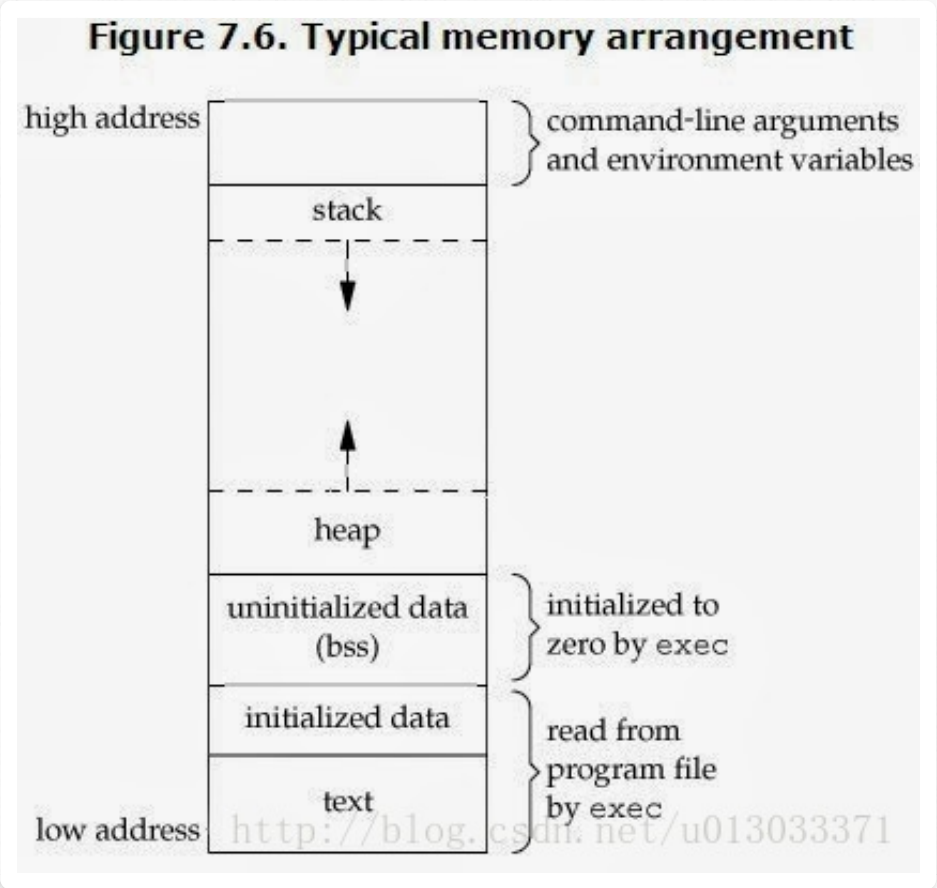
聊聊 C 语言的内存模型与指针

简介

内存模型就是对内存进行使用的一个规范。通过内存模型，我们能了解 C 语言是如何对内存进行划分，如何使用每一部分内存的。

划分

下图是APUE中的一个典型 C 语言内存空间分布图:



程序代码区

存放可执行文件的二进制数据

静态数据区

也称全局数据区，包含的数据类型比较多，如全局变量、静态变量、一般常量、字符

串常量。
全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。
常量数据（一般常量、字符串常量）存放在另一个区域。

堆区

在 C 语言中手动进行内存分配，就是在这个区获取内存

栈区

在 C 语言执行过程中，由系统自动分配的内存在这个区。如 函数的参数，局部变量等。

命令行参数区

存放命令行参数和环境变量的值，如通过main()函数传递的值。

未初始化全局变量与未初始化局部变量区别

未初始化的全局变量的默认值是 0，而未初始化的局部变量的值却是任意值。
代码示例：

```
#include <stdio.h>
int global;
int main()
{
    int local;
    printf("global = %d\n", global);
    printf("local = %d\n", local);
    return 0;
}
```

输出内容：

```
$ ./test
global = 0
local = 324952530
$ ./test
global = 0
local = 163410386
```

所在分区代码示例

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int a = 0;  // 静态数据区
char *p1;  // 静态数据区
int main()
{
    int b;  // 栈区
    char s[] = "abc";  // 栈区
    char *p2;  // 栈区
    char *p3 = "123456"; // 123456\0 静态数据区，p3在栈上，体会与 char s[]="
abc"; 的不同
    static int c = 0;  // 静态数据区
    p1 = (char *)malloc(10),  // 堆区
    p2 = (char *)malloc(20);  // 堆区
    // 123456\0 放在常量区，但编译器可能会将它与p3所指向的"123456"优化成一个地方
    strcpy(p1, "123456");
}
```

指针

指针用途

指针用于存储内存地址。如：

