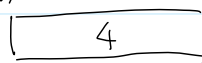


复习

2024年4月29日 9:37

0xABCD



int i = 4;

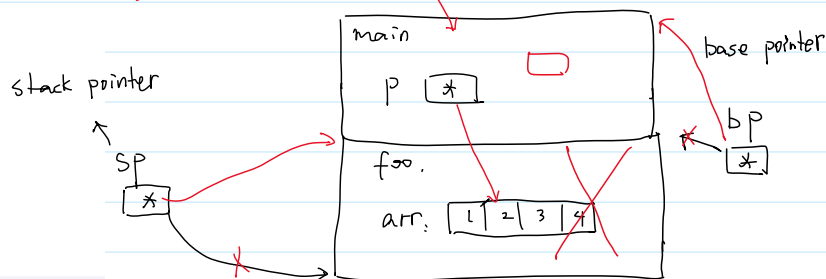
表达式:

i → 4

&i → 0xABCD

函数调用相关的信息

形参、局部变量、返回地址...



教训: 不要返回指向当前栈帧的指针!

```
int* foo(void) {  
    int arr[] = { 1, 2, 3, 4 };  
    return &arr[1];  
}
```

```
int main(void) {  
    int* p = foo();  
    printf("*p = %d\n", *p);  
    printf("*p = %d\n", *p);  
    return 0;  
}
```

Microsoft Visual Studio 调试

*p = 2
*p = -858993460

作业讲解

2024年4月29日 9:58

```
int main(void) {
    long total_sec = 9527;
    int hour, minute, second;
    // 指针可以作为返回值来用
    split_time(total_sec, &hour, &minute, &second);
    printf("%d:%d:%d\n", hour, minute, second);
    return 0;
}

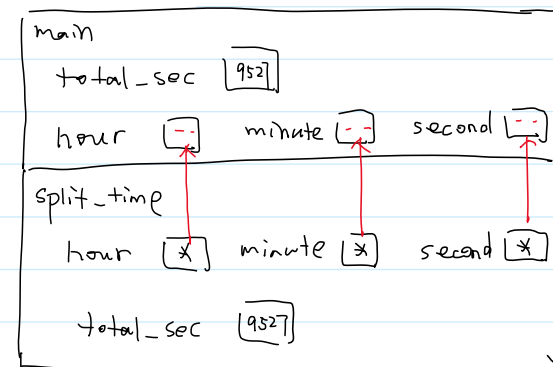
void split_time(long total_sec, int* hour, int* minute, int* second) {
    *second = total_sec % 60;
    *minute = (total_sec / 60) % 60;
    *hour = (total_sec / 60 / 60) % 24;
}
```

值传递

$total_sec \% 60$

$total_sec / 60 \% 60$

$total_sec / 60 / 60 \% 24$



指针常量和常量指针

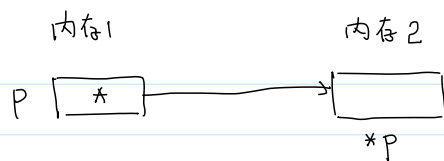
2024年4月29日 10:17

```
{ int i = 10, j = 20;  
  int* p = &i;
```

```
  *p = 10; → 写
```

```
  p = &j; → 写
```

对内存1、内存2都有写权限



```
int i = 10, j = 20;
```

```
const int* p = &i;
```

```
I  
p = &j;
```

```
// *p = 10;
```

pointer to const, 常量指针

对内存1有写权限.

内存1

内存2

P [*]

*p

(没有写权限)

constant pointer: 指针常量.

```
int i = 10, j = 20;
```

```
int* const p = &i;
```

```
I  
// p = &j;
```

```
*p = 10;
```

对内存2有写权限

内存1

内存2

P [*]

*p

(没有写权限)

```
int i = 10, j = 20;
```

```
const int* const p = &i;
```

```
// p = &j;
```

```
// *p = 10;
```

内存1

内存2

P [*]

*p

(没有写权限)

(没有写权限)

本质: 限制变量的写权限.

