1：

介绍栈

1.1：综述内存存放方式【1】

《编译原理》7.1 写道：

内存区域有五个：代码区，静态区，堆区，栈区，空闲内存，其关系如下：

<table>
<tr><td>栈区</td></tr>
<tr><td>↓
空闲内存
↑</td></tr>
<tr><td>堆区</td></tr>
<tr><td>静态区</td></tr>
<tr><td>代码区</td></tr>
</table>

代码区存储可执行目标代码

静态区存储编译时就能确定的数据，比如全局常量和编译器产生的数据

堆和栈区是动态区，其大小可以随着程序运行改变。

堆区一般用于管理有长生命周期的数据，比如对象。New+delete,malloc+free 都可以用来管理堆区数据

在很多编程语言中，栈区域被用于管理使用过程，函数或方法的运行时刻存储。每当一个过程被调用，存放该过程局部变量的空间被压入栈。当这个过程结束时，该空间被弹出这个栈。

举例：

```
 void Frog()
{
char s=’a’;
}
```

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

在这个程序里，执行 Frog 函数时，Frog 函数的局部变量 s 被压入栈。待 Frog 函数执行完，s被压出栈。在递归调用函数的时候，栈内存分配遵循“先进后出”原则。关于栈的性质，后续我会详细验证。

1.2 栈的性质【2】：

栈的空间是连续的，一般情况下，栈顶地址比栈底地址高。我的程序 1（stack_test）测试了这部分内容。

栈的空间上限默认为 1M，不同编译器栈上限大小可能不同（程序 2 测试了这部分内容）。但是可以修改（静态，动态两种方法）。我的程序 3（stack_get）部分将详细介绍这部分内容。

栈空间是系统自动在执行时分配，如果没有足够的空间，将会报错 stack overflow。如果函数退栈或者释放局部变量，其栈空间也会被释放。相比堆，栈存储效率高。

举例：如果栈顶的内存地址为 1000，执行一次函数需要 200 的内存，那么执行一次函数后（还没释放内存时）消耗了 200 的内存，当前栈地址变为 800。

而执行一次函数后，再递归执行一次函数，此时消耗了总共 400 的内存，当前栈地址变为 600.如果第二次执行结束，函数退栈一次，会释放 200 的栈内存，栈地址变回 800.如果函数全部退栈，那么栈地址变回 1000.

如果栈底（上限）的内存地址为 1，那么栈内存上限就是 999，递归执行 5 次函数后，栈地址变为 0,0<1，就会爆栈，此时系统会抛出 stack overflow 异常。

2：程序 1 stack_test 项目

测试栈空间的连续性，以及递归，退栈对栈空间大小的影响

2.1 代码部分

原理：进行函数反复递归，直到爆栈。

首先每次递归记录一下函数最后一个变量的地址，计算相邻两次递归的尾地址之差。如果差始终相等，则可以说明栈地址连续。

如果栈地址连续，计算爆栈时函数的尾地址和栈首，如果始终近似相等，则可以说明（在不进行修改的情况下）栈上限大小相同。

代码如下

文件 stack_test.h:

```
#ifndef STACK_TEST_H
#define STACK_TEST_H
class stack_test
{
public:
        int mode, num;//mode1:执行1到爆栈，2:执行2到爆栈,3：执行3到爆栈
                      //-1：执行1到20退栈，-2执行-2到20退栈，-3：执行-3到20退栈
        double sum; //总共内存大小
        int dsum; //一个函数的内存大小
        char* start,*endline,*last_end; //分别为栈首地址，目前调用的函数的栈尾地址，上一次调用函数的栈尾地址
        stack_test(int i):mode(i),num(0),start(NULL),endline(NULL),last_end(NULL),dsum(0),sum(0){}
        ~stack_test(){}
        void control(); //主控界面
        void swing1();  //递归函数1
        void swing2();  //递归函数2
        void swing3();  //递归函数3
        void moveon();  //控制递归与退栈
};
#endif
```

