计算机程序设计A
期末复习

王浩宇 2024.12

第一章 预备知识
- 进制转换
- 整数
- 浮点数
- 字符
- 基本逻辑结构

第二章 程序设计入门
- 关于C语言
- 变量的定义
- 宏定义
- 常量的定义
- C语言的运算符
- C语言的标准输入输出

第三章 结构化程序设计
- 数据类型及其特点
- 表达式
- 流程控制
- 数组
- 结构体
- 基础算法

第四章 模块化程序设计
- 函数
- 文件包含
- 常用的库函数
- 函数的递归调用

第五章 系统级程序设计
- 指针
- 链表

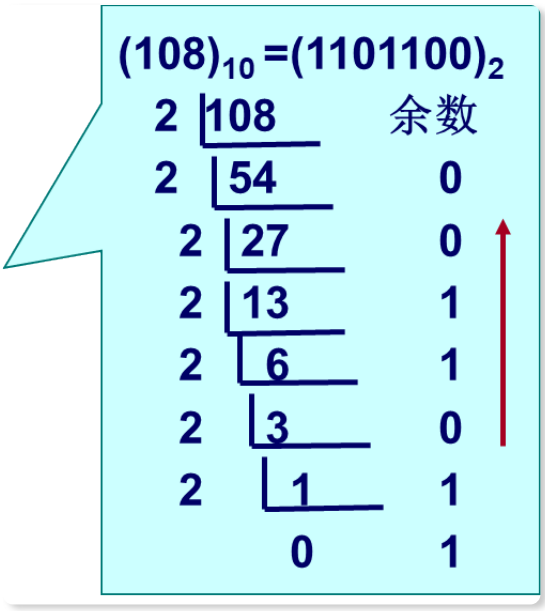
- 作业与实验常见问题

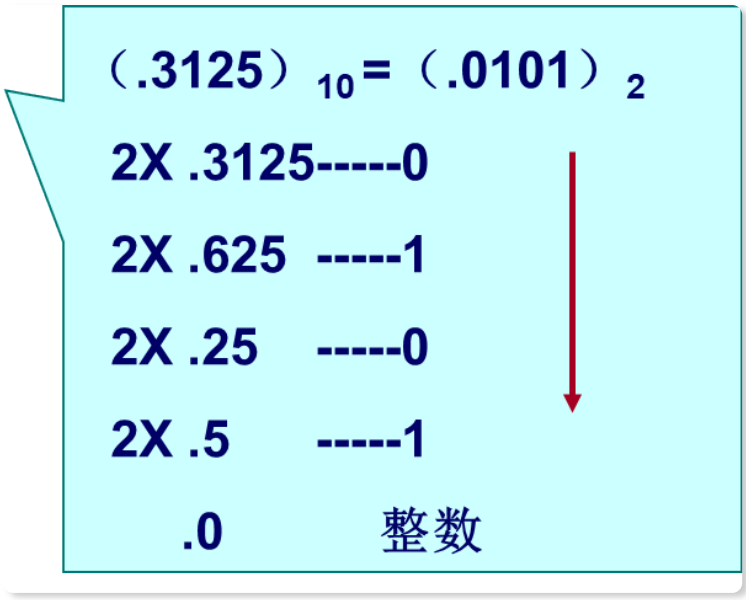
第一章 预备知识

进制转换

十进制转二进制

- 整数部分：除2取余，逆序排列
- 小数部分：乘2取整，正序排列





二进制转十进制

整数和小数部分：按权展开求和

> **例：111.101的十进制表示**
>
> - 整数部分：$1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 7$
> - 小数部分：$1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0.625$
>
> 所以，$(111.101)_2 = (7.625)_{10}$

二进制、八进制与十六进制的转换

- 二进制转八进制：每三位二进制数转换为一位八进制数

- 二进制转十六进制：每四位二进制数转换为一位十六进制数
- 八进制转二进制：每一位八进制数转换为三位二进制数
- 十六进制转二进制：每一位十六进制数转换为四位二进制数

> **例：二进制数 `1101.101` 转换为八进制数**
>
> - 整数部分
>
> `1101` 补充为 `001101`，每三位一组转换为八进制数。`001` 转换为 `1`，`101` 转换为 `5`，所以整数部分为 `15`
> - 小数部分
>
> `101` 每三位一组转换为八进制数。`101` 转换为 `5`，所以小数部分为 `5`
>
> 所以，二进制数 `1101.101` 转换为八进制数为 `15.5`。转换为十六进制同理，上述计算中三位一组改为四位一组即可。

> **例：八进制数 `17.5` 转换为二进制数**
>
> - 八进制 `1` 转换为三位二进制数 `001`
> - 八进制 `7` 转换为三位二进制数 `111`
> - 八进制 `5` 转换为三位二进制数 `101`
>
> 所以，八进制数 `17.5` 转换为二进制数为 `001111.101`，去除前导零为 `1111.101`。十六进制转换同理，每一位十六进制数转换为四位二进制数即可。

整数

整数的表示

- 原码：用最高位表示符号，0表示正数，1表示负数。其余位表示数值。
- 反码：正数的反码与原码相同，负数的反码是对原码除符号位外取反。
- 补码：正数的补码与原码相同，负数的补码是对原码除符号位外取反，然后加1。

> **例：求十进制数 `-5` 的8位原码、反码和补码**
> 已知 `5` 的二进制表示为 `101`
>
> - 原码：`10000101`（符号位为1，数值部分前补充 `0` 为 `0000101`）
> - 反码：`11111010`（符号位为1，数值部分从原码数值部分 `0000101` 取反为 `1111010`）

> - 补码：`11111011`（符号位为1，补码数值部分为反码数值部分加1）

性质：

- 正数的原码、反码、补码相同
- 从原码求补码的操作称为补码运算。对一个数进行两次补码运算，结果为原数。
- 补码的加减法运算：
  - $(X+Y)_{补}=(X)_{补}+(Y)_{补}$
  - $(X-Y)_{补}=(X)_{补}+(-Y)_{补}$

C语言中整数的类型

- `char`：1字节。$0\sim2^8-1$`(unsigned)`，$-2^7\sim2^7-1$`(signed)`
- `short`：2字节。$0\sim2^{16}-1$`(unsigned)`，$-2^{16}\sim2^{15}-1$`(signed)`
- `int`：4字节。$0\sim2^{32}-1$`(unsigned)`，$-2^{32}\sim2^{31}-1$`(signed)`
- `long`：4字节。$0\sim2^{32}-1$`(unsigned)`，$-2^{32}\sim2^{31}-1$`(signed)`
- `long long`：8字节。$0\sim2^{64}-1$`(unsigned)`，$-2^{63}\sim2^{63}-1$`(signed)`

整数的溢出

整数的溢出是指整数运算的结果超出了该类型的表示范围。当发生溢出时，计算机会将结果截断为该类型的表示范围内的值，并按照补码的规则进行解析。例如，`unsigned char`类型的数值范围为`0~255`，当计算`255+1`时，二进制位`11111111+00000001=100000000`，截断溢出的一位`1`，结果为`00000000`，即`0`。

