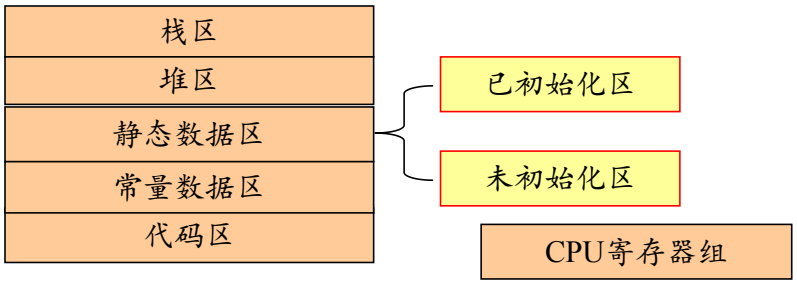
C 语言的深度挖掘 (二)

C程序中的内存管理问题

C/C++程序运行时的内存结构

- 未被作为初始化使用的常量和被const修饰的全局变量存储在常量数据区
- 全局变量、用static修饰的局部变量都存储在静态数据区。
- 程序指令和被作为初始化使用常量都存储在代码区。
- 大部分函数的形参和局部变量都存储在栈区。
- 程序中动态分配的内存都存储在堆区。
- 一小部分函数形参和局部变量存储在CPU寄存器组中。



代码区？数据区？

```
int add(int x, int y) {
   return x + y;
}
typedef int (*FP)(int, int);
int main( ) {
    unsigned char buff[1024];
    unsigned char *ps = (unsigned char*)add, *pd = buff;
    void *pp = buff;
    while(1) {
        *pd = *ps;
        if(*ps == 0xc3)  break;
        ps++; pd++;
    }
    FP fp = (FP)pp;
    printf("3 + 5 = %d\n", fp(3, 5) );
    return 0;
}
```

变量的生存期

- 把程序运行时一个变量占有内存空间的时间段称为该变量的**生存期**。C++把变量的生存期分为：**静态**、**自动**和**动态**三种。
- **静态生存期**：全局变量都具有静态生存期，它们的内存空间从程序开始执行时就进行分配，直到程序结束才被收回。
- **自动生存期**：局部变量和函数形参一般都具有自动生存期，它们的内存空间在程序执行到定义它们的复合语句(包括函数体)时才分配，当定义它们的复合语句执行结束时内存被收回。
- **动态生存期**：具有动态生存期的变量的生存时间是由程序员自由控制的，其内存空间用new操作符分配，用delete回收。
- 在定义局部变量时，可以为它们加上存储类修饰符auto、static和register来指出它们的生存期。
- 定义为static存储类型的局部变量具有静态生存期，它们也被存放在静态数据区。

关键字*volatile*的作用

```
int flag;

void onInterupt()
{
    flag = 0;
}

void thread1()
{
    flag = 1;
    while(flag)
    {
        // do something
    }
}
```

main函数为空居然也有输出?

```
#include <stdio.h>

int f(), g = f();
void main()
{
}
int f()
{
    printf("Hello world!\n");
    return 0;
}
```

关键字*extern*的作用

错了!

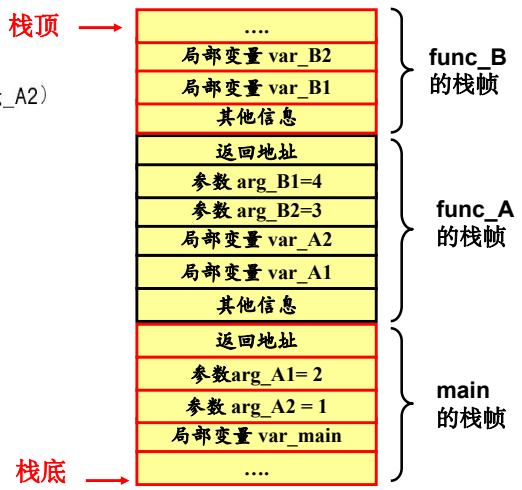
```
/*  a.cpp  */
extern int gv1;
extern char *ar1;
extern int fun(float, int);
int gv2 = 100;
char ar2[20];
void main()
{
    gv1 = 3, gv2 = 5;
    ar2[0] = 'A';
    fun(3, 5);
}
```

```
/*  b.cpp  */
extern int gv2;
extern char ar2[];
int gv1 = 89;
char ar1[] = "Hello!";

int fun(float fp, int ip)
{
    gv2 = 99;
    ar2[0] = 'B';
    /* ... ... */
}
```