文件 stack_test.cpp:

```
#include<iostream>
#include<fstream>
#include<iomanip>
#include<Windows.h>
#include"stack_test.h"
using namespace std;
ofstream out("data_out.txt");
void stack_test::control()
{
        if (mode == 1||mode==-1) swing1();
        if (mode == 2||mode==-2) swing2();
        if (mode == 3 || mode == -3) swing3();
}
void stack_test::swing1()
```

```
{
	char k='a';//定义一个1B的变量来获取当前内存地址
	if (num == 0) //第一次执行函数，初始化地址
	{
		start = &k;
		last_end = &k;
		out << "The start place is " << long long(start) << endl;
	}
	else
	{
		if (num > 1)last_end = endline; //
		endline = &k;
		dsum = last_end - endline ; //这个函数的栈内存大小
		sum = double(start - endline)/ 1024 / 1024;  //总共用的栈内存大小
		out << "The " << num << " th set" << endl;
		out << "Present place is " << long long(endline)<<endl;
		out << "The memory for this set is " << dsum  << " B" << endl;
		out << "Having used stack for " << sum << setprecision(2) << " MB" <<
endl;
		out << "-----------------------------------------------" << endl;
	}
	num++;
	moveon();
	//退栈的一系列操作
	num--;
	endline = &k;
	sum = double(start - endline) / 1024 / 1024;
	out << "Withdraw to the " << num << " th set" << endl;
	out << "Present place is " << long long(endline) << endl;
	out << "Having used stack for " << sum << setprecision(2) << " MB" << endl;
	out << "-----------------------------------------------" << endl;
}
//swing2,swing3和1操作大体相同，不再赘述
void stack_test::swing2()
{
	int map[1000];
	char k='a';
	if (num == 0)
	{
		start = &k;
		last_end = &k;
		out << "The start place is " << long long(start) << endl;
	}
	else
	{
		if (num > 1)last_end = endline;
		endline = &k;
		dsum = last_end - endline ;
		sum = double(start - endline)  / 1024 / 1024;
		out << "The " << num << " th set" << endl;
		out << "Present place is " << long long(endline) << endl;
		out << "The memory for this set is " << dsum << " B" << endl;
		out << "Having used stack for " << sum << setprecision(2) << " MB" <<
endl;
		out << "-----------------------------------------------" << endl;
	}
```

```
        num++;
        moveon();
        num--;
        endline = &k;
        sum = double(start - endline) / 1024 / 1024;
        out << "Withdraw to the " << num << " th set" << endl;
        out << "Present place is " << long long(endline) << endl;
        out << "Having used stack for " << sum << setprecision(2)<< " MB" << endl;
        out << "-----------------------------------------------" << endl;
}
void stack_test::swing3()
{
        int map[300];
        char k='a';
        if (num == 0)
        {
                start = &k;
                last_end = &k;
                out << "The start place is " << long long(start) << endl;
        }
        else
        {
                if (num > 1)last_end = endline;
                endline = &k;
                dsum = last_end - endline;
                sum = double(start - endline)  / 1024 / 1024;
                out << "The " << num << " th set" << endl;
                out << "Present place is " << long long(endline) << endl;
                out << "The memory for this set is " << dsum << " B" << endl;
                out << "Having used stack for " << sum << setprecision(2) << " MB" <<
endl;
                out << "-----------------------------------------------" << endl;
        }
        num++;
        moveon();
        num--;
        endline = &k;
        sum = double(start - endline) / 1024 / 1024;
        out << "Withdraw to the " << num << " th set" << endl;
        out << "Present place is " << long long(endline) << endl;
        out << "Having used stack for " << sum << setprecision(2) << " MB" << endl;
        out << "-----------------------------------------------" << endl;
}
void stack_test::moveon()
{
        if (mode < 0 && num >= 20) return;
        if (mode == 1||mode==-1) swing1();
        if (mode == 2||mode==-2) swing2();
        if (mode == 3||mode==-3) swing3();
}
```

文件 main

```
#include<iostream>
#include<fstream>
#include"stack_test.h"
```

```
using namespace std;
#pragma comment(linker, "/STACK:1048576,1048576")
int main(int argv, char* argc[])
{
        int n;
        cout << "Please input an integer of 1,2,3,-1,-2,-3 as the mode" << endl;
        cin >> n;

        stack_test test(n);
        test.control(); //先退栈模式
        test.mode = 0 - n;
        test.control(); //如果没爆栈，再来一次不退栈模式
        return 0;
}
```

