浙江大学 计算机科学与技术学院

COLLEGE OF COMPUTER SCIENCE AND TECHNOLOGY

ZHEJIANG UNIVERSITY

《程序设计专题》朋辈辅学

week 1 期中复习

章越

课程安排

- 期中复习
  - 指针进阶
  - 链表
  - 模块化程序设计
  - 文件进阶
  - libgraphics和simpleGUI库
- 习题答疑
- 大程指导
- 适当拓展

指针进阶

数据类型

众所周知，C语言主要包含以下类型

- 基本数据类型
  - 整型：short, int, unsigned int...
  - 浮点型：float, double
  - 字符型：char
  - 空类型：void(不能用于定义变量)
- 构造数据类型
  - 数组
  - 结构：struct
  - 联合：union
  - 枚举：enum
- 指针类型

结构和联合

结构体分配足够的空间存储其所有成员，而共用体分配空间仅存储最大的成员。

```
1 struct {
2    int a;  // 4字节
3    int b;  // 4字节
4 } s1; // s1占用8个字节
5
6 union {
7    int a;
8    int b;
9 } u1; // u1占用4个字节
```

结构和联合

结构体分配足够的空间存储其所有成员，而共用体分配空间仅存储最大的成员。

```
struct {
   int a;  // 4字节
   int b;  // 4字节
} s1; // s1占用8个字节

union {
   int a;
   int b;
} u1; // u1占用4个字节
```

结构和联合

结构体分配足够的空间存储其所有成员，而共用体分配空间仅存储最大的成员。

```
1 struct {
2    int a;   // 4字节
3    int b;   // 4字节
4 } s1; // s1占用8个字节
5
6 union {
7    int a;
8    int b;
9 } u1; // u1占用4个字节
```

共同体变量中起作用的成员是最后一次存放的成员，在存入新的成员后原有的成员失去了作用

结构和联合

字节对齐：

占用空间大小能被其包含的所有基本数据类型的大小所整除

```
struct {
   int a;  // 4字节
   char b; // 1字节
} s1; // s1占用8个字节
union S {
   char a; // 1字节
   int b[4]; // 4*4=16字节
   double c; // 8字节
} x;  // x占用16个字节
```

union大小必须满足：

1. 大小足够容纳最宽的成员；
2. 大小能被其包含的所有基本数据类型的大小所整除。

结构和联合

字节对齐：

占用空间大小能被其包含的所有基本数据类型的大小所整除

```
struct {
   int a;  // 4字节
   char b; // 1字节
} s1; // s1占用8个字节
union S {
   char a; // 1字节
   int b[4]; // 4*4=16字节
   double c; // 8字节
} x;  // x占用16个字节
```

union大小必须满足：

1. 大小足够容纳最宽的成员；
2. 大小能被其包含的所有基本数据类型的大小所整除。

结构和联合

一个小练习：

```
union U {
    char s[9];
    int n;
    double d;
} x;

sizeof(x) == ?
```

结构和联合

一个小练习：

```
union U {
    char s[9];
    int n;
    double d;
} x;

sizeof(x) == ?
```

```
16
```

枚举

语法：

```
enum 枚举名 { 枚举值1, 枚举值2, … };
```

e.g.

```
enum week{ Mon = 1, Tues = 2, Wed = 3, Thurs = 4, Fri = 5,
  Sat = 6, Sun = 7 } day;

day = Mon;
```

注意：对于没有赋值的元素，其值为前一元素加1 初值为0

e.g.

```
enum Test {x, y = 4, z};  // x==0, z==5
```

指针基础回顾

定义

```
类型名 *指针变量名
```

e.g.

```
1 int *a;
2 int x;
3 a = &x;
4
5 *a = 10;  // equal to x = 10;
6
7 int *b;
8 b = (int *)malloc(sizeof(int));
9 free(b);
```

