# 汇编语言课程作业

**————报告5**

2251334 倪朗恩

## 存储区域划分

使用block.c进行验证,验证了局部变量、全局变量、两种静态变量、函数等的存储地址。

#include<stdio.h>

#include<stdlib.h>

int global\_var;

int init\_global\_var = 0;

static int static\_global\_var;

void function()

{

    return;

}

int main()

{

    int local\_var;

    static int static\_local\_var;

    int\* heap\_var = (int\*)malloc(sizeof(int));

    printf("local\_var: %p\n", &local\_var);

    printf("static\_local\_var: %p\n", &static\_local\_var);

    printf("heap\_var: %p\n", heap\_var);

    printf("uninitialized global\_var: %p\n", &global\_var);

    global\_var = 0;

    printf("initialized global\_var: %p\n", &global\_var);

    printf("preinitialized global\_var: %p\n", &init\_global\_var);

    printf("static\_global\_var: %p\n", &static\_global\_var);

    printf("function: %p\n", function);

    free(heap\_var);

    return 0;

}

## 栈大小

使用stack.cpp进行验证，递归创建数组，不断输出，进行快速的估计。（环境为VSCode）

#include<iostream>

using namespace std;

void testStack()

{

    int i[1024];

    cout << i << endl;

    try{

        testStack();

    }

    catch(...){

        cout << "proximate stack size: "<< i;

    }

    return;

}

int main()

{

    testStack();

    system("pause");

    return 0;

}

使用C++代码是为了更好的处理栈溢出error，开1024大小的int数组是速度和精度的权衡。

## 嵌套

使用了网站compiler explorer对代码recursive.c进行验证，结果保存在compile.asm。

#include<stdio.h>

int interFunction(int i, int j)

{

    int k = i + j;

    return k;

}

int outerFunction(int i, int j)

{

    int k = interFunction(i, j);

    return k;

}

int main()

{

    int i = 10, j = 20;

    int k = outerFunction(i, j);

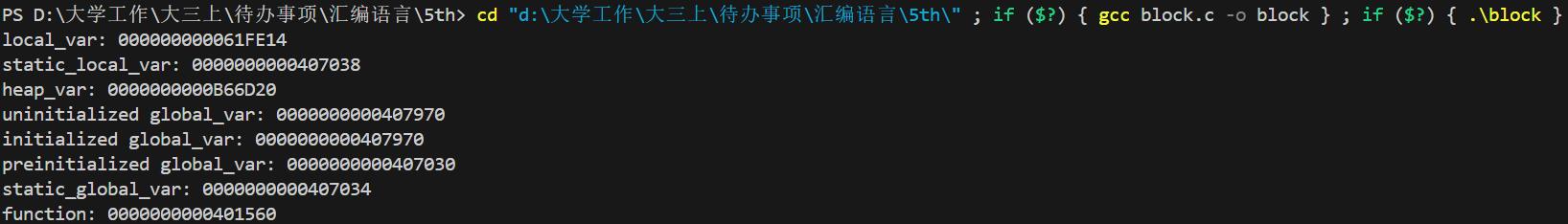
    printf("%d", k);

    return 0;

}

## 实验结果

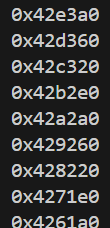
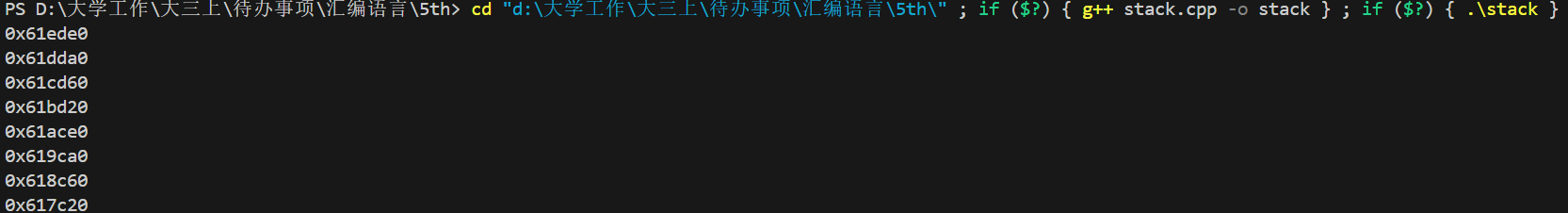
1. 变量存储空间



* **堆变量 (heap\_var)**:
  + 动态分配在堆中。
  + 输出地址为 0000000000716D20。
* **局部变量 (local\_var)**:
  + 存储在栈中。
  + 输出地址为 000000000061FE14。
* **未初始化的全局变量 (global\_var)**:
  + 存储在 BSS 段（未初始化的全局数据区）。
  + 输出地址为 0000000000407970。
* **已初始化的全局变量 (init\_global\_var)**:
  + 存储在数据段（已初始化的全局数据区）。
  + 输出地址为 0000000000407030。
* **静态局部变量 (static\_local\_var)**:
  + 存储在全局数据段（静态数据区）。
  + 输出地址为 0000000000407038。
* **静态全局变量 (static\_global\_var)**:
  + 存储在 BSS 段（未初始化的全局数据区）。
  + 输出地址为 0000000000407034。
* **函数地址 (function)**:
  + 存储在代码段（文本段）。
  + 输出地址为 0000000000401560。