对比只执行一个函数直到爆栈，以及执行这个函数到 20 次，之后再退栈到 0，之后再执行到爆栈的栈内存情况，如果二者内存情况近似相同，则说明退栈对栈内存上限和连续性没影响。

对比一直执行三个不同函数直到爆栈的结果，如果都近似相同，则说明不管怎么执行函数，栈内存上限和连续性都不变。

说明：函数 swing1,swing2,swing3 是三个内存大小不同的递归执行函数

2.2 测试部分

详细的测试文件在 result 文件夹里

Result1:输入指令 1，一直执行 swing1 直到爆栈

Result2:输入指令 2，一直执行 swing2 直到爆栈

Result3:输入指令 3，一直执行 swing3 直到爆栈

Result-1:输入指令-1，执行 swing1，之后退栈，再执行直到爆栈

Result-2:输入指令-2，执行 swing2，之后退栈，再执行直到爆栈

Result-3:输入指令-3，执行 swing3，之后退栈，再执行直到爆栈

测试结果：

数据	栈首地址	栈尾地址	栈内存上限	每个函数内存大小
Result1	370094109380	370093089300	0.97MB	656B

Result2	246175232532	246174219156	0.97MB	6032B
Result3	593402787988	593401767860	0.97MB	2272B
Result-1	299674039924	299673019188	0.97MB	656B
Result-2	220917788004	220916774628	0.97MB	6032B
Result-3	593402787988	593401767860	0.97MB	2272B

而且，对于三组退栈的数据，其退栈完毕时，已用栈内存都是 0,

2.3 结论:

每组数据的栈首，栈尾地址都不一样，说明内存分配的栈首位地址是随机的

每组数据的栈首地址都>栈尾地址，说明栈内存分配是由上到下的

每组数据，执行同一个函数，消耗的内存大小，也就是地址变化量大小，都是一样的，说明栈内存是一个一个连续的块。但是值得指出的是，每个块的内存大小比我定义的局部变量大不少，这个在我的队友夏天航同学的文档里有解释。

每组数据，爆栈时已用的栈内存都是 0.97MB，说明在不修改内存上限的情况下，同一个编译器的栈内存大小上限恒定

3：程序 2 stack_different_OS 项目

测试不同操作系统和编译器下默认栈空间上限大小

3.1 代码部分

代码思路与程序 1 完全相同，只是在不同编程环境，操作系统（win10 下的 VS2017,MacOS 下的 Clang,Ubantu16.04 下的 GDB）进行测试

代码如下

```
#include<iostream>
#include<fstream>
using namespace std;
char* start = NULL;
char* endline = NULL;
char* last_end = NULL;
double sum = 0;
double dsum = 0;
```

```
ofstream out("data_out.txt");
void swing1(int i,bool mode);
void swing2(int i,bool mode);
void swing3(int i,bool mode);
int main(int argv,char* argc[])
{
	int n;
	cout << "Please input an integer of 0,1,2 or 3 as the mode" << endl;
	cin >> n;
	if (n == 1)
	{
		while (1)
		{
			swing1(0,1);
		}
	}
	if (n == 2)
	{
		while (1)
		{
			swing2(0,1);
		}
	}
	if (n == 3)
	{
		while (1)
		{
			swing3(0,1);
		}
	}
	if (n == 0)
	{
		swing1(0,0);
	}
	return 0;
}
void swing1(int i,bool mode)
{
	char ii='a';
	if (i == 0)
	{
		start = &ii;
		last_end = &ii;
	}
	else
	{
		if (i > 1)last_end = endline;
		endline = &ii;
		dsum = double(last_end - endline) ;
		sum = double(start - endline)  / 1024 / 1024;
		out << "The " << i << " th set" << endl;
		out << "The memory for this set is " << dsum << " B" << endl;
		out << "Having used stack for " << sum << " MB" << endl;
		out << "-----------------------------------------------" << endl;
	}
	i++;
```

```
        if (mode == 1) swing1(i,1);
        else swing2(i,0);
}
void swing2(int i,bool mode)
{
        int map[1000];
        char ii='a';
        if (i == 0)
        {
                start = &ii;
                last_end = &ii;
        }
        else
        {
                if (i > 1)last_end = endline;
                endline = &ii;
                dsum = double(last_end - endline);
                sum = double(start - endline) / 1024 / 1024;
                out << "The " << i << " th set" << endl;
                out << "The memory for this set is " << dsum << " B" << endl;
                out << "Having used stack for " << sum << " MB" << endl;
                out << "-------------------------------------------------" << endl;
        }
        i++;
        if (mode == 1) swing2(i, 1);
        else swing3(i,0);
}
void swing3(int i,bool mode)
{
        char map[1000];
        char ii='a';
        if (i == 0)
        {
                start = &ii;
                last_end = &ii;
        }
        else
        {
                if (i > 1)last_end = endline;
                endline = &ii;
                dsum = double(last_end - endline) ;
                sum = double(start - endline) / 1024 / 1024;
                out << "The " << i << " th set" << endl;
                out << "The memory for this set is " << dsum << " B" << endl;
                out << "Having used stack for " << sum << " MB" << endl;
                out << "-------------------------------------------------" << endl;
        }
        i++;
        if (mode == 1) swing3(i, 1);
        else swing1(i,0);
}
```

