

```
D:\codeblocks\examples\undefined\bin\Debug\undefined.exe

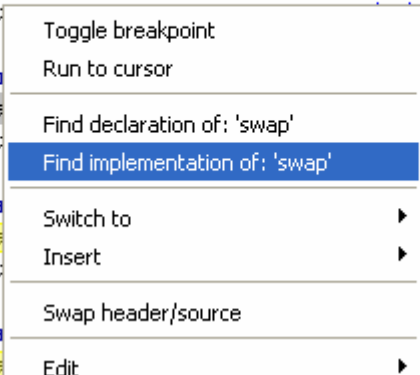
swap(int&, int&) called.
1 2293728 这个整数怎么回事
1.13222e-317 3.3 这个浮点数怎么回事
swap(T*, T*) called. 调用哪个函数交换的，怎么没有提示？
f e
swap(T*, T*) called.
string 1 string 2 字符串值没有交换

Process returned 0 (0x0)   execution time : 1.702 s
Press any key to continue.
```

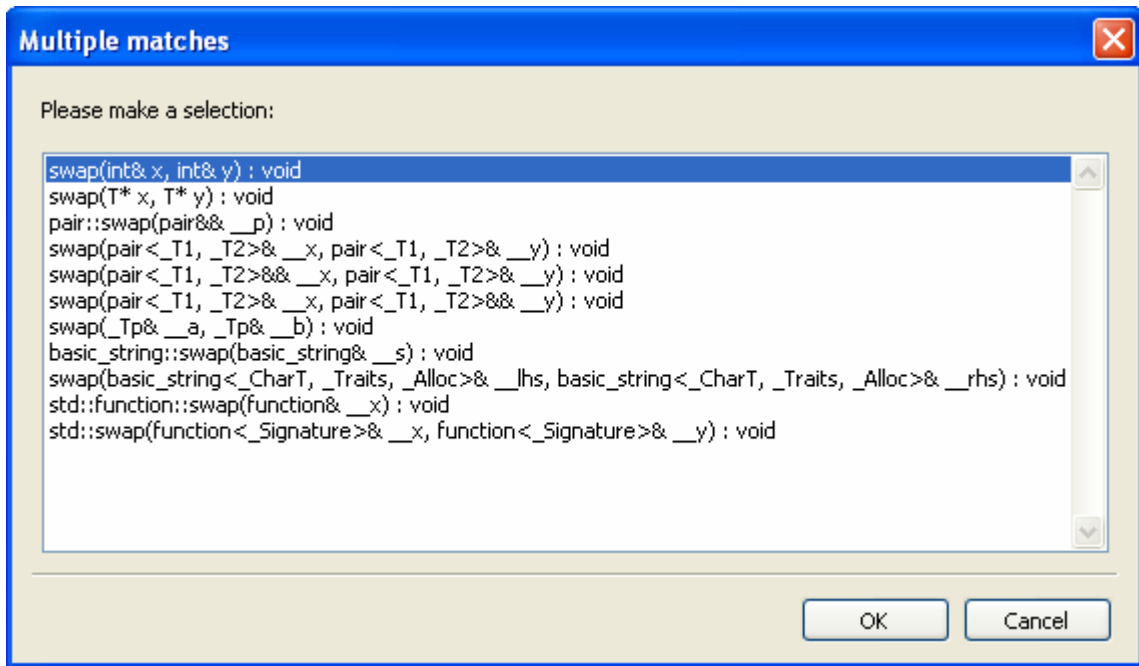
```
8  int main()
9  {
10     int a, b = 1; 局部变量a没有赋值
11     swap(a, b);
12     cout << a << ' ' << b << "\n\n";
13
14     double c = 3.3, d; 局部变量d没有赋值
15     swap(c, d); 调用哪个swap?
16     cout << c << ' ' << d << "\n\n";
17
18     char e = 'e', f = 'f';
19     swap(&e, &f);
20     cout << e << ' ' << f << "\n\n";
21
22     char* str1 = "string 1", * str2 = "string 2";
23     swap(str1, str2); 为什么不能交换两个字符串的值
24     cout << str1 << ' ' << str2 << "\n\n";
```

先改正容易修改的错误，给a和d赋初值，假设a=0, d=0.1，保存当前文件。然后看看第15行，到底调用了哪个swap，因为我们的swap.hpp没有跟这个调用相匹配的函数，去能编译不出现语法错误，实在蹊跷，那我们就看到底调用了哪个swap。选中swap(显示灰色)，按下鼠标右键，在弹出的快捷菜单中选择Find implementation of: 'swap'，见下图。

```
8  int main()
9  {
10     int a = 0, b = 1;
11     swap(a, b);
12     cout << a << ' ' << b << "\n\n";
13
14     double c = 3.3, d = 0.1;
15     swap(c, d);
16     cout << c << ' ' << d << "\n\n";
17
18     char e = 'e', f = 'f';
19     swap(&e, &f);
20     cout << e << ' ' << f << "\n\n";
21
22     char* str1 = "string 1", * str2 = "string 2";
23     swap(str1, str2);
24     cout << str1 << ' ' << str2 << "\n\n";
```



弹出一个对话框，显示了所有可能的swap函数头，见下图。



居然有这么多的swap，前两个是我们自己定义的，后面的是系统自带的。不要忘记前面的using namespace std;这会污染整个std命名空间，这样使用是OOP的大忌！注释掉using namespace std;，加一行using std::cout;然后重新编译，看看结果如何。

```
1 // main.cpp
2 #include <iostream>
3
4 //using namespace std;
5 using std::cout;
6 #include "swap.hpp"
7
8 int main()
9 {
10     int a = 0, b = 1;
11     swap(a, b);
12     cout << a << ' ' << b << "\n\n";
13
14     double c = 3.3, d = 0.1;
15     swap(c, d);
16     cout << c << ' ' << d << "\n\n";
17
18     char e = 'e', f = 'f';
19     swap(&e, &f);
20     cout << e << ' ' << f << "\n\n";
21 }
```

