

一. 系统内存空间分布情况

系统内存通常分为以下几个部分：

1. 代码区：存放程序的机器指令，由编译器生成。
2. 全局/静态变量区：存放全局变量和静态变量。
3. 堆区：用于动态分配内存（`malloc/free`）。
4. 栈区：存放函数调用过程中创建的局部变量、函数参数和返回地址。

对此，用 c 语言代码展示系统内存空间分布情况：

代码部分：

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

// 全局变量
int global_var = 10;
// 静态变量
static int static_var = 20;

void memory_layout() {
    // 局部变量（栈区）
    int local_var = 30;
    // 动态分配的内存（堆区）
    int* heap_var = (int*)malloc(sizeof(int));
    *heap_var = 40;

    printf("地址分布:\n");
    printf("代码区: %p (函数 memory_layout 地址)\n", (void*)memory_layout);
    printf("全局变量区: %p (全局变量 global_var 地址)\n", (void*)&global_var);
    printf("静态变量区: %p (静态变量 static_var 地址)\n", (void*)&static_var);
    printf("栈区: %p (局部变量 local_var 地址)\n", (void*)&local_var);
    printf("堆区: %p (动态分配 heap_var 地址)\n", (void*)heap_var);

    // 释放动态分配的内存
    free(heap_var);
}

int main() {
    memory_layout();
    return 0;
}
```

x86 平台下运行结果：

```
Microsoft Visual Studio 调试控制台
地址分布:
代码区: 00E512B7 (函数 memory_layout 地址)
全局变量区: 00E5A000 (全局变量 global_var 地址)
静态变量区: 00E5A004 (静态变量 static_var 地址)
栈区: 003CFC9C (局部变量 local_var 地址)
堆区: 009A02E8 (动态分配 heap_var 地址)
E:\c++\project\12周课堂作业\Debug\12周课堂作业.exe (
```

在运行结果中，我们可以看到：

1. 代码区：地址 **00E512B7**，代码区存放的是程序的机器指令。程序运行时，CPU 从这个区域读取指令并执行，上述地址为函数 `memory_layout` 的地址。代码区通常位于程序的只读内存中，并且大小是固定的。
2. 全局变量区：地址 **00E5A000**，存放的是程序中定义的全局变量（在这里为 `global_var`），全局变量区在程序加载时被分配，并且持续到程序结束。这个区域和静态变量区相邻，通常位于堆区以下。
3. 静态变量区：地址为：**00E5A004**，存放的是程序中的静态变量（`static_var`）。静态变量与全局变量类似，但作用域受到限制，仅在其声明的文件或函数内可见。地址紧挨着全局变量区域，可以看出全局和静态变量共享一个区域。
4. 栈区：地址为：**003CFC9C**，栈区用于存放函数调用中的局部变量（`local_var`）、函数参数和返回地址。栈从高地址向低地址方向增长。函数调用嵌套时，新分配的栈帧位于旧栈帧之下。
5. 堆区：地址为：**009A02E8**，堆区存放动态分配的内存（如 `malloc` 分配的内存）。地址是调用 `malloc` 后分配的内存地址。堆区从低地址向高地址增长，与栈区相反。

在 x64 下运行：

```
Microsoft Visual Studio 调试控制台
地址分布:
代码区: 00007FF72D33106E (函数 memory_layout 地址)
全局变量区: 00007FF72D33D000 (全局变量 global_var 地址)
静态变量区: 00007FF72D33D004 (静态变量 static_var 地址)
栈区: 000000DF648FF4E4 (局部变量 local_var 地址)
堆区: 00000296F5CE9CC0 (动态分配 heap_var 地址)
E:\c++\project\12周课堂作业\x64\Debug\12周课堂作业.exe (进程 2868
```

在 32 位架构和 64 位架构下运行程序时主要的不同体现在：

1. **地址宽度**：x86 地址为 32 位，打印出的内存地址位 8 位 16 进制数，程序的每个内存段通常会限制在 4GB 的地址空间内。x64 地址为 64 位，但通常只使用了 48 位有效地址空间，内存地址为 16 位 16 进制数，内存布局更宽松，栈、堆等分配的空间通常较大。
2. **地址分布**：x86 代码区、全局/静态变量区、堆区 和 栈区 的起始地址距离较近，因为地址空间有限。栈区和堆区的增长空间较小，容易导致栈溢出或堆分配失败。x64 地址空间更大，各内存区域之间间隔更远。栈通常分配更大的默认空间（如 8 MB 或更多），堆的动态增长也更灵活。
3. **栈大小**：x86 默认栈大小一般是 1 MB。因为地址空间有限，深层递归或大局部变量会更容易导致栈溢出。x64 默认栈大小一般是 8 MB（某些平台可能更大），更大的栈允许更多的递归深度和更大的局部变量空间。

二. 栈空间大小及修改方式

默认栈空间大小取决于操作系统和编译器。例如，在 Linux 系统上，默认栈大小通常是 8 MB。栈大小可以通过以下方式修改

1. 修改程序栈的大小（Linux 为例，修改为 16MB）：