浮点数

浮点数使用“尾数+阶码”的形式表示，尾数和阶码都是用二进制表示的。浮点数的值 $X=$(尾数)$\times2^{阶码}$。规定尾数第一位一定为1。

> 实际上浮点数就是二进制科学计数法，以此来表示更大范围的数值。

- 单精度浮点数`float`：1位符号位、8位阶码、23位尾数，共4字节，精度为6位有效数字。取值范围为3.4×10^{-38}~3.4×10^{38}。
- 双精度浮点数`double`：1位符号位、11位阶码、52位尾数，共8字节，精度为15位有效数字。取值范围为1.7×10^{-308}~1.7×10^{308}。

- 长双精度浮点数 `long double` 的长度和精度因平台和编译器而异。

> **精度有效数字的计算方式**
> 以单精度浮点数为例，其尾数有23位，即最多有23位有效数字。但是，浮点数的尾数是规格化的，即尾数的最高位一定是1，所以实际上只有22位有效数字。因此，单精度浮点数的精度为 $floor(lg(2^{23})) = 6$ 位有效数字。

字符

字符的表示

- ASCII码：使用7位二进制数表示128个字符，包括控制字符、可显示字符和扩展字符。（符号位为0）
- 扩展ASCII码：使用8位二进制数表示256个字符，包括ASCII码和扩展字符。
- 其他编码：GB2312、GBK、Unicode、UTF-8等。

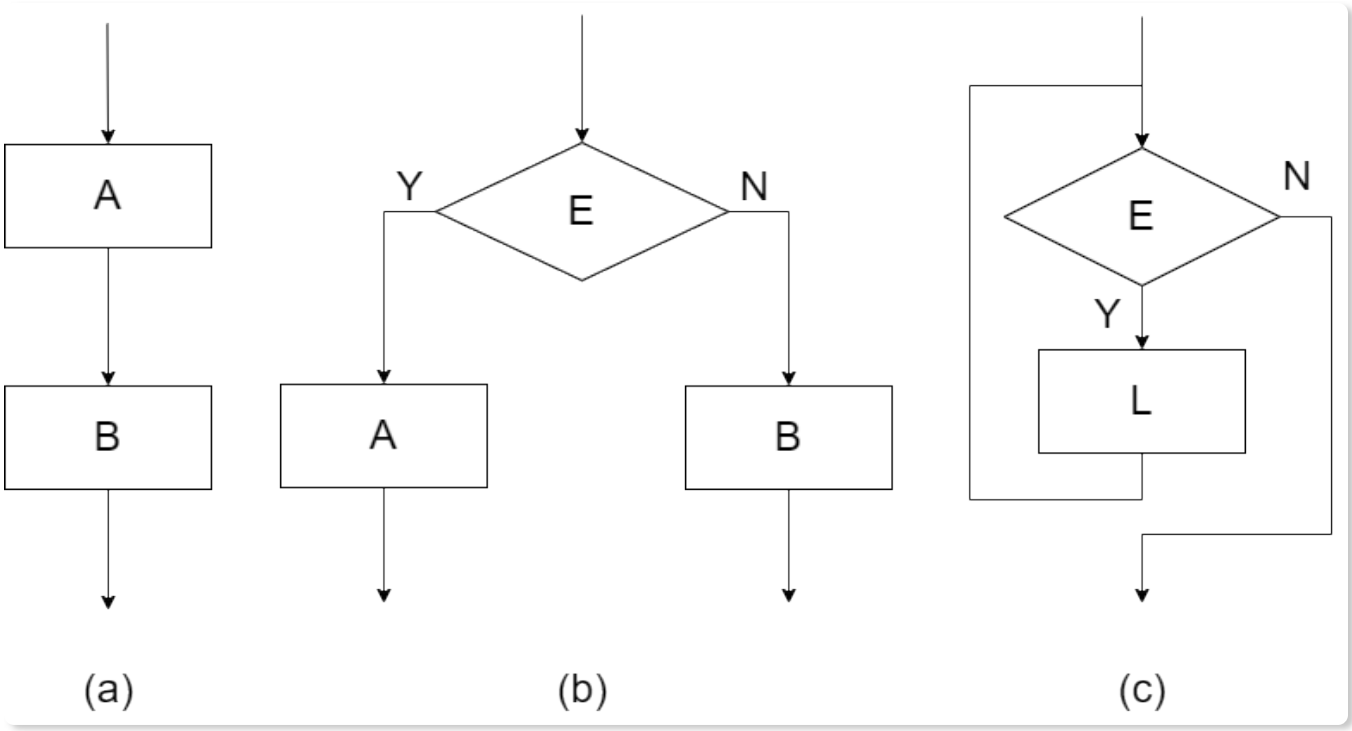
转义字符（看起来是两个字符，但被解析为一个字符）

- `'\n'`：换行
- `'\t'`：制表符
- `'\\'`：反斜杠
- `'\0'`：空字符
- ...

字符串

字符串是由字符组成的有序序列。在C语言中，字符串以字符数组的形式存储。

基本逻辑结构



顺序结构(a)

程序按照代码的顺序执行，不发生跳转。

选择结构(b)

根据条件`E`选择执行不同的代码块。

> 对应的C语言语句有`if`、`if-else`、`switch-case`。

循环结构(c)

根据条件`E`重复执行代码块`L`。

> 对应的C语言语句有`while`、`do-while`、`for`。

流程图绘制

第二章 程序设计入门

关于C语言

- C语言介于高级语言和汇编语言之间，是一种中级语言。
- C语言是一种**结构化**语言，程序由函数组成。
- C语言是一种**面向过程**的语言，程序由一系列函数调用组成。
- C语言是一种**编译型**语言，程序需要经过编译器编译成机器码后才能运行。
- C语言是一种**静态类型**语言，变量在声明时需要指定类型。

> **编译与解释**
> 编译：将高级语言源代码转换为机器码的过程。
>
> - 以C语言为代表的编译型语言，需要完整的程序编译无误后才能运行。
>
> 解释：将高级语言源代码逐行转换为机器码并执行的过程。
>
> - 以Python为代表的解释型语言，逐行解释执行，遇到错误即停止。几乎每条语句都能单独翻译执行。

变量的定义

程序在内存中运行，程序运行过程中需要的内容存放在内存中。变量是程序运行过程中存放数据的一种形式，**C语言通过定义变量来为这些数据分配内存空间，C语言中的变量必须先定义后使用。**

变量定义的命名规范

- 变量名由字母、数字和下划线组成，不能以数字开头。
- 变量名区分大小写。
- 变量名不能是C语言的保留字（如`int`、`float`、`if`等）。

变量的作用域

宏定义

宏定义是一种预处理指令，用于定义一个标识符代表一个常量或一个代码片段。宏定义的格式为`#define 标识符 值`，例如`#define PI 3.14159`。在程序中输入`PI`时，会被替换为`3.14159`。