在不同编程环境下分别输入 0，1,2,3，并分别收取输出文档即可

3.2 测试部分

在 win10 的 VS2017 下，栈空间上限大小在 0.97MB 左右，在 linux 的 gdb 下，栈空间上限大小在 8MB 左右，在 MacOS 的 Clang 下,栈空间上限大小在 8MB 左右。测试文件在项目的 result 文件夹里。

3.3 结论

不同版本的 VS 栈空间大小上限大致相同，不同编译器（GDB 和 VS）默认栈空间上限大小不同

4：程序 3 stack_get 项目

一个类 stack_get 及其测试，能实现自动读取当前栈大小

4.1 理论基础

因为经过测试，（如果不修改）栈内存上限大小不变，而且连续，所以只需要三个变量：

栈首地址 Pstart, 当前使用的栈内存地址 Pthis,栈尾（下限）地址 Pend

这样就可以求出已使用的栈内存大小为 Pstart-Pthis, 剩余可用栈内存大小为 Pthis-Pend

获取栈首地址的方法：开始执行 main 的时候声明并定义一个局部变量 char s= ‘a’ ;

获取当前地址的方法：在当前位置声明并定义一个局部变量 char t= ‘a’ ;

（为什么用 char：char 大小为 1 字节，这样可以直接相减内存地址，不需要比特转换为字节的单位转换）

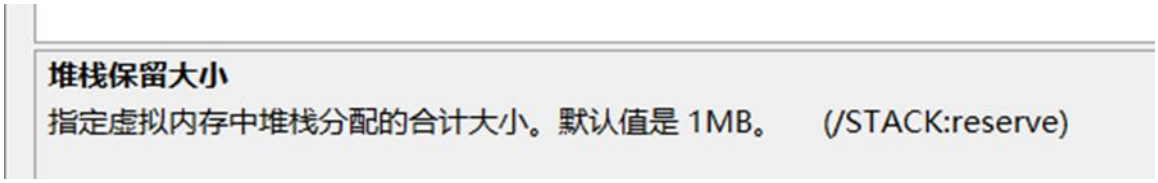
代码如下：

```
stack_get::stack_get()

{

        char first='a';

        init = &first; //栈首地址
```

```
}

void stack_get::get_result()

{

        char temp='a';

        present = &temp; //当前栈地址

}
```

栈内存上限获取方法有四种

1：在 VS 中选择项目—属性—链接器—系统—堆栈保留大小，可以设置和查看栈上限大小，默认为 1MB（但经过测试，应该在 0.97MB 左右）



2：在VS中静态设置栈大小（代码为#pragma comment(linker, "/STACK:2000000,2000000")，两个数字代表设置的内存上限字节数）后，使用NtCurrentTeb来获取内存大小。代码如下【3】

```
NT_TIB *tib = (NT_TIB*)NtCurrentTeb();
DWORD stackBase = (DWORD)tib->StackBase;
DWORD stackLimit = (DWORD)tib->StackLimit;
max = stackBase - stackLimit;
```