```
ulimit -s 16384
```

2. 通过代码限制栈深度： 使用大数组或者深递归会导致栈溢出。可以通过合理规划栈空间或切换到堆区避免问题。

代码展示：

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

void stack_test(int depth) {
    int stack_array[1024]; // 每次递归消耗约 4 KB 栈空间
    printf("Recursion depth: %d, stack address: %p\n", depth, (void*)&stack_array);

    // 深度限制，防止栈溢出
    if (depth < 200) {
        stack_test(depth + 1);
    }
}

int main() {
    printf("启动栈测试:\n");
    stack_test(1);
    return 0;
}
```

代码运行结果：

```
E:\c++\project\12周课堂作业\Debug\12周课堂作业.exe
启动栈测试:
Recursion depth: 1, stack address: 00BAEACC
Recursion depth: 2, stack address: 00BAD4A8
Recursion depth: 3, stack address: 00BABE84
Recursion depth: 4, stack address: 00BAA860
Recursion depth: 5, stack address: 00BA923C
Recursion depth: 6, stack address: 00BA7C18
Recursion depth: 7, stack address: 00BA65F4
Recursion depth: 8, stack address: 00BA4FD0
Recursion depth: 9, stack address: 00BA39AC
Recursion depth: 10, stack address: 00BA2388
Recursion depth: 11, stack address: 00BA0D64
Recursion depth: 12, stack address: 00B9F740
Recursion depth: 13, stack address: 00B9E11C
Recursion depth: 14, stack address: 00B9CAF8
Recursion depth: 15, stack address: 00B9B4D4
Recursion depth: 16, stack address: 00B99EB0
Recursion depth: 17, stack address: 00B9888C
Recursion depth: 18, stack address: 00B97268
Recursion depth: 19, stack address: 00B95C44
Recursion depth: 20, stack address: 00B94620
Recursion depth: 21, stack address: 00B92FFC
Recursion depth: 22, stack address: 00B919D8
Recursion depth: 23, stack address: 00B903B4
Recursion depth: 24, stack address: 00B8ED90
Recursion depth: 25, stack address: 00B8D76C
Recursion depth: 26, stack address: 00B8C148
Recursion depth: 27, stack address: 00B8AB24
Recursion depth: 28, stack address: 00B89500
```

在运行结果中，每次递归调用时，程序会分配一个局部数组 `stack_array[1024]`，这会消耗约 4 KB 的栈空间。通过打印 `stack_array` 的地址，我们观察到递归深度的增加如何影响栈空间的使用。

运行结果分析：

1. 栈地址逐渐减小：栈从高地址向低地址增长，每次递归都会分配约 4 KB 的局部变量空间，导致栈地址按固定步长下降。
2. 步长（约 5636 字节）：递归函数中，局部数组 `stack_array[1024]`（每个 `int` 占 4 字节，总大小为 4096 字节）是主要的栈空间消耗，实际消耗大于 4096 字节，因为还包括函数调用的开销（如返回地址、局部变量等）。
3. 递归深度限制：每次递归都消耗栈空间，如果递归深度过大（超出栈的容量），会导致 栈溢出（Stack Overflow） 错误，在 x86 平台上，默认栈大小一般为 1 MB，在本例中最多支持约 250 层深度。
4. 局部变量地址变化：递归中，局部变量的地址每次都会重新分配，并位于更低的栈地址。

详细解释：

1. 栈帧结构：每次函数调用会创建一个新的 栈帧，包含局部变量、函数参数、返回地址等信息。每个栈帧的大小取决于局部变量和调用上下文。
2. 栈地址规律：栈地址递减表明栈从高地址向低地址增长。栈的增长步长约为局部变量大小加上函数调用的附加开销（如寄存器保存等）。
3. 递归终止：本程序设置递归深度为 200，即 `depth < 200`，因此不会达到栈的最大容量。如果移除深度限制或增加数组大小，可能会引发栈溢出。