指针基础回顾

定义

```
类型名 *指针变量名
```

e.g.

```
1 int *a;
2 int x;
3 a = &x;
4
5 *a = 10;  // equal to x = 10;
6
7 int *b;
8 b = (int *)malloc(sizeof(int));
9 free(b);
```

内存模型

- Stack
  - 存储调用的函数、自动变量等
  - 返回时会自动返还空间
- Heap
  - 用户手动申请
  - 需手动释放，否则会造成内存泄漏
- Globals
  - 全局变量和静态变量
- Constants
  - 常量
- Code
  - 源代码

动态内存分配

1. malloc

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

在内存的Heap中分配连续空间，长度为size，返回一个指向该空间起始地址的指针

若申请失败，则返回NULL

e.g. 分配n个整数大小的空间

```
int *p = (int *)malloc(n * sizeof(int));
if (p == NULL) {
  printf("malloc failed\n");
  exit(1);
}
```

动态内存分配

2. calloc

```
void *calloc(unsigned n, unsigned size);
```

在内存的Heap中分配连续空间，长度为n*size，返回一个指向该空间起始地址的指针，并把该空间全部初始化为0

若申请失败，则返回NULL

e.g. 分配n个整数大小的空间

```
int *p = (int *)calloc(n, sizeof(int));
if (p == NULL) {
  printf("malloc failed\n");
  exit(1);
}
```

动态内存分配

3. realloc

```
void *realloc(void *ptr, unsigned size);
```

用于调整ptr指针指向的空间大小。在内存的Heap中重新分配连续空间，长度为size，返回一个指向该空间起始地址的指针。并保证该块内容与原块一致。

若申请失败，则返回NULL，原来ptr指向的内容不变。

e.g. 重新分配n+1个整数大小的空间

```
p = (int *)realloc(p, (n+1) * sizeof(int));
if (p == NULL) {
  printf("realloc failed\n");
  exit(1);
}
```

动态内存分配

4. free

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

用于释放ptr指针指向的空间，ptr指向该空间的首地址。

当某个内存不再使用时，切记要及时释放，否则会产生内存泄漏。

e.g.

```
free(p);
p = NULL;
```

动态内存分配

4. free

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

用于释放ptr指针指向的空间，ptr指向该空间的首地址。

当某个内存不再使用时，切记要及时释放，否则会产生内存泄漏。

e.g.

```
free(p);
p = NULL;
```

普通指针

函数的形参只传递值，是单向传递

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

可用指针作为函数参数来实现参数的双向传递

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

普通指针

调用这个函数会发生什么？

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

普通指针

函数返回指针

```
1  int *f() {
2    int a = 10;
3    return &a;
4  }
5
6  int *g() {
7    int *p = (int *)malloc(sizeof(int));
8    *p = 10;
9    return p;
10 }
11
12 int *h() {
13   static int d[10];
14   return d;
15 }
```

普通指针

函数返回指针

```
 1 int *f() {
 2   int a = 10;
 3   return &a;
 4 }
 5
 6 int *g() {
 7   int *p = (int *)malloc(sizeof(int));
 8   *p = 10;
 9   return p;
10 }
11
12 int *h() {
13   static int d[10];
14   return d;
15 }
```

普通指针

函数返回指针

```
1  int *f() {
2    int a = 10;
3    return &a;
4  }
5
6  int *g() {
7    int *p = (int *)malloc(sizeof(int));
8    *p = 10;
9    return p;
10 }
11
12 int *h() {
13   static int d[10];
14   return d;
15 }
```

字符串指针

两个字符串直接比较时，比较的是首地址

```
char *s1 = (char *)malloc(10 * sizeof(char));
char *s2 = (char *)malloc(10 * sizeof(char));
s1[0] = s2[0] = 'a';
s1[1] = s2[1] = '\0';
if (s1 == s2) {
    printf("equal\n");
} else {
    printf("not equal\n");
}
```

输出结果?