需要头文件<windows.h>

代码如下：文件夹stack_windows下的stack_windows.cpp

```
#include<iostream>
#include<iomanip>
#include<Windows.h>
#include<fstream>
using namespace std;
ofstream out("data_out.txt");
//这里有设置栈空间上限的语句
void test(int num)
{
```

```
        //手动爆栈来获取已使用栈内存，和之前的代码方法相似
        static char* start_flag;
        static char* this_flag;
        if (num == 0)
        {
                char start = 'a';
                start_flag = &start; //获取首地址
                this_flag = start_flag;
        }
        else
        {
                char c = 'a';
                this_flag = &c;
                out << "The " << num << "th data" << endl;
                double sum = double(start_flag - this_flag) / 1024 / 1024;
                out << setprecision(2) << sum << "MB of stack has been used" << endl;
                out << "-------------------------------------------------" << endl;
        }
        num++;
        test(num);
}
int main()
{
        NT_TIB *tib = (NT_TIB*)NtCurrentTeb();
        DWORD stackBase = (DWORD)tib->StackBase;
        DWORD stackLimit = (DWORD)tib->StackLimit;
        double limit = double(stackBase - stackLimit)/1024/1024;
        out << "The stack limit is " << setprecision(2) << limit << "MB" << endl;
        //以上代码用来自动获取栈内存上限
        test(0);
        return 0;
}
```

结果：

如果没有手动设置栈空间上限，则读取的栈空间上限为0.078MB,实际上限为0.98MB

如果设置了栈空间上限，就能够正确读取。但是实际上限不一定等于设置上限。

输入：手动修改#pragma comment(linker, "/STACK:2000000,2000000")中的两个数字到MB数*1024*1024，然后执行

设置上限	读取上限	实际上限
0.098MB	0.098MB	0.23MB
0.19MB	0.19MB	0.23MB
0.48MB	0.48MB	0.98MB
0.77MB	0.77MB	0.98MB
1MB	1MB	1MB
2MB	2MB	2MB

2.9MB	2.9MB	3MB
4.8MB	4.8MB	5MB
8.1MB	8.1MB	9MB
8MB	8MB	8MB
9.5MB	9.5MB	10MB
16MB	16MB	16MB

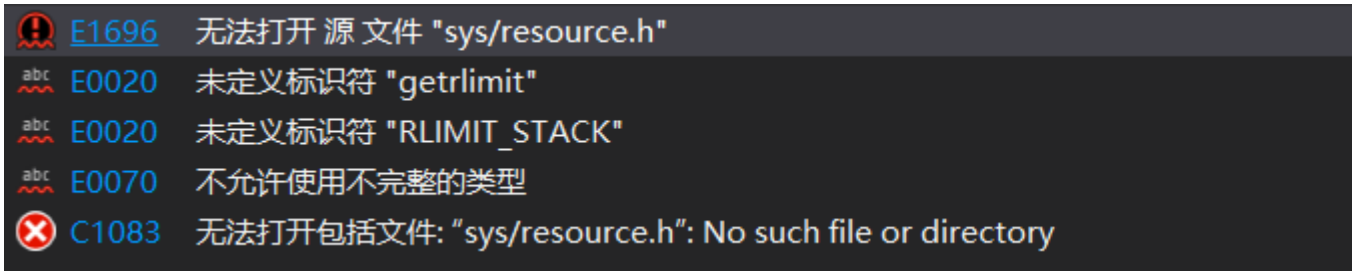
结论：这些代码能够正确读取设置的栈上限，但是设置栈上限其实是一个向上取整的过程：系统可能有些固定的 level，设置栈上限时向上取整，取整的范围包括正整数大小的栈内存，还有 0.98MB,0.23MB。而且，如果不初始化，则无法正确读取栈上限。

3：在 linux,GDB 编译器下，可以用 getrlimit 函数查询栈内存上限【4】

int getrlimit(int *resource*, struct rlimit **rlim*);