三. 课堂作业

程序展示了两个函数的嵌套调用过程，打印调用栈中各个函数的地址和变量地址。

```
#include <stdio.h>

void second_function(int val) {
    int second_var = val;
    printf("Inside second_function:\n");
    printf("  Address of second_var: %p\n", (void *)&second_var);
}

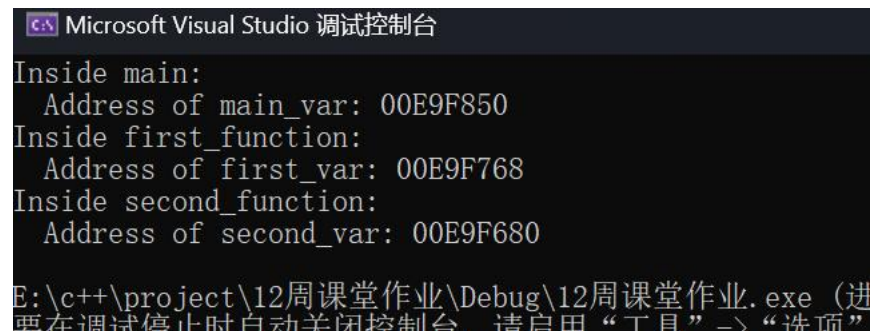
void first_function(int val) {
    int first_var = val;
    printf("Inside first_function:\n");
    printf("  Address of first_var: %p\n", (void *)&first_var);

    // 嵌套调用
    second_function(first_var + 1);
}

int main() {
    int main_var = 1;
    printf("Inside main:\n");
    printf("  Address of main_var: %p\n", (void *)&main_var);

    // 调用 first_function
    first_function(main_var);
    return 0;
}
```

运行结果展示（x86 环境下）：



```
Microsoft Visual Studio 调试控制台
Inside main:
  Address of main_var: 00E9F850
Inside first_function:
  Address of first_var: 00E9F768
Inside second_function:
  Address of second_var: 00E9F680
E:\c++\project\12周课堂作业\Debug\12周课堂作业.exe (进
要在调试停止时自动关闭控制台，请启用“工具”->“选项”
```

反汇编结果：

```
int main() {
002B1880  push        ebp
002B1881  mov         ebp, esp
```



```

002B1883 sub      esp,0D0h
002B1889 push     ebx
002B188A push     esi
002B188B push     edi
002B188C lea      edi,[ebp-10h]
002B188F mov      ecx,4
002B1894 mov      eax,0CCCCCCCCh
002B1899 rep stos  dword ptr es:[edi]
002B189B mov      eax,dword ptr [__security_cookie (02BA040h)]
002B18A0 xor      eax,ebp
002B18A2 mov      dword ptr [ebp-4],eax
002B18A5 mov      ecx,offset _689CA4AD_12 周课堂作业\12 周课堂作业\demo@c
(02BC008h)
002B18AA call     @_CheckForDebuggerJustMyCode@4 (02B1339h)
    int main_var = 1;
002B18AF mov      dword ptr [main_var],1
    printf("Inside main:\n");
002B18B6 push     offset string "Inside main:\n" (02B7B30h)
002B18BB call     _printf (02B10D7h)
002B18C0 add      esp,4
    printf(" Address of main_var: %p\n", (void*)&main_var);
002B18C3 lea      eax,[main_var]
002B18C6 push     eax
002B18C7 push     offset string " Address of main_var: %p\n" (02B7B94h)
002B18CC call     _printf (02B10D7h)
002B18D1 add      esp,8

    // 调用 first_function
    first_function(main_var);
002B18D4 mov      eax,dword ptr [main_var]
002B18D7 push     eax
002B18D8 call     _first_function (02B13E8h)
002B18DD add      esp,4

    return 0;
002B18E0 xor      eax,eax
}
002B18E2 push     edx
002B18E3 mov      ecx,ebp
002B18E5 push     eax
002B18E6 lea      edx,ds:[2B1914h]
002B18EC call     @_RTC_CheckStackVars@8 (02B11EFh)
002B18F1 pop      eax

```

002B18F2	pop	edx
002B18F3	pop	edi
002B18F4	pop	esi
002B18F5	pop	ebx
002B18F6	mov	ecx, dword ptr [ebp-4]
002B18F9	xor	ecx, ebp
002B18FB	call	@__security_check_cookie@4 (02B1154h)
002B1900	add	esp, 0D0h
002B1906	cmp	ebp, esp
002B1908	call	__RTC_CheckEsp (02B1258h)
002B190D	mov	esp, ebp
002B190F	pop	ebp
002B1910	ret	
002B1911	nop	dword ptr [eax]
002B1914	add	dword ptr [eax], eax
002B1916	add	byte ptr [eax], al
002B1918	sbb	al, 19h
002B191A	sub	eax, dword ptr [eax]
002B191C	hlt	
002B191D	??	??????
002B191E	??	??????
}		
002B191F	inc	dword ptr [eax+eax]
002B1922	add	byte ptr [eax], al
002B1924	sub	byte ptr [ecx], bl
002B1926	sub	eax, dword ptr [eax]
002B1928	ins	dword ptr es:[edi], dx
002B1929	popad	
002B192A	imul	ebp, dword ptr [esi+5Fh], 726176h

