WEEK 7

方便的操作字符串

c语言标准库为我们提供了一系列的函数，用于操作字符串和内存，它们被声明在`string.h`中。本章所有的示例代码均在操作系统Ubuntu 22.04和CPU13700KF,gcc版本11.4.0编译运行。

比较

1. `int strcmp(const char *lhs, const char *rhs)`

   用于比较两个字符串的字典序。

   - 当`lhs`小于`rhs`时，返回负值。
   - 当`lhs`大于`rhs`时，返回正值。
   - 当`lhs`和`rhs`相等时，返回0。

   当`lhs`或`rhs`为`NULL`时，该函数的行为是未定义的。

   了解更多可以浏览strcmp - cppreference.com。

2. `int strncmp(const char* lhs, const char* rhs, size_t count)`

   和`strcmp`不同，`strncmp`最多比较`count`个字符，当比较到其中一个字符串结尾的时候，它也会停止比较。返回值和`strcmp`相同。

   了解更多可以浏览strncmp - cppreference.com。

3. `int memcmp(const void* lhs, const void* rhs, size_t count)`

   和`strncmp`类似，`memcmp`会比较内存中的`count`个字节，不同的是，它不会在`'\0'`处停止比较。返回值和`strcmp`相同。

   了解更多可以浏览memcmp - cppreference.com。

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

typedef struct
{
    int v0;
    int v1;
} Object;

const char *relation(int r) {
    if (r == 0) return "equal";
    if (r < 0) return "less";
    if (r > 0) return "greater";
}

int main() {
    const char *string = "Missing Semester";
    printf("%s\n", relation(strcmp(string, "Missing Semester"))); // equal
```

```c
    printf("%s\n", relation(strcmp(string, "Missing Files")));    // greater
    printf("%s\n", relation(strcmp(string, "Missing files")));    // less
    printf("%s\n", relation(strncmp(string, "Miss her", 4)));     // equal
    printf("%s\n", relation(memcmp(&(Object){1, 2}, &(Object){1, 3},
sizeof(Object)))); // less

    int v0 = 0x12345678, v1 = 0x78563412;
    printf("%s\n", relation(memcmp(&v0, &v1, sizeof(int)))); // greater WHY?

    int arr0[4] = {0x12, 0x34, 0x56, 0x78};
    int arr1[4] = {0x12, 0x34, 0x57, 0x00};
    printf("%s\n", relation(memcmp(arr0, arr1, sizeof(arr0)))); // less
    return 0;
}
```

> **奇怪的输出**
>
> 第26行中，明明v0在数值上小于v1，为什么`memcmp`比较的结果是v0大于v1？在其它平台上编译运行，结果是否会不一样？尝试用数据存储的知识解释。

字符串的长度

`size_t strlen(const char* str );`

用于求字符串的长度。了解更多可以浏览strlen, strnlen_s - cppreference.com

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

int main() {
    const char *string = "Missing Semester";
    printf("%ld\n", strlen(string)); // 16
    printf("%ld\n", strlen(string + 8)); // 8
    return 0;
}
```

复制和填充

1. `char *strcpy(char *restrict dest, const char *restrict src)`

   将`src`中的字符拷贝到`dst`中，**这个函数在`dest`溢出的时候的行为是未定义的，它不会告诉你`dest`溢出了！**它返回`dest`的。了解更多可以浏览memmove, memmove_s - cppreference.com。

2. `char *strncpy(char *restrict dest, const char *restrict src, size_t count)`

   和`strcpy`类似，但是它最多只复制`count`个字符，且不会在末尾加上`'\0'`。了解更多可以浏览strncpy, strncpy_s - cppreference.com。

3. `void* memcpy(void *restrict dest, const void *restrict src, size_t count)`

   `void* memmove(void* dest, const void* src, size_t count);`

将 `src` 中 `count` 个字节拷贝到 `dest` 中。**`memcpy` 假定 `src` 和 `dest` 两块区域没用重叠，而 `memmove` 则不作这个假定**。了解更多可以浏览memcpy, memcpy_s - cppreference.com memmove, memmove_s - cppreference.com。