需要引用头文件<sys/resource>

但是在 vs 编译器不过编译，因为没有这个头文件



测试代码如下：

文件：stack_linux 文件夹下的 stack_linux.cpp

```
#include<iostream>
#include<sys/resource.h>//getrlimit需要的头文件
#include<iomanip>
#include<fstream>
using namespace std;
ofstream out("data_out.txt");
struct rlimit lim_get;
void test(int num)
{
//手动爆栈来获取已使用栈内存，和之前的代码方法相似
static char* start_flag;
```

```
static char* this_flag;
if(num==0)
{
char start='a';
start_flag=&start; //获取首地址
this_flag=start_flag;
}
else
{
char c='a';
this_flag=&c;
out<<"The "<<num<<"th data"<<endl;
double sum=double(start_flag-this_flag)/1024/1024;
out<<setprecision(2)<<sum<<"MB of stack has been used"<<endl;
out<<"--------------------------------------------------"<<endl;
}
num++;
test(num);
}
int main()
{
getrlimit(RLIMIT_STACK,&lim_get);
double limit=double(lim_get.rlim_cur)/1024/1024;
out<<"The stack limit is "<<setprecision(2)<<limit<<"MB"<<endl;
//以上三句代码用来自动获取栈内存上限
test(0);
return 0;
}
```

这个能在 linux 的 GDB 编译器下编译，需要先用命令行移动到代码本地地址，然后用 g++ -o stack_linux.exe stack_linux.cpp 生成可执行文件 stack_linux.exe，之后 gdb stack_linux.exe 执行,然后使用 run 运行程序，得到结果。

结果在 data_out.txt 中，我手动爆栈得到的栈内存上限和读取的栈内存上限都是 8MB，这也和我之前测试的 linux 下 GDB 默认栈内存上限一致，说明这个代码的确可以用来获取栈内存上限

4：VS 有一个.net 的变量---VCLinkerTool.StackReserveSize 可以查询和修改栈内存上限，但是只有 C++/CLI 才能调用这个变量【5】

4.2 代码部分

初始化：用一个 char 类型的临时变量的地址当做栈头。

代码如下

```
stack_get::stack_get()
{
```

```
        char first='a';
        init = &first; //栈首地址
        max = 1024 * 1024;
}
```

之后有三种获取栈上限方法：

方法 1：用 NtCurrentTeb 自动获取栈上限大小，但是前提是必须要事先设置栈上限大小，否则无法读取默认值

代码如下：

```
void stack_get::limit_set()
{
        NT_TIB *tib = (NT_TIB*)NtCurrentTeb();
        DWORD stackBase = (DWORD)tib->StackBase; //获取栈首地址
        DWORD stackLimit = (DWORD)tib->StackLimit; //获取栈下限地址
        max = stackBase - stackLimit; //计算得到栈内存上限，单位为Byte
}
```

方法 2：默认栈上限大小为 1M

代码如下：

```
void stack_get::limit_set(int a)

{

        max = 1024 * 1024;

}
```

方法 3：让用户自行输入栈上限大小

代码如下：

```
void stack_get::limit_set(char c)
{
        std::cout << "Please input an integer of the unit of stack limit" << std::endl;
        std::cout << "1 means MB,2 means KB,3 means Byte" << std::endl;
        cp_integer m,n; //引用了代码构件库里的cp_integer文件，能正确读入int类型整数并且能抛出完整异常
        int k;
        m.input(1, 3);
        std::cout << "Now input an integer as the number of stack limit" << std::endl;
        k = m.get();
```

```
        if (k == 1) //MB模式
        {
                std::cout << "It should >=1 and <=64" << std::endl;
                n.input(1, 64);
        }
        if (k == 2) //KB模式
        {
                std::cout << "It should >=64 and <=65536" << std::endl;
                n.input(64, 65536);
        }
        if (k == 3) //byte模式
        {
                std::cout << "It should >=65536 and <=67108864" << std::endl;
                n.input(65536, 67108864);
        }
        int p = n.get();
        if (k == 1) max = p * 1024 * 1024;
        if (k == 2) max = p * 1024;
        if (k == 3) max = p;
}
```

