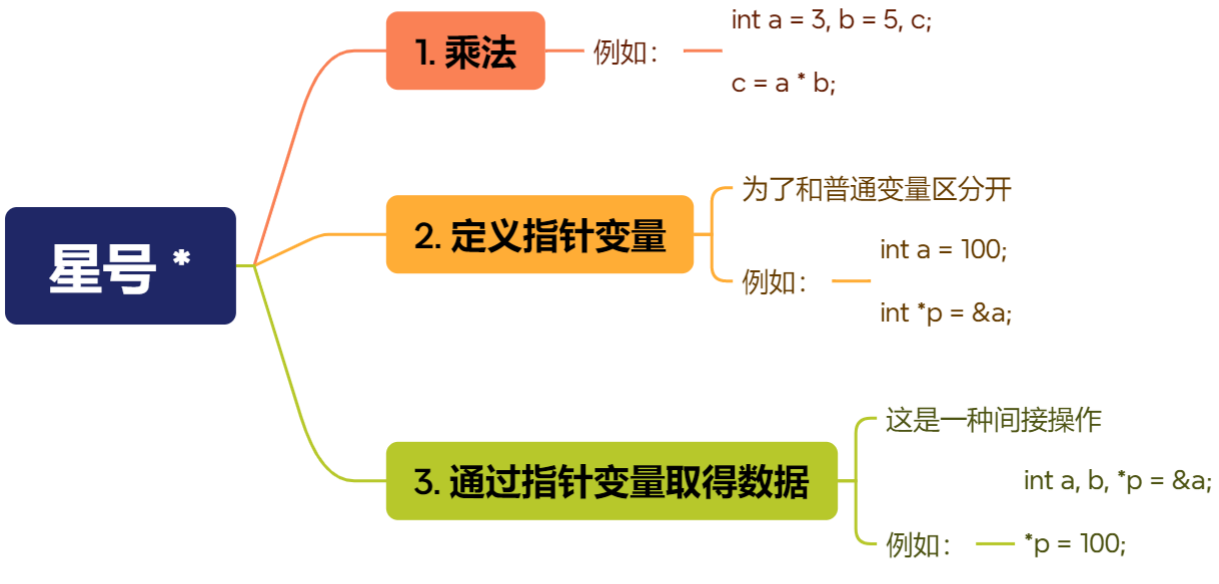
② 在指针变量 `p` 前加上 `*`，就可以得到地址对应的数据，称为**解引用**。在这里，我们就得到了指针 `p` 所指向的地址的内容。



③ 定义指针变量时必须带 `*` ，给指针变量赋值时不能带 `*` 。

6.2.2 对星号 `*` 的总结

在我们目前所学到的语法中，星号 `*` 主要有三种用途。



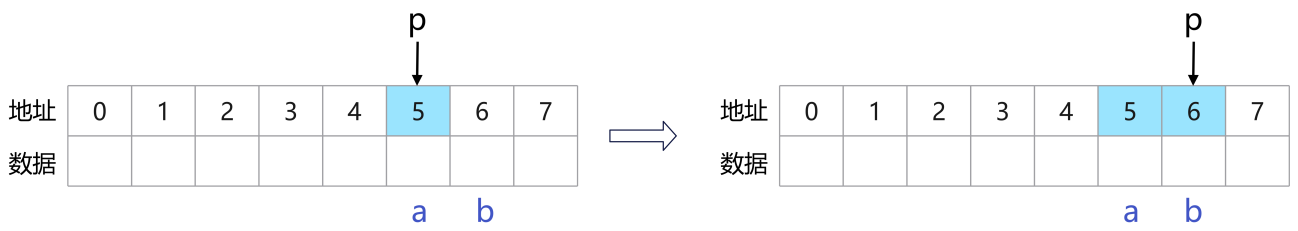
6.2.3 指针用法

1. 基本操作

- 指针变量可以进行加减运算，此时这些运算符的操作对象都是 **内存地址。**

  假设变量 a 的地址是 5，变量 b 的地址是 6，用指针变量 p 指向 a 。

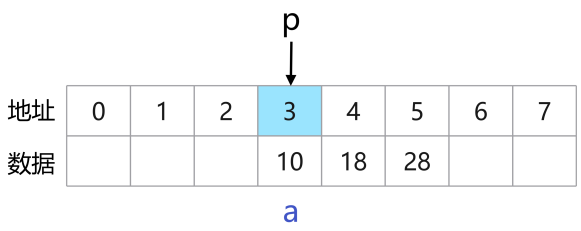
```
int *p = &a;
p++;
```



2. 五个辨析

- 看下列代码：

```
int a = 10;
int *p = &a;
```



基于上述代码，执行下列表达式运算或语句后，`a`、`*p` 和 `p` 的值是多少?

