内存管理

动态内存分配

> c语言中的一切操作都是基于内存。
>
> 变量和数组都是内存的别名，分配这些内存由编译器在编译期间决定。
>
> 定义数组必须指定数组长度，数组长度在编译器就必须决定。

> malloc free用于执行动态内存分配和释放
>
> malloc分配的是一块连续内存，以字节为单位，不带任何类型信息
>
> free用于将动态内存归还系统

```
void * malloc(size_t size);
void free(void * pointer);
//malloc实际分配内存可能比请求多一点
//请求的动态内存无法满足时malloc返回NULL
//free参数为NULL时，函数直接返回
```

```
void* calloc(size_t num,size_t size);//(单元数目，单元大小)
void* realloc(void* pointer, size_t new_size);

//calloc能以类型大小为单位申请内存并初始化为0
//calloc会将返回的的内存初始化为0

//realloc用于修改一个原先已经分配的内存块大小
//在使用realloc之后应该使用其返回值
//当Pointer的第一个参数为NULL时，等价于malloc
```

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

int main()
{
    int i = 0;
    int* pI = (int*)malloc(5 * sizeof(int));
    short* pS = (short*)calloc(5, sizeof(short));

    for(i=0; i<5; i++)
    {
        printf("pI[%d] = %d, pS[%d] = %d\n", i, pI[i], i, pS[i]);
    }

    pI = (int*)realloc(pI, 10);

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

```
    }

    free(pI);
    free(pS);

    return 0;
}
```

栈 堆 存储区

> 栈：在程序中用于维护函数调用上下文，没有栈就没有函数，没有局部变量。 后进先出
>
> 函数运行结束时会自动销毁
>
> 堆：由malloc系列函数或new分配的内存，生命周期由free或delete决定。空间较大。
>
> 堆管理方式：空闲链表法，位图法，对象池法
>
> 静态存储区：用于全局变量和静态变量，在编译期大小就已经确定，随着程序运行分配空间，直到运行结束。

内存分布

```
int global_init_var = 84;  //.data section
int global_uniit_var;      //.bss section

void func1(int i)   //.text
{
    printf("%d\n",i);
}

int main(){
    static int static_var = 85; //.data
    static int static_var2; //.bss
    int a=1;
    int b;
    func1(static_var + static_var2+a+b); //.text
    return 0;
}
```

> 运行之后的地址空间布局
>
> *高地址*

栈
堆
.bss
.data
.text
..l.....未映射区域

低地址

堆栈段在程序运行后才正式存在， 存局部变量

.bss段存放未初始化的全局变量和静态变量

.text存放可执行代码

.data存放已经初始化的全局变量和静态变量

.rodata存放程序中的常量值，只读区

```
#include<stdio>

int main()
{
    char * p ="hello";  //常量在.rodata，只读，修改就会出错
    p[0]='1';
    printf(p);  //显示段错误
}
```

函数指针对应存放代码段的地址

野指针

通常是因为指针变量中保存的值不是合法的内存地址

不是NULL指针，是指向不可用内存的指针

NULL指针不容易用错

c语言中没有任何手段判断一个指针是不是野指针

来源

局部指针变量未被初始化

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

struct Student
{
    char* name;  //name未初始化
    int number;
};
```

```
int main()
{
    struct Student s;

    strcpy(s.name, "Delphi Tang"); // OOPS!

    s.number = 99;


    return 0;
}
```

使用已经释放过的指针

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

void func(char* p)
{
    printf("%s\n", p);
    free(p);   //已经释放了
}

int main()
{
    char* s = (char*)malloc(5);

    strcpy(s, "Delphi Tang");

    func(s);

    printf("%s\n", s); // OOPS! , s已经释放了

    return 0;
}
```

指针所指向的变量在指针之前被销毁

```
#include <stdio.h>

char* func()
{
    char p[] = "Delphi Tang";  //p是局部数组

    return p;
}

int main()
{
    char* s = func();  //s指向一片被释放的栈空间

    printf("%s\n", s); // OOPS!
```

```
    return 0;
}
```

经典错误

结构体指针未初始化

没有为结构体指针分配足够内存

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

struct Demo
{
    int* p;   //p未初始化
};

int main()
{
    struct Demo d1;
    struct Demo d2;

    int i = 0;

    for(i=0; i<10; i++)
    {
        d1.p[i] = 0; // OOPS!
    }

    d2.p = (int*)calloc(5, sizeof(int)); //分配内存不够

    for(i=0; i<10; i++)
    {
        d2.p[i] = i; // OOPS!
    }

    free(d2.p);

    return 0;
}
```

内存分配成功但是未初始化

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

int main()
{
    char* s = (char*)malloc(10);

    printf(s); // OOPS!

    free(s);

    return 0;
}
```

数组越界

```
#include <stdio.h>

void f(int a[10])
{
    int i = 0;

    for(i=0; i<10; i++)
    {
        a[i] = i; // OOPS!
        printf("%d\n", a[i]);
    }
}

int main()
{
    int a[5];

    f(a);

    return 0;
}
```

内存泄漏

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

void f(unsigned int size)
{
    int* p = (int*)malloc(size*sizeof(int));
    int i = 0;

    if( size % 2 != 0 )
    {
        return; // OOPS!,没有释放
    }

    for(i=0; i<size; i++)
    {
        p[i] = i;
        printf("%d\n", p[i]);
    }

    free(p);
}

int main()
{
    f(9);
    f(10);

    return 0;
}
```

多次指针释放

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

void f(int* p, int size)
{
    int i = 0;

    for(i=0; i<size; i++)
    {
        p[i] = i;
        printf("%d\n", p[i]);
    }

    free(p);
}

int main()
{
    int* p = (int*)malloc(5 * sizeof(int));

    f(p, 5);

    free(p); // OOPS!  ，第二次free

    return 0;
}

//遵循原则：谁申请谁释放
```

使用已释放的指针

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

void f(int* p, int size)
{
    int i = 0;

    for(i=0; i<size; i++)
    {
        printf("%d\n", p[i]);
    }

    free(p);
}

int main()
{
    int* p = (int*)malloc(5 * sizeof(int));
    int i = 0;

    f(p, 5);  //p已经释放了

    for(i=0; i<5; i++)
    {
```

```
        p[i] = i; // OOPS!
    }

    return 0;
}
```

规则

1. malloc申请了内存之后，应立即检查指针值是否为NULL,防止使用值为NULL的指针

```
int * p=(int *)malloc(size*sizeof(int));
if(p)
{
  ...
}
free(p);
```

2. 牢记数组长度，防止数组越界，考虑使用柔性数组

```
typedef struct _soft_array
{
    int len;
    int array[];
}SoftArray;

int i=0;
SoftArray* sa = (SoftArray *)malloc(sizeof(SoftArray)+sizeof(int)*10);

sa->len = 10;
for(i=0;i<sa->len;++i)
{
    sa->array[i]=i;
}
```

3. 动态申请操作必须和释放操作匹配，防止内存泄漏和多次释放

```
void f(){
    int* p =(int*)malloc(5);
    free(p);
}
```

4. free指针之后必须立即赋值为NULL

```
int * p=(int*)malloc(10);
free(p);
p=NULL;
```

这样指针要么为空要么为合法地址