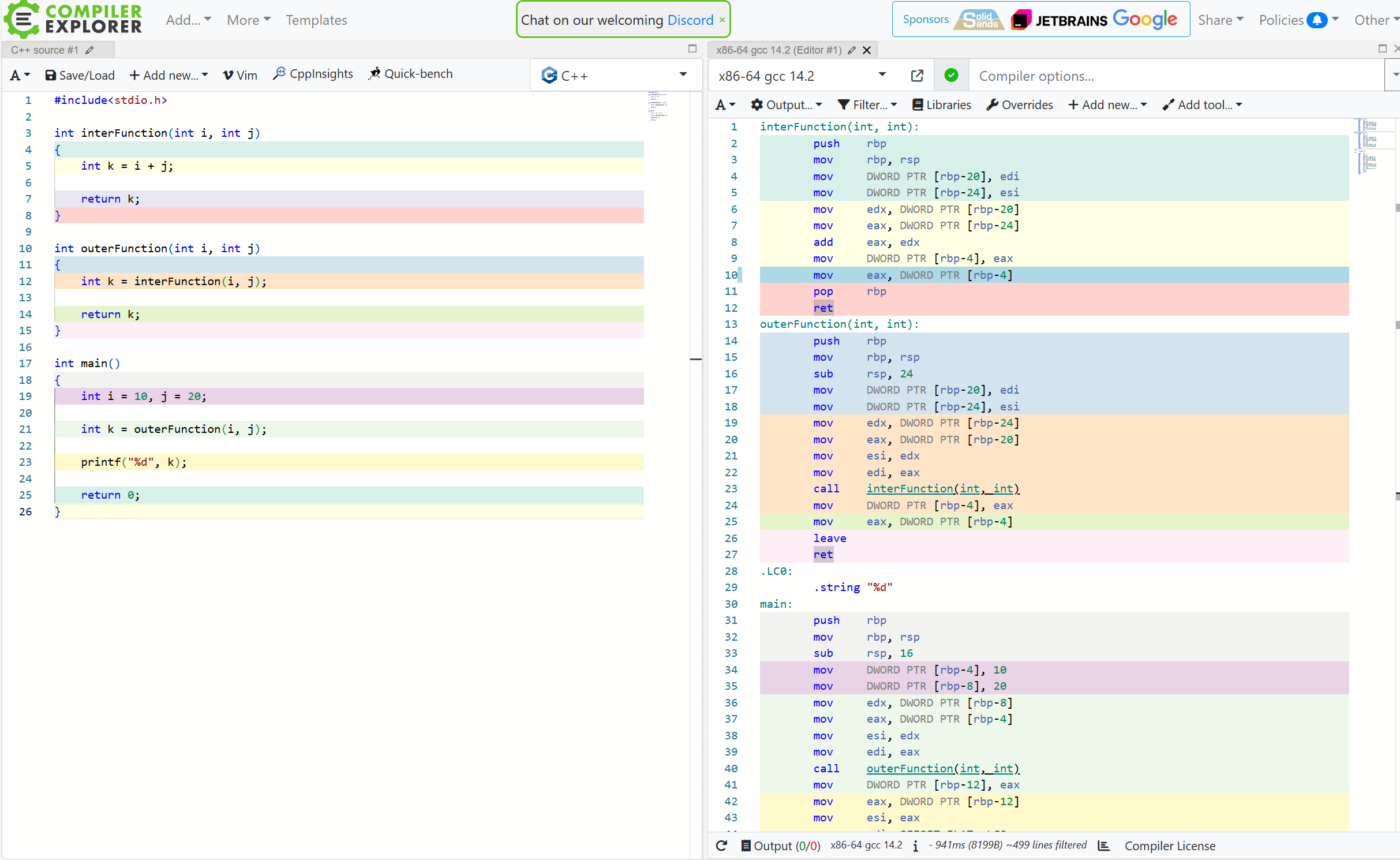
在当前运行环境下（Windows VSCode），显然有以下结论：

1. 函数/代码段存储在最低位
2. 初始化/实例化的全局变量，与静态变量（静态全局与静态局部），存储在同一区域，地址高低依照定义顺序，地址高于代码段
3. 已定义但未赋值的全局变量，存储位置略高于静态变量段，赋值后地址不变
4. 局部变量存储在栈，地址高于未赋值的全局变量段
5. 申请变量位于堆，经过多次尝试，发现每次堆地址都有明显变化，但基本有结论存储地址最高
6. 栈大小