调用：在要读取内存的位置调用函数 get_result()就可以了

具体的应用方法：

1：声明变量 stack_get* g;

2：在主函数开始处初始化变量：g=g->initialize();

3: 随后设置模式：g->choose();

4：在想要输出的地方调用 g->get_result();

具体代码:文件夹 stack_get

文件 stack_get.h

```
#ifndef STACK_GET_H
#define STACK_GET_H
class stack_get
{
private:
        stack_get(); //采用单体模式

        ~stack_get();
        char* init, *present;
        int max;
        static stack_get* stack_app;
public:
        void limit_set();//修改栈大小后自动获取栈上限
        void limit_set(int a);//默认栈上限1M
        void limit_set(char c);//用户手动输入栈上限
```

```
        void choose();
        void get_result();
        void get_result(int i);
        static stack_get* initialize();
};
#endif
```

文件 stack_get.cpp

```
#include<iostream>
#include<fstream>
#include <windows.h>
#include"stack_get.h"
#include"cp_integer.h"

stack_get::stack_get() //设置栈首地址
{
        char first='a';
        init = &first; //栈首地址
        max = 1024 * 1024;
}
stack_get* stack_get::stack_app = NULL; //单体模式的定义

stack_get::~stack_get()
{
        if (stack_app != NULL) delete(stack_app);
}
void stack_get::get_result() //获取结果
{
        char temp='a';
        present = &temp; //当前栈地址
        double answer = (double(max) - double(init - present)) / 1024 / 1024;  //当前剩
余栈内存
        std::cout << "There is still " << answer << " MB of stack memory left" <<
std::endl;
}
stack_get* stack_get::initialize() //获取实例对象的方法
{
        if (stack_app == NULL) stack_app = new stack_get();
        return stack_app;
}
void stack_get::limit_set() //初始化1：用teb获取栈大小
{
        NT_TIB *tib = (NT_TIB*)NtCurrentTeb();
        DWORD stackBase = (DWORD)tib->StackBase; //获取栈首地址
        DWORD stackLimit = (DWORD)tib->StackLimit; //获取栈下限地址
        max = stackBase - stackLimit; //计算得到栈内存上限，单位为Byte
}
void stack_get::limit_set(int a) //初始化2：设置栈大小为默认1M，但是不修改栈实际大小
{
        max = 1024 * 1024;
}
void stack_get::limit_set(char c) //初始化3：用户手动输入栈大小，但是不修改实际栈大小
{
        std::cout << "Please input an integer of the unit of stack limit" << std::endl;
        std::cout << "1 means MB,2 means KB,3 means Byte" << std::endl;
```

```
	cp_integer m,n; //引用了代码构件库里的cp_integer文件，能正确读入int类型整数并且能抛出完整异常
	int k;
	m.input(1, 3);
	std::cout << "Now input an integer as the number of stack limit" << std::endl;
	k = m.get();
	if (k == 1) //MB模式
	{
		std::cout << "It should >=1 and <=64" << std::endl;
		n.input(1, 64);
	}
	if (k == 2) //KB模式
	{
		std::cout << "It should >=64 and <=65536" << std::endl;
		n.input(64, 65536);
	}
	if (k == 3) //byte模式
	{
		std::cout << "It should >=65536 and <=67108864" << std::endl;
		n.input(65536, 67108864);
	}
	int p = n.get();
	if (k == 1) max = p * 1024 * 1024;
	if (k == 2) max = p * 1024;
	if (k == 3) max = p;
}
void stack_get::choose() //主控模块
{
	std::cout << "Mode 1: After you set the limit, select it to get the stack limit" << std::endl;
	std::cout << "Mode 2: Use the default stack limit of 1MB" << std::endl;
	std::cout << "Mode 3: Manually set the stack limit" << std::endl;
	std::cout << "Now input an integer from 1 to 3 as the mode" << std::endl;
	cp_integer m;
	int k;
	m.input(1, 3);
	k = m.get();
	if (k == 1) limit_set();
	if (k == 2) limit_set(1);
	if (k == 3) limit_set('c');
}
```