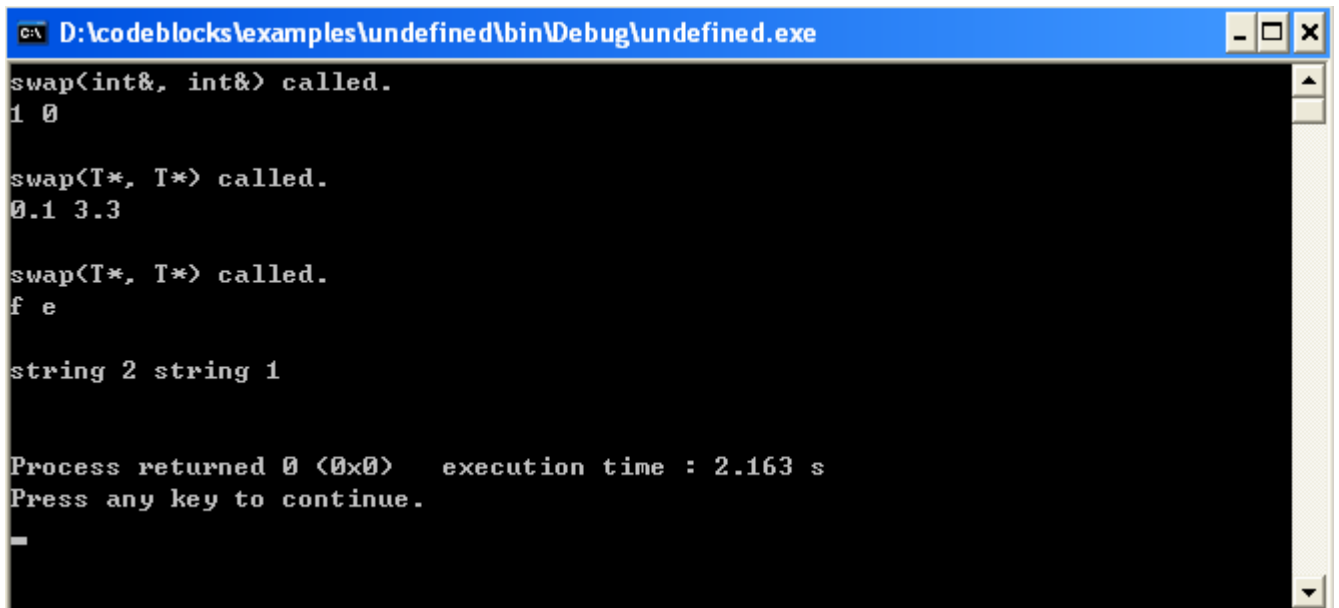
Logs & others

Code::Blocks Search results Build log Build messages Debugger Thread search

File	Line	Message
D:\codeblocks\...		In function 'int main()':
D:\codeblocks\...	15	error: invalid initialization of reference of type 'int&' from
D:\codeblocks\...	16	error: in passing argument 1 of 'void swap(int&, int&)'
D:\codeblocks\...	22	warning: deprecated conversion from string constant to 'char*'
D:\codeblocks\...	22	warning: deprecated conversion from string constant to 'char*'
=== Build finished: 2 errors, 2 warnings ===		

最可能的匹配是`void swap(int&, int&)`, 但是参数类型却不合适, 因此编译器报错。知道原因就好, 改正方法非常简单, 改成`swap(&c, &d)`;就可以了。

接下来, 我们看看为何两个字符串的值不能交换吧。从输出信息可知, 调用`void swap(T*, T*)`;但是这个函数实现中有个判断是否相等的比较`if(*x != *y)`, 而此刻的`*x`和`*y`都是字符's'(两个字符串的首字符相等), 自然后面的三个语句根本就不执行, 不要忘记判断字符串是否相等的C语言库函数是`strcmp`, 而绝非直接判断是否相等。那怎么修改呢? 最简单的办法是用C++的`string`处理这个问题, 把字符串当成对象来处理。`main`函数前加一行`#include <string>`, 然后用`std::string`定义字符串, 即`std::string str1("string 1"), str2("string 2")`; 交换两个字符串最好的办法是用`std::string`的成员函数`swap`, 这样不需要复制字符串内容, 仅仅交换指针就可以了, 也就是`str1.swap(str2)`;。讲过上述这些修改后, 保存, 重新编译并运行得到的结果如下图。



```
swap<int&, int&> called.
1 0

swap<T*, T*> called.
0.1 3.3

swap<T*, T*> called.
f e

string 2 string 1

Process returned 0 (0x0) execution time : 2.163 s
Press any key to continue.
-
```

3.5 调试程序

伴随着编写的程序愈来愈复杂, 我们往往很难一次性编译成功并运行得到我们期望的结果(虽无语法错误, 但可能有逻辑错误), 这时需要对程序进行调试以便定位错误。调试程序有时需要在程序中的某些地方设置一些特殊“点”, 让程序运行到该位置停下来, 有时需要检查某些变量的值, 以帮助我们检查程序中的逻辑错误。

调试程序之前, 首先要确认已经设置了Produce debugging symbols [-g]选项, 按照如下顺序去找(->表示下一步): Project -> Build options... -> find_min -> Debug -> Compiler Flags -> Produce debugging symbols [-g], 如果该项已经配置, 则可以调试程序了, 否则前面打勾, 点击OK按钮设置好。

下面将会通过具体的例子简要讲述调试器的使用。

3.5.1 面向过程风格的程序

假设我们有一个随机数集合, 该集合内随机数个数随机(为了便于观察结果是否正确, 我们假设该集合中随机数少于100个), 要求找出最小的那个随机数。

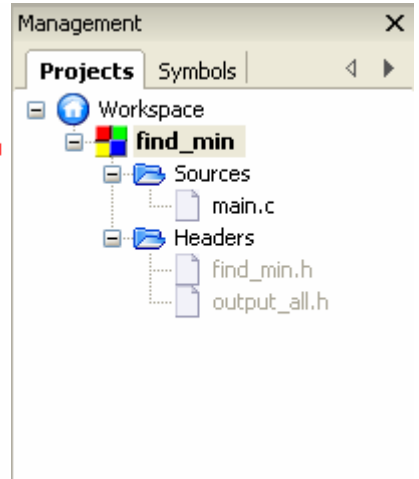
从一批数据中找到我们需要的某一个方法很多，为了简单起见，我们使用顺序搜索算法。用C语言编写程序实现以上题目要求，写了三个文件，一个源文件(main.c)，两个头文件，其中一个头文件(find_min.h)用来找出最小的随机数，一个头文件(output_all.h)用来输出这个随机数集中的所有随机数，代码见下贴图。

```
1 // main.c
2 #include <stdio.h> // for printf
3 #include <stdlib.h> // for malloc, free, srand, rand, system
4 #include <time.h> // for time
5 #include "find_min.h"
6 #include "output_all.h"
7
8 int main()
9 {
10     srand(time(NULL)); // 用时间作为随机数种子
11     // 随机数为SZ个, 0 <= SZ < 100
12     const unsigned MOD = 100, SZ = rand() % MOD;
13     // 在堆上开辟一段空间, 存储产生的随机数
14     unsigned* arr = (unsigned*)malloc(SZ * sizeof(SZ));
15     unsigned k = 0;
16     for(; k < SZ; ++k)
17         arr[k] = rand();
18
19     output_all(arr, SZ); // 输出产生的所有随机数
20     // 找到最小的那个随机数, 并返回它的下标
21     unsigned min = find_min(arr, SZ);
22     // 如果集合非空必然存在最小值, 输出它
23     if(0 != SZ)
24         printf("minimum in all: %d\n", arr[min]);
25
26     free(arr); // 释放分配的堆内存空间
27
28     system("PAUSE"); // 让系统等待, 以便我们查看结果是否正确
29     return 0;
30 }
31
32 // output_all.h
33 #ifndef OUTPUT_ALL_H
34 #define OUTPUT_ALL_H
35
36 #include <stdio.h> // for printf
37 void output_all(const unsigned* v, unsigned n)
38 {
39     // 输出结果中每行最多12个随机数
40     const unsigned NL = 12, M = NL + 1;
41     unsigned i = 0;
42     for(; i < n; ++i)
43     {
44         printf("%5d ", v[i]);
45         if(NL == i % M) // 控制换行
46             printf("\n");
47     }
48     printf(0 == n ? "" : "\n"); // 随机数集合为空不换行
49 }
50
51 #endif
52
```



```


1 // find_min.h
2 #ifndef FIND_MIN_H_
3 #define FIND_MIN_H_
4
5 unsigned find_min(const unsigned* v, unsigned n)
6 {
7     unsigned j = 0, i = 1;
8     for (; i < n; ++i)
9         if (v[i] < v[j])
10             j = i; // 记住最小随机数的下标
11     return j;
12 }
13
14 #endif
15