4. `void *memset(void *dest, int ch, size_t count);`

用 `(unsigned char)ch`（只取最低8位）填充 `dest` 中的 `count` 个字节。了解更多可以浏览memset, memset_explicit, memset_s - cppreference.com。

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

int main() {
  const char *string = "Missing Semester";
  char buffer[20];
  strcpy(buffer, string);
  printf("%s\n", buffer); // Missing Semester

  memset(buffer, 'a', 20);
  buffer[19] = 0;
  strncpy(buffer, string, 7);
  printf("%s\n", buffer); // Missingaaaaaaaaaaaa

  int array[4] = {0x12, 0x34, 0x56, 0x78};
  int *b = malloc(sizeof(array));
  memcpy(b, array, sizeof(array));
  printf("%d\n", memcmp(array, b, sizeof(array)) == 0); // 1
  free(b);
  return 0;
}
```

查找和匹配

1. `void* memchr(const void* ptr, int ch, size_t count)`

   返回在 `ptr` 中第一次出现 `(unsigned char)ch` 的位置（指针），如果没有找到，则返回 `NULL`。

2. `char* strchr(const char* str, int ch)`

   返回在 `str` 中第一次出现 `(char)ch` 的位置（指针），如果没有找到，则返回 `NULL`。

3. `char* strrchr(const char* str, int ch);`

   返回在 `str` 中最后一次出现 `(char)ch` 的位置（指针），如果没有找到，则返回 `NULL`。

4. `size_t strspn(const char* dest, const char* src)`

   返回 `dest` 中第一个不在 `src` 中的字符的索引。

5. `size_t strcspn(const char *dest, const char *src)`

   返回 `dest` 中第一个在 `src` 中的字符的索引。

6. `char *strstr(const char *str, const char *substr)`

   返回 `str` 中 `substr` 第一次出现的位置。

7. `char *strtok(char *str, const char *delim);`

   将`str`分解为一组小的字符串。**这个函数会更改`str`！**下面是一个示例。

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

int main() {
    char string[48] = "I am studying the Missing Semester";
    char *tok = strtok(string, " ");
    while (tok != NULL) {
        printf("%s\n", tok);
        tok = strtok(NULL, " ");
    }
    /*
    I
    am
    studying
    the
    Missing
    Semester
    */
    return 0;
}
```

拼接

1. `char *strcat(char *restrict dest, const char *restrict src)`

   将`src`拼接到`dest`的末尾。返回`dest`。

2. `char *strncat(char *restrict dest, const char *restrict src, size_t count)`

   将`src`中最多`count`个字符拼接到`dest`末尾。返回`dest`。

> **什么是`size_t`**
>
> 在标准中，`size_t`被规定为操作符`sizeof`返回值的类型。

用什么类型作为标签呢？——枚举类型

定义枚举类型

枚举是 C 语言中的一种基本数据类型，用于定义一组具有离散值的常量，它可以让数据更简洁，更易读。枚举类型通常用于为程序中的一组相关的常量取名字，以便于程序的可读性和维护性。

我们可以用宏来记录星期和数字的对应关系：

```
#define MON  1
#define TUE  2
#define WED  3
#define THU  4
#define FRI  5
#define SAT  6
#define SUN  7
```

也可以声明一个枚举类型来表示：

```
enum day
{
      MON, TUE, WED, THU, FRI, SAT, SUN
};
```

这个枚举类型定义了常量`MON=0`，`TUE=1`，`WED=2`……枚举类型被当做`int`来处理。

我们也可以指定枚举中的值等于什么：

```
enum color
{
     RED, ORANGE=2, YELLOW, GREEN
};
```

`ORANGE`及其以后得值将从2开始编号，也就是说`YELLOW=3`，`GREEN=4`。

使用枚举类型

```
#include <stdio.h>

enum color {RED=1, GREEN, BLUE};
int main()
{
    enum color favorite_color;
    printf("Input your favorite color: (1. red, 2. green, 3. blue): ");
    scanf("%u", &favorite_color);
    switch (favorite_color)
    {
    case RED:
        printf("You like red best.");
        break;
    case GREEN:
        printf("You like green best.");
        break;
    case BLUE:
        printf("You like blue best.");
        break;
    default:
        printf("You like other colors best.");
    }
```

```
    return 0;
}
```