系统栈与过程调用-win vc

```
int func_B(int arg_B1, int arg_B2)
{
    int var_B1, var_B2;
    return 0;
}
int func_A(int arg_A1, int arg_A2)
{
    int var_A1, var_A2;
    func_B(4, 3);
    return 0;
}
void main()
{
    int var_main;
    func_A(2, 1);
}
```



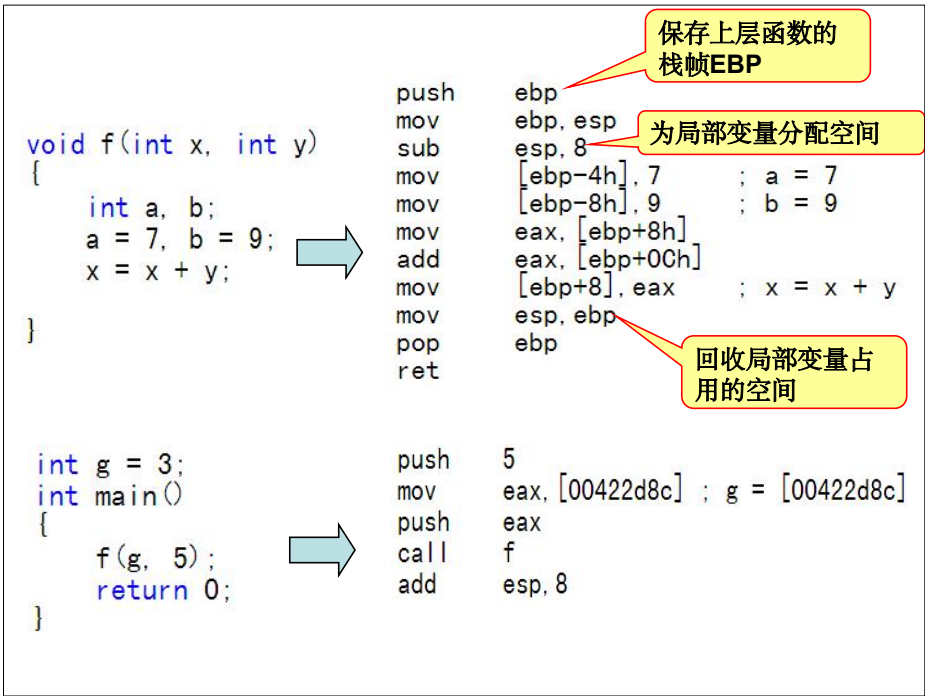
CPU对过程调用的支持

相关的寄存器：

1. ESP：存放一个指针，该指针指向系统栈最上面一个**栈帧**的栈顶，即整个系统栈的栈顶。
2. EBP：存放一个指针，该指针指向系统栈最上面一个**栈帧**的栈底，即当前栈帧的栈底。有时也被称为栈帧寄存器。
3. EIP：指令寄存器，存放一个指针，指向下一条等待执行的指令地址。

相关的机器指令：

push *operand*	**sub** *ESP*, 1 ; **mov** [*ESP*], *operand* ;
pop *operand*	**mov** *operand*, [*ESP*] ; **add** *ESP*, 1 ;
call *Label*	**push** *EIP* ; **jmp** *Label* ;
call *operand*	**push** *EIP* ; **jmp** *operand*;
ret	**pop** *EIP* ;



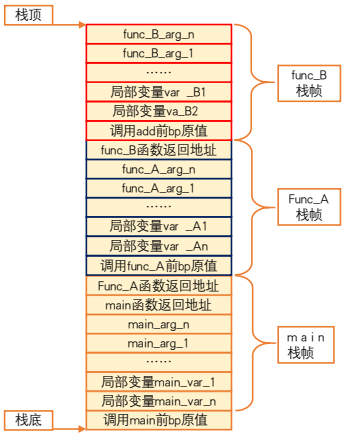
注：图中每个格都表示4个字节

一个小结论：

函数的参数都在EBP所指示的内存地址的正偏移处，函数内部的局部变量都在EBP所指示的内存地址的负偏移处。

系统栈与过程调用-Ubuntu gcc

```
int func_B(int arg_B1, int arg_B2)
{
    int var_B1, var_B2;
    return 0;
}
int func_A(int arg_A1, int arg_A2)
{
    int var_A1, var_A2;
    func_B(4, 3);
    return 0;
}
void main()
{
    int var_main;
    func_A(2, 1);
}
```



为什么C语言不支持这样的语法?

```
int f(int x, int y)
{
    int n = 100;
    int a[n];
    /*
    ...
    */
}
```

输出什么?