```




上面贴图中，右图是工程find_min的展开图。为了节约篇幅，这里不讲述find_min怎么建立的(前面已经讲述了怎么建立工程)，也不详细解释代码(代码中有注释)，仅仅讲调试过程。

首先，要在程序中设置一个点，让程序运行到这个位置暂停，把光标置于您想启动跟踪程序运行过程的这行代码的前一行，再用Debug下拉菜单中或者调试程序快捷菜单上的Run to Cursor  按钮启动调试器。(启动调试器的另一种方法是，用Debug下拉菜单中的Toggle breakpoint按钮先设置断点，然后点击Start  开始按钮来启动调试器。)



接下来就要告诉跟踪程序运行到哪里再次停下来，以便我们检查此刻的运行结果，我们可以把光标置于该行代码前，然后再次点击Run to Cursor  按钮，则程序运行到此处就停下来，此处称谓断点，可以用Debug下拉菜单中的Toggle breakpoint按钮设置断点，在一个断点前再次Toggle breakpoint，则该段点就被删除，如果想删除所有断点，可以用Debug下拉菜单中的Remove all breakpoints按钮。

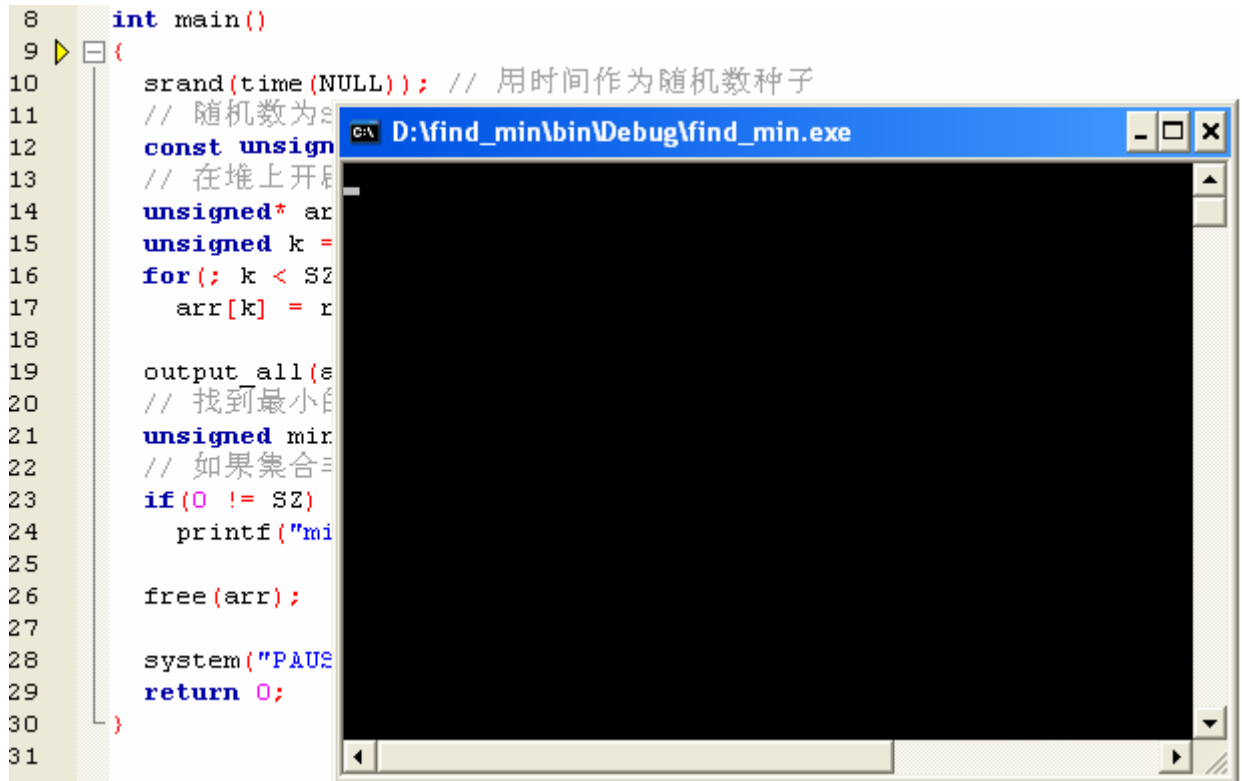
如果我们希望每次总是运行到程序下一行前面，则可以点击Debug下拉菜单中或者调试程序的快捷菜单上按钮Next line ，如果我们不想每次执行一行代码，希望执行的单位更小，可以逐条执行指令 (Next instruction )，如果运行到某个程序块(例如，调用某个函数)还可以选择按钮Step into ，则运行到该代码块内，如果希望跳出该代码块，则可以选择按钮Step out 。如果希望终止调试器，可以选择Continue  按钮，则调试器会自然运行到结束，也可以用Stop debugger ，则调试器就会被强行终止，调试过程结束。

在程序执行到输出语句前，屏幕始终不显示任何信息。如果我们希望查看程序中变量的值，则可以从Debug下拉菜单中选择Debugging windows子菜单的Watches按钮，也可以从快捷菜单按钮  下找到Watches按钮。