新的存储方式——联合体、位域

联合体

联合体的声明和使用和结构体很类似，只是联合体的所有变量都"挤"在同一片空间之中。

声明联合体

```
union Object
{
    int a[4];
    int v0;
};
```

可以用`union Object`来表示这个联合体类型。

使用联合体

联合体对类型的访问和结构体很像。

```
#include <stdio.h>

union Object
{
    int a[4];
    int v0;
};


int main() {
    union Object o;
    o.v0 = 0xff;
    printf("%d\n", o.v0); // 255
    o.a[0] = 0x12;
    printf("%d\n", o.a[0]); // 18
    return 0;
}
```

联合体的内存模型

联合体的所有成员共用了一段内存。联合体的大小取决于最大成员的大小。如`union Object`的大小为`int[4]`的大小，即16字节。

标准并没有规定union的内存模型应该是什么样的，应该如何存储数据，所以下面只给出一种可能的便于理解的存储方式。下面的读取方式实际上属于一种未定义行为！

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

```
union Object
{
    int a[4];
    int v0;
};


int main() {
    union Object o;
    o.v0 = 0xff;
    printf("%d\n", o.a[0]); // (possible) 255
    printf("%d\n", &o.v0 == &o.a[0]); // (possible)1
    return 0;
}
```

可以看到，实际上`v0`使用的空间和`a[0]`重合了，写入`v0`的时候，实际上也更改了`a[0]`的值。如图是`union Object`内存空间使用。

再次强调：这只是一种可能得内存分布，具体取决于编译器的实现！

```
┌----┬---------------┐
|v0  |               |
├----┼----┬----┬----┤
|a[0]|a[1]|a[2]|a[3]|
├----┴----┴----┴----┤
|union Object       |
└-------------------┘
```

在使用联合体时，应该在为一个成员赋值后，只使用这一个成员，直到另一个成员被赋值。

联合体内嵌套结构体

联合体内部可以嵌套结构体。在嵌套匿名结构体时，结构体的成员可以直接使用`union`类型访问。

```
union object
{
    int a[4];
    struct
    {
        int v0, v1, v2, v3;
    };
};
```

对于如上定义的联合体，声明`union object o`，我们可以直接使用`o.v0`对匿名结构体中的`v0`进行访问。

示例：使用联合体来记录不同的类型

```c
#include <stdio.h>

typedef struct object
{
    enum{CHAR, INT, DOUBLE} tag;
    union
    {
        char c;
        int i;
        double d;
    };
} Object;

void print_object(const Object o)
{
    switch(o.tag)
    {
        case CHAR: printf("%c\n", o.c); break;
        case INT: printf("%d\n", o.i); break;
        case DOUBLE: printf("%lf\n", o.d); break;
    }
}

int main()
{
    Object o = {CHAR, 'a'};
    print_object(o);
    o.tag = INT;
    o.i = 123;
    print_object(o);
    o.tag = DOUBLE;
    o.d = 1.5;
    print_object(o);
    return 0;
}
```

位域

位域

有时候，我们用`int`存储数据的时候，并不需要完整的32位，我们可以显示地在结构体中标注一个`int`、`unsinged int`所需要的空间的大小。

```c
#include <stdio.h>

struct pack
{
    unsigned int v0 : 1;
```

```
    unsigned int v1 : 2;
};

int main()
{
    struct pack p;
    p.v0 = 1;
    printf("%d\n", p.v0); // 1
    p.v0 = 2;
    printf("%d\n", p.v0); // 0
    return 0;
}
```

上面的代码声明了`v0`只占一位，`v1`只占两位。由于2需要两位来表示，v0只能取其最低位0。

位域类型不能做`sizeof`操作。

示例：用位域做拓展

下面的两个函数将用20位表示的数字`v`分别做符号拓展和无符号拓展。

```
int signed_extend(int v) {
    struct {int t: 20} t = {.t=v};
    return (int)t.t;
}

unsigned int unsigned_extend(int v) {
    struct {unsigned int t: 20} t = {.t=v};
    return (unsigned int)t.t;
}
```