文件 cp_integer.h 和 cp_integer.cpp 放在附录了

文件 main.cpp

```
#include<iostream>
#include<fstream>
#include"stack_get.h"
#pragma comment(linker, "/STACK:1048576,1048576")
using namespace std;
stack_get* g; //定义指针
ifstream in("data_in.txt");
ofstream out("data_out.txt");
```

```
void stack_get::get_result(int i) //适合文件流输出的代码，除输出方式外和控制台版本完全
一样
{
	char temp = 'a';
	present = &temp;
	double answer = (double(max) - double(init - present)) / 1024 / 1024;
	out << "There is still " << answer << " MB of stack memory left\n";
}
void stack_test(int num) //测试代码
{
	static int total=0;
	static int outplace_num=0;
	static int outplace[30];
	if (num == 0) //文件读入总递归次数，总输出次数，和输出的位置
	{
		in >> total;
		in >> outplace_num;
		for (int i = 1; i <= outplace_num; i++)
		{
			int p;
			in >> p;
			outplace[i] = p;
		}
	}
	int i;
	bool whether = 0;
	for (i = 1; i <= outplace_num; i++)
	{
		if (num == outplace[i])
		{
			whether = 1;
			break;
		}
	}
	if (whether == 1)
	{
		out << "The " << num << "th data" << endl;
		g->get_result(1);
	}
	num++; //递归
    if (num <= total) stack_test(num); //判断是否退栈
	num--; //退栈
	whether = 0;
	for (i = 1; i <= outplace_num; i++)//判断是否输出
	{
		if (num == outplace[i])
		{
			whether = 1;
			break;
		}
	}
	if (whether == 1)//输出
	{
		out << "The " << num << "th data" << endl;
		g->get_result(1);
	}
```

```
}
int main()
{
        g = g->initialize(); //初始化实例对象
        g->choose(); //选择模式
        stack_test(0);
        stack_test(1);
        return 0;
}
```

4.3 测试部分（详细测试数据在 result 文件夹）

我的读入：读入文件 data_in.txt 都如下：

3600

13

1 300 600 900 1200 1500 1800 2100 2400 2700 3000 3300 3600

数据文件	类型	首数据的内存差	末数据的内存差	每 300 组数据的剩余内存差
1M_mode1	内存大小 1M，不退栈	1M	0.08M	0.076M
1M_mode2	内存大小 1M，不退栈	1M	0.08M	0.076M
1M_mode3	内存大小 1M，不退栈	1M	0.08M	0.076M
2M_mode1	内存大小 2M，不退栈	2M	1.08M	0.076M
2M_mode3	内存大小 2M，不退栈	2M	1.08M	0.076M
Withdraw_ mode1	内存大小 1M，退栈	1M 退栈后 1M	0.08M 退栈后 0.08M	0.076M 退栈后 0.076M

Withdraw_ Mode2	内存大小 1M，退栈	1M 退栈后 1M	0.08M 退栈后 0.08M	0.076M 退栈后 0.076M
Withdraw_ mode3	内存大小 1M，退栈	1M 退栈后 1M	0.08M 退栈后 0.08M	0.076M 退栈后 0.076M

结论：三个模式的数据大致相同，而且不管设置内存多大，是否退栈，结果都大致一致，可以说明这个程序有正确性

局限：

1：方法 2,3 需要用户手动用编译器/控制台获取栈内存上限

2：方法 1 要求用户事先设置栈上限，而且设置的大小有一定要求，详细见 stack_windows 的测试

3：VCLinkerTool.StackReserveSize 这个变量获取内存方法更好，但是它是.net 的变量，不是 C++的变量，无法直接调用

6：引用：

【1】《编译原理》第二版，7.1 存储组织

【2】《编译原理》第二版，7.2 空间的栈式分配

【3】How to get thread stack information on Windows?—Stack Overflow
https://stackoverflow.com/questions/3918375/how-to-get-thread-stack-information-on-windows

【4】setrlimit(2) - Linux man page
https://linux.die.net/man/2/setrlimit

【5】VCLinkerTool.StackReserveSize Property
https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.vcprojectengine.vclinkertool.stackr

eservesize?view=visualstudiosdk-2017