下面用示意图说明调试find_min的过程。

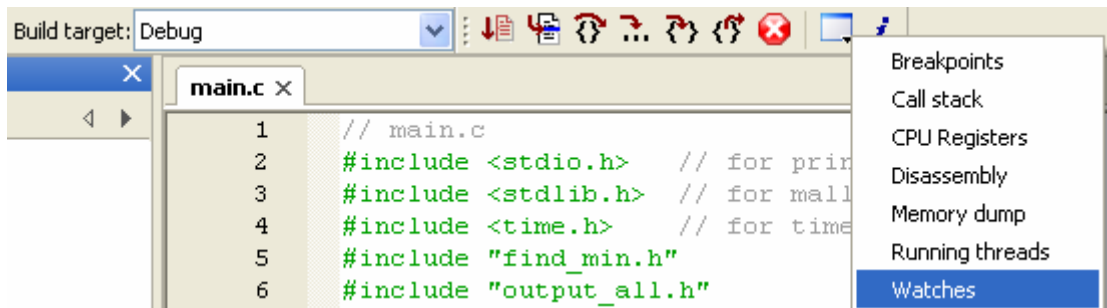
(1) 启动调试器

把光标置于int main之前，点击Run to cursor  按钮，则可以看到{前面有个黄色的箭头 ，而且还会出现了一个没有任何输出信息的对话框，如下图。

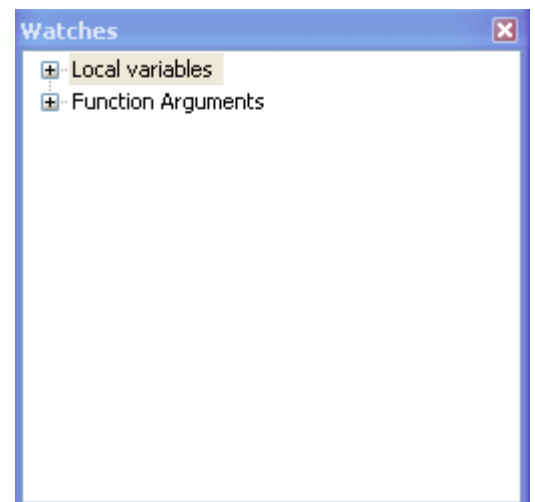


(2) 打开观察窗口

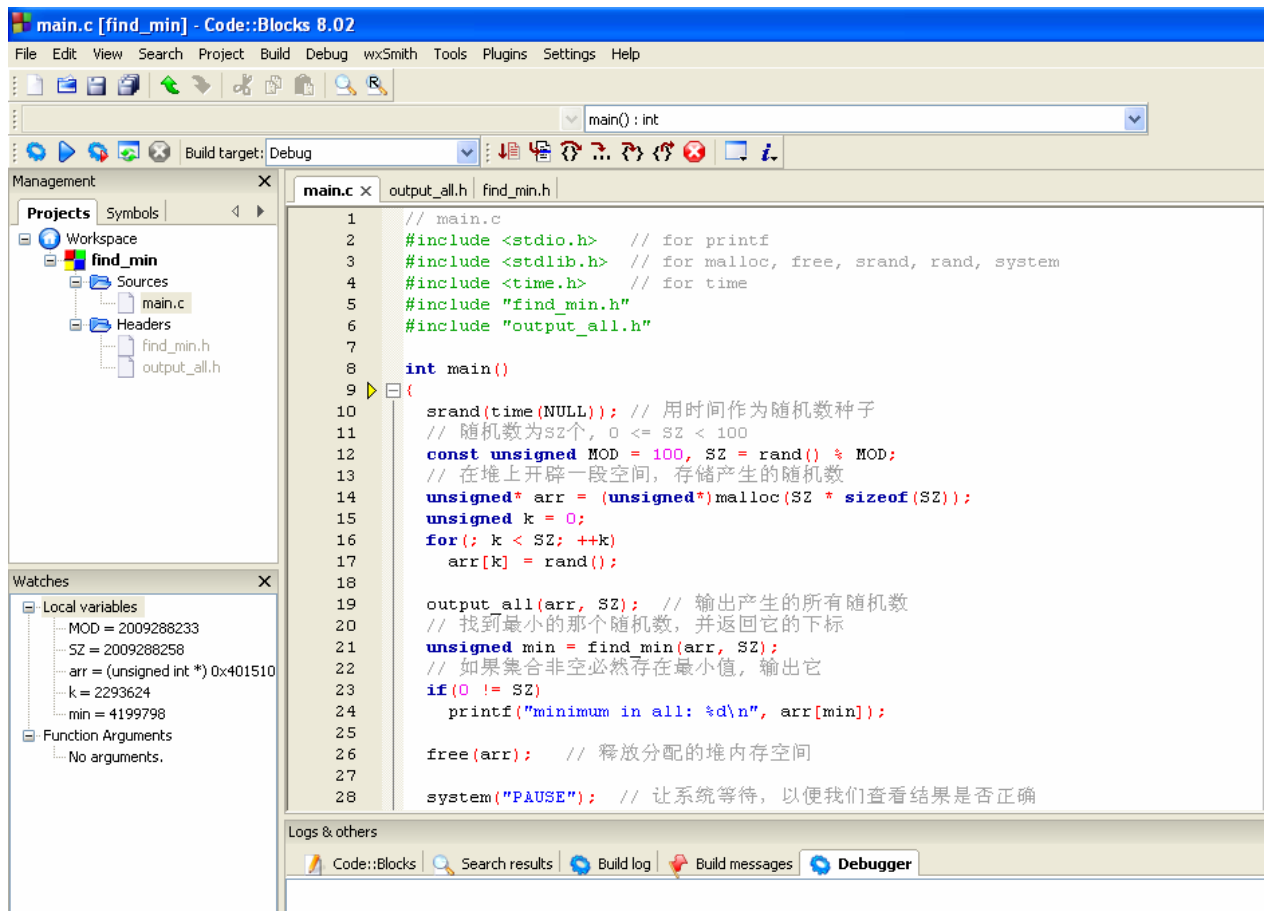
为了查看程序运行中变量值的变化情况，需要打开观察变量的窗口，用快捷菜单上  按钮的下拉菜单中Watches打开即可(也可以用Debug -> Debugging windows -> Watches打开)。