①

```
#include <stdio.h>
void f()
{
    int i, a[10];
    for(i=0; i<=10; i++)
    {
        a[i] = 3;
    }
}

int main()
{
    f();
    printf("main() function\n");
    return 0;
}
```

输出什么？

②

```
#include <stdio.h>

void f()
{
    int  a[10], i;
    for(i=0; i<=10; i++)
    {
        a[i] = 3;
    }
}
int main()
{
    f();
    printf("main() function\n");
    return 0;
}
```

输出什么？

③

```
#include <stdio.h>

void f()
{
    char i, a[10];
    for(i=0; i<=10; i++)
    {
        a[i] = 3;
    }
}
int main(int argc, char* argv[])
{
    f();
    printf("main function()\n");
    return 0;
}
```

常量成了变量?

```
#include <stdio.h>

void f()
{
    const float PI = 3.14259;
    int a[10];
    a[10] = 0;
    printf("%f\n", PI);
}
int main()
{
    f();
    printf("main() function\n");
    return 0;
}
```

变量可见性与生存期的区别

```
void f()
{
    int a = 3, *p = NULL;
    if(3==a)
    {
        int b = 5;
        p = &b;
    }
    int c = 7;
    printf("%d\n", *p);
}

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

如何攻破密码验证程序?

```
#define PASSWORD "1234567"
int verify_password(char *password) {
    int valid_flag = 0;
    char buffer[8];
    valid_flag = strcmp(password, PASSWORD);
    strcpy(buffer, password);
    return valid_flag;
}
void main() {
    char inputstr[1024];
    for(int i=0; i<3; i++) {
        printf("Please input password: ");
        scanf("%s", inputstr);
        if(verify_password(inputstr)==0)
        {
            printf("You have passed the verification!\n\n");
            break;
        }
        else
            printf("Incorrect password!\n\n");

    }
}
```

程序“飞了”

```
#include <stdio.h>

void g()
{
    printf("g() function\n");
}
void f()
{
    int a[10];
    a[11] = (int)g;
}
int main()
{
    f();
    printf("main() function\n");
    return 0;
}
```

程序又“飞了”

```
#include <stdio.h>

void g()
{
    printf("g() function\n");
}
void f(int x, int y)
{
    *(&x-1) = (int)g;
}
int main()
{
    f(3, 5);
    printf("main() function\n");
    return 0;
}
```

程序“飞”到哪儿了？

```
void g()
{
    printf("g() function\n");
}
void f()
{
    int a[10];
    a[12] = a[11];
    a[11] = (int)g;
}
int main(int argc, char* argv[])
{
    int a[10];
    f();
    __asm { sub esp, 4 }
    printf("main() function!\n");
    return 0;
}
```

关于缓冲区溢出攻击

请参阅论文：

《Smashing The Stack For Fun And Profit 》

有安全漏洞的程序

```
#include <stdio.h>

int login()
{
    char buff[100];
    printf("Pleas input your password:");
    scanf("%s", buff);
    .....
    return 0;
}
```

如何在栈上动态分配内存?

```
/*Allocates memory on the stack.
  <malloc.h>                        */

void *_alloca( size_t size );

#include <malloc.h>

void f()
{
    char *p;
    p = (char*)_alloca(20);
    memcpy(p, "Hello World!", 13);
    printf("%s\n", p);
}
int main()
{
    f();
    return 0;
}
```

有错吗?

```
#include <stdio.h>
#include <stdlib.h>
void getmemory(char *p)
{
    p = (char *) malloc(100);
    strcpy(p, "hello world");
}
int main( )
{
    char *str = NULL;
    getmemory(str);
    printf("%s/n", str);
    free(str);
    return 0;
}
```

有问题吗?

```
#include <stdio.h>

char *GetMemory()
{
    char p[] = "Hello World!";
    return p;
}

int main()
{
    char *str = GetMemory();
    printf(str);
    return 0;
}
```

有问题吗?