① `printf(“%d”, *p++);`

```
等效于
    printf(“%d”, *p);
    p++;

a:10
*p:10
p:4
```

② `*p += 1;`

```
等效于
    *p = *p + 1;

a:11
*p:11
p:3
```

③ `*p = a * 2;`

```
等效于
    a = a * 2;

a:20
*p:20
p:3
```

④ if 分支判断

```
if(*p > 10) {
    p++;
} else {
    *p /= 2;
}

a:5
*p:5
p:3
```

⑤ `p += 2;`

```
a:10
*p:28
p:5
```

6.3 指针和数组

0. 预备备

先看一段代码：

```c
int main() {
    int arr[41] = {0, 2, 4, 6};
    int *p = arr; // arr 就是 &arr[0]

    printf("%d %d %d\n", p, arr, &arr[0]); // 三个都是同一个地址

    printf("%d %d %d\n", p, p + 1, p + 2);
    printf("%d %d %d\n", &arr[0], &arr[1], &arr[2]); //两句输出效果一样。这个取的是“地址”

    printf("%d %d %d\n", *p, *(p + 1), *(p + 2));
    printf("%d %d %d\n", arr[0], arr[1], arr[2]); //两句输出效果一样。这个取的是“值”
}
```

解释几个点：

① 数组存储在一段连续的内存空间上，其中 `arr` 实际上是 `&arr[0]`，是数组首元素的地址。

② `&arr[1]` 是首元素地址往后的一个 int 类型大小的偏移量的地址，也就是数组元素 `arr[1]` 的地址，`&arr[2]`、`&arr[3]` 以此类推。（见 6.6 节，会有解释）

地址	0	1	2	3	4
元素	-	arr[0]	arr[1]	arr[2]	arr[3]

1. 利用指针声明字符串

在上一节课数组中，我们知道字符串是用字符数组的形式来声明的，在学了指针之后，我们了解输出数组的名称，就相当于输出数组首元素的地址。

- 因而我们可以类似的有这么一种声明字符串的方式：用字符指针变量指向一个字符串常量，通过字符指针变量引用字符串常量。

  ```c
  char *str = "abcd";
  ```

  因为 C 语言对字符串常量 `abcd` 是按字符数组处理的，会在内存中开辟一个字符数组，用来存放字符串常量。但是这个字符数组是没有名字的，因此，它不能通过数组名来引用，只能通过指针变量来引用。

  在不致引起误解的情况下，为了简便，有时也可说 str 指向字符串 "abcd"，但应当理解为 “指向字符串的第 1 个字符 a ”。

  ```c
  char *str = "abcd";
  printf("%c %c\n", str[1], str[2]);
  printf("%s", str);

  输出结果：
  b c
  abcd
  ```

2. 声明不同类型的指针变量

- 指针变量的类型不止有 int 和 char，还有之前我们学过的 short、float 等数值数据类型。

  ```c
  int *p1;
  short *p2;
  float *p3;
  ```

6.4 结构体指针

- 定义：结构体指针，就是指向结构体类型的**指针变量**。他的声明方式是这样的：

```
struct Student {
    char *name;
    int id;
    int score;
};

struct Student *stu;
```

- 我们访问结构体变量中的元素时，我们是使用 `.` 操作符实现的，如：

```
struct Student {
    char *name;
    int id;
    int score;
};

struct Student stu;

stu.name = "LiHua";
```

- 访问指针指向的结构体变量中的数据就应该是：

```
(*stu).name = "LiHua";
```

解释几个点：

① 先通过解引用 `*` 获得地址当中的结构体的数据，然后再通过点运算符 `.` 访问结构体中的成员变量 `name` 。

C 语言提供了 `->` 指向运算符：

```
stu->name = "LiHua";
```

解释几个点：

① 第二种写法中，`->` 是一个新的运算符，习惯称它为 “箭头”，有了它，可以通过结构体指针直接取得结构体成员；这也是 `->` 在 C 语言中的唯一用途。

这俩种方式都能够访问到指针指向的结构体变量中的数据。

- 看一个例子：

```
int main() {
    struct Student {
        char *name;
        int id;
        int score;
        bool sex;
    };

    struct Student stu1;
    stu1.name = "LiHua";

    struct Student *stu = &stu1;

    printf("stu1. name = %s \n(*stu).name = %s \nstu -> name = %s \n", stu1.name, (*stu). name, stu -> name);
}

输出结果：
stu1. name = LiHua
(*stu).name = LiHua
stu -> name = LiHua
```

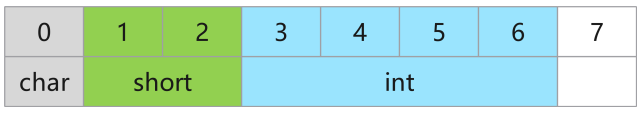
6.5 内存的动态申请与释放

内存的动态申请与释放是一项重要的技术，他可以在程序运行时动态地分配和释放内存空间，使得程序具有更大的灵活性和可扩展性。如：已经申请一个 8 个元素的数组，想要在这个基础上再加 2 个元素，要怎么做呢?