Watches窗口见右图。



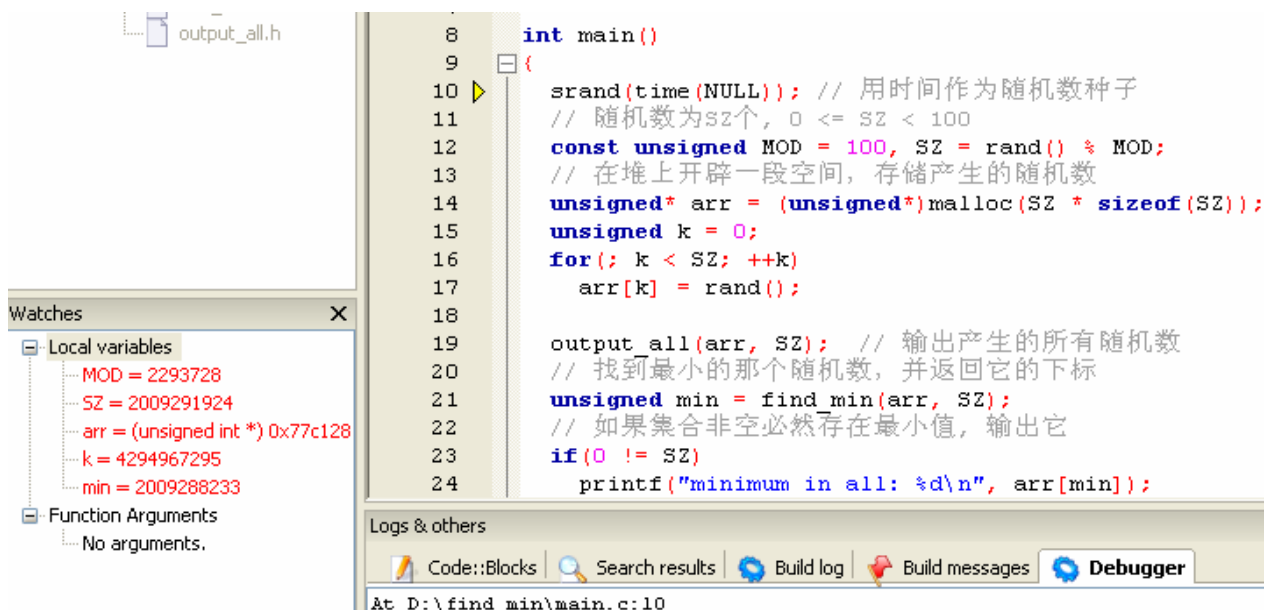
为了方便观察整个调试过程布局，拖动Watches窗口到左下角，并展开各个变量，如下图。




此时，从Watches窗口中可以看到定义的局部变量都是随机值，因为程序尚未执行到给这些变量赋值的语句。

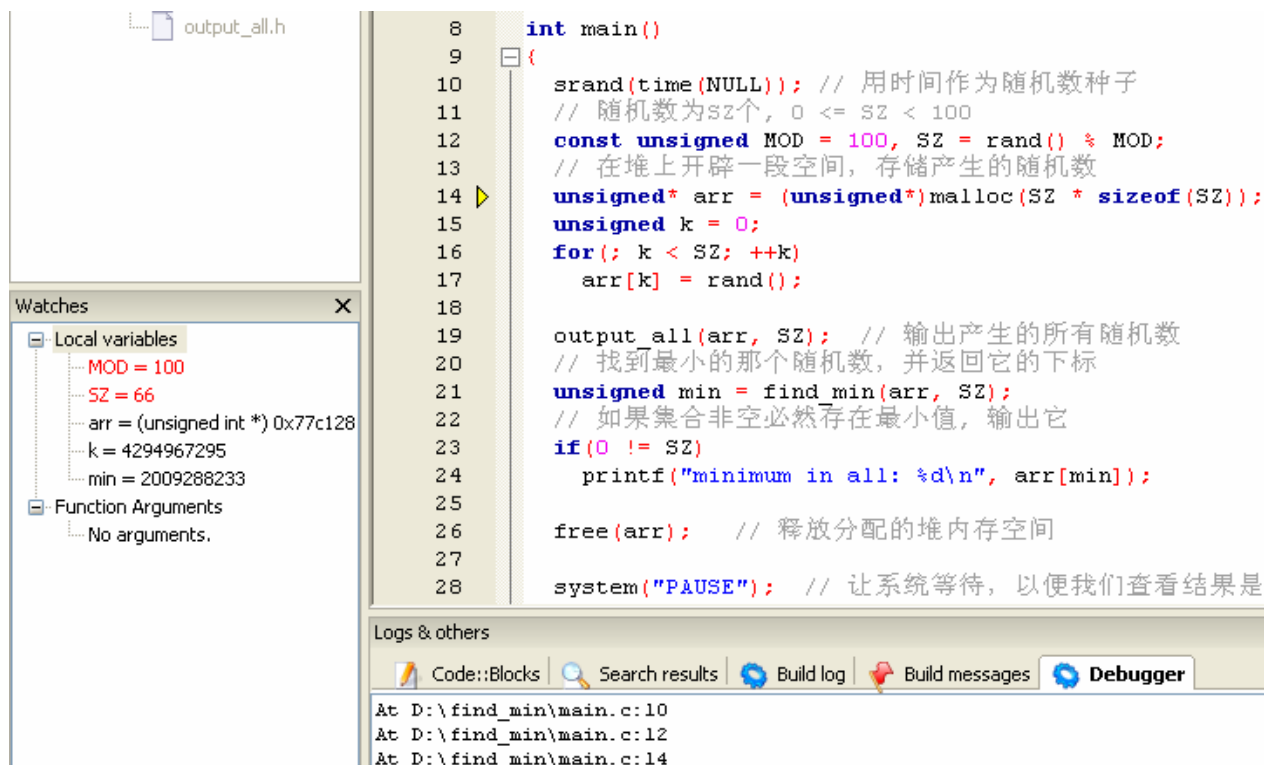
(3) 执行到下一行(第10行代码)

见下面的截图，输出信息窗口仍然什么都不显示，观察变量值的窗口局部变量的值都是系统赋予的随机值(左下角红色文字部分)。




(4) 执行到第14行

把光标放到第14行代码前面，点击Run to Cursor 按钮，见下图。



由于第12行语句已经被执行，从观察窗口可以看到局部变量MOD = 100, SZ = 33 (观察窗口红色文字部分)，此外日志窗口的调试器Debugger栏目也出现了一些文字，显示了main.c被执行的代码。