常量的定义

常量是程序运行过程中不可改变的数据，C语言中常量可以分为字面常量、符号常量和常变量。

- 字面常量：直接出现在程序中的常量，如`123`、`3.14`、`'a'`、`"hello"`。
- 符号常量：使用`#define`指令定义的常量，如`#define PI 3.14159`。
- 常变量：使用`const`关键字定义的常量，如`const int MAX=100`。

> **符号常量与常变量的区别：**
>
> - 符号常量是在预处理阶段进行替换，不会分配内存空间（可以理解为编译时对代码进行全文替换）；常变量是在编译阶段分配内存空间，有类型信息。
> - 符号常量不需要指定类型，常变量需要指定类型。
> - 符号常量不能被修改；常变量只是不允许被赋值，但由于其是有内存空间的变量，所以可以通过指针修改其值。

C语言的运算符

运算符与优先级

运算符类型	包含的运算符	优先级
括号和结构体元素	`()`、`[]`、`{}`、`.`、`->`	1
单目运算符	正号`+`、负号`-`、`++`、`--`、取值`*`、`&`、`!`、`~`、`sizeof`	2
乘除	`*`、`/`、`%`	3
加减	`+`、`-`	4

运算符类型	包含的运算符	优先级
移位	`<<`、`>>`	5
关系	`<`、`<=`、`>`、`>=`	6
相等	`==`、`!=`	7
位与	`&`	8
位异或	`^`	9
位或	`|`	10
逻辑与	`&&`	11
逻辑或	`||`	12
三目运算符	`? :`	13
赋值	`=`、`+=`、`-=`、`*=`、`/=`、`%=`、`&=`、`|=`、`^=`、`<<=`、`>>=`	14
逗号	`,`	15

自增自减运算符

- `++`：自增运算符，使变量的值加1。
- `--`：自减运算符，使变量的值减1。

> 前缀和后缀运算符的区别：
>
> - 前缀运算符：`++i`，先自增再取值。
> - 后缀运算符：`i++`，先取值再自增。

> 自增自减运算符只能用于变量，不能用于常量和表达式！如`++(a+b)`中的`a+b`是一个表达式，不能直接对其进行自增操作；`++5`中的`5`是一个常量，不能直接对其进行自增操作。

C语言的标准输入输出

格式化字符串

在C语言的标准输入输出函数中，使用格式化字符串来指定输入输出的数据类型和格式。格式化字符串中可以包含普通字符和格式控制符。其中，格式控制符以`%`开头，用于指定输入输出的数据类型和格式。

下面是常用的格式控制符与数据类型的对应关系：

格式控制符	数据类型	格式控制符	数据类型	格式控制符	数据类型
`%d`	int	`%ld`	long	`%lld`	long long
`%u`	unsigned int	`%lu`	unsigned long	`%llu`	unsigned long long
`%f`	float	`%lf`	double	`%Lf`	long double
`%c`	char	`%s`	字符串	`%p`	指针
`%x`	16进制整数	`%o`	8进制整数	`%e`	科学计数法
`%g`	自动选择`%f`或`%e`	`%%`	输出`%`	...	...

`%a.bf`：`a`表示输出的总宽度，`b`表示小数点后的位数。如`%8.2f`表示输出宽度为8，小数点后保留2位。

scanf()函数

函数定义：`int scanf(const char *format, ...);`

- `format`：格式化字符串，用于指定输入的数据类型和格式。
- `...`：省略号表示可变参数，为接收输入的变量的**地址**。变量地址与格式控制符一一对应。
- 返回值：成功返回读取的参数个数，失败返回`EOF`。

printf()函数

函数定义：`int printf(const char *format, ...);`

- `format`：格式化字符串，用于指定输出的数据类型和格式。
- `...`：省略号表示可变参数，为输出的变量 **（不是地址！除非你本身要输出的就是地址）**。变量与格式控制符一一对应。
- 返回值：成功返回输出的字符个数，失败返回负数。

从文件输入输出

- `fscanf()`：从指定文件中读取数据。
  函数定义：`int fscanf(FILE *stream, const char *format, ...);`
  - `stream`：文件指针，指向要读取的文件指针。从文件读入时，通过语句`FILE *stream=fopen("filename", "r");`打开文件；从键盘输入时为`stdin`；从屏幕输出时为`stdout`。
  - 后续参数写法与`scanf()`函数相同。
  - 返回值与`scanf()`函数相同。
- `fprintf()`：向指定文件中写入数据。
  函数定义：`int fprintf(FILE *stream, const char *format, ...);`
  - `stream`：文件指针，指向要写入的文件指针。向文件写入时，通过语句`FILE *stream=fopen("filename", "w");`打开文件；向屏幕输出时为`stdout`。
  - 后续参数写法与`printf()`函数相同。
  - 返回值与`printf()`函数相同。

其他标准输入输出函数

- `getchar()`：从标准输入中读取一个字符。用法：`char ch=getchar();`
- `putchar()`：向标准输出中输出一个字符。用法：`putchar(ch);`
- `gets()`：从标准输入中读取一整行。用法：`char str[100]; gets(str);`
- `puts()`：向标准输出中输出一整行并换行。用法：`puts(str);`
- `fgets()`：从指定文件流中读取一整行。用法：`char str[100]; fgets(str, sizeof(str), stream);` `stream`用法同`fscanf()`函数。
- `fputs()`：向指定文件中输出一个字符串。用法：`fputs(str, stream);`
- `getche()`：从标准输入中读取一个字符并显示在屏幕上。用法：`char ch=getche();`

> `fgets()` 函数与 `gets()` 函数的区别
>
> - `gets()` 函数从标准输入中读取一行字符串，会将换行符替换为 `\0` 。
> - `fgets()` 函数从指定文件中读取一行字符串，会保留换行符并在其后添加 `\0` 。
>
> 因此，使用 `fgets()` 函数时需要注意，读取的字符串结尾会有一个换行符，需要手动去除。

第三章 结构化程序设计

数据类型及其特点

- 整型
- 浮点型
- 字符型

> 注意：浮点数由于精度问题，可能出现误差。在判断两个浮点数是否相等时，尽量通过判断它们的差的绝对值是否小于一个很小的数（如 `1e-6` ）来判断，而非直接使用 `==` 运算符。

表达式

C语言中，表达式是由运算符和操作数组成的式子，无论是简单的单个变量、常量，还是复杂运算的表达式，都可以看作是一个C语言的表达式。表达式有唯一的值。

计算表达式的值需要按照运算符的优先级进行计算。

> **算数表达式例：有 `int a=-5, b=-4, c=-3, k;`**
>
> - `a<=b` 的值为 `1` ，因为 `-5<=-4` 为真。
> - `a+b>c-b` 等价于 `(a+b)>(c-b)` ，值为 `0` ，因为 `-5-4>-3+4` 为假。
> - `k=3>=c` 等价于 `k=(3>=c)` ，值为 `1` ，因为 `3>=-3` 为真。
> - `a==-5<=c` 等价于 `a==(-5<=c)` ，值为 `0` ，因为 `-5==1` 为假。
> - `a<b<c` 等价于 `(a<b)<c` ，值为 `0` ，因为 `1<-3` 为假。
> - `a==a==a` 等价于 `(a==a)==a` ，值为 `0` ，因为 `1==-5` 为假。
> - ...

