

俞楚凡 ICS Lab 3.31

1. 函数局部存储的管理

对于练习一，我编写的代码如下：

```
ainfinity@AInfinity: ~/ics/lab5$ cat recursive.c
#include <stdio.h>

void test(int i) {
    if (i > 10) return;

    int a[11];
    a[i] = i;
    for (int j = 0; j < 11; ++j)
        printf("%d ", a[j]);
    printf("\n");
    test(i + 1);
}

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

这是一个简单的递归函数。从0开始递归，每次重新声明一个局部数组a，并且将对应位置赋值为i，随后打印输出。程序循环11次后停止。

以下为其输出：

```
ainfinity@AInfinity:~/ics/lab5$ ./recursive
0 0 0 0 0 0 0 0 0 0 0 0
-395356592 1 176672208 32687 12 0 18350080 0 -395356168 32765 1
-395356704 32765 2 32687 0 0 178222528 32687 -395356168 32765 1
-395356816 32765 176672208 3 0 0 178222528 32687 -395356168 32765 1
-395356928 32765 176672208 32687 4 0 178222528 32687 -395356168 32765 1
-395357040 32765 176672208 32687 0 5 178222528 32687 -395356168 32765 1
-395357152 32765 176672208 32687 0 0 6 32687 -395356168 32765 1
-395357264 32765 176672208 32687 0 0 178222528 7 -395356168 32765 1
-395357376 32765 176672208 32687 0 0 178222528 32687 8 32765 1
-395357488 32765 176672208 32687 0 0 178222528 32687 -395356168 9 1
-395357600 32765 176672208 32687 0 0 178222528 32687 -395356168 32765 10
ainfinity@AInfinity:~/ics/lab5$
```

可以观察到，在每一次调用时，**a**数组被初始化成了相同的值，随后对应的**i**位置上的值被初始化成了**i**。

在上一层循环中，程序对于数组**a**的赋值只在当次循环中有效，而无法影响到之后的循环中。这是因为，对于每一次函数调用，虽然调用的是同一个函数，但是系统会为每一个函数开辟一个独立的栈帧空间。因此，在这些栈帧空间中的**a**数组的“副本”也是独立的。这就意味着，每一个**a**数组被独立的赋了一次值，故而不会相互影响。

2. 二维数组的内存管理与系统限制

test.c:编写的代码如下：

```
1  #include <stdio.h>
2
3  #define N 720
4
5  int main() {
6      long a[N][N];
7      for (int i = 0; i < N; ++i) {
8          for (int j = 0; j < N; ++j) {
9              a[i][j] = i * N + j;
10         }
11     }
12     printf("%d", a[N - 1][N - 1]);
13
14     return 0;
15 }
```

经过反复试验，**windows**下得出使得程序没有合法输出的最小**N**值为720.在**linux**中，这个值则是1024.

查阅资料可知，**linux**等系统对于在栈中声明的局部变量的占用空间大小是有限制的，一般是8M。当数组空间开销过大的时候就很有可能发生栈溢出。在**linux**下， $1024 \times 1024 \times 8 / 1024 / 1024$ 正好是8，加上其他的一些开销会达到栈空间上限。而在**windows**中，图中数组所占用的空间为 $720 \times 720 \times 4 / 1024 / 1024$ ，约为1.9775390625M，加上其他的一些开销，会超过2M，也就是**windows**系统对于栈空间的限制。

这样的限制对于大多数程序来说并没有明显的影响，原因是因为，第一，程序员通常会避免在函数中声明过大的局部变量，因此系统对于栈空间的限制通常可以满足程序的运行需求。

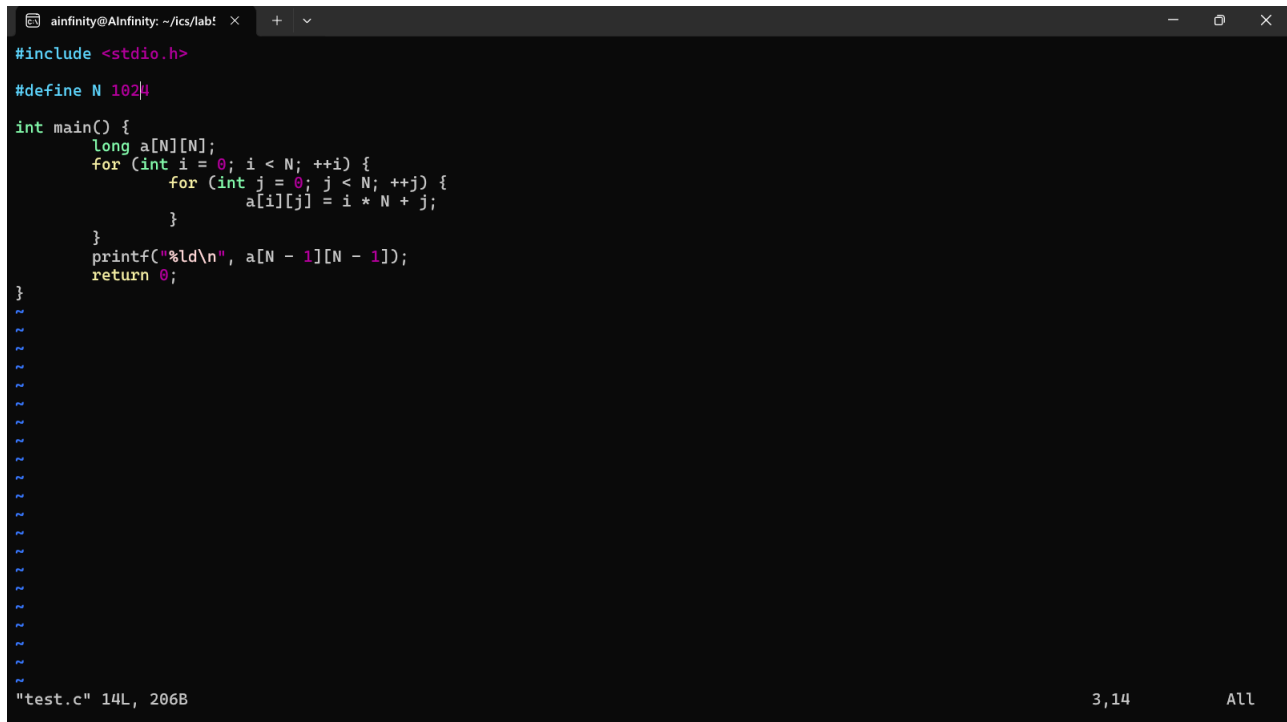
第二，大部分程序的递归所使用的嵌套层次不会达到栈空间上限。

第三，如果不可避免的出现栈空间不足，开发者可以手动调高栈空间。

我们首先启动WSL Ubuntu 24.04.1 LTS系统，查看系统栈空间限制：