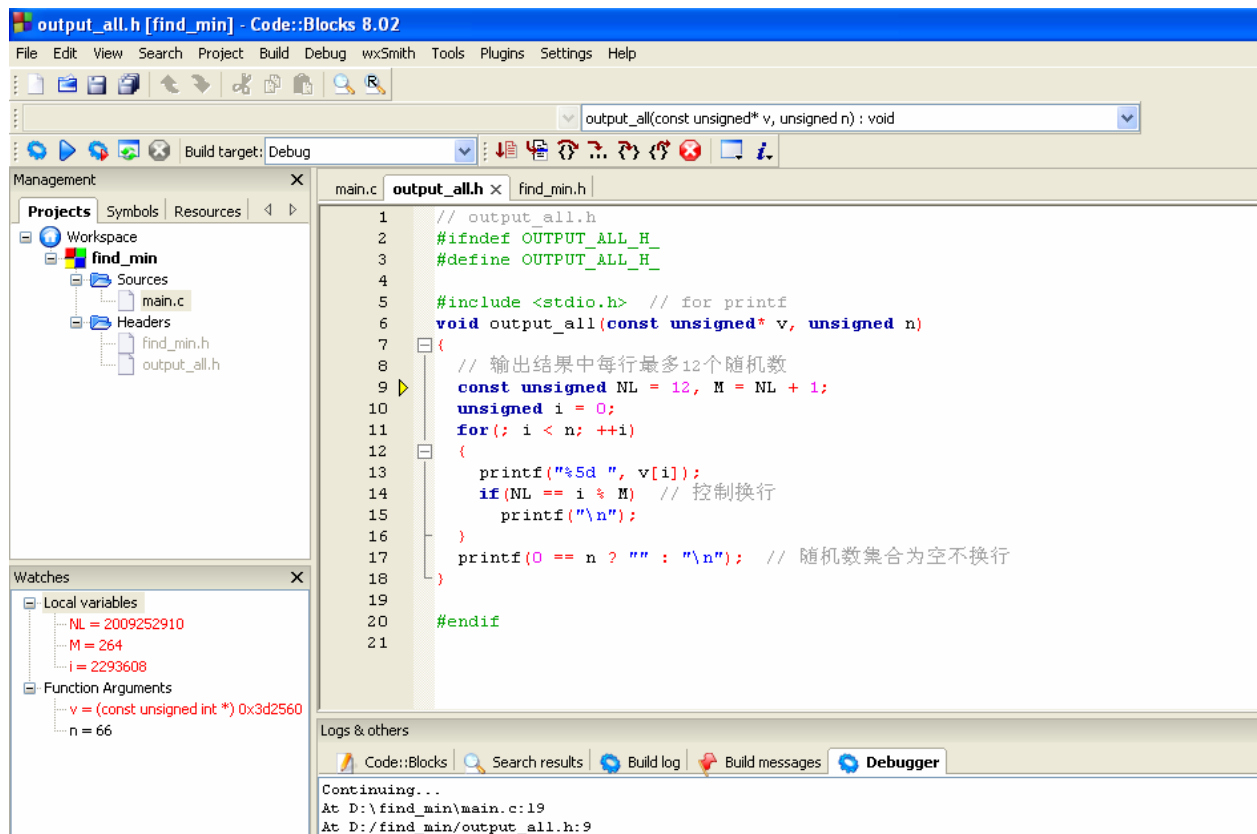
(5) 执行到第17行

把光标放到第17行代码前面，点击Run to Cursor 按钮，见下图。

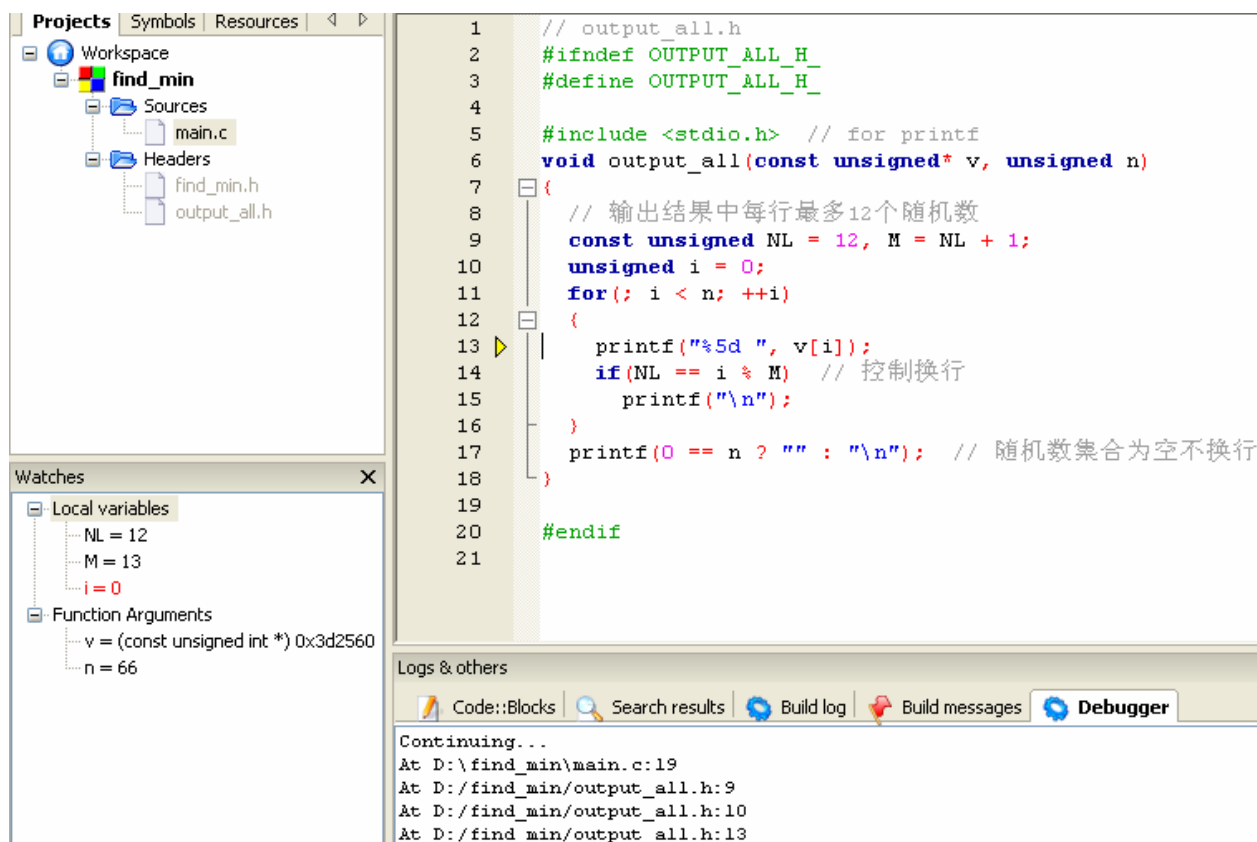


(6) 执行到output_all函数

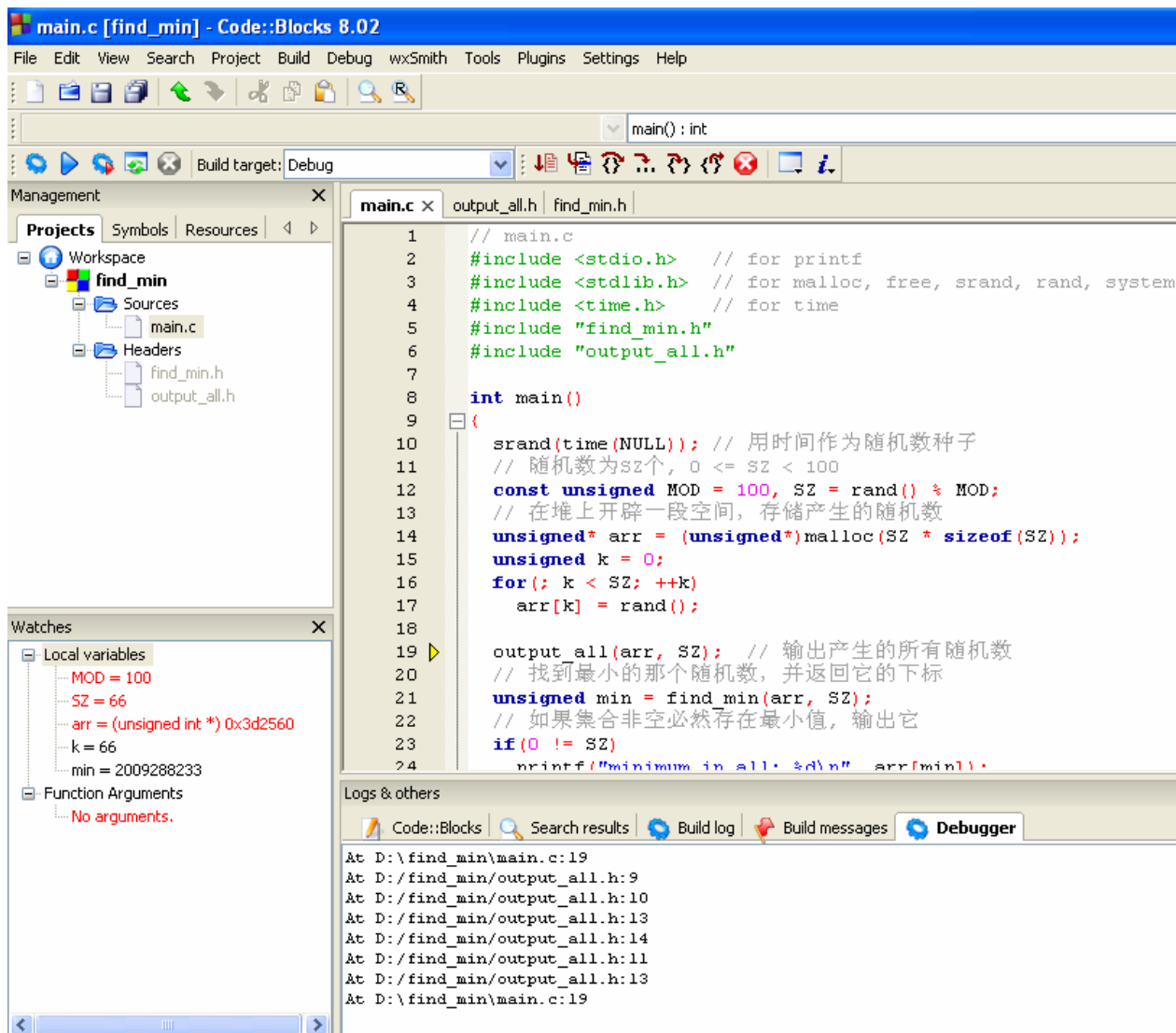
把光标置于第19行前，执行到此行，然后用鼠标点击Step into，则出现如下界面。



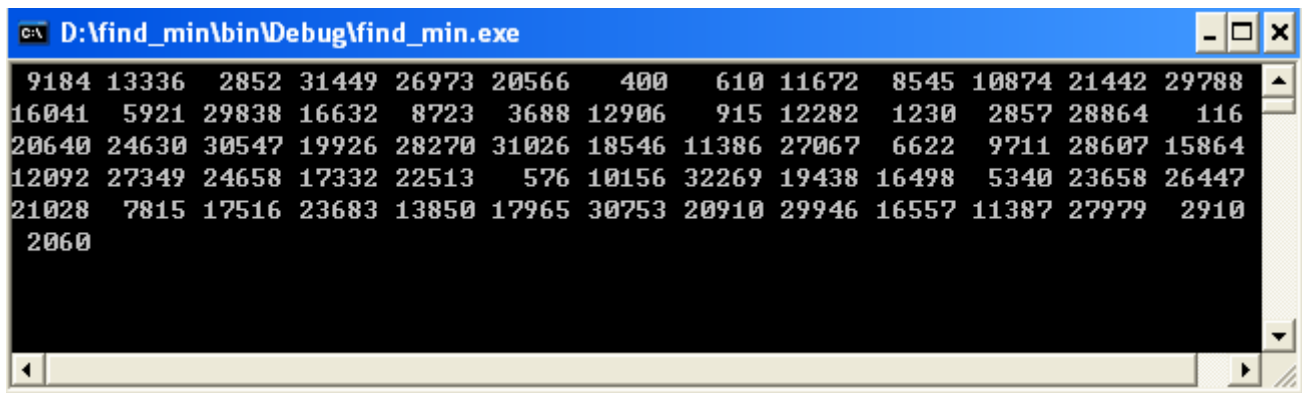