由于之前我们都是静态申请内存空间，无法处理这种情况，所以，才需要根据程序的动态执行进行动态的内存申请。

0. 回顾：不同数据类型占据的内存大小

在第二章讲解基本数据类型的时候，我们说过 int 类型比 short 类型表示的范围大，那 int 占据的空间理应更多。假设不考虑内存对齐方式，**内存空间的大小单位是字节**（计算机内存空间大小单位确实是字节），不同数据类型在内存中空间分配应该是这样的：



char 占据 1B = 8 bit

short 占据 2B = 16 bit

int 占据 4B = 32 bit

可以利用 `sizeof(类型)` 获取类型占据空间的大小。（单位为字节）

1. 内存的动态申请与释放

- C 语言使用 `malloc` 动态申请内存空间，使用 `free` 来动态释放内存空间，他们的具体使用方式是这样的：

```c
int main() {
    int *arr;
    int size; //数据的规模

    scanf("%d", &size);
    arr = (int*) malloc (size * sizeof(int)); //申请 size 大小的数组
    free(arr); //释放 arr的空间
}
```

解释几个点：

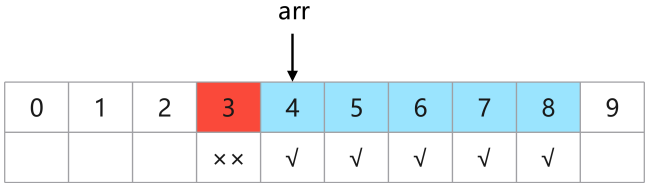
① 在这里，我们可以手动输入数据的规模 `size`，并让指针指向了一块 `size` 个 int 类型数据的内存空间，在这里可以等价理解为声明了一个数组，数组名为 `arr`。

② `malloc` 前面使用要加上 `(int*)` 进行强制类型转换。

那什么叫做 “动态” 呢?

声明指针时，内存是静态分配的。申请内存空间时，是动态分配的。因此，在执行语句 `arr = (int*) malloc (size * sizeof(int));` 时，我们需要先在内存中找到满足大小的空间。

假设这里 size = 5，表格中地址 3 已被使用，我们要怎么分配这么大的空间呢?



显然，0 1 2 这三个位置不够分配五个空间，所以，编译器选择 4 5 6 7 8 作为分配的空间，并让 `arr` 指向这块空间的首地址。这样就完成了内存的动态分配。

③ 使用 `free` 时，必须提供需要释放内存的**起始地址**，因此，必须保存好 `malloc` 返回的指针值，若丢失，则所分配的空间无法回收，称**内存泄漏**。（这里释放的时候，只需要释放 `arr` 。）

④ `malloc` 和 `free` 需配对使用。编译器不负责动态内存的释放，需要程序员手动释放。

⑤ 内存不允许重复释放。同一空间的重复释放也是危险的，因为该内存空间可能已另分配。

2. 更简便的申请与释放方式：new 与 delete

> 考研初试和复试机试都可以使用！

- `new` 和 `delete` 是 C++ 用于管理内存的两个运算符，对应于 C 语言中的 `malloc` 和 `free` ，但是 `malloc` 和 `free` 是函数，`new` 和 `delete` 是运算符，在这里，我们依然不深究其中的原理，只了解用法，感兴趣的同学可以自己去探究查询相关资料进行学习。

  C 语言代码：

```c
int main() {
    int *arr1;
    int size; //数据的规模
    scanf("%d", &size);

    arr1 = (int*) malloc (size * sizeof(int)); //申请 size 大小的数组

    free(arr1); //释放 arr的空间
}
```

  C++ 语言代码：

```cpp
#include <iostream>
using namespace std;

int main() {
    int size;
    int *arr2;
    scanf("%d", &size);

    arr2 = new int(size);

    delete(arr2);
}
```

通过上面的使用对比，我们可以很明显的感觉到 `new` 和 `delete` 使用起来更加方便。因为我们在使用 `new` 动态开辟空间时，并不需要计算所开辟空间类型的大小。因为后面跟着类型，`new` 会自动计算出类型的大小。

第七章　函数

函数就是一段封装好的，可以重复使用的代码，它使得我们的程序更加模块化，不需要编写大量重复的代码。

7.0　C 语言中的函数和数学中的函数

美国人将函数称为 “Function”。Function 除了有 “函数” 的意思，还有 “功能” 的意思，中国人将 Function 译为 “函数” 而不是 “功能”，是因为 C 语言中的函数和数学中的函数在使用形式上有些类似，例如：

- C 语言中有 length = strlen(str)
- 数学中有 y = f(x)

你看它们是何其相似，都是通过一定的操作或规则，由一份数据得到另一份数据。

不过从本质上看，将 Function 理解为 “功能” 或许更恰当，C 语言中的函数往往是独立地实现了某项功能。一个程序由多个函数组成，可以理解为「一个程序由多个小的功能叠加而成」。

7.1　函数的定义

- 定义：函数是指将一组能完成一个功能或多个功能的语句放在一起的代码结构。
- 在 C 语言程序中，至少会包含一个函数，也就是主函数 `main()`。
- C 语言中的函数就是一个完成某项特定的任务的一小段代码，C 语言的程序其实是由许多个函数组合而成的。我们也可以把函数理解为一个子程序，由一个或多个语句块组成，负责完成某项特定任务。函数一般都有**输入参数**和**返回值**，提供对过程的封装和实现细节的隐藏。