反汇编截图：

```

反汇编  x demo.c
地址(A): main(...)
查看选项
int main() {
002B1880 push     ebp
002B1881 mov      ebp, esp
002B1883 sub      esp, 0D0h
002B1889 push     ebx
002B188A push     esi
002B188B push     edi
002B188C lea      edi, [ebp-10h]
002B188F mov      ecx, 4
002B1894 mov      eax, 0CCCCCCCCh
002B1899 rep stos  dword ptr es:[edi]
002B189B mov      eax, dword ptr [__security_cookie (02BA040h)]
002B18A0 xor      eax, ebp
002B18A2 mov      dword ptr [ebp-4], eax
002B18A5 mov      ecx, offset _689CA4AD_12周课堂作业\12周课堂作业\demo@c (02BC008h)
002B18AA call     @__CheckForDebuggerJustMyCode@4 (02B1339h)
    int main_var = 1;
002B18AF mov      dword ptr [main_var], 1
    printf("Inside main:\n");
002B18B6 push     offset string "Inside main:\n" (02B7B30h)
002B18BB call     _printf (02B10D7h)
002B18C0 add      esp, 4
    printf("  Address of main_var: %p\n", (void*)&main_var);
002B18C3 lea      eax, [main_var]
002B18C6 push     eax
002B18C7 push     offset string "  Address of main_var: %p\n" (02B7B94h)
002B18CC call     _printf (02B10D7h)
002B18D1 add      esp, 8

    // 调用 first_function
    first_function(main_var);
002B18D4 mov      eax, dword ptr [main_var]
002B18D7 push     eax
002B18D8 call     _first_function (02B13E8h)
002B18DD add      esp, 4

```

反汇编解释:

1. 函数栈帧的初始化

```

002B1880 push     ebp
002B1881 mov      ebp, esp
002B1883 sub      esp, 0D0h

```

push ebp 和 mov ebp, esp: 保存上层函数的栈帧指针, 并为当前函数建立栈帧。

sub esp, 0D0h: 为局部变量分配栈空间 (0xD0 = 208 字节)。

2. 保存寄存器和初始化

```

002B1889 push     ebx
002B188A push     esi
002B188B push     edi
002B188C lea      edi, [ebp-10h]
002B188F mov      ecx, 4
002B1894 mov      eax, 0CCCCCCCCh
002B1899 rep stos  dword ptr es:[edi]

```

push 保存 ebx, esi, edi 等寄存器。

栈区域 [ebp-10h] 被用 0xCCCCCCC 填充，这是用于调试的标记值，表明该内存还未被使用。

3. 安全性检查

```
002B189B  mov     eax,dword ptr [__security_cookie]
002B18A0  xor     eax,ebp
002B18A2  mov     dword ptr [ebp-4],eax
```

用于防止 栈溢出攻击。通过 __security_cookie 校验栈帧的完整性。

4. 变量声明与操作

```
002B18AF  mov     dword ptr [main_var],1
```

将 main_var 初始化为 1，对应代码 int main_var = 1;。

5. 打印语句

```
002B18B6  push    offset string "Inside main:\n"
002B18BB  call    _printf
002B18C0  add     esp,4
002B18C3  lea     eax,[main_var]
002B18C6  push    eax
002B18C7  push    offset string "  Address of main_var: %p\n"
002B18CC  call    _printf
002B18D1  add     esp,8
```

使用 printf 打印字符串 Inside main:\n 和 main_var 的地址。

lea eax, [main_var]: 将 main_var 的地址加载到寄存器 eax。

6. 调用 first_function

```
002B18D4  mov     eax,dword ptr [main_var]
002B18D7  push    eax
002B18D8  call    _first_function
002B18DD  add     esp,4
```

将 main_var 的值压栈，并调用 first_function。

7. 返回值设置与清理栈

```
002B18E0  xor     eax,eax
002B190D  mov     esp,ebp
002B190F  pop     ebp
002B1910  ret
```

设置返回值为 0 (xor eax, eax)。

恢复栈帧指针和寄存器，返回调用者。