从上图可以看出，由于执行到函数output_all的第9行，函数参数v和n已经被赋值，但是NL, M, i三个局部变量都是系统赋予的随机值，但是执行到第13行前的时候，三个局部变量都已被正常赋值，见下图。



点击Step out按钮，则跳出output_all函数体，此时output_all函数已经执行完毕，见下图。



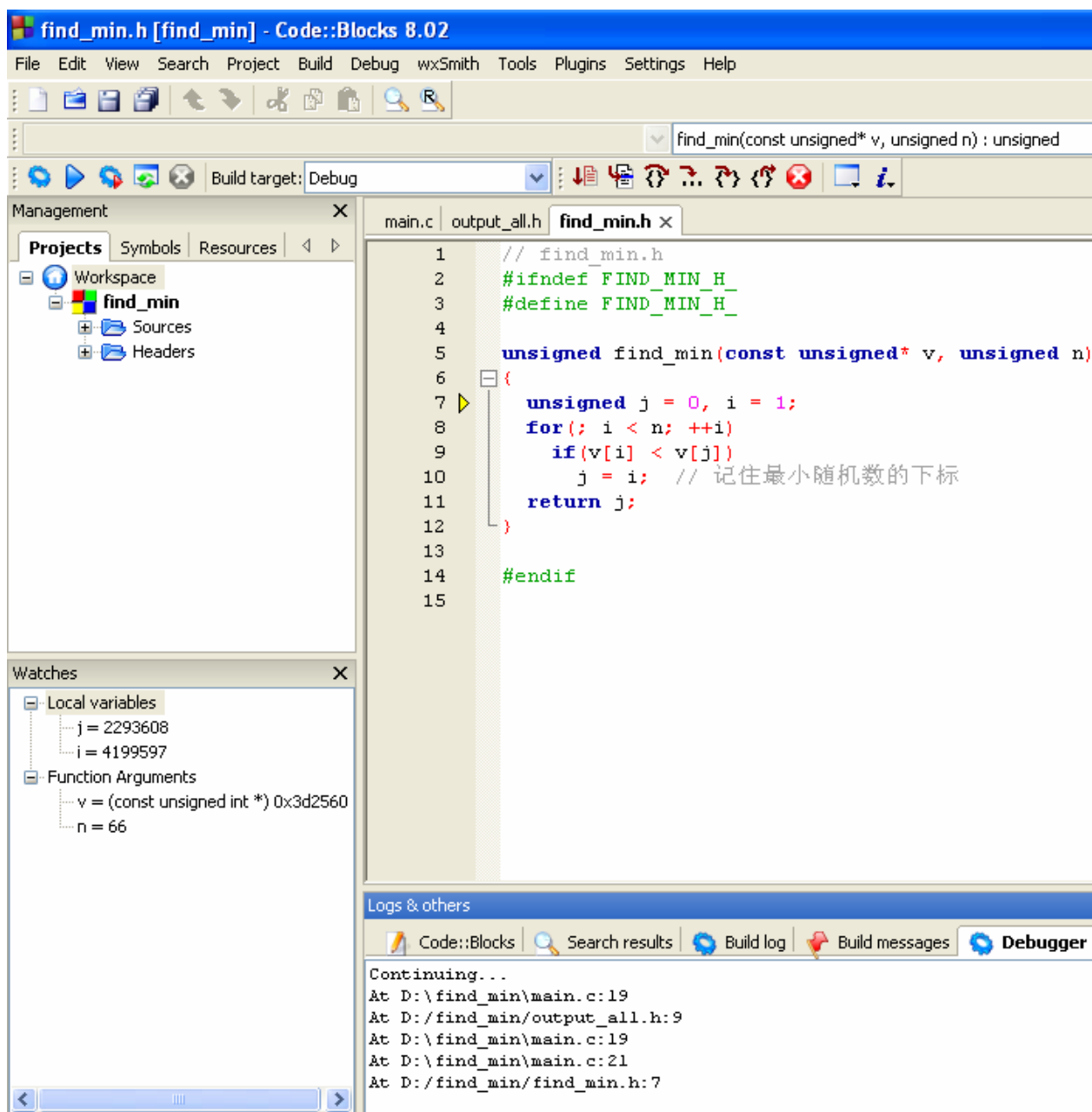
从上图日志Debugger栏目的信息可以看出，output_all函数已经执行完毕，重新进入main.c函数体，并已经执行完第19行代码，并且屏幕显示了一些信息(output_all函数刚执行完毕后的结果)，见下图。