7.2　函数的分类

1. 库函数

> C语言在发布时已经为我们封装好了很多函数，它们被分门别类地放到了不同的头文件中（暂时先这样认为），使用函数时引入对应的头文件即可。这些函数都是专家编写的，执行效率极高，并且考虑到了各种边界情况，各位读者请放心使用。

- 库函数就是存放在函数库中的函数，具有明确的功能、入口调用参数和返回值。
- C 语言自带的函数称为**库函数**（Library Function）。库（Library）是编程中的一个基本概念，可以简单地认为它是一系列函数的集合，在磁盘上往往是一个文件夹。C 语言自带的库称为标准库（Standard Library），其他公司或个人开发的库称为第三方库（Third-Party Library）。

  比如 `printf` 和 `scanf`，这两个函数都是库函数。除此之外，还有 `strcpy` 字符串拷贝，`pow` 计算 n 的 k 次方等库函数。常见的库函数有：IO 函数、字符串操作函数、字符操作函数、内存操作函数、时间/日期函数、数学函数等。

2. 自定义函数

- 显然库函数并不能干所有的事情，不然就不会有程序员了。所以自定义函数是更加重要的，所谓自定义，就是函数的所有功能都是程序员来设计并实现的。
- 例：计算 3 的 n 次方（n>0）

  普通 for 循环实现

```c
int main() {
    int n;
    int result = 1;
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        result *= 3;
    }

    printf("%d", result);
    return 0;
}
```

利用 `pow` 函数实现

```c
# include<stdio.h>
# include<math.h>

int main() {
    int n;
    int result;
    scanf("%d", &n);
    result = pow(3, n); //计算 3 的 n 次方
    printf("%d", result);
}
```

相比起 for 循环，`pow` 函数更加简洁，通过函数名就可以知道这个函数的功能，power function 是幂函数，所以函数的功能就是求了的 n 次幂。

7.3 函数的组成与使用

1. 有返回值的函数

- 函数的组成是这样的：

```
函数类型 函数名(参数类型 变量名...) {
    实现的功能;
    return 返回值;
}
```

其中，函数名要能代表这段程序功能，这样能让其他人看到这个函数名时，就能大致了解这个函数的功能。

例如：

```c
int twoSum(int num1, int num2) {
    int sum = num1 + num2;
    return sum;
}

int result = twoSum(1, 2);
```

`twoSum` 为函数名； `num1` 和 `num2` 是两个参数，他们的类型都是 int 。

2. 无返回值的函数

- 如果不需要返回值，则可以写成这样：

```
//void是无返回值类刑
void 函数名(参数类型 变量名...) {
    实现的功能;
}
```

```
//void是无返回值类刑
void 函数名(参数类型 变量名...) {
    return;
    实现的功能; //执行了return之后，后面的语句不会被执行
}
```

例如：打印正整数。

```c
void printPositiveNum(int num) {
    if (num <= 0) {
        return;
    }
    printf("%d", num);
}
```

3. 函数的基本调用

- 看下面一段代码，看看到底函数是个什么情况。

```c
void printwords() {
    printf("hello world");
}

int main() {
    printwords();
    return 0;
}
```

上面这段代码等价于：

```c
int main() {
    printf("hello world");
    return 0;
}
```

上面两段代码的功能其实是一致的，第一段代码只是把打印的功能放进了单独的函数里。我们可以理解为，函数在主函数中调用。第一段代码展开后就是下面这块代码段。

4. 无类型、有参数的函数

- 让这个函数接受一个 int 类型的参数，并让函数输出这个参数的两倍：

```c
void printwords(int num) {
    printf("%d", num*2);
}

int main() {
    printWords(10);
    return 0;
}
```

在这段程序中，`printWords` 函数需要一个外部给他的 int 类型参数，并将这个参数使用一个 int 变量 `num` 接收。接着，我们就可以对这个 `num*2`，然后将他输出，就实现了我们想要的功能。

5. 有类型、有参数的函数

- 有返回值的函数：两数之和：

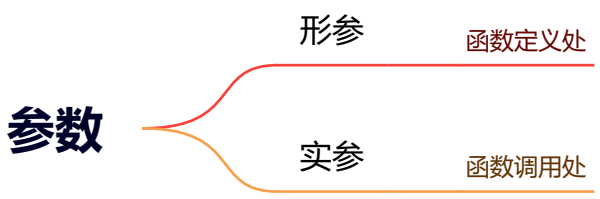
```c
int twoSum(int num1, int num2) {
    return num1 + num2;
}
int main() {
    int a = 10;
    int b = 5;
    int sum = twoSum(a, b);
    printf("%d", sum);
}
```

在这段程序中，`twoSum` 函数需要两个外部给他的 int 类型参数，并将这两个参数使用 int 变量 `num1` 和 `num2` 接收。接下来，我们将其相加，并将相加的结果作为返回值。在主函数中，我们要用一个 int 类型变量来保存 `twoSum` 函数的返回值。

