# C语言基础 2023/12/9

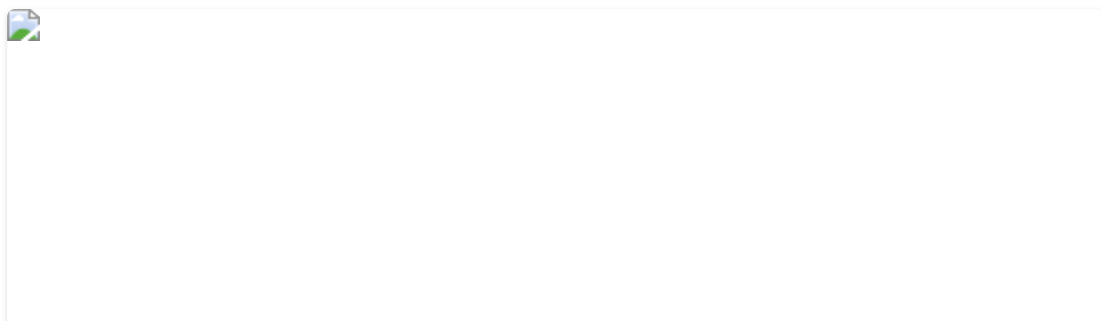
## 数组, 指针与函数

娄雨轩 建筑与城市规划学院

# 目录

# 内存

## 简单来说, 内存就是存放数据的容器



# 堆与栈

堆和栈都是一段连续的内存区域, 区别在于在栈区的资源会自动释放, 在堆区的资源则(一般来说)不会

| — | 堆区资源 | 栈区资源 |
| --- | --- | --- |
| 内存分配 | 动态分配 | 自动分配 |
| 内存释放 | 需要手动释放 | 自动释放 |
| 内存效率 | 较低 | 较高 |
| 内存大小 | 较大(数GB) | 较小(1~8MB) |
| 生命周期 | 程序结束时释放 | 函数执行完毕时释放 |

# 内存的开辟

一般来说, 我们不需要通过额外手段在栈区开辟内存, 我们只要声明一个变量, 即在栈区开辟了这个变量对应的内存

```
int a = 1;
printf("a : %d, address: %p", a, &a);
```

虽然栈区的内存管理很方便(不需要我们特别管理)而且读写速度快(比堆稍快几倍)但他只有可怜的几兆内存

```
int matrix[1024 * 1024]; // 可能StackOverflow
```

所以我们应该在堆区开辟内存, 需要借助这个函数:

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

由于最小的存储单元是Byte, 所以我们要先计算好所需要被分配的内存大小, 再传入这个函数, 他的返回值为分配得到的内存的起始地址

```
// 开辟大内存空间正确的姿势
int* matrix = (int*)malloc(1024 * 1024 * sizeof(int));
for(int i =0 ; i < 1024 * 1024; ++ i) { // 可以将matrix看作一个数组访问
  matrix[i];
}
```

# 内存释放

`malloc`的全称时间上是memory allocate(内存分配), 我们通过malloc分配的内存一定要自己手动释放(交还给计算机), 否则会造成安全隐患(memory leak 内存泄漏), 负责内存释放的函数是`void free(void* ptr)`,并且你应该要在free()之后将原本指向这段内存的指针重新指向NULL或nullptr.

```
int main() {
  float* arr = (float*) malloc(100 * sizeof(float));
  ...
  ...
  return 0;
}
```

```
int main() {
  float* arr = (float*) malloc(100 * sizeof(float));
  ...
  free(arr);
  arr = nullptr;
  return 0;
}
```

什么时候释放, 在何处释放, 由谁来释放是个很复杂的概念, 但是程序员们通过制定一系列"君子协定"来约束这一件事.

## 资源与所有权简介

`资源`一般指一段连续或不连续的内存, `所有权`是指, 一个内存资源是在哪, 由谁开辟的, 那么这个对象具有这个资源的所有权, 谁具有资源的所有权, 谁负责清理和回收资源.