> **逻辑表达式例：有 `int a=1, b=0, c=2, k;`**
>
> - `a&&b` 的值为 `0`，因为 `1&&0` 为假。
> - `a||b` 的值为 `1`，因为 `1||0` 为真。
> - `a&&b||c` 等价于 `(a&&b)||c`，值为 `1`，因为 `0||2` 为真。
> - ...

> 逗号表达式：逗号表达式的值为最后一个表达式的值。如 `k=a,b,c`，则 `k` 的值为 `c`。

流程控制

判断

- `if-else` 语句：条件为真执行语句块1，否则执行语句块2。

```
if (条件表达式){
    语句块1;
}
else{
    语句块2;
}
```

- `switch-case` 语句：表达式的值与 `case` 后的常量进行匹配，匹配成功执行对应的语句块，匹配失败执行 `default` 语句块。

```
switch (表达式){
    case 常量1:
        语句块1;
        break;
    case 常量2:
        语句块2;
        break;
    ...
    default:
        语句块n;
}
```

    - `switch-case` 语句中，`case` 后的常量必须是整型常量，且不能重复。

    - `default`语句块是可选的，用于处理没有匹配的情况。
    - `break`语句用于结束当前`case`的执行，如果没有`break`语句，程序会继续执行下一个`case`的语句块。

- 三目运算符：根据条件表达式的真假执行不同的表达式。

```
表达式1?表达式2:表达式3;
```

当表达式1为真时，执行表达式2；当表达式1为假时，执行表达式3。逻辑上等价于：

```
if (表达式1){
    表达式2;
}
else{
    表达式3;
}
```

> 三目运算符的值为表达式2或表达式3的值，取决于表达式1的真假。

循环

- `while`循环：当条件表达式为真时执行语句块。

```
while (条件表达式){
    语句块;
}
```

- `do-while`循环：先执行一次语句块，然后判断条件表达式是否为真，为真则继续执行语句块。

```
do{
    语句块;
}while (条件表达式);
```

- `for`循环：执行一次初始化表达式，然后判断条件表达式是否为真，为真则执行语句块，然后执行更新表达式，再次判断条件表达式是否为真，如此循环。

```
for (初始化表达式; 条件表达式; 更新表达式){
    语句块;
}
```

> - `for`循环中，初始化表达式只在循环开始时执行一次，更新表达式在每次循环结束后执行一次。
> - `for`循环中的三个表达式都是可选的，可以缺省。

跳转

- `break`语句：用于结束当前循环。
- `continue`语句：用于结束当前循环的本次迭代，继续下一次迭代。即跳过本次循环体中剩余的语句，直接进行下一次循环。

数组

数组是一种存储多个相同类型数据的数据结构，数组中的每个元素都有一个唯一的下标，用于访问该元素。**数组的下标从0开始，最大下标为数组长度减1。**

数组的存储

- 数组元素依次存储在内存中的连续位置。
- 数组名是数组首元素的地址，即`&a[0]`。（但使用`sizeof`和`&`运算符时，数组名与数组首元素的地址并不等价）
- 数组名本身是一个常量，不能被赋值，因此不能使用形如`a=b`的赋值语句来复制数组。
- 数组的长度是数组元素的个数，可以通过`sizeof`运算符计算。
- 数组的长度需要是一个常量表达式，不能是变量

  > 在同学们日常的编程中可以注意到，实际上数组的长度可以是变量，但是这是C99标准中的内容，并非所有编译器都支持，所以在编写代码时应该尽量避免使用变量作为数组的长度。
  > 例如定义`int a[n];`可以通过`const int m=n; int a[m];`实现以规避这个问题。

数组的赋值

- 数组的每个元素都是一个变量，可以通过下标访问和赋值：`a[i]=x;`

- 数组在定义时可以直接赋初值：`int a[5]={1,2,3,4,5};`；当只赋部分初值时，未赋值的元素默认为0；当没有赋值时，数组元素的值是不确定的。

多维数组

数组本质上都是一维的，多维数组可以看作是一维数组的扩展。多维数组的元素依然是连续存储的，只是在访问时需要根据多维数组的下标计算出对应的一维数组下标。例如二维数组`int a[3][4]`，可以看作是一个包含3个（包含4个`int`类型变量的一维数组）的一维数组，而访问`a[i][j]`时，实际上是访问了这个一维数组中的第`i*4+j`个`int`类型变量。

> **多维数组的行数可以省略，而列数不能省略。**
> 由上面提到的，访问`a[3][4]`中的元素`a[i][j]`实际上是访问了a中的第`i*4+j`个`int`类型变量，程序也是这样计算的，所以在定义多维数组时，列数（例中为`4`）不可省略，而行数（例中为`3`）不涉及计算，可以省略。

字符数组

元素类型为`char`的数组称为字符数组，用于存储字符串。**字符串在字符数组中存储时，以额外的`\0`结尾**，表示字符串的结束，因此字符数组的长度至少比字符串长度多1。

- 字符数组的初始化：`char str[6]="hello";`，未赋值的元素默认为`0`。
- 字符数组的输入输出：使用`%s`格式控制符。
  - 输入：`scanf("%s", str);`（`str`本身就是地址，不需要加`&`）
  - 输出：`printf("%s", str);`

  > scanf()读取字符串时，遇到空格、制表符、回车等空白字符会停止读取，因此无法读取带空格的字符串。可以使用`gets()`函数读取带空格的字符串。

- 字符数组之间不能直接使用`=`赋值，需要使用`strcpy()`函数。

```
char src[6]="hello";
char dest[6];
strcpy(dest, src);
```

结构体

结构体是一种用户自定义的数据类型，用于存储多个不同类型的数据。结构体中的数据称为成

员，结构体的成员可以是基本数据类型、数组、指针、结构体等。定义出的结构体类型和基本数据类型一样，可以用于定义变量、数组、指针等。通过`.`运算符可以访问结构体的成员，通过`->`运算符可以访问结构体指针指向的结构体的成员。

> **例如：定义一个包含学生信息的结构体**
>
> ```
> typedef struct{
>     char name[20];
>     int age;
>     float score;
> } Student; // 定义了一个名为Student的结构体类型
> ```
>
> 后续可以通过语句`Student stu;`定义一个名为`stu`的结构体变量，通过`stu.name`、`stu.age`、`stu.score`访问结构体的成员；可以通过`Student *p=&stu;`定义一个指向`stu`的结构体指针，通过`p->name`、`p->age`、`p->score`访问结构体的成员。

基础算法

排序

下方`swap`函数表示交换两个变量的值。

- 冒泡排序

```
for(i=0; i<n-1; i++)
    for(j=0; j<n-i-1; j++)
        if(a[j]>a[j+1])
            swap(a[j], a[j+1]);
```

- 选择排序

```
for(i=0; i<n-1; i++){
    min=i;
    for(j=i+1; j<n; j++)
        if(a[j]<a[min])
            min=j;
    swap(a[i], a[min]);
}
```

- 插入排序

```
for(i=1; i<n; i++){
    temp=a[i];
    for(j=i-1; j>=0 && a[j]>temp; j--)
        a[j+1]=a[j];
    a[j+1]=temp;
}
```

查找

- 顺序查找
- 二分查找

```
int binary_search(int a[], int l, int r, int val){
    while(l<=r){
        int mid=(l+r)/2;
        if(a[mid]==val) return mid;
        else if(a[mid]<val) l=mid+1;
        else r=mid-1;
    }
    return -1;
}
```

其他基础算法思路

- 递推
- 枚举
- 回溯
- 贪心

- 分治
- 动态规划
- ...