```
#include <stdio.h>

char *GetMemory()
{
    char *p = "Hello World!";
    return p;
}

int main()
{
    char *str = GetMemory();
    printf(str);
    return 0;
}
```

C与汇编的混合编程(win)

```
#include <stdio.h>

int g = 0;
void f(int x, int y)
{
    printf("x=%d, y=%d\n", x, y);
}
int main()
{
    int a, b;
    __asm {
        mov dword ptr [g], 3
        mov dword ptr [a], 4
        mov dword ptr [ebp-8], 5
        push 7
        push 6
        call f
        add esp, 8
    }
    printf("g=%d, a=%d, b=%d\n", g, a, b);
    return 0;
}
```

C与汇编的混合编程(Linux)

```
1 /*inline.c*/
2 #include<stdio.h>
3 int g = 0;
4 void f(int x, int y)
5 {printf("x = %d, y = %d\n",x,y);}
6 int main()
7 {
8    int a = 10, b;
9    __asm__ __volatile__("movl %2, %%eax;"  //b = a + 100
10          "add $100, %%eax;"
11          "movl %%eax, %0;"
12          "movl $5, %%eax;"              // g = 5
13          "movl %%eax, %1;"
14          "movl %0, %%edi;"              //f(b,g)
15          "movl %2, %%esi;"
16          "callq f;"
17          :"=m"(b),"=r"(g)               //output
18          :"r"(a)                        //input
19          :"%eax","%edi","%esi");        //modify
20    printf("Result a = %d, b = %d, g = %d\n",a,b,g);
21    return 0;
22 }
```

输出什么?

```
int main()
{
    char *str = "Hello World!";
    str[0] = 'h';
    printf("%s\n", str);
    return 0;
}
```

存储位置是否相同?

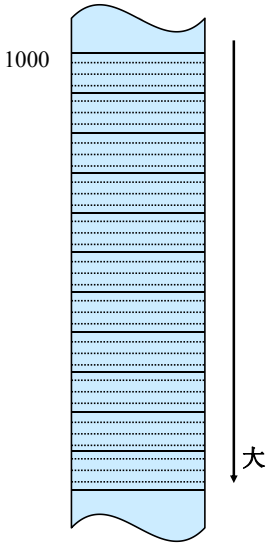
```
char *str1 = "Hello World!";
int gval = 10;
int main()
{
    char *str2 = "Hello World!";
    printf("%u, %u \n", str1, str2);
    return 0;
}
```

Calling Conventions

方式	编译开关	参数传递方式	谁负责清栈	C修饰名称
__cdecl	/Gd	从右向左压栈	函数调用者	_function
__stdcall	/Gz	从右向左压栈	被调用函数	_function@number
__fastcall	/Gr	前两个参数通过ECX和EDX两个寄存器传送，其它参数从右向左压栈	被调用函数	@funciton@number
__pacal		从左向右压栈	被调用函数	
thiscall		从右向左压栈，this指针存放在ECX中		
WINAPI CALLBACK APIENTRY PASCAL	同__stdcall方式			

堆内存管理方法初探

```
int *p1 = (int*) malloc(sizeof(int));
char *p2 = (char*) malloc(sizeof(char));
……
free(p1) ;
free(p2);
```



自由内存区表

基地址	长度
1000	100

占用内存区表

基地址	长度

堆内存管理方法初探

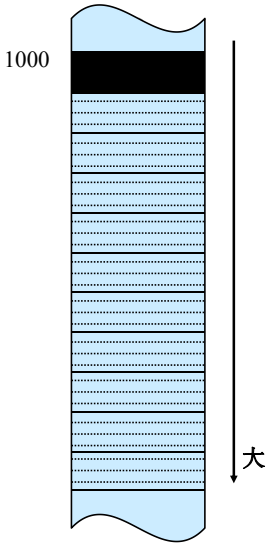
```
int *p1 = (int*) malloc(sizeof(int));
char *p2 = (char*) malloc(sizeof(char));
……
free(p1) ;
free(p2);
```

自由内存区表

基地址	长度
1004	96

占用内存区表

基地址	长度
1000	4



堆内存管理方法初探

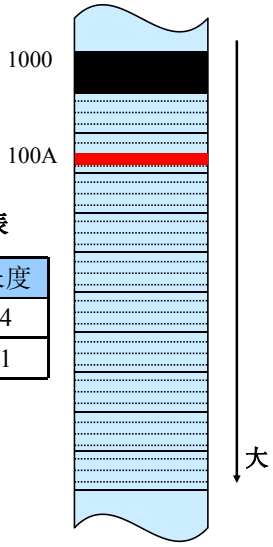
```
int *p1 = (int*) malloc(sizeof(int));
char *p2 = (char*) malloc(sizeof(char));
……
free(p1) ;
free(p2);
```

自由内存区表

基地址	长度
1004	6
100B	89

占用内存区表

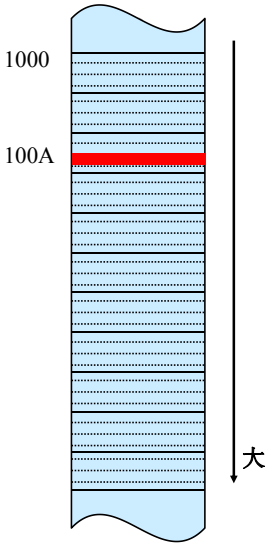
基地址	长度
1000	4
100A	1



- 最先适配算法
- 最佳适配算法

堆内存管理方法初探