7.4 指针和引用作为函数参数

如果把函数比喻成一台机器，那么参数就是原材料，返回值就是最终产品；从一定程度上讲，函数的作用就是根据不同的参数产生不同的返回值。

C 语言函数的参数会出现在两个地方，分别是**函数定义处**和**函数调用处**，这两个地方的参数是有区别的。



1. 形参（形式参数）

- 在 C 语言中，形参是指在**函数定义处**用于表示参数的变量，它们位于函数名后面的括号中。例如：

```c
int twoSum(int a, int b) {
    return a + b;
}
```

在这个函数中，a 和 b 就是形参。形参通常是在函数定义里声明的局部变量，它们只在函数内部（局部）有效，也就是说，它们只能被函数内部的代码使用。

2. 实参（实际参数）

- 实参是指在**函数被调用处**给出的参数包含了实实在在的数据，会被函数内部的代码使用，所以称为**实际参数**，例如：

```c
int twoSum(int a, int b) {
    return a + b;
}

int result = twoSum(1, 2);
```

在 `twoSum` 这个函数调用中，`1` 和 `2` 就是实参。当函数被调用时，实参的值被复制到相应的形参中，在函数内部使用的是形参，而不是实参，这就意味着，在函数内部如果形参的值发生变化，并不会影响到实参。

> 这里就有点像英语语法里面的 “形式主语 it ” 和 “实际主语” 这俩个概念。

3. 指针作为函数参数

想要让自定义函数对主函数中传递过去的参数进行操作，可以使用**指针**。我们在学习指针的时候学过，利用指针指向变量，指针的值为变量的地址，利用解引用修改指针的值中存放的数据，等效于修改了指针指向的变量。

- 看一个例子：

```c
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int main() {
    int a = 10;
    int b = 5;
    printf("a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("a=%d, b=%d", a, b);
}
```

解释几个点：

① 在 `swap` 函数中将参数类型定义为指针类型。

② 在进行参数传递的时候，实参的类型为地址。那么，在 `swap` 函数中接收到的就是主函数中的 `a` 的地址和 `b` 的地址。

③ 接下来，通过解引用，把数据 `a` 的值给 `t` ，把数据 `b` 的值给 `a` ，再把数据 `t` 的值给 `b` 。

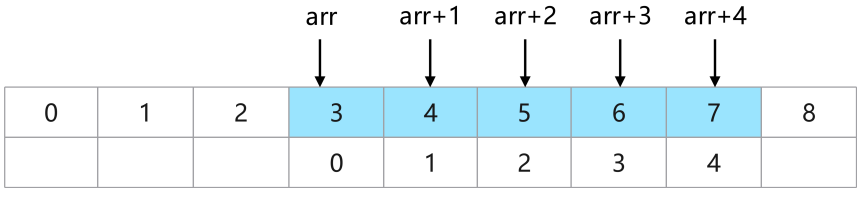
4. 数组作为函数参数

> 这个在考研初试和复试中用不到，瞄一瞄就行。

在指针这个章节，我们提到过，数组名的值，其实就是数组首元素的地址，知道了首元素的地址和数组的长度，我们就可以用访问地址的方式来访问数组元素了。

- 看一个例子：把数组中所有元素的值都加 1 。

  假设数组中的首地址为 3 ，数组长度为 5 。




```
void increaseValue(int arr[], int arrsize) {
    for (int i = 0; i < arrsize; i++) {
        arr[i] += 1;
    }
}
```

解释几个点：

① `increaseValue` 函数用一个数组变量 `arr` 和一个 int 类型的变量 `arrsize` 接收调用方传过来的数组和数组的长度。

② 利用 for 循环，让数组中每一个元素的值都加 1 。

③ 由于传过来的数组值是地址，在 `increaseValue` 中，对数组进行操作，也会影响到函数调用方的数组中的值。

5. 引用 & 作为函数参数

> 考研用这个！

- 引用是 C++ 的概念，引用的作用是给已存在的变量起一个别名。

```
int main() {
    int b = 10;
    int &a = b;
    b = 12; //此时 a=b，都为 12
    a = 8;  //此时 a=b，都为 8
}
```

解释几个点：

① 在变量前使用和取地址符一样的引用符 "&"。

② 声明 `a` 对 `b` 的引用不会开辟新的内存空间，此时 `a` 和 `b` 共享一块内存空间。

③ 需要注意的是，引用类型和引用实体应是**同种类型**的。

④ 因为 `a` 和 `b` 指向同一块内存空间，所以对 `b` 的修改，等效于对 `a` 的修改，同样的，对 `a` 的修改等效于对 `b` 的修改。

> 看下面的例子。考研就用这个嗷！

- 这种特性和指针非常相似，都是对同一块内存空间的数据进行操作，因此我们也可以利用 C++ 的引用特性，完成 `a` 和 `b` 的数据交换。

```
void swap(int &a, int &b) {
    int t = a;
    a = b;
    b = t;
}

int main() {
    int a = 10;
    int b = 5;
    printf("a = %d, b = %d\n", a, b);
    swap(a, b);
    printf("a=%d, b=%d", a, b);
}
```

7.5 递归函数

1. 在函数中调用函数

- 在函数中，也可以调用其他的函数，比如在 `printWords` 函数中，我们调用了库函数中的 `printf` 函数：

```c
void printWords(int num) {
    printf("%d", num * 2);
}
```