# 指针与数组

数组就是指针是一种十分错误的说法(同样地还有引用就是指针这种说法也是同等离谱的错误)

思考: 可以这样做吗

```
int arr[] = (int*) malloc(100 * sizeof(int))
```

# 指针类型初探

指针是指值为内存地址的变量

## 声明一个指针

对于一个类型T,我们声明**保存这个类型的变量的地址的变量**的方式为T*

```
int* p_i; // p_i可以保存某个int类型的变量的地址
float* p_f; // p_f可以保存某个float类型的变量的地址
```

## 初始化一个指针

```
int a = 0;
int* p = &a; // p 保存了a的地址 或 称 p指向a
// 如果暂时没有需要保存的地址，将其置为NULL
int* q = NULL; // NULL是C的一个宏,其值为((void*)(0)) 或 ((long long)(0))
```

我们还可以通过解引用运算符+指针来访问到一个值

```
*p = 1 // 此时修改a为1
```

也可以修改p的指向(即改变p保存的地址)

```
int b = 2, c = 3;
p = &b; // *p = 2
```

# 指针与生存期

```
int* p = NULL;
{
  int a = 1;
  p = &a;
}
*p = 1;
```

# 指针与生存期

```
int* p = NULL;
{
  int a = 1;
  p = &a;
}
*p = 1;
```

经典的UB, 指针指向了一个已经消亡的对象, 并且试图对这个地址写入数据

# 指针与生存期

```
int* p = NULL;
{
  int a = 1;
  p = &a;
}
*p = 1;
```

经典的UB, 指针指向了一个已经消亡的对象, 并且试图对这个地址写入数据

解决方法1

使指针变量的生存期和可见性与被指向的变量一致(或小于)

```
{
  int a = 1;
  int* p = &a;
  ...
  // p消亡, a消亡
}
*p = 1; // 错误: 未定义标识符(此处p不可见)
```

# 指针与生存期

```
int* p = NULL;
{
  int a = 1;
  p = &a;
}
*p = 1;
```

经典的UB, 指针指向了一个已经消亡的对象, 并且试图对这个地址写入数据

解决方法1

使指针变量的生存期和可见性与被指向的变量一致(或小于)

```
{
  int a = 1;
  int* p = &a;
  ...
  // p消亡, a消亡
}
*p = 1; // 错误: 未定义标识符(此处p不可见)
```

解决方法2

在临时变量消亡或即将消亡时改变p的指向

```
int* p = NULL;
{
  int a = 0;
  p = &a;
  // do something...
  ...
  p = NULL;
}
if (p != NULL) {
  // do something with p
}
```

# 指针与const

**指向常量的指针(pointer to const)** 不能用于修改所指对象的值,**想要存放常量对象的地址,只能使用指向常量的指针**, 但常量指针可以指向非常量, 只是你不能通过这个指针修改它, 并且与其他常量一样, **你必须在声明时初始化它**.

```
int a = 0;
const int* p = &a; // p是指向a的常量指针
*p = 1; // ERROR:
int b = *p // OK, 只读
const int c = 1;
int* q = c; // ERROR:
const int* q = c; // OK
const int* _q; // ERROR: 必须初始化
```

# 顶层const

如前所述, 指针本身就是一个变量, 它又可以指向另一个变量. 因此, 指针本身是不是常量以及指针所指的是不是常量就是两个相互独立的问题.

所以我们用**顶层const(top-level const)**表示指针本身是一个常量, 用**底层const(low-level const)**来表示指针所指的变量是一个常量.

换言之, 顶层const不允许我们修改指针的指向(所保存的地址), 而可以修改指针指向的对象的值, 底层const不允许我们修改指针指向的对象的值, 而可以修改指针的指向(所保存的地址).