传入参数和传出参数

2024年4月29日 10:18

```
void foo(const int* p) {  
    // *p = 100;  
}
```

传入参数：在函数里面，不会（不能够）

通过指针变量修改它指向对象

传出参数：返回值。

在函数里面，可以通过指针变量修改它指向对象

```
void foo(int* p) {  
    *p *= 2;  
}  
  
int main(void) {  
    int i = 10;  
    foo(&i);  
    printf("i = %d\n", i);  
    return 0;  
}
```

Microsoft Visual Studio 调试器

i = 20

min max

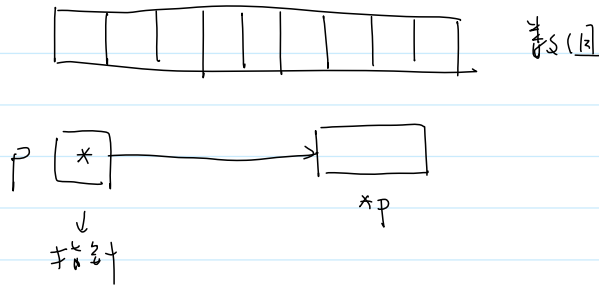
传入参数

传出参数：返回值

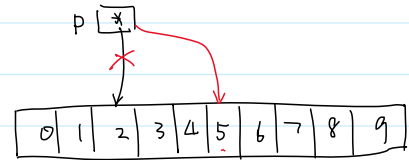
```
void min_max(const int arr[], int n, int* pmin, int* pmax) {  
    *pmin = arr[0];  
    *pmax = arr[0];  
    for (int i = 1; i < n; i++) {  
        if (arr[i] < *pmin) {  
            *pmin = arr[i];  
        } else if (arr[i] > *pmax) {  
            *pmax = arr[i];  
        }  
    }  
}
```

指针的算术运算

2024年4月29日 11:05



```
int main(void) {  
    int arr[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };  
    int* p = &arr[2];  
  
    // 指针加上一个整数, 结果还是一个指针  
    p = p + 3;  
    printf("*p = %d\n", *p);  
    return 0;  
}
```



指针类型不是整数!

0x00b3fc6c (2)
0x00b3fc78 (5)

向右偏移3个单位

→ 对象类型的大小

```
// 指针减去上一个整数, 结果还是一个指针  
int* p = &arr[8];  
p = p - 3; → 向左偏移3个单位  
printf("*p = %d\n", *p);  
return 0;
```

Microsoft Visual Studio 调试
*p = 5

```
// 两个指针相减, 结果是一个整数  
int* p = &arr[8];  
int* q = &arr[2];  
printf("p - q = %ld\n", p - q);  
printf("q - p = %ld\n", q - p);
```

→ 相隔几个单位

→ 4个字节

Microsoft Visual Studio 调试
p - q = 6
q - p = -6

0x003efcc4 (8)
0x003efcac (2)

4
- 4
18 (16) → 24 (16)

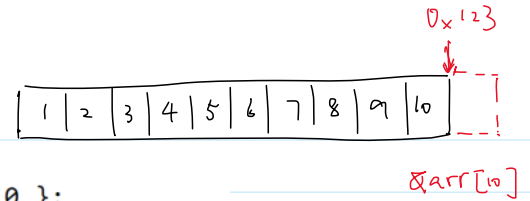
12 1. 4 5 6 7 8 9 10 11 12 13 14

定义: 指针的比较运算.

$$\begin{array}{lll} p - q == 0 & \Rightarrow & p == q; \quad (==, !=) \\ p - q > 0 & \Rightarrow & p > q; \quad (>, >=) \\ p - q < 0 & \Rightarrow & p < q; \quad (<, <=) \end{array}$$

指针和数组的关系

2024年4月29日 11:27



// 1. 指针处理数组

```
int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
int sum = 0;  
for (int* p = &arr[0]; p < &arr[10]; p++) {  
    sum += *p;  
}  
printf("sum = %d\n", sum);  
return 0;
```

addr = addr + 4
越界? 不会!

Microsoft Visual Studio 调试
sum = 55

```
sum = 0;  
for (int i = 0; i < 10; i++) {  
    sum += arr[i];  
}  
printf("sum = %d\n", sum);
```

$i_addr = base_addr + i * 4$

```
int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
int sum = 0;  
int* p = &arr[0];  
while (p < &arr[10]) {  
    sum += *p++;  
}  
printf("sum = %d\n", sum);
```

$\rightarrow * \text{和} ++ \text{的} (组合)$

Microsoft Visual Studio 调试
sum = 55

* 和 ++ 的组合

- A. $*p++$, $*(p++)$: 值为 $*p$, 副作用为 p 自增
- B. $(*p)++$: 值为 $*p$, 副作用为 $*p$ 自增
- C. $*++p$, $*(++p)$: 值为 $*(p+1)$, 副作用为 p 自增
- D. $++*p$, $++(*p)$: 值为 $(*p)+1$, 副作用为 $*p$ 自增

// 2. 在必要的时候, 数组可以退化成指向它索引为0元素的指针

```
int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
int sum = 0;  
for (int* p = arr; p < arr + 10; p++) {  
    sum += *p;  
}  
printf("sum = %d\n", sum);
```

数组

Microsoft Visual Studio 调试
sum = 55

long long a1 = 100;
int a2 = a1;

总结: ① 数组作为参数

`fun(arr);`

② 数组在赋值表达式的右边.