第四章 模块化程序设计

函数

函数是一段完成特定任务的代码块，可以通过函数名调用函数。函数可以接收参数，也可以返回值。函数的定义包括函数类型、函数名、参数列表和函数体。

函数的参数

- 形参：函数**定义**时的参数，用于接收函数调用时传入的实参。
- 实参：函数**调用**时传入的参数，用于给函数的形参赋值。

参数传递的性质

- 主函数在调用子函数时，计算出实参的值，将实参的值转换为形参的类型后传给实参；
- 形参有自己的存储空间，实参的值传递给形参后，**形参的值改变不会影响实参的值**；
- 形参只在函数内部有效，函数调用结束后形参的地址被释放，不可再被使用。

> 一般对于需要修改实参的值的情况，可以通过传递指针的方式实现。例如需要修改主函数中 `a` 的值，可以通过传递 `&a` 的方式，将 `a` 的地址作为参数传递给形参 `int *p`，然后通过 `*p` 修改 `a` 的值。

数组作为函数的参数

由于数组名是数组首元素的地址，因此数组名作为函数参数时，实际上是传递了数组的地址，因此形参是一个指针。

- 有以下几种方式定义形参接受一维数组，通过这些形式接收参数后，在函数中都可以将 `a` 看作一个数组使用，也可以使用指针的方式访问数组元素：
  - `int a[]`：形参是一个数组，可以接受任意长度的数组。
  - `int a[N]`：形参是一个数组，可以接受长度为 `N` 的数组，`N` 需要是一个全局定义的常

量或宏定义。
    - `int *a`：形参是一个指针，可以接受任意长度的数组。
- 有以下几种方式定义形参接受二维数组：
    - `int a[][M]`：形参是一个二维数组，可以接受任意行数、列数为`M`的二维数组，`M`需要是一个全局定义的常量或宏定义。
    - `int a[N][M]`：形参是一个二维数组，可以接受行数为`N`、列数为`M`的二维数组，`N`、`M`需要是一个全局定义的常量或宏定义。
    - `int (*a)[M]`：形参是一个指针，可以接受任意行数、列数为`M`的二维数组。
    - `int *a`：形参是一个指针，实参不能直接传递`a`而需要传递`a[0]`或`&a[0][0]`或`(int *)a`

  其中前三种方式在函数中使用时，可以直接使用`a[i][j]`访问二维数组元素；第四种方式在函数中使用时，由于列数未知，因此只能通过指针的方式（形如`*(a+i*M+j)`的方式）访问二维数组元素。

函数的返回值

当需要从函数中返回一个值时，可以使用`return`语句。`return`语句可以返回一个值，也可以不返回值（取决于函数的返回类型）。

- `return`语句可以提前结束函数的执行，返回到函数调用的地方。函数执行到任意一个`return`语句时，都会结束函数的执行。
- 函数的返回值类型由函数的类型决定，如果函数没有返回值，返回类型为`void`。当返回值类型和函数类型不一致时，会将返回值转换为函数类型。

函数的声明与定义

- 函数的声明：告诉编译器函数的存在，函数的声明包括函数的返回类型、函数名和参数列表。
- 函数的定义：实现函数的功能，包括函数的返回类型、函数名、参数列表和函数体。

函数的声明或定义必须在函数调用之前，否则编译器无法识别函数。当函数定义在主函数之后时，需要在主函数之前进行函数的声明；当函数定义在主函数之前时，可以不进行函数的声明。

> 函数声明中的参数名可以省略，只需要参数类型即可。例如`int max(int, int);`。
> **注意：函数声明的结尾需要加分号！**

> 函数不能嵌套定义

变量的作用域

变量的作用域是变量在程序中有效的范围。C语言中变量的作用域有以下几种：

- 局部变量：定义在函数内部的变量，只在函数内部有效。
- 全局变量：定义在函数外部的变量，整个程序中有效。
- 形参：函数的参数，只在函数内部有效。
- 静态变量：使用`static`关键字定义的变量，只在定义的函数中有效，但是其值在函数调用结束后不会被释放。

变量的就近原则：在当前作用域中存在同名变量时，优先使用当前作用域中定义的变量，而不会向更大的作用域中寻找变量。

> 多次调用函数时，静态局部变量只赋一次初值。因为赋初值是在分配内存时进行的，而静态局部变量只在第一次调用时分配内存，因此只会在此时赋一次初值。

文件包含

C语言中，可以通过`#include`预处理指令包含头文件，头文件中包含了函数的声明、宏定义等内容。头文件的作用是将函数的声明和定义分离，使得程序结构更加清晰，提高代码的可读性和可维护性。

- `#include <文件名>`：用于包含系统头文件，编译器会在系统目录下查找头文件。
- `#include "文件名"`：用于包含用户自定义的头文件，编译器会在当前目录下查找头文件。因此，用户自定义的头文件应该放在当前目录下，本课程中一般情况下头文件和源文件需要放在同一目录下。

`include`的工作原理：`#include`指令会将指定的头文件内容复制到当前文件中，因此头文件与源文件中不能有重复的定义，否则会出现重定义错误。

常用的库函数

`stdio.h`库函数

`scanf`，`printf`，`getchar`，`putchar`，`gets`，`puts`，`fgets`，`fputs`，`fscanf`，`fprintf`，`getche`等。

`string.h` 库函数

- `strlen(const char *s)`：返回字符串`s`的长度。
- `strcmp(const char *s1, const char *s2)`：比较字符串`s1`和`s2`的大小，相等返回`0`，`s1<s2`返回负数，`s1>s2`返回正数。
- `strcpy(char *dest, const char *src)`：将字符串`src`复制到`dest`中。
- `strcat(char *dest, const char *src)`：将字符串`src`连接到`dest`的末尾。
- ...

`stdlib.h` 库函数

- `srand(unsigned int seed)`：设置随机数种子。
- `rand()`：生成一个随机数。
- `malloc(size_t size)`：分配`size`字节的内存空间。
- `free(void *ptr)`：释放内存空间。
- ...