**一个指针可以即带有顶层const又带有底层const**

```
int i = 0;
int * const p1 = &i; // 不能改变p1的指向，这是一个顶层const
const int ci = 42; // 不能改变ci的值，这是一个常量
const int* p2 = &c; // 允许改变p2的指向，这是一个底层const
p2 = &i // OK
const int* const p3 = p2 // 靠右的const是顶层const，靠左的const是底层const
```

# 快速记忆const

## 唯一原则: const的左结合律

const关键字默认和优先与左侧的关键字结合, 与表示类型的关键字结合时为底层const, 意为不可修改其指向的变量的值, 与*结合时为顶层const, 意为不可修改其本身的值(即指针的指向)

```
const int i = 0; // const左侧没有关键字，故与右侧关键字int结合，不能修改i的值
const int* p = &i; // const左侧没有关键字，故与右侧关键字int结合，底层const，不能修改p所指变量即i的值
int const* q = &i; // const优先与左侧关键字int结合，与const int* q完全相同
int* const _p = &i; // const优先与左侧*结合，顶层const，不能修改_p的指向
_p = NULL; // ERROR: 不能修改_p的值(指向)
// 注意: 没有const* T这种写法
```

# 数组

## 数组的声明

数组声明的方式为: T arr[numeric constexpr]; 或T arr[] = {.....}

```
int a[100]; // Correct
float b[100]; // Correct
#define N 100
int c[N]; //Correct
const int n = 100;
int D[n]; // Correct?
constexpr int m = 100; // After C++11
int E[m]; // Correct After C++17
int A[] = {1, 2, 3, 4, 5}; // Correct
int B[5] = {1, 2, 3, 4, 5}; // Correct Same With A
```

```
int C[5] = {0}; // Correct
int D[10] = {0, 0, 0, 0, 0,}; // Correct Same With C
int E[10] = {1}; // Correct
for(int i = 0; i < 10; ++ i) {
  printf("%d", E[i]);
} // {1, 0, 0, 0, 0}
```

如果数组元素没有被初始化, 那么数组元素值的默认初值为0;

# sizeof

sizeof(expr)关键字是一个**运算符**, **而不是函数**他可以获取某个变量, 基本类型或复合类型在内存中所占的大小(单位: Byte) 你可以使用`sizeof expr`也可以使用`sizeof(expr)`

# sizeof

sizeof(expr)关键字是一个**运算符**, **而不是函数**他可以获取某个变量, 基本类型或复合类型在内存中所占的大小(单位: Byte) 你可以使用`sizeof expr`也可以使用`sizeof(expr)`

```
printf("%lld", sizeof(int)); // CORRECT
printf("%lld", sizeof int); // ERROR;
int a =0 ;
printf("%lld %lld", sizeof(a), sizeof a); // CORRECT;
printf("%lld", sizeof("HelloWorld")) // CORRECT
printf("%lld", sizeof "HelloWorld") // CORRECT
// Guess:
char ch = 'A';
printf("%lld", sizeof(ch));
printf("%lld", sizeof('A'));
printf("%lld", sizeof 'A')
```

# sizeof

sizeof(expr)关键字是一个**运算符**, **而不是函数**他可以获取某个变量, 基本类型或复合类型在内存中所占的大小(单位: Byte) 你可以使用`sizeof expr`也可以使用`sizeof(expr)`

```
printf("%lld", sizeof(int)); // CORRECT
printf("%lld", sizeof int); // ERROR;
int a =0 ;
printf("%lld %lld", sizeof(a), sizeof a); // CORRECT;
printf("%lld", sizeof("HelloWorld")) // CORRECT
printf("%lld", sizeof "HelloWorld") // CORRECT
// Guess:
char ch = 'A';
printf("%lld", sizeof(ch));
printf("%lld", sizeof('A'));
printf("%lld", sizeof 'A')
```

## sizeof 内的表达式**实际上不会被执行**

```
int a = 0;
int b = sizeof(++a);
printf("%d", a);
```

# 指针与数组之区别

观察下面的代码:

```
int arr[10];
int* ptr = (int*)malloc(sizeof(int));
printf("sizeof arr: %lld\n", sizeof(arr));
printf("len of arr: %lld\n", sizeof(arr)/sizeof(int));
printf("sizeof ptr: %lld\n", sizeof ptr);
printf("sizeof \"hello\": %lld\n", sizeof "hello");
```