```
int *p1 = (int*) malloc(sizeof(int));
char *p2 = (char*) malloc(sizeof(char));
……
free(p1) ;
free(p2);
```



自由内存区表

基地址	长度
1000	10
100B	89

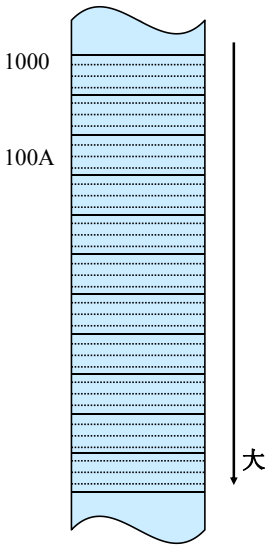
占用内存区表

基地址	长度
~~1000~~	~~4~~
100A	1

堆的紧缩问题

堆内存管理方法初探

```
int *p1 = (int*) malloc(sizeof(int));
char *p2 = (char*) malloc(sizeof(char));
……
free(p1) ;
free(p2);
```



自由内存区表

基地址	长度
1000	100

占用内存区表

基地址	长度
~~1000~~	~~4~~
~~100A~~	~~1~~

使用malloc和free的注意事项

1. 刚刚分配的动态内存的初始值是不确定的
2. 不能对同一指针(地址) 连续两次进行free操作
3. 不能对指向静态内存区(全局变量)或栈内存区(局部变量)的指针应用free (但可以对空指针NULL应用free)。
4. 对一个指针应用free之后，它的值不会改变，但它指向了一个无效的内存区，这时称该指针为“悬空指针”。
5. 如果没有及时释放某块动态内存，并且将指向它的指针指向了别处，就会造成“内存泄漏”。
6. 执行malloc和free函数有一定的代价，所以对于较小的变量不应该放在动态内存之中，并且尽量避免频繁地分配和释放动态内存。

使用堆内存时的常见错误

1. 内存分配未成功，却使用了它。
2. 内存分配虽然成功，但是尚未初始化就引用它。(误认为初始值为0)
3. 内存分配成功并且已经初始化，但操作越过了内存的边界。
4. 忘记了释放内存，造成内存泄露。
5. 释放了内存却继续使用它。

关于悬空指针

- 一个指针变量，如果不为NULL且没有指向有效的内存地址，都称为“悬空指针”
- 通过悬空指针访问其指向的内存区会使程序产生不可预知的错误。
- 如何避免悬空指针：
  - 定义指针变量时坚持对其进行正确的初始化
  - 在用free或delete释放内存之后，应及时将相应的指针置为NULL

悬空指针的例子(一)

```
void somefuncion()
{
        int *p;
        … … … …
        *p = 7;
        … … … …
}
```

```
void somefuncion()
{
        int *p = NULL;   //正确地进行初始化
        … … … …
        *p = 7;
        … … … …
}
```

悬空指针的例子(二)

```
int main()
{
        int *p = NULL;
        p = (int*)malloc(sizeof(int));
        *p = 5;
        free(p);
        // …  do something
        *p = 7;
        printf("%d", *p);
        … …
}
```

```
free(p);
p = NULL;
```

内存泄漏的例子(一)

```
void MyFunction(int nSize)
{
    char* p= new char[nSize];
    if( !SomeFunc() ){
        printf("Error");
        return;
    }
    …//using the string pointed by p;
    delete p;
}
```

内存泄漏的例子(二)

```
char *TransToEng(const char *inputStr)  // 将中文翻译成英文
{
        char *outputStr = (char*) malloc(… …);
        … … /* 翻译 */
        return outputStr;
}

int main()
{
        char *chineseStr = "欢迎光临";
        char *englishStr = TransToEng("欢迎光临");
        printf("%s", englishStr);
}
```

如何避免内存泄漏

1. 运行检测法
   - 定义自己的malloc和free函数，或者对new和delete进行重载，在运行时跟踪记录动态内存的分配和释放情况
   - 利用专用的检测工具，如BoundsChecker、Purify和Performance Monitor
2. 利用复杂的程序设计技术(C++)
   - 智能指针技术
   - 为C++增加垃圾回收机制(可参考《C++编程艺术》艺术)