字符串指针

两个字符串直接比较时，比较的是首地址

```
char *s1 = (char *)malloc(10 * sizeof(char));
char *s2 = (char *)malloc(10 * sizeof(char));
s1[0] = s2[0] = 'a';
s1[1] = s2[1] = '\0';
if (s1 == s2) {
    printf("equal\n");
} else {
    printf("not equal\n");
}
```

输出结果?

```
"not equal"
```

字符串指针

一些注意点

1．指针使用前需要先分配内存空间

```
char *color[5];
for (int i = 0; i < 5; i++)
    scanf("%s", color[i]);
```

段错误!

2．初始化时赋值的字符串为常量字符串，不允许修改

```
char *str = "hello";
str[0] = 'H';
```

这种写法不合法！会导致段错误

字符串指针

一些注意点

1．指针使用前需要先分配内存空间

```
char *color[5];
for (int i = 0; i < 5; i++)
    scanf("%s", color[i]);
```

段错误!

2．初始化时赋值的字符串为常量字符串，不允许修改

```
char *str = "hello";
str[0] = 'H';
```

这种写法不合法！会导致段错误

字符串指针

一些注意点

1．指针使用前需要先分配内存空间

```
char *color[5];
for (int i = 0; i < 5; i++)
    scanf("%s", color[i]);
```

段错误!

2．初始化时赋值的字符串为常量字符串，不允许修改

```
char *str = "hello";
str[0] = 'H';
```

这种写法不合法！会导致段错误

字符串指针

字符串处理相关函数(string.h库)

函数名	功能
int strcmp(const char *str1, const char *str2)	如果返回值小于 0，则表示 str1 小于 str2。
char *strcat(char *dest, const char *src)	把 src 所指向的字符串追加到 dest 所指向的字符串的结尾（dest会被修改）。返回一个指向最终的目标字符串 dest 的指针
char *strcpy(char *dest, const char *src)	把 src 所指向的字符串复制到 dest（dest会被修改），返回一个指向最终的目标字符串 dest 的指针。

数组与指针

数组名是一个常量指针，指向数组的首元素

```
int a[10] = {0};
int *p = a;
for (int i = 0; i < 10; i++) {
  printf("%d ", *(p+i));  // equal to a[i]
}

int b[10] = {1};
a = b;  // error 常量指针不可修改
```

以下代码有什么问题?

```
char color[ ][7] = {"red", "blue", "yellow", "green", "black"};
char *tmp = color[0];
color[0] = color[4];
color[4] = tmp;
```

数组与指针

那如果我们把代码改成这样呢？

```
1  char* pcolor[] = {"red", "blue", "yellow", "green", "black"};
2  // 以下代码可行吗?
3  char *tmp = pcolor[0];
4  pcolor[0] = pcolor[4];
5  pcolor[4] = tmp;
6
7  // 以下代码呢?
8  char tmp = pcolor[0][0];
9  pcolor[0][0] = pcolor[4][0];
10 pcolor[4][0] = tmp;
```

数组与指针

那如果我们把代码改成这样呢？

```
1  char* pcolor[] = {"red", "blue", "yellow", "green", "black"};
2  // 以下代码可行吗？
3  char *tmp = pcolor[0];
4  pcolor[0] = pcolor[4];
5  pcolor[4] = tmp;
6
7  // 以下代码呢？
8  char tmp = pcolor[0][0];
9  pcolor[0][0] = pcolor[4][0];
10 pcolor[4][0] = tmp;
```

指针数组

数组的各个元素都是指针类型，那这个数组就是指针数组

```
int *a[5];
for (int i = 0; i < 5; i++) {
  a[i] = (int *)malloc(sizeof(int));
  scanf("%d", a[i]);
}
```

以下两个指针数组有什么区别？

```
const char *color[5] = {"red", "green", "blue", "yellow",
  "white"};

char * const color[5] = {"red", "green", "blue", "yellow",
  "white"};
```