# 指针与数组之区别

观察下面的代码:

```
int arr[10];
int* ptr = (int*)malloc(sizeof(int));
printf("sizeof arr: %lld\n", sizeof(arr));
printf("len of arr: %lld\n", sizeof(arr)/sizeof(int));
printf("sizeof ptr: %lld\n", sizeof ptr);
printf("sizeof \"hello\": %lld\n", sizeof "hello");
```

## 区别之1: 数组包含长度信息, 而指针不包含长度信息

# 指针的运算

C为指针定义了一系列运算符:

1. 解引用运算符operator*
2. 加法运算符operator+
3. 减法运算符operator-
4. 自增运算符operator++
5. 自减运算符operator–

# 指针的运算

C为指针定义了一系列运算符:

1. 解引用运算符operator*
2. 加法运算符operator+
3. 减法运算符operator-
4. 自增运算符operator++
5. 自减运算符operator–

`operator*`

```
int a = 0;
int* p = &a;
printf("%d", *p); // 通过*p访问p所指向的变量
```

# 指针的运算

C为指针定义了一系列运算符:

1. 解引用运算符operator*
2. 加法运算符operator+
3. 减法运算符operator-
4. 自增运算符operator++
5. 自减运算符operator–

`operator*`

```
int a = 0;
int* p = &a;
printf("%d", *p); // 通过*p访问p所指向的变量
```

`operator+`

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int* p = a; // p指向数组的第一个元素
p = p + 5; // p现在指向数组的第六个元素
printf("%d", *p); // 输出5，这是数组的第六个元素的值
printf("%p", a);
printf("%p", p);
```

`operator-`

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int* p = a + 5; // p指向数组的第六个元素

// 指针的减法运算符
p = p - 3; // p现在指向数组的第三个元素
printf("%d\n", *p); // 输出2，这是数组的第三个元素的值
printf("%p", a);
printf("%p", p);
```

## `operator-`

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int* p = a + 5; // p指向数组的第六个元素

// 指针的减法运算符
p = p - 3; // p现在指向数组的第三个元素
printf("%d\n", *p); // 输出2，这是数组的第三个元素的值
printf("%p", a);
printf("%p", p);
```

## `operator++`

```
p++; // p现在指向数组的第四个元素 同 p = p + 1或 p -= 1
printf("%d\n", *p); // 输出3，这是数组的第四个元素的值
printf("%p", a);
printf("%p", p);
```

## `operator-`

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int* p = a + 5; // p指向数组的第六个元素

// 指针的减法运算符
p = p - 3; // p现在指向数组的第三个元素
printf("%d\n", *p); // 输出2，这是数组的第三个元素的值
printf("%p", a);
printf("%p", p);
```

## `operator++`

```
p++; // p现在指向数组的第四个元素 同 p = p + 1或 p -= 1
printf("%d\n", *p); // 输出3，这是数组的第四个元素的值
printf("%p", a);
printf("%p", p);
```

## `operator--`

```
// 指针的自减运算符
p--; // p现在指向数组的第三个元素 同p = p-1或p-=1;
printf("%d\n", *p); // 输出2，这是数组的第三个元素的值
printf("%p", a);
printf("%p", p);
```

# 当我们在访问数组成员时, 我们究竟在访问什么

When we visit the elements of an array, what are we visiting actually?

学习完指针, 我们现在有2种访问数据的方式:

1. 通过变量的标识符(变量名)
2. 通过变量的地址 + 解引用运算符 ( *p )

所以 `arr[i]` 是怎么一回事?

# 当我们在访问数组成员时, 我们究竟在访问什么

When we visit the elements of an array, what are we visiting actually?

学习完指针, 我们现在有2种访问数据的方式:

1. 通过变量的标识符(变量名)
2. 通过变量的地址 + 解引用运算符 ( *p )

所以 `arr[i]` 是怎么一回事?