`int *p = arr;`

③ 数组号与算术运算.

`arr + 3;`

// 3. 指针也支持取下标运算。

```
int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```
int* p = arr;
```

```
int sum = 0;
```

```
I for (int i = 0; i < 10; i++) {  
    sum += p[i];  
}
```

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

$p[i] \Leftrightarrow *(p+i)$

$\Leftrightarrow *(i+p)$

$\Leftrightarrow i[p]$

```
I int sum_array(int arr[], int n) {  
    int sum = 0;  
    for (int i = 0; i < n; i++) {  
        // sum += *(arr + i);  
        // sum += arr[i];  
        sum += i[arr];  
    }  
    return sum;  
}
```


字符串字面值

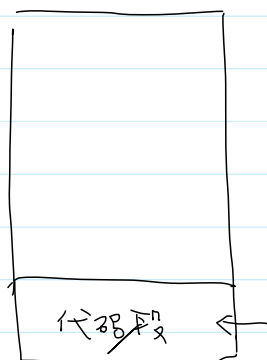
2024年4月29日 14:40

1. 书写方式.

```
int main(void) {  
    printf("I love xixi -- From peanut\n");  
  
    printf("I love xixi\  
    -- From peanut\n");  
  
    printf("I love xixi"  
        " -- From peanut\n");  
    return 0;  
}
```

2. 内存模式.

字符串字面值在代码段 → 不可被修改.



"ABC" →

A	B	C	\0
---	---	---	----

 数组名
↳ 空字符.

" " →

\0

void 空类型
\0 空字符
NULL 空指针
"" 空字符串

3. 字符串字面值支持的操作.

常量数组能支持的操作, 字符串字面值都支持.

```
// 10 -> 'A'  
// 11 -> 'B'  
// ...  
// 15 -> 'F'  
char digit_to_hex(int digit) {  
    return "0123456789ABCDEF"[digit];  
}
```

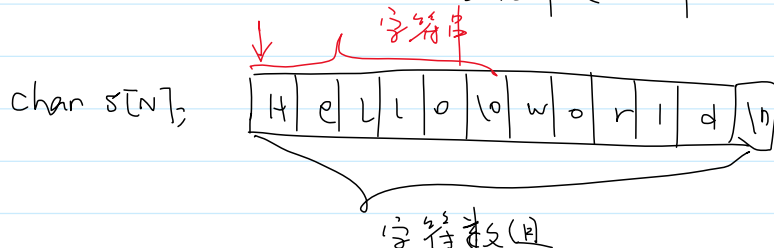
字符串变量

2024年4月29日 15:09

总结: ① C语言没有字符串类型! \rightarrow string, String

② C语言中的字符串依赖字符数组存在!

③ C语言中字符串是一种“逻辑类型”,



注意事项: A. 字符串必须以 '\0' 结尾!

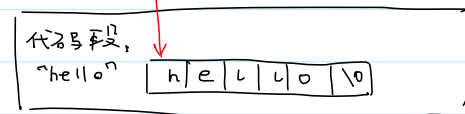
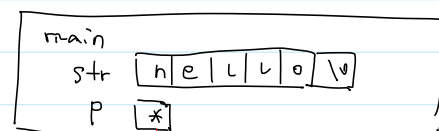
B. 在C语言中求字符串的长度不是 $O(1)$ 的时间复杂度

```
int main(void) {  
    char str[] = "Hello world\n";  
    for (int i = 0; i < strlen(str); i++) {  
        char c = str[i];  
    }  
  
    return 0;  
}
```

$\Rightarrow O(n)$

声明和初始化,

```
// 注意事项  
char str[] = "hello"; // "hello": 数组的初始化式  
char* p = "hello";    // "hello": 字符串字面值
```



#. 读和写 (和用户交互)

1. printf + %s. 输出字符串.

```
char str[] = "Hello world";
printf("%s\n", str);
printf("%.5s\n", str);
```

p, 最多输出 p 个字符



scanf + %s.

```
char str[MAX_LINE];
scanf("%s", str); // 不需要加取地址运算符!
printf("%s", str);
```



%s 匹配规则: 忽略前置空白字符, 读取字符填入字符数组, 遇到空白字符结束.