指针数组

练习题：对于以下程序，能够正确表示二维数组 t 的元素地址的表达式是()

```
int main(void) {
    int k, t[3][2], *pt[3];
    for ( k = 0; k < 3; k++) {
        pt[k] = t[k];
    }
    return 0;
}
```

A. &t[3][2]
B. *pt[0]
C. *(pt+1)
D. &pt[2]

指针数组

练习题：对于以下程序，能够正确表示二维数组 t 的元素地址的表达式是()

```
int main(void) {
    int k, t[3][2], *pt[3];
    for ( k = 0; k < 3; k++) {
        pt[k] = t[k];
    }
    return 0;
}
```

A. &t[3][2]
B. *pt[0]
C. *(pt+1)
D. &pt[2]

Ans: C

数组指针

指针数组是一个以指针为元素的数组(指针的数组)

数组指针是一个指向数组的指针(数组的指针)

注意: []运算的优先级高于*

```
char *color1[5];  // 指针数组

char (*color2)[5];  // 数组指针
```

二级指针

二级指针即指针的指针

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

二级指针作为函数参数

```
void f(int **p) {
  *p = (int *)malloc(sizeof(int));
  **p = 10;
}
```

小练习：输入指向数组的指针和数组大小，完成数组的动态创建

```
void ArrayCreate(double **array, int size);
```

二级指针

我们来看以下两种回答是否正确：

```
1 void ArrayCreate(double **array, int size) {
2     array = (double **)malloc(sizeof(double *) * size);
3 
4     *array = (double *)malloc(sizeof(double) * size);
5 }
```

二级指针

我们来看以下两种回答是否正确：

```
1 void ArrayCreate(double **array, int size) {
2     array = (double **)malloc(sizeof(double *) * size);
3
4     *array = (double *)malloc(sizeof(double) * size);
5 }
```

二级指针

我们来看以下两种回答是否正确：

```
1 void ArrayCreate(double **array, int size) {
2     array = (double **)malloc(sizeof(double *) * size);
3 
4     *array = (double *)malloc(sizeof(double) * size);
5 }
```

二级指针

我们来看以下两种回答是否正确：

```
1 void ArrayCreate(double **array, int size) {
2     array = (double **)malloc(sizeof(double *) * size);
3 
4     *array = (double *)malloc(sizeof(double) * size);
5 }
```

指针形参实参匹配

二维数组名不是二级指针，而是一个数组指针

即二维数组的每个元素其实是一个一维数组

```
void f(int (*a)[10]);
void g(int **a);

int a[10][10];  // 相当于 int (*a)[10];
f(a); // 实参和形参相互匹配
g(a); // 实参和形参不匹配
```

指针形参实参匹配

形参中只能忽略第一维，忽略后面的维度会导致编译器无法正确寻址

```
void f(int a[][10]);  // ok
void f(int a[10][]);  // error
void f(int a[][]);  // error
```

什么叫无法正确寻址？

```
double a[m][n];
a[i][j]: a + i * n * sizeof(double) + j * sizeof(double)
```

指针形参实参匹配

练习题：

下列哪些实参（左边）和形参（右边）能够匹配的是（多选）。

```
A. double a[10][10]    double **b
B. double a[10][10]    double b[][]
C. double a[10][10]    double b[10][]
D. double a[10][10]    double b[][10]
E. double a[10][10]    double (*b)[10]
F. double *a[10]       double **b
G. double (*a)[10]     double **b
H. double **a          double *b[]
I. double **a          double b[][]
```

指针形参实参匹配

练习题：

下列哪些实参（左边）和形参（右边）能够匹配的是（多选）。

```
A. double a[10][10]    double **b
B. double a[10][10]    double b[][]
C. double a[10][10]    double b[10][]
D. double a[10][10]    double b[][10]
E. double a[10][10]    double (*b)[10]
F. double *a[10]       double **b
G. double (*a)[10]     double **b
H. double **a          double *b[]
I. double **a          double b[][]
```

```
Ans: DEFH
```