根据刚才我们所了解到的对一个指针变量所使用的operator+与operator-, 实际上是以这两个运算符的右操作数为偏移量, 根据原来指针所保存的地址值进行地址的偏移, 具体的偏移数值为: 右操作数 x sizeof(指针的类型) 那么如果我们希望访问数组arr的第i个元素, 我们可以这样获取其地址:

```
int arr[100] = {0}, i = 5;
int* p = arr + i; // p保存了第i个元素(第5个元素)的地址
int is_same = (p + i == &a[i]); // True
```

由此推断出:

```
int i = 5;
int is_same = (*(p + i) == a[i]); // True
```

# WARNING: 一些有一点邪恶但是有可能你会在其他项目中看到的技巧

如果你不能十分熟练地玩弄指针, 请不要学习这样做, 本页的介绍纯属希望大家认识这种做法的合法性以及危险性.

# WARNING: 一些有一点邪恶但是有可能你会在其他项目中看到的技巧

如果你不能十分熟练地玩弄指针, 请不要学习这样做, 本页的介绍纯属希望大家认识这种做法的合法性以及危险性.

## 知识回顾

我们可以通过一个指针加或减一个数得到一个新的地址, 该地址与原来指针所保存的地址的差为操作数乘以指针的类型 那么:

```
char ch = '0';
char* p_ch = &ch; // p_ch + 1 = &ch + 1Byte
int integer = 0;
int* p_int = &integer; // p_int + 1 = &integer + 4Byte;
T sometype;
T* p_sometype = &sometype; // p_sometype + 1 = &sometype + sizeof(T)Byte;
```

# WARNING: 一些有一点邪恶但是有可能你会在其他项目中看到的技巧

如果你不能十分熟练地玩弄指针, 请不要学习这样做, 本页的介绍纯属希望大家认识这种做法的合法性以及危险性.

## 知识回顾

我们可以通过一个指针加或减一个数得到一个新的地址, 该地址与原来指针所保存的地址的差为操作数乘以指针的类型 那么:

```
char ch = '0';
char* p_ch = &ch; // p_ch + 1 = &ch + 1Byte
int integer = 0;
int* p_int = &integer; // p_int + 1 = &integer + 4Byte;
T sometype;
T* p_sometype = &sometype; // p_sometype + 1 = &sometype + sizeof(T)Byte;
```

```
int num = 1234567;
printf("%d", num); // ouput: 1234567
char* p = (char*)&num;
int is_same = (p + 4 == &num + 1) // True
p[0] = 0;p[1] = 0;
p[2] = 0;p[3] = 0;
printf("%d", num); // output: ?
```

# 函数的概念

C语言中的函数是一段可以被重复使用的代码，它可以执行特定的任务。函数可以有参数，这些参数是传递给函数的值，函数可以使用这些值来执行其任务。 在C语言中，函数的定义包括函数的名称、返回类型和参数列表。形如:

```
Decoration(可选的) ReturnType FunctionName(Args...) Decoration(可选的) {
  FunctionBody...
  return ;
}
```

例如:

```
int add(int a, int b) { return a + b; }
```

函数的参数可以是任何类型，包括基本类型（如int、float及其指针等）、数组、结构体、甚至是其他函数。

函数会将每个参数都独自拷贝一份供内部使用:

```
void func(int x) { printf("value: %d address: %p", x, &x); }
int x = 0;
printf("value: %d address: %p", x, &x);
func(x);
```

Q: 形如`scanf("%d", &num);`这个函数为什么需要传递num的地址.

# 函数的声明

函数可以将声明和定义分开写例如:

```
void func();
...
...

void func() {
  // 注意，定义需要和函数声明具有严格相同的签名
}
```

函数在声明时可以不用写参数的名称, 只需要写明类型即可

```
void func(char, int ,float, int*);
...
...
void func(char ch, int integer, float f, int* p)
```

# 函数的意义和规范

1. 函数可以在你完成一个复杂的任务时减轻你的心智负担 复杂任务---- 子任务1 – … |— 子任务2 – … |— 子任务3 – … … 当你在完成复杂任务时, 你可以将其拆分成多个子任务函数, 在负责子任务的函数中你只需要关注当前的输入输出和当前的子任务逻辑, 而不用去过多考虑全局因素.
2. 函数可以减少代码量 一个超过5行并且多次重复出现的代码段, 你应该将其编写为函数.