大致估计为0x1F2a80=2042496Byte~2MB，且可以观察到：

1. 栈从大到小被分配
2. 每次调用该函数大概要额外消耗64字节
3. 嵌套函数编译



interFunction(int, int):

        push    rbp

        mov     rbp, rsp

        mov     DWORD PTR [rbp-20], edi

        mov     DWORD PTR [rbp-24], esi

        mov     edx, DWORD PTR [rbp-20]

        mov     eax, DWORD PTR [rbp-24]

        add     eax, edx

        mov     DWORD PTR [rbp-4], eax

        mov     eax, DWORD PTR [rbp-4]

        pop     rbp

        ret

outerFunction(int, int):

        push    rbp

        mov     rbp, rsp

        sub     rsp, 24

        mov     DWORD PTR [rbp-20], edi

        mov     DWORD PTR [rbp-24], esi

        mov     edx, DWORD PTR [rbp-24]

        mov     eax, DWORD PTR [rbp-20]

        mov     esi, edx

        mov     edi, eax

        call    interFunction(int, int)

        mov     DWORD PTR [rbp-4], eax

        mov     eax, DWORD PTR [rbp-4]

        leave

        ret

.LC0:

        .string "%d"

main:

        push    rbp

        mov     rbp, rsp

        sub     rsp, 16

        mov     DWORD PTR [rbp-4], 10

        mov     DWORD PTR [rbp-8], 20

        mov     edx, DWORD PTR [rbp-8]

        mov     eax, DWORD PTR [rbp-4]

        mov     esi, edx

        mov     edi, eax

        call    outerFunction(int, int)

        mov     DWORD PTR [rbp-12], eax

        mov     eax, DWORD PTR [rbp-12]

        mov     esi, eax

        mov     edi, OFFSET FLAT:.LC0

        mov     eax, 0

        call    printf

        mov     eax, 0

        leave

        ret

根据提供的汇编代码，我们可以逐步解析每个函数的作用和执行流程。

**1. interFunction(int, int)**

（1）push rbp 和 mov rbp, rsp

设置栈帧基址

（2）mov DWORD PTR [rbp-20], edi

mov DWORD PTR [rbp-24], esi

将传入的两个参数（edi 和 esi）保存到栈中

（3）mov edx, DWORD PTR [rbp-20]

mov eax, DWORD PTR [rbp-24]

从栈中读取这两个参数

（4）add eax, edx

将两个参数相加，结果存放在 eax 中

（5）mov DWORD PTR [rbp-4], eax

将结果保存到栈中的一个临时变量

（6）mov eax, DWORD PTR [rbp-4]

将结果从栈中读回 eax

（7）pop rbp 和 ret

恢复栈帧并返回

**2. outerFunction(int, int)**

（1）push rbp 和 mov rbp, rsp

设置栈帧基址

（2）sub rsp, 24

为局部变量分配栈空间

（3）mov DWORD PTR [rbp-20], edi

mov DWORD PTR [rbp-24], esi

将传入的两个参数（edi 和 esi）保存到栈中

（4）mov edx, DWORD PTR [rbp-24]

mov eax, DWORD PTR [rbp-20]

从栈中读取这两个参数

（5）mov esi, edx

mov edi, eax

将参数传递给 interFunction

（6）call interFunction(int, int)

调用 interFunction

（7）mov DWORD PTR [rbp-4], eax

将 interFunction 的结果保存到栈中的一个临时变量

（8）mov eax, DWORD PTR [rbp-4]

将结果从栈中读回 eax

（9）leave 和 ret

恢复栈帧并返回

**3. main**

（1）push rbp

mov rbp, rsp

设置栈帧基址

（2）sub rsp, 16

为局部变量分配栈空间

（3）mov DWORD PTR [rbp-4], 10

mov DWORD PTR [rbp-8], 20

初始化两个整数变量 10 和 20

（4）mov edx, DWORD PTR [rbp-8]

mov eax, DWORD PTR [rbp-4]

从栈中读取这两个整数

（5）mov esi, edx

mov edi, eax

将这两个整数传递给 outerFunction

（6）call outerFunction(int, int)

调用 outerFunction

（7）mov DWORD PTR [rbp-12], eax

将 outerFunction 的结果保存到栈中的一个临时变量

（8）mov eax, DWORD PTR [rbp-12]

将结果从栈中读回 eax

（9）mov esi, eax

mov edi, OFFSET FLAT:.LC0

准备调用 printf，格式字符串为 "%d"

（10）mov eax, 0

call printf

调用 printf 打印结果

（11）mov eax, 0

设置返回值为 0

（12）leave 和 ret

恢复栈帧并返回

.string “%d“个人猜测为可执行文件传入参数

## 问题与心得

1. 可以在Ubuntu系统上（本人有双系统）实验，使用不同IDE（VS2022）实验，探索heap等其他存储空间你大小，但由于时间问题，没有实施
2. 更好的了解了系统的配置