- 自然，我们也可以在自定义函数中，调用自定义函数：

```c
void printResult(int result) {
    printf("三角形的面积是 %d", result);
}

void trisquare(int a, int b, int c) {
    int p = (a + b + c) / 2;
    double square = pow((p * (p-a) * (p-b) * (p-c)) , 0.5);
    printResult(square);
}

int main() {
    int a, b, c;
    int result = 0;
    scanf("%d %d %d", &a, &b, &c);
    trisquare(a, b, c);
    return 0;
}
```

2. 递归

> 递归，就是自己调用自己。

- 程序调用自身的编程技巧称为递归。

  一个过程或函数在其定义或说明中有直接或间接调用自身的一种方法，它通常把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解。（大问题转化为小问题）

  递归策略只需少量的程序就可描述出解题过程所需要的多次重复计算，大大地减少了程序的代码量。

- 递归的两要素：① 递归基；② 递归关系

① 递归基：递归函数的**终止条件**。用于指示函数不再继续递归调用，常用 if 语句实现。

- 例：

```c
void func() {
    printf("你一定是研究生！\n");
    func();
    printf("初试能打五百分！\n");
}

int main() {
    func();
    return 0;
}
```

解释几个点：

① 我们会发现这个 func 函数永远在执行，永远都不会结束，这样就会造成**程序崩溃**。之所以会这样，是因为我们没有给出这个递归函数一个**递归基**。

② 递归关系：这个找的是数学上的关系。

> 是使用递归解决问题的关键，通常可以通过递归关系将一个大型复杂的问题层层转化为一个与原问题相似的较小规模的问题来进行求解。（大问题转化为小问题）

3. 递归练习1：阶乘

- 例：

$$n! = n \times (n-1) \times (n-2) \cdots \times 2 \times 1$$

前后两项之间有如下**关系**：

$$n! = n \times (n-1)!$$

我们可以根据这个关系设置递归关系。

而当 $n = 1$ 时，我们知道 $1! = 1$，所以，我们可以根据这个设置递归基。那么，我们就可以写出我们的递归函数了：

```c
int fact(int n) {
    if (n == 1){ //递归基
        return 1;
    }
    return n * fact(n - 1); //递归关系
}

int main() {
    int n;
    scanf("%d", &n);
    int result = fact(n);
    printf("%d! = %d", n, result);
}
```

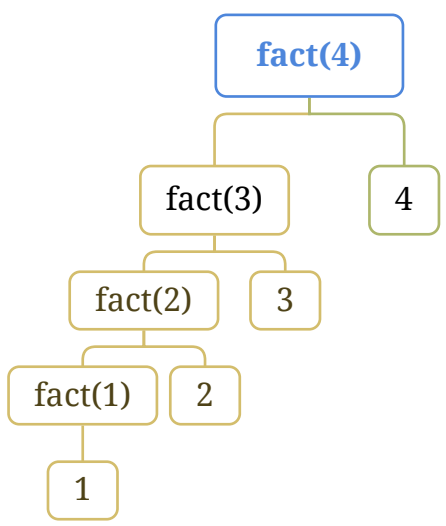
解释几个点：

① 首先定义一个 `fact` 函数来实现阶乘功能，它需要一个参数 `n` 用于计算 n 的阶乘。

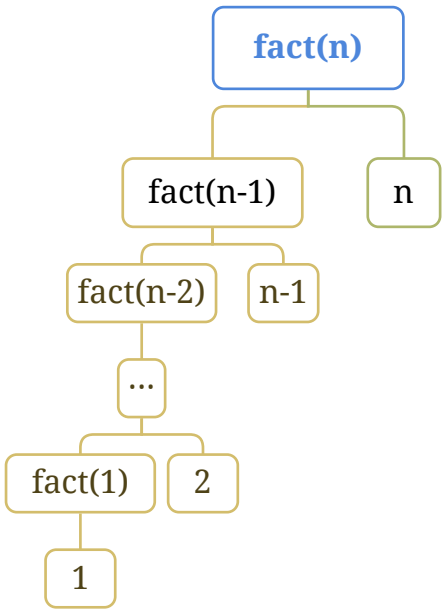
② `fact` 函数的核心代码部分利用了**递归关系**：$n! = n \times (n-1)!$