# 函数规范: 如何设计函数的内容

1. 一个函数应当只关注并处理符合其语义的一件事. 如果一个函数的本意是处理某个数据, 进行某个数学运算, 处理某个逻辑, 那么他就不应该进行输入输出(IO, printf或scanf), 你应当适时地将输入作为参数, 而不是在函数内通过scanf接受用户输入, 除非你设计函数的本意如此.
2. 一个函数的命名应当符合其语义. 除了临时使用的函数或在某种数学问题下具有共识的命名的函数, 否则拒绝诸如func(), f(), g()这种命名的函数.
3. 函数的参数传递限定, 也应当符合其语义.
4. 函数应当遵守其所有权的约定, 即不参与所有权不属于自身的资源的开辟与释放, 除非语义如此, 并且自己负责在函数调用期间的资源的开辟与回收.
5. 函数内部也可以调用其他函数

6. 一个函数如果内部不引用任何外部变量(静态变量, 全局变量等), 也不会改变任何外部状态(任何外部变量等), 不论在任何时候调用这个函数, 只要是保持相同的输入, 就一定会得到相同的输出, 那么这个函数就是**纯的**, 被称为**纯函数**, 我们应当尽量设计**纯函数**.

# 函数的参数传递与所有权

函数允许接收若干值或者指针为变量作为函数的参数.

| 参数类型: | T | const T | T* | const T* |
| --- | --- | --- | --- | --- |
| 所有权: | 资源的拥有者 | 资源的拥有者 | 资源的共享者 | 资源的共享者 |
| 是否可修改: | True | False | True | False |
| 是否需要释放 | 函数结束时自动释放 | 函数结束时自动释放 | 不需要也不允许释放 | 不需要也不允许释放 |

# 数组作为函数的参数传递

有一下3种写法都可以被接受:

```
void func1(T* arr) {

}
```

```
void func2(T arr[]) {

}
```

```
void func3(T arr[10]) {

}
```

```
T arr[100] ={ 0 };
func1(arr) //Correct
func2(arr) //Correct
func3(arr) //Correct
```

# 数组作为函数的参数传递

以数组为函数参数在参数传递时, 数组与指针的兼容实际上是一种"退化行为", 并且这是不可避免的

```
void func1(T* arr) {

}
```

```
void func2(T arr[]) {

}
```

Q: 如何证明func1的参数传递和func2的参数传递完全等价

# 数组作为函数的参数传递

以数组为函数参数在参数传递时, 数组与指针的兼容实际上是一种"退化行为", 并且这是不可避免的

```
void func1(T* arr) {

}
```

```
void func2(T arr[]) {

}
```

Q: 如何证明func1的参数传递和func2的参数传递完全等价

```
void func1(T* arr) {
  printf("%lld", sizeof(arr));
}

void func2(T arr[]) {
  printf("%lld", sizeof(arr));
}
```

结论: 当数组作为参数传递时, 数组会退化成指向其头部位置的指针, 并且丢失其长度信息.

# 指针的参数传递

指针作为参数传递有一下几种情况:

1. 某个需要被修改的对象的地址作为参数传入 (传T*)
2. 某个不适合被拷贝的对象的地址作为参数传入 (一般传const T*)
3. 某个数组的头地址作为参数传入(传T*与一个整形类型的参数作为数组长度的指示)

# 练习

1. 实现swap函数, 要求: 接受两个数的地址, 交换这两个数的值, 签名如下:

```
void swap(int* a, int* b);
```

思考, 为什么swap(int a, int b); 不行 2. 实现冒泡函数, 要求, 接受一个指向数组头部地址的指针及其长度, 对这个数组进行一次冒泡

```
void bubble(int* arr, int length);
```

1. 实现冒泡排序, 要求, 接受一个指向数组头部地址的指针及其长度, 对其进行排序

```
void bubbleSort(int* arr, int length);
```

1. 请编写一个C语言函数，该函数接受一个指向数组头部的指针和数组的长度，然后将数组中的元素反转。你不能使用额外的数组来存储元素，只能在原数组上进行操作。

函数签名如下：

```
void reverseArray(int* arr, int length);
```