`math.h` 库函数

- `sqrt(double x)`：求`x`的平方根。
- `pow(double x, double y)`：求`x`的`y`次方。
- `fabs(double x)`：求`x`的绝对值。
- `sin(double x)`：求`x`的正弦值。
- ...

`time.h` 库函数

- `time_t time(time_t *t)`：获取当前时间。
  > 使用时间作为随机数种子：`srand(time(NULL));`

函数的递归调用

递归是指函数直接或间接调用自身的过程。递归函数包括两个部分：递归调用和递归终止条件。递归函数的调用过程是通过不断调用自身来解决问题的过程，递归终止条件是递归调用的结束条件，当满足终止条件时，递归调用结束。如果递归终止条件设置不当，会导致递归调用无法结

束，出现死循环。

第五章 系统级程序设计

指针

程序的内存结构

计算机在运行程序时，会将程序加载到内存中运行。程序在内存中的存储结构包括以下几个部分：

- 代码段：存放程序的代码。
- 数据段：存放程序的全局变量和静态变量。
- 栈：存放函数的局部变量、函数的参数、函数的返回地址等。
- 堆：存放动态分配的内存空间。
- ...

什么是指针

上面我们知道，各种变量都存储在内存之中，有特定的地址空间来存放其中的数据。一般情况下，程序中可以直接使用变量名来访问变量；二般情况下，也可以通过直接访问变量所在的内存空间来访问变量。**指针就是用来存储变量的地址的变量。**

指针的定义

定义变量时，在变量名前加`*`，即可定义一个指针变量，表示指向对应类型的变量的指针变量。例如：`int *p;`定义了一个指向`int`类型变量的指针变量`p`。

指针变量的大小

与系统有关。

- 在32位系统中，指针变量的大小为4字节；
- 在64位系统中，指针变量的大小为8字节。

指针变量的取值范围

与系统有关。

- 在32位系统中，指针变量的取值范围为`0x00000000-0xFFFFFFFF`（与`unsigned int`取值范围相同）
- 在64位系统中，指针变量的取值范围为`0x0000000000000000-0xFFFFFFFFFFFFFFFF`（与`unsigned long long`取值范围相同）

空指针

空指针是指向空地址的指针，即指针变量未指向任何内存空间。空指针的值为`0`或`NULL`（也是0），表示指针变量未初始化。空指针在程序中可以用于判断指针变量是否初始化，避免野指针的出现。

野指针

指针变量未初始化，或者指向的内存空间已经释放，但指针变量未置空，此时指针变量指向的内存空间是不确定的，称为野指针。程序中野指针的存在会导致程序运行出现不可预知的错误，因此在使用指针变量时，**应该始终保证指针变量指向的内存空间是有效的！！！**

> **实践中的问题：为什么没有给指针初始化程序还能正常运行?**
> 这是一个巧合，因为指针变量未初始化时，指针变量指向的内存空间是不确定的，有可能是有效的内存空间，也有可能是不可使用的内存空间（如其他程序占用，系统保留等）。当指针恰好指向可用的内存空间时，程序运行正常；当指针指向不可用的内存空间时，程序就会出现错误，被被系统终止运行甚至影响其他程序的运行。
> **因此绝不能对野指针指向的内存空间进行操作，更严格地说，应该避免野指针的出现。**

指针的运算

取地址运算符`&`

`&`运算符用于取变量的地址，返回变量的地址。例如定义变量`int a`，则`&a`表示变量`a`的地址。可以使用语句`int *p=&a`定义一个指针变量`p`，使得`p`指向变量`a`。

解引用运算符`*`

`*`运算符用于获取指针变量指向的内存空间的值，返回指针变量指向的内存空间的值。例如定

义指针变量 `int *p`，则 `*p` 表示指针变量 `p` 指向的内存空间的值。

> 从上面的描述中可以看出，`&` 运算符和 `*` 运算符是一对互逆运算符，`&*p` 等价于 `p`，`*&a` 等价于 `a`。因此在代码中，连续的 `&*` 或 `*&` 运算符可以忽略。

指针的算数运算

指针变量可以进行加减运算，运算的结果是指针变量指向的内存空间的地址。例如定义指针变量 `int *p`，则 `p+1` 表示指针变量 `p` 指向的内存空间的下一个地址。指针的加减移动的单位是指针变量的类型大小，例如 `int` 类型的指针加1时，其指向的地址移动的是4个字节，`char` 类型的指针加1时，其指向的地址则只移动的是1个字节。

> **例：已知 `int a` 的地址为 `0xA1919810`，定义指针 `int *p=&a`，则有：**
>
> - `p` 的值为 `0xA1919810`；
> - `p+1` 的值为 `0xA1919814`，即 `p` 的值加4，而非 `0xA1919811`；
> - `p++` 后，`p` 的值变为 `0xA1919814`。

指针之间可以比较大小，比较的是指针变量指向的内存空间的地址。例如定义指针变量 `int *p1, *p2`，则 `p1>p2` 表示指针变量 `p1` 指向的内存空间的地址大于指针变量 `p2` 指向的内存空间的地址。

指针的下标运算

指针变量可以通过下标运算访问指针指向的内存空间的值。例如定义指针变量 `int *p`，则 `p[i]` 等价于 `*(p+i)`，表示指针变量 `p` 指向的内存空间的第 `i` 个元素的值。用法与数组类似，但是指针变量的下标运算不会进行越界检查，因此**需要保证指针指向的内存空间是有效的。**

> 需要注意，定义指针变量时运算符 `*` 低于 `[]`。因此 `int *p[5]` 首先解析 `p[5]`，因此是一个包含5个元素的数组，随后解析 `*`，因此数组元素是指向某种类型的指针，最后解析 `int` 得到基本元素类型，从而确定 `p` 是一个包含5个 `int *` 类型指针的数组；而 `int (*p)[5]` 首先解析 `*`，因此 `p` 是一个指针，随后解析 `[5]`，因此指针指向的是一个包含5个元素的数组，最后解析 `int` 得到基本元素类型，从而确定 `p` 是一个指向包含5个 `int` 类型元素的数组的指针。（有点绕，最好多看几遍想一想）

> 事实上，C语言中的下标运算都是通过指针运算实现的。在编译时，`a[i]` 会被转换为 `*(a+i)`，因此下标运算中的下标可以是负数。但是**使用时务必注意是否会出现越界访问的问题，因为C语言中不会对数组的下标进行越界检查！**

细看指针与数组

以二维数组`int a[3][4]`为例：

- `a`是`int (*)[4]`类型的指针，指向包含4个`int`类型元素的数组；
- `a[0]`是`int *`类型的指针，指向包含4个`int`类型元素的数组的第一个元素；
- `a[0][0]`是`int`类型的变量，表示二维数组的第一个元素；