③ 由于 `fact` 是一个有返回值的函数，所以函数的结果是一个确定的值，那么这个函数的结果自然也可以作为返回值。

④ 看一下 4 的阶乘 `fact(4)` 的**递归关系**图：



⑤ n 的阶乘也是同理：



4. 递归练习2：斐波那契数列

- 例：

  斐波那契数列是这样的一串数列：

  $$1、1、2、3、5、8、13、21\cdots$$

  不难发现，有这样的规律：（作为递归关系）

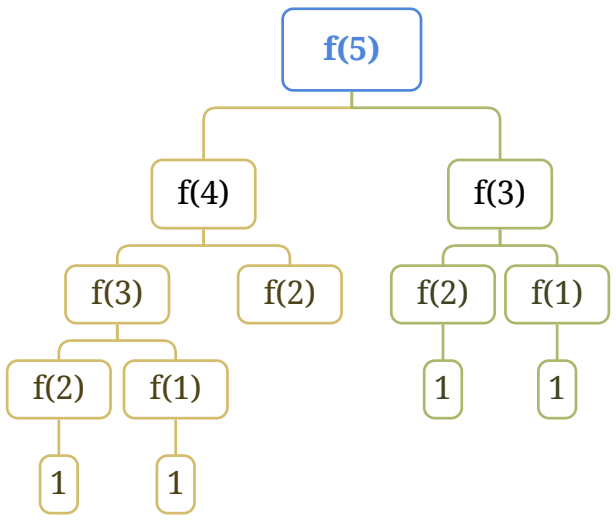
  $$f(n)=f(n-1)+f(n-2)$$

  其中，$n>2$。当$n=2$时，$f(n)=1$，这个就可以看做递归函数的递归基。

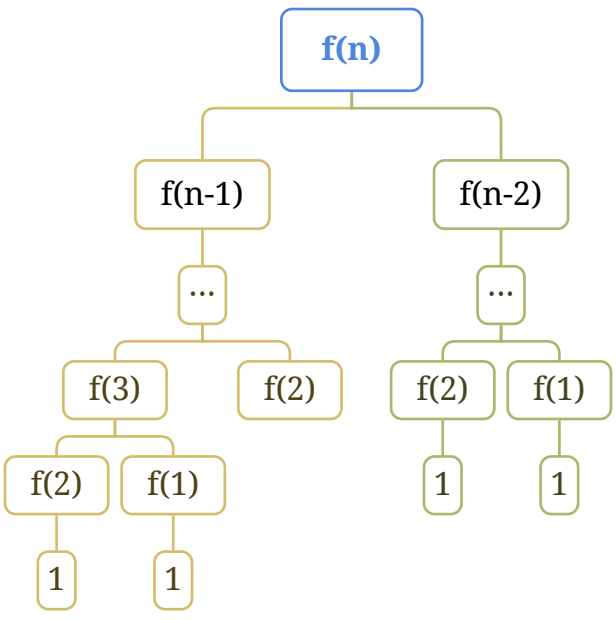
  我们发现，$f(n)$总是依赖于$f(n-1)$和$f(n-2)$的值，假如我们知道$f(n-1)$和$f(n-2)$的值，那么我们就可以顺理成章的知道$f(n)$的值。知道了**递归关系**和**递归基**，我们就可以轻松地采用递归的方式得到$f(n)$：

```c
int fibonacci(int n) {
    if (n == 1 || n == 2){ //递归基
        return 1;
    }
    return fibonacci(n - 1) + fibonacci(n - 2); //递归关系
}
```

① 看一下 5 的斐波那契数列 `f(5)` 的**递归关系**图：



② n 的斐波那契数列也是同理：



递归是一个难点，尤其是找递归关系是一个数学上的难点，这里没有技巧可言，大家只能多多进行题目练习，多积累。但即使是这样，也会遇到那种无法解决的题目，所以各位对 408 的算法题可以进行战略性取舍。我们的目标是总分过线，而不是某一门很牛！

C 语言基础知识到这里就结束了，祝大家继续拿下 408 的剩余内容！

结构体合集

这里的内容是关于各类数据结构的定义。这也是是最基础的一步，希望大家可以通过这份文档，掌握各类数据结构的定义。

这里的第一章就是 “编程基础”。

第二章　线性表

2.1　顺序表结构体定义

这里包含静态和动态两种方式。大家要注意，通常的情况下，静态的时候用数组的方式，动态都会使用到开辟释放空间。

注意：对于内存空间的申请，一定要写会，尽量去理解，实在理解不了，给我**每天写一遍，写个一周。**

- 静态顺序表

```
#define MaxSize 50  //定义线性表最大长度
//顺序表定义
typedef struct {
   int data[MaxSize];
   int length;      //实际数据的个数：用来判断是否达到最大空间
} Sqlist;

//考试的时候可以用下面的这种定义，直接定义一个数组
int A[maxSize];
int n;
```

- 动态顺序表

```
typedef struct {
    ElmeType *data;
    int size;       //用来记录表中数据的实际个数方便比较
    int capacity;   //设置用来记录L的最大空间，若是达到最大空间则需要扩容
} SeqList;
```

- 关于动态申请空间，一定一定要会哈。

```
L.data = (ElemType*)malloc(sizeof(Elemtype)*InitSize);
```

2.2　链表结构体定义

每一个结点不仅要放数据域，还要有一个指针域。

- 单链表

```
typedef struct LNode {
    ElemType data;
    struct LNode *next;
} LNode, *LinkList; //这两个是等价的
```

- 双链表

```
typedef struct DNode {
    ElemType data;
    struct LNode *prior, *next;
} DNode, *DLinkList; //这两个是等价的
```

第三章　栈、队列

3.1　栈结构体定义

- 顺序栈

```c
//顺序栈定义
typedef struct {
    int data[maxSize];
    int top; //栈顶指针
} SqStack;

//假设元素为int型，可以直接使用下面方法进行定义并初始化
int stack[maxStack];
int top=-1;
```

