运算符的优先级

1. 下标 []、函数调用 ()、结构选择 .
2. 一元运算符 指针 *、自增 ++、自减 --
3. 二元运算符
   数学运算符 *、/、%、+、-
   移位运算符 <<、>> \\
   关系运算符 <、>、<=、>=
   逻辑运算符 &、^、|
   赋值运算符 =、*=、+=、&=、<<=
   条件运算符（三元运算符） ? :
4. 逗号运算符 ,

NULL、'\0'、0 之间的区别

NULL 既表示 0 也表示空指针，存在二义性。

'\0' 常作为字符串的结尾。

nullptr 是为了消除 NULL 的二义性而创造的。

初始化数组或字符串

```
// 初始化一维数组
int arr[10]{ 0 };
int arr[] = {0, 1, 2};
// 初始化二维数组
int arr[3][4] = { 0 };
int arr[3][3] = {{0,1}, {0,0,2}, {3}};
// 初始化单个字符串
char *str = "ABCD";
char str[] = "ABCD";
char str[] = {'A', 0};
// 初始化多个字符串
char *str[] = {"ABC", "DEF"};
```

修改数组或字符串

```
char arr[] = "hello";
arr[0] = 'X'; // 正确，可以修改栈区
char *ptr = "world";
ptr[0] = 'X'; // 错误，不能修改常量区
```

复制字符串

```
// 复制到栈区
char str[] = "hello";
char buf[10];                  // 开辟栈区空间
int sz = sizeof(str) / sizeof(str[0]);
//strcpy(buf, str);            // UNIX 习惯
strcpy_s(buf, sz, str);        // 不能用 b = a
strcmp(buf, str) == 0;         // 不能用 b == a
// 复制到堆区
size_t len = strlen(str);
char *p = NULL;            // NULL 存在二义性
p = (char *)malloc(sizeof(char) * (len + 1));
strcpy_s(buf, len + 1, str);// 不能忘记 + 1
if (strcmp(buf, str) == 0)  // 不能用 b == a
  free(p);                 // 释放动态开辟的空
间
p = NULL;                  // 避免出现野指针
```

动态开辟空间 malloc / free

```
// 申请和释放一维数组
int *num = (int *)malloc(sizeof(int) * 1024);
if (num != NULL)
  free(num);
// 申请和释放二维数组
char **ptr = (char**)malloc(sizeof(char*) * 1024
for (int i = 0; i < 1024; i++)
  ptr[i] = (char *)malloc(sizeof(char) * 30);
for (int i = 0; i < 1024; i++)
  if (ptr[i] != NULL)
    free(ptr[i]);
if (ptr != NULL)
  free(ptr);
```

类型转换 char chs[100]{ 0 }

```
ptr = (char *) str.c_str();// string -> char*
double d = atof("0.23");  // string -> double
int i = atoi("1021");     // string -> int
long l = atol("303992");  // string -> long
sprintf(chs, "%f", 2.3);  // 保存到 chs
char *ptr = chs;          // char[] -> char*
strcpy(chs, ptr, len);    // char* -> char[]
string str = chs;         // char[] -> string
(unsigned)num;            // 强制转换
```

哈希表 #include "uthash.h"

```
// 创建哈希结构
struct my_struct {
  int id;              // key
  char name[10];       // value
  UT_hash_handle hh;  // make it hashable
};
// 定义哈希表
struct my_struct *users = NULL;
// 向哈希表中添加数据
void add_user(struct my_struct *s) {
  HASH_ADD_INT( users, id, s );
}
// 从哈希表中检索数据
struct my_struct *find_user(int user_id) {
  struct my_struct *s;

  HASH_FIND_INT( users, &user_id, s );
  return s;
}
// 删除哈希表中的一条数据
void delete_user(struct my_struct *user) {
  HASH_DEL( users, user);
}
```