```
ainfinity@AInfinity:~/ics/lab5$ ulimit -s
8192
```

随后，在ubuntu下编译源程序：

A screenshot of a code editor window titled 'ainfinity@AInfinity: ~/ics/lab5'. The code is a C program named 'test.c'. It includes <stdio.h> and defines a constant N as 1024. The main function declares a long array 'a' of size N. It then uses two nested for loops to iterate over the array, calculating a value for each element. The calculation is 'a[i][j] = i * N + j;'. After the loops, it prints the value of 'a[N - 1][N - 1]' and returns 0. The editor shows the file is 14 lines long and 206 bytes. The status bar at the bottom right indicates '3,14' and 'All'.

```
ainfinity@AInfinity:~/ics/lab5$ ./test
Segmentation fault (core dumped)
```

将系统栈空间限制调大：

```
ainfinity@AInfinity:~/ics/lab5$ ulimit -s 65532
ainfinity@AInfinity:~/ics/lab5$ gcc test.c -o test
ainfinity@AInfinity:~/ics/lab5$ ./test
1048575
```

重新运行程序，发现程序正常运行。

3. 数组的访问方式

1. 地址计算

编写 `address.c` 程序如下：

```
ainfinity@AInfinity: ~/ics/lab5$ cat address.c
#include <stdio.h>

long a[10], b[10][100];

int main() {
    for (int i = 0; i < 10; ++i) a[i] = i;
    for (int i = 0; i < 10; ++i)
        for (int j = 0; j < 100; ++j) b[i][j] = i * 100 + j;
    printf("%lu\n", sizeof(long));
    printf("%p %p %ld %ld", (a + 1), (a + 1) - a, (b + 1), (b + 1) - b);
    printf("%p %p %ld %ld", (a + 2), (b + 2), (a + 2) - a, (b + 2) - b);
    printf("%p %p %ld %ld", (a + 12), (b + 12), (a + 12) - a, (b + 12) - b);
    return 0;
}

"address.c" 14L, 443B 10, 21-28 All
```

编译并运行。

```
ainfinity@AInfinity:~/ics/lab5$ ./address
8
0x55b2a2514048 0x55b2a25143c0 1 10x55b2a2514050 0x55b2a25146e0 2 20x55b2a25140a0 0x55b2a2516620 12 12
```

可以发现，数组在系统内存中是以一块连续的线性空间的形式存储的，即使是多维数组也会被展开成“一维数组”的形式。每一个元素的空间是元素类型所占的字节数。故而，每当我访问下一个元素时，地址将会偏移八个单位。但是差值仍然是1， 2， 12而不是8， 16， 96，这是因为 `a + 1` 实际代表的是将指针指向数组的后一个元素，而不是简单的+1运算，实际上指针偏移了八位。

2. 数组越界访问行为

编写程序如下：


```

int main() {
    int m = 5, n = 6;
    int a[1009][1009];

    int i = 3, j = 5;
    printf("number (%d, %d) is %d\n", i, j, mapToVal(m, n, i, j));

    int num = 23;
    int ii, jj;
    valToMap(m, n, num, &ii, &jj);
    printf("position is (%d, %d)\n", ii, jj);

    return 0;
}

```

程序内置的测试用例是在5*6的矩阵下输出(3, 5)位置的数，以及输出23对应的位置。当然，这是可以随意修改的。

程序的设计思想是先确定某个自然数在矩阵中处于从外到里分层的第几层，再以上下左右的顺序依次遍历判断是否遍历到了这个数。在程序中，k是指示层数的标志。

测试结果如下。

```

ainfinity@AInfinity:~/ics/lab5$ vim address_3.c
ainfinity@AInfinity:~/ics/lab5$ gcc address_3.c -o address_3
ainfinity@AInfinity:~/ics/lab5$ ./address_3
number (3, 5) is 8
position is (2, 4)

```