```
int num = 10;
int *p1 = &num;
char *p2 = (char *)malloc(20);
```

指针转换

指针之间可以转换。转换的方式就是指针变量前面增加括号，并在括号中写明要转换成的指针类型即可。例如：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main()
{
```

```
    int num = 97;
    int *p = &num;
    char *p2 = (char *)p;
    printf("%c\n", *p2);
}
```

输出内容

```
$ ./test
a
```

void *ptr

void *ptr 指针使用前必须转换为指定类型。代码示例：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main()
{
    int num = 97;
    int *p = &num;
    void *p2 = (char *)p;
    int sum = num + *p2;
    printf("%d", sum);
}
```

编译的时候报错

```
$ gcc ./ctest.c -o test
./ctest.c:9:19: error: invalid operands to binary expression ('int' and 'vo
    int sum = num + *p2;
              ~~~ ^ ~~~
1 error generated.
```

引用 与 解引用

- 引用就是取一个变量的内存地址。进行引用的方式就是在变量前面加&符号。
- 解引用就是获取指针所指向内存中存储的内容。解引用的方式就是在指针前面加星号*。

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

```c
#include<stdlib.h>
#include<string.h>
int main()
{
    int num = 97;
    int *p = &num;
    *p = 100;
    printf("*p=%d, num=%d\n", *p, num);
}
```

输出结果

```
$ ./test
*p=100, num=100
```

双重指针

指针作为一个参数传入示例:

```c
void fun(char * pa)
{
    printf("pa的地址: %d\n",pa);
    puts(pa);
    char *pb = "bb";
    printf("pb的地址: %d\n",pb);
    puts(pb);
    pa=pb;   //在这里发生了改变
    printf("pa的地址: %d\n",pa);
    puts(pa);
}

int main(void)
{
    char *p ="aa" ;
    printf("p的地址: %d\n",p);
    printf("改变前p = ");
    puts(p);
    printf("------------------\n");
    fun(p);
    printf("------------------\n");
    printf("p的地址: %d\n",p);
    printf("改变后p = ");
    puts(p);
    return 0;
}
```

输出内容：

```
$ ./test
p的地址：0x105a1cf6b
改变前p = aa
------------------
pa的地址：0x105a1cf6b
aa
pb的地址：0x105a1cf56
bb
pa的地址：0x105a1cf56
bb
------------------
p的地址：0x105a1cf6b
改变后p = aa
```

指针地址作为一个参数传入示例：

```
void fun(char ** pa)
{
    printf("pa的地址：%d\n",*pa);
    puts(*pa);
    char *pb = "bb";
    printf("pb的地址：%d\n",pb);
    puts(pb);
    *pa=pb;   //这里pa是双重指针 pb是一重指针
    printf("pa的地址：%d\n",*pa);
    puts(*pa);
}

int main(void)
{
    char *p ="aa" ;
    printf("p的地址：%d\n",p);
    printf("改变前p = ");
    puts(p);
    printf("------------------\n");
    fun(&p);
    printf("------------------\n");
    printf("p的地址：%d\n",p);
    printf("改变后p = ");
    puts(p);
    return 0;
}
```

输出内容：

```
$ ./test
p的地址: 0x10d915f6b
改变前p = aa
------------------
pa的地址: 0x10d915f6b
aa
pb的地址: 0x10d915f56
bb
pa的地址: 0x10d915f56
bb
------------------
p的地址: 0x10d915f56
改变后p = bb
```

栈 Stack

栈的大小一般是有限制的。超出限制会报错。

栈存储的信息

- 方法内的本地量
- 方法参数
- 方法的返回值

栈空间的使用由编译器控制

- 每次调用的时候进行分配，每调用一次分配一次
- 当方法退出的时候，栈帧就会消耗
- 栈的空间分配是从高位往地位分配的

代码示例

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

char * fun()
{
    char str[] = "123456";
    return str;
}

int main(void)
{
    char *p ="aa" ;
```

```
    p = fun();
    printf("%s\n", p);
    return 0;
}
```

编译的时候，报错信息如下：

```
$ gcc ./ctest.c -o test
./ctest.c:8:12: warning: address of stack memory associated with local vari
    return str;
           ^~~
1 warning generated.
```

报错的原因是，str 数组在栈上分配。当方法调用退出后，栈上的信息就释放了。返回一个释放的信息，是有问题的。

堆 Heap

堆是需要程序员自己调用相应的方法进行申请和释放的。进行内存的申请需要使用 malloc 等方法。内存的释放使用 free 方法。

代码示例

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

int main(void)
{
    int *p = (int *) malloc(10 * sizeof(int));
    // 申请完内存后，要检测返回值。验证是否申请内存失败。
    if (p == NULL) {
        printf("malloc error\n");
        return 0;
    }
    // 释放内存前要检测指针，避免多次释放
    if (p != NULL) {
        free(p);
        p = NULL;
    }
    return 0;
}
```

思考题

写一段程序，实现二位数组的效果。