- 动态栈

```c
typedef struct {
    ElemType * data;
    int top;
    //int size; 这里可以不需要size，因为top就是指向最后的位置
    int capacity; //这个时候是需要容量的，方便扩容
} SqStack;
```

- 链栈

```c
// 链栈的存储结构
typedef struct StackNode {
    int data;
    struct StackNode *next;
} StackNode, *LinkStack;
```

3.2　队列结构体定义

- 顺序队列

```c
//顺序队列定义
typedef struct {
    int data[maxSize];
    int front;
    int rear;
} SqQueue;
```

- 链队

```c
//链队定义
//1.队节点定义
typedef struct QNode
{
    int data;           //数据域
    struct QNode *next; //指针域
} QNode;

//2.类型定义
typedef struct
{
    QNode *front;       //队头指针
    QNode *rear;        //队尾指针
} LiQueue;
```

第四章　串

4.1　定长顺序存储结构体定义

- 串的定长顺序存储好像没有什么结构上的要求，直接使用静态线性表的结构体定义即可。

```
#define MAXLEN 255      //预定义最大串长为255
typedef struct {
    char ch[MAXLEN];    //每个分量存储一个字符
    int length;         //串的实际长度
} SString;
```

4.2　堆分配结构体定义

- ```
  typedef struct {
      char *ch;   //按照串长分配存储区，ch指向串的基地址
      int length; //串的长度
  } HString;
  ```

4.3　块存储结构体定义

- ```
  //块链存储
  typedef struct Chunk {
      char ch[3];         //这里我定义的是放三个值
      struct Chunk *next;
  } Chunk;
  ```

- 定义头尾指针的作用是 方便进行连接操作；连接的时候别忘了处理第一个串尾的无效字符。

```
typedef struct {
    Chunk *head, *tail; //串的头尾指针
    int curlen;         //串的当前长度 (链表的节点数)
} LString;
```

第五章　树与二叉树

正如我们所熟知的存储方式一般有两种，顺序存储或者链式存储。若是使用顺序存储，则一般二叉树为了能反映二叉树中结点之间的逻辑关系，只能添加并不存在的空结点构造树像完全二叉树一样，每一个 结点与完全二叉树上的结点对应，再存储到一维数组的相应分量中。这样有可能造成空间的极大浪费，所以这里我们使用链式存储，为了方便各种操作，链式存储中也分为了几种方式，这里就不多赘述了， 用到一个写一个。

5.1　链式存储结构体定义

```c
//二叉树的存储结构，一个数据域，2个指针域
typedef struct BiTNode {
    char data;
    struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;
```

5.2　线索二叉树结构体定义

```c
typedef struct BiThrNode {
    struct BiThrNode* Lchild;
    int Ltag;
    Elemtype data;
    int Rtag;
    struct BiThrNode *Rchild;
} BiThrNode;
```

5.3　哈夫曼树结构体定义

```c
typedef struct HNode {
    char data;      //数据，非叶节点为NULL
    double weight;  //权重
    int parent;     //双亲，-1表示没有双亲，即根节点
    int lchild;     //左孩子，数组下标。-1表示无左孩子，即叶节点
    int rchild;     //右孩子
} Hnode;
```

5.4　树之双亲表示法结构体定义

```c
typedef struct Snode {
    char data;
    int parent;
} PTNode;

typedef struct {
    PTNode tnode[MAX_SIZE]; // 存放树中所有结点
    int n;                  // 结点数
}
```

5.5　孩子兄弟表示法结构体定义

```c
#define tree_size 100   //宏定义树中结点的最大数量
#define TElemType int   //宏定义树结构中数据类型

typedef struct PTNode {
    TElemType data;     //树中结点的数据类型
    int parent;         //结点的父结点在数组中的位置下标
} PTNode;

typedef struct {
    PTNode nodes[tree_size]; //存放树中所有结点
    int r, n;                //根的位置下标和结点数
}PTree;
```

第六章　图

6.1　邻接矩阵法结构体定义

- ```c
  //图的邻接矩阵定义
  typedef struct {
      int no;          //顶点编号
      char info;
  } VertexType;

  typedef struct {
      int edges[maxSize][maxSize];
      int n, e;                    //顶点个数和边个数
      VertexType vex[maxSize];     //存放节点信息
  } MGraph;
  //上面的定义如果没记住，也要记住里面的元素，如n，e等含义
  ```

6.2　邻接表法结构体定义

- ```c
  //结点的定义
  typedef char VertexType;
  typedef int EdgeType;

  #define MaxVex 100

  typedef struct EdgeNode {      //边表结点
      int adjvex;                //邻接点域，存储邻接顶点对应的下标
      EdgeType weight;           //用于存储权值，对于非网图可以不需要
      struct EdgeNode *next;     //链域，指向下一个邻接点
  } EdgeNode;

  typedef struct VertexNode {    //顶点表结点
      VertexType data;           //顶点域，存储顶点信息
      EdgeNode *firstedge;       //边表头指针
  }VertexNode, AdjList[MaxVex];

  typedef struct {
      AdjList adjList;
      int numVertexes, numEdges; //图中当前顶点数和边数
  }
  ```