- `&a`是`int (*)[3][4]`类型的指针，指向一个包含3行4列的二维数组；
- `&a[0]`是`int (*)[4]`类型的指针，与`a`相同，指向包含4个`int`类型元素的数组；
- `&a[0][0]`是`int *`类型的指针，与`a[0]`相同，指向包含4个`int`类型元素的数组的第一个元素；

- `a+1`是`int (*)[4]`类型的指针，指向二维数组的第二行，与`a[1]`相同，指向`a[1][0]`元素所在地址；（即对`a`加1，指针移动一整行）
- `a[0]+1`是`int *`类型的指针，指向二维数组的第一行的第二个元素；（即对`a[0]`加1，指针移动一个元素）

- `*a`是`int *`类型，是二维数组的第一行，可以看作一个一维数组，指向第一行的第一个元素；
- `*a+1`是`int *`类型，指向第一行的第二个元素；
- `*(a+1)`是`int *`类型，指向二维数组的第二行的第一个元素；（注意和上一条优先级的区别）
- `**a`是`int`类型，是二维数组的第一个元素，即第一行的第一个元素；
- `**a+1`是`int`类型，是第一行的第一个元素加1；
- `**(a+1)`是`int`类型，是第二行的第一个元素。
- `*(*(a+1)+1)`是`int`类型，是第二行的第二个元素。
- ...

通过上面的例子，可以更好地理解数组作为参数时形参的定义与使用

动态分配内存

程序运行时，有时需要用到动态长度的内存空间，之前提到，数组的长度需要是一个常量表达式，不能是变量。因此，需要通过动态分配内存的方式来实现动态长度的内存空间。另外，出于安全问题，指针不应该指向未初始化的内存空间，除了可以让指针指向已经定义的变量的内存空

间外，也可以通过动态分配内存的方式，让指针指向一块分配的内存空间，这样就能有效避免指针操作未分配内存空间的问题。

分配内存空间

在C语言中，可以通过`malloc`函数动态分配内存空间，`malloc`函数的原型为：

```
void *malloc(size_t size);
```

`malloc`函数接收一个参数`size`，表示需要分配的内存空间的大小，返回一个`void *`类型的指针，指向分配的内存空间的首地址。`malloc`函数分配的内存空间是连续的，可以通过指针变量访问内存空间中的数据。

> **例如：可以用如下代码分配长为`n`的一维数组和`n*n`的二维数组**
>
> ```
> // 分配一维数组
> int *a=(int *)malloc(n*sizeof(int));
> // 分配二维数组
> int **b=(int **)malloc(n*sizeof(int *));
> for(int i=0; i<n; i++)
>     b[i]=(int *)malloc(n*sizeof(int));
> ```

C语言中函数`calloc`函数也可以动态分配空间，`calloc`函数的原型为：

```
calloc(size_t n, size_t size);
```

比起`malloc`函数，`calloc`会将分配的空间全部初始化为参数`n`。

内存空间释放

在使用动态分配的内存空间后，需要通过`free`函数释放内存空间，`free`函数的原型为：

```
void free(void *ptr);
```

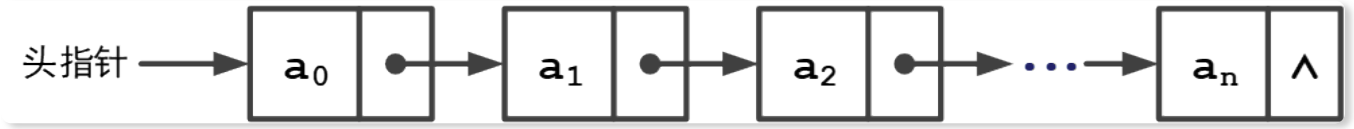
在释放内存空间后，指针变量指向的内存空间将不再可用，应该及时指针变量置空，避免野指针的出现。

链表

链表是一种常见的数据结构，用于存储数据。链表中的每个元素称为节点，节点中至少包含数据和指向下一个节点的指针。由于每个节点需要存储多种类型的数据，因此链表中的节点一般使用结构体来定义。链表的节点之间通过指针相连，形成一个链式结构。在访问链表时，需要从头节点开始，通过指针依次访问下一个节点，直到访问到目的节点。

由于链表的节点是通过指针相连的，因此链表节点的地址空间不需要连续，可以动态分配内存空间，因此链表在插入、删除节点操作时，不需要像数组一样移动其后的所有元素，比起数组更加灵活，同时也可以实现动态长度的存储。

链表的基本结构如下：



每个方框表示一个节点，a_i表示节点`i`的数据，箭头表示指向下一个节点的指针，`^`表示指向`NULL`的空指针，表示链表的结束。

链表的定义

链表的定义包括链表的节点结构和链表的头指针。

- 链表的节点结构：链表的节点结构一般使用结构体定义，结构体中包含数据和指向下一个节点的指针。

```
typedef struct Node{
    int data;
    struct Node *next;
} Node;
```

- 链表的头指针：链表的头指针指向链表的第一个节点，通过头指针可以访问整个链表。

```
Node *head;
```

链表的创建

头插法

头插法是指在链表的头部插入新节点，新节点成为链表的第一个节点。头插法的步骤如下：

1. 创建一个新节点；
2. 将新节点的指针指向原头节点；
3. 将头指针指向新节点。
4. 重复1-3，直到所有节点插入完毕。
5. 返回头指针。

示例代码如下：

```
Node *CreateList(){
    Node *head=NULL, *p;
    int num;
    while(scanf("%d", &num)!=EOF){
        p=(Node *)malloc(sizeof(Node));
        p->data=num;
        p->next=head;
        head=p;
    }
    return head;
}
```

> 头插法流程中可以看出，后加入的节点距离头节点更近，因此头插法得到的链表在访问时是逆序的。

尾插法

尾插法是指在链表的尾部插入新节点，新节点成为链表的最后一个节点。尾插法的步骤如下：

1. 创建一个新节点；
2. 将新节点的指针指向`NULL`；
3. 将原尾节点的指针指向新节点；
4. 将新节点作为尾节点。
5. 重复1-4，直到所有节点插入完毕。
6. 返回头指针。

示例代码如下：

```
Node *CreateList(){
    Node *head=NULL, *p, *tail;
    int num;
    while(scanf("%d", &num)!=EOF){
        p=(Node *)malloc(sizeof(Node));
        p->data=num;
        p->next=NULL;
        if(head==NULL)
            head=p;
        else
            tail->next=p;
        tail=p;
    }
    return head;
}
```

> 尾插法得到的链表在访问时是顺序的。

链表的遍历

链表的遍历是指从头节点开始，依次访问链表中的每个节点。链表的遍历可以通过指针变量依次访问链表中的每个节点，直到访问到链表的结束。

例如，遍历链表并输出链表中的数据：

```
void PrintList(Node *head){
    Node *p=head;
    while(p!=NULL){
        printf("%d ", p->data);
        p=p->next;
    }
}
```

链表的插入

链表的插入需要考虑几种可能的情况：

- 在链表的头部插入新节点；
- 在两个节点之间插入新节点；
- 在链表的尾部插入新节点（操作中等价于在最后一个节点和`NULL`之间插入新节点，可以与上一种情况合并）。