> **更一般的位拓展器**
>
> 尝试使用位运算编写更加一般的`signed_extend`和`unsigned_extend`，使它们可以接受`v`的有效位数作为作为参数。

指向函数的指针

指针可以指向变量，那是否可以指向函数呢？答案是肯定的。

函数指针

函数指针类型用如下方式定义：

`函数返回值 (*函数名)(函数参数类型列表)`

比如函数`void swap(int *a, int *b)`可以用`void (*f)(int *, int *)=swap`。

我们还可以用`typedef`：

```
typedef void (*function_t)(int *, int *);
function_t f = swap;
```

我们可以用函数指针调用函数：

```
int a, b;
f(&a, &b);
```

用函数指针模拟函数的覆盖(override)

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

#define ANIMAL_HEADER char name[20]; void (*make_sound)(struct animal *this);

typedef struct animal
{
    ANIMAL_HEADER;
} Animal;

typedef struct dog
{
    ANIMAL_HEADER;
    char favorite[20];
} Dog;

typedef struct cat
{
    ANIMAL_HEADER;
    char goodAt[20];
} Cat;

static void dog_make_sound(Animal *this_) {
    Dog *this = (Dog *)this_;
    printf("I am dog %s, I like %s best.\n", this->name, this->favorite);
}

void dog_init(Dog *dog, char *name, char *favorite) {
    dog->make_sound = dog_make_sound;
    strncpy(dog->name, name, 19);
    strncpy(dog->favorite, favorite, 19);
}

static void cat_make_sound(Animal *this_) {
    Cat *this = (Cat *)this_;
    printf("I am cat %s, I am good at %s.\n", this->name, this->goodAt);
}

void cat_init(Cat *cat, char *name, char *goodAt) {
    cat->make_sound = cat_make_sound;
    strncpy(cat->name, name, 19);
    strncpy(cat->goodAt, goodAt, 19);
}

```

```c
int main()
{
    Dog dog;
    dog_init(&dog, "Bob", "meat");
    Cat cat;
    cat_init(&cat, "Tom", "catch mouse");
    Animal *animals[2] = {(Animal *)&dog, (Animal *)&cat};
    for (int i = 0; i < 2; i++) {
        animals[i]->make_sound(animals[i]);
    }
    return 0;
}
```

变参数函数

`printf`、`scanf`等函数的参数个数、类型是可变的，这是如何实现的呢?

使用指针实现变参数

我们可以将一个指针作为函数的参数，然后通过某种我们自己规定的规则解析指针指向的地址，就比如解析数组求和。

```c
#include <stdio.h>

int sum(int *p) {
    int s = 0;
    for (int i = 0; p[i] != 0; i++) {
        s += p[i];
    }
    return s;
}

int main()
{
    int a0[] = {1, 2, 3, 4, 5, 0};
    printf("%d\n", sum(a0)); // 15
    int a1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0}; // 55
    printf("%d\n", sum(a1));
    return 0;
}
```

使用标准库提供的方法

其实C语言提供了对变参数的支持。这是`printf`的原型：`int printf(const char* restrict format, ...)`其中`...`提供了对变参数的支持。下面通过一个例子来介绍。

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

int sum(int count, ...) {
```

```
    va_list args;
    va_start(args, count);
    int s = 0;
    for (int i = 0; i < count; i++) {
        s += va_arg(args, int);
    }
    va_end(args);
    return s;
}

int main()
{
    printf("%d\n", sum(3, 1, 2, 3)); // 6
    printf("%d\n", sum(2, 1, 2)); // 3
    return 0;
}
```

在处理变参数的时候，我们先定义了一个`va_list`类型的变量`args`，然后通过`va_start`来初始化，其中`va_list`的第二个参数是函数声明中`...`前的最后一个参数。然后我们就可以通过`va_arg`依次取出其中的参数。最后通过`va_end`释放空间。

> **变参数函数实现的原理**
>
> 尝试搜索标准库提供的变参数函数的原理，想想和第一种方法的相同和不同。