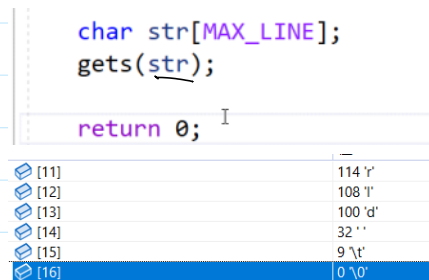
缺点: ① 不能够存储空白字符.
② 不会检查数组越界

2. puts / gets

```
char str[MAX_LINE] = "Hello world";
```

```
printf("%s\n", str);
puts(str);
```

puts(str) 等价于 printf("%s\n", str)



gets(str):

从 stdin 中读取一行数据, 存入字符数组, 并将 '\n' 替换成 '\0'.

缺陷: 不会检查数组越界.

```
fgets(str, MAX_LINE, stdin);
```

和 gets 的区别:

- ① 会检查是否越界
- ② 会存储 '\n' 符, 并在后面添加 '\0'.

字符串库函数

2024年4月29日 16:28

#3. 操作.

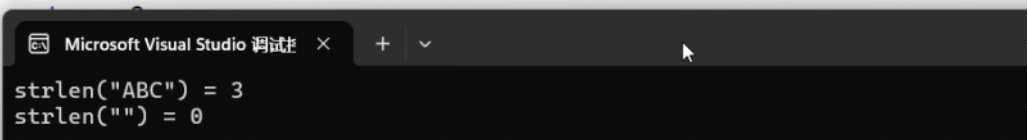
<string.h>

API 使用

↓
实现.

1. strlen:

```
printf("strlen(\"ABC\") = %d\n", strlen("ABC")); // 3
printf("strlen(\"\") = %d\n", strlen("")); // 0
```



Microsoft Visual Studio 调试

```
strlen("ABC") = 3
strlen("") = 0
```

strlen 不会计算 '\0'.

惯用法:

```
const char* p = s;
```

```
while (*p) {
    p++;
}
```

```
return p - s;
```

⇒ 搜索字符串的末尾, *p == '\0'

↓
遍历字符串.

2. strcpy & strncpy

```
char s1[MAXLINE]; // s1是一个数组
// strcpy(s1, "Hello world");
strncpy(s1, "Hello world", MAXLINE - 1); // 1 for '\0'
s1[MAXLINE - 1] = '\0';
```

```

char* my_strcpy(char* s1, const char* s2) {
    // 版本一
    //while (*s2 != '\0') {
    //    *s1 = *s2; // 复制
    //    s1++;
    //    s2++;
    //} // *s2 == '\0';
    // *s1 = '\0';
    //return s1;

    // 版本二
    char* p = s1;
    while (*s1++ = *s2++) // *s2
        ; // 复制
    // *s2 == '\0'
    return p;
}

```

堆内存法

复制字符串, 并且 '\0' 也复制了

```

// count: 最多复制count个字符
char* my_strncpy(char* s1, const char* s2, int count) {
    char* p = s1;
    while ((count-- > 0) && (*s1++ = *s2++))
        ;
    // count == 0 || *s2 == '\0'
    return p;
}

```

3. strcat & strncat.

```

// 字符串的拼接
// strcat = string + concatenate
char s1[MAXLINE] = "Hello "; // s1是一个数组
// strcat(s1, "world\n");
strncat(s1, "world\n", MAXLINE - strlen(s1) - 1); // 1 for '\0'
s1[MAXLINE - 1] = '\0';

```

```

char* my_strcat(char* s1, const char* s2) {
    // 1. 搜索s1的末尾
    char* p = s1;
    while (*s1) {
        s1++;
    } // *s1 == '\0'
    // 2. 复制s2
    while (*s1++ = *s2++)
        ;
    return p;
}

```

```
// count: 最多拼接count个字符
char* my_strncat(char* s1, const char* s2, size_t count) {
    // 1. 搜索s1的末尾
    char* p = s1;
    while (*s1) {
        s1++;
    } // *s1 == '\0'
    // 2. 复制s2
    while ((count-1) && (*s1++ = *s2++))
        ;
    return p;
}
```