例如，在一个升序排列的链表中插入一个新节点：

```
Node *InsertList(Node *head, Node *newNode){
    Node *p=head;
    // 在空链表中插入新节点
    if(head==NULL){
        head=newNode;
        newNode->next=NULL;
        return head;
    }
    // 在头部插入新节点
    if(newNode->data<head->data){
        newNode->next=head;
        head=newNode;
        return head;
    }
    // 在中间或末尾插入新节点
    while(p->next!=NULL && p->next->data<newNode->data)
        p=p->next;
    newNode->next=p->next;
    p->next=newNode;
    return head;
}
```

链表的删除

链表的删除也需要考虑几种可能的情况：

- 删除链表的头节点；
- 删除链表的中间节点；
- 删除链表的尾节点。

例如，删除链表中一个值为`num`的节点：

```
Node *DeleteList(Node *head, int num){
    Node *p=head, *q;
    // 删除头节点
    if(head->data==num){
        head=head->next;
        free(p);
        return head;
    }
    // 删除中间或尾节点
    while(p->next!=NULL && p->next->data!=num)
        p=p->next;
    if(p->next==NULL) // 未找到值为num的节点
        return head;
    q=p->next;
    p->next=q->next;
    free(q);
    return head;
}
```

链表的销毁

链表的销毁是指释放链表中的所有节点，释放链表占用的内存空间。链表的销毁需要依次释放链表中的每个节点，直到链表的结束。

```
void DestroyList(Node *head){
    Node *p=head, *q;
    while(p!=NULL){
        q=p;
        p=p->next;
        free(q);
    }
}
```

链表的拓展

- 循环链表：循环链表尾节点的下一节点指针指向头节点，形成一个环形结构。循环链表的遍历和普通链表类似，只是遍历时需要判断是否到达头节点。

- 双向链表：双向链表的每个节点有两个指针，分别指向前一个节点和后一个节点。双向链表的插入和删除操作比起单向链表都更加方便，因为双向链表可以直接访问前一个节点，而不需要从头节点开始遍历。
- 十字链表：链表是一种一维的数据结构，正如两层一维数组可以构成二维数组一样，链表也可以组成更复杂的数据结构。十字链表是一种用于存储稀疏矩阵的链表，通过行链表和列链表存储矩阵的非零元素，可以有效节省存储空间。
- 邻接表：邻接表是一种用于存储图的链表，通过顶点链表和边链表存储图的顶点和边，可以有效存储图的信息。
- ...

作业与实验常见问题

数据范围

在编写程序时，需要注意数据类型的取值范围，避免数据溢出。

数据量范围

除了要注意数据类型的取值范围外，还需要注意数据量的范围。在编写程序时，需要考虑到数据量的范围，在定义变量尤其是数组时，需要考虑到数据量的范围，避免数组越界等问题。

> 例如，当需要处理的数据量为 `1000` 时，定义数组时应该至少定义为 `int a[1000]`，而非 `int a[100]`，同时注意数组 `a` 下标的范围只有 `0-999`，而访问 `a[1000]` 是越界的！

表达式值的类型

C语言中，表达式的值的类型是由表达式中的操作数的类型决定的，当表达式中有多种数据类型，且数据类型间可计算的，结果类型为操作数中范围最大的类型。

例如，`int a=3, b=2;`，则 `a/b` 的值是 `1`，而不是 `1.5`，因为 `a` 和 `b` 都是 `int` 类型，`a/b` 的值也是 `int` 类型，即整数除法，结果取整。

如果 `int a=3; double b=2;`，则 `a/b` 的值就是 `1.5`，因为 `a` 和 `b` 的类型不同，`a` 是 `int` 类型，`b` 是 `double` 类型，`a/b` 的值是 `double` 类型，即浮点数除法。

字面常量如果不是小数，则默认为 `int` 类型，如果是小数，默认为 `double` 类型。例如 `3/2` 的值

是`1`，而`3.0/2`的值是`1.5`。

定义数组时，数组长度不可为变量

C语言中，定义数组时，数组的长度必须是一个常量表达式，不能是变量。尤其！！不能！！是！！没有赋值的变量！！！

例如，有同学不慎写出如下代码：

```
int main(){
    int n;
    char str1[n], str2[n];
    scanf("%d", &n);
    scanf("%s", str1);
    scanf("%s", str2);
    printf("%s\n", str1);
    printf("%s\n", str2);
}
```

输入如下：

```
5
hello
world
```

输出如下：

```
world
world
```

这段代码中，`n`在`scanf()`语句中被赋值的，而在定义数组时，`n`是未赋值的，因此`str1`和`str2`的长度是不确定的，可能是`0`，也可能是其他值，因此在输入时，`str1`和`str2`的长度是不确定的，可能会出现越界访问的问题。

在本例中，编译器将未赋初值的`n`默认初始化为`0`，所以`str1`和`str2`的长度为`0`，长度为0的数组并不会分配空间，所以`str1`和`str2`的起始地址相同，`scanf()`函数分别读入`str1`和`str2`的内容后，实际上是向相同的地址先后写入了`hello`和`world`，`world`覆盖

了`hello`，因此输出时`str1`和`str2`的内容都是`world`。

同样的问题还有如下代码：

```
int main(){
    int n;
    int A[n][n];
    printf("%d %d %d %d",A,A+1,*(A+1),(*(A+1)+1));
}
```

输出：

```
6487440 6487440 6487440 6487444
```

这里可以看出，由于列数为0，所以`A[0]`和`A[1]`地址相同，所以读入二维数组会导致只有最后一行的数据能够被正确读入，而之前的行都会被覆盖。

这些都是定义数组时数组长度为变量导致的，因此在定义数组时，数组的长度尽量使用常量表达式，避免使用变量，尤其不能是未赋值的变量。

字符串结尾的`\0`

C语言中，字符串是以`\0`结尾的字符数组，因此在定义字符串时，需要保证字符串的长度至少比字符串长度多1，以容纳字符串结尾的`\0`。例如，当需要存储字符串`hello`时，需要至少定义`char str[6]`，而非`char str[5]`，否则会产生数组越界的问题。在手动处理字符串时，也需要注意结尾是否有`\0`和位置是否正确，避免字符串处理函数出现问题。

指针的赋值

众所周知，变量可以在初始化时进行赋值，也可以在后续使用中进行赋值。例如`int a=5;`和`int a; a=5`是等价的。但是使用指针变量时需要注意，虽然看起来都是`*p=5`，但是`p`的初始化和赋值是不同的。例如：

```
int *p=5; // p指向地址0x00000005
int *q=new int; *q=5; // q指向的地址存放5
```

事实上`int *p`和`int* p`是等价的，后者这个写法也许能帮助大家更好地理解这里的不同：`int* p=5`定义一个`int*`类型的变量`p`并将它赋值为`5`；`*p=5`则是将`p`指向的内存空间的值赋为`5`。这样就很直观了。

野指针