**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：

|  |
| --- |
| 栈区 |
| 空闲内存 |
| 堆区 |
| 静态区 |
| 代码区 |

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()

{

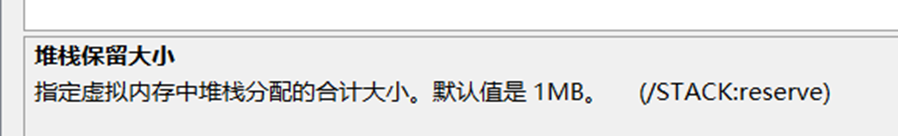
char temp='a';

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 |

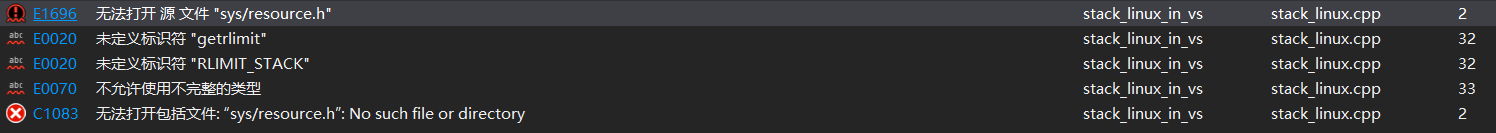
结论：这些代码能够正确读取设置的栈上限，但是设置栈上限其实是一个向上取整的过程：系统可能有些固定的level，设置栈上限时向上取整，取整的范围包括正整数大小的栈内存，还有0.98MB,0.23MB。而且，如果不初始化，则无法正确读取栈上限。

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?](https://stackoverflow.com/questions/3918375/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.stackreservesize?view=visualstudiosdk-2017>