(7) 执行到find_min函数


把光标置于main函数第21行前，然后点击Run to Cursor按钮，再点击Step into按钮，则进入

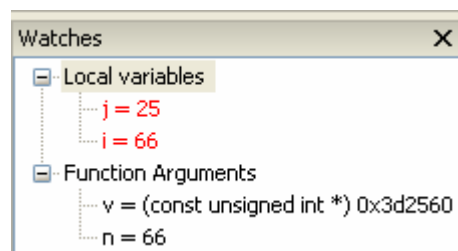
find_min函数体，见下图。

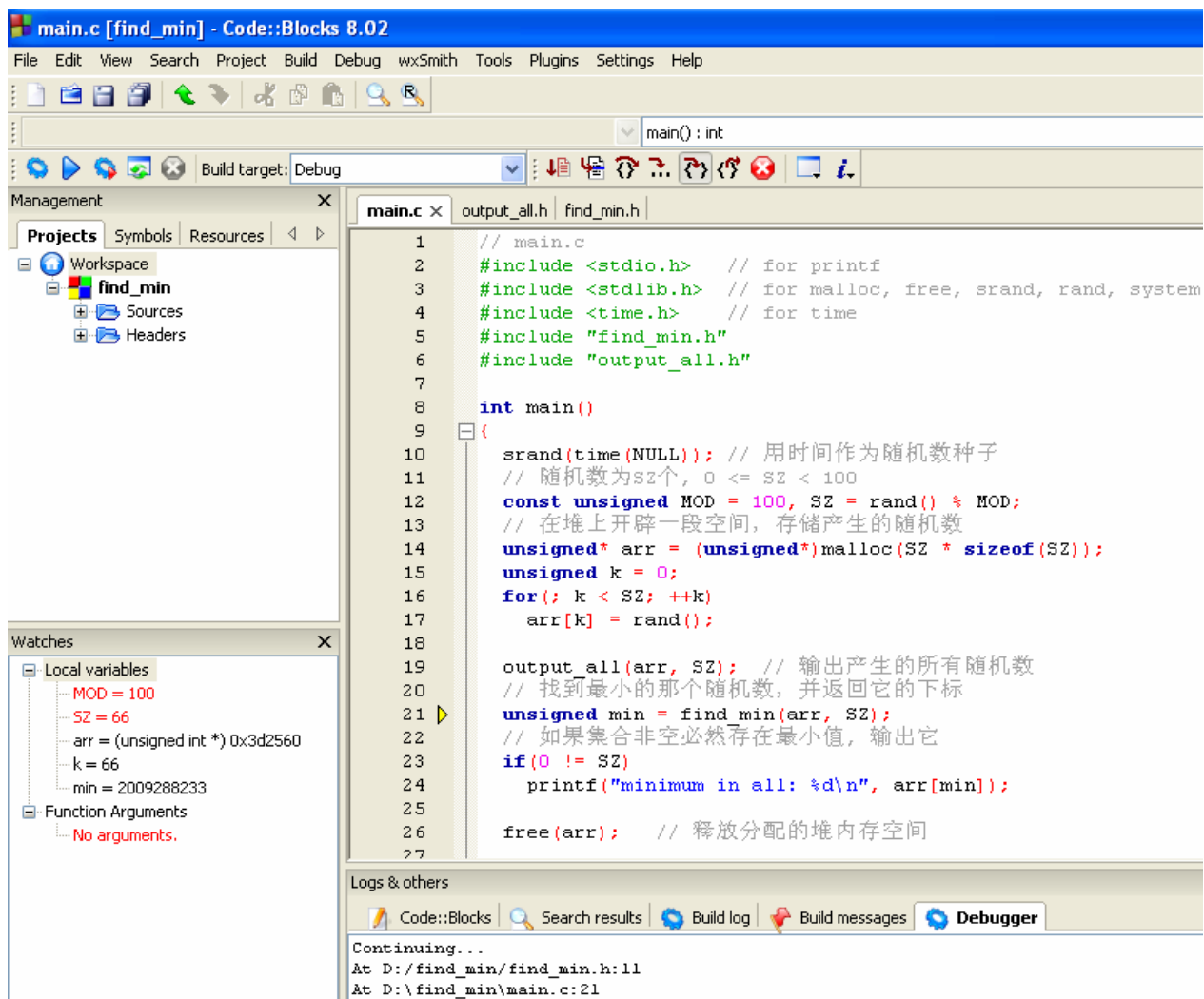


从上图可以看出，已经执行到`find_min`函数的第7行代码前，此时`find_min`函数的参数已经被赋值，但是局部变量`j`和`i`仍然是系统赋予的随机值(因为第7行代码尚未执行)。

把光标置于第11行代码前，点击Run to Cursor  按钮，此时局部变量`j`已经得到确定值25(最小值下标)，由于循环体已经执行完毕，因此`i`得到66(跟传递进来的数组大小值相同，等于该次`n`的值)，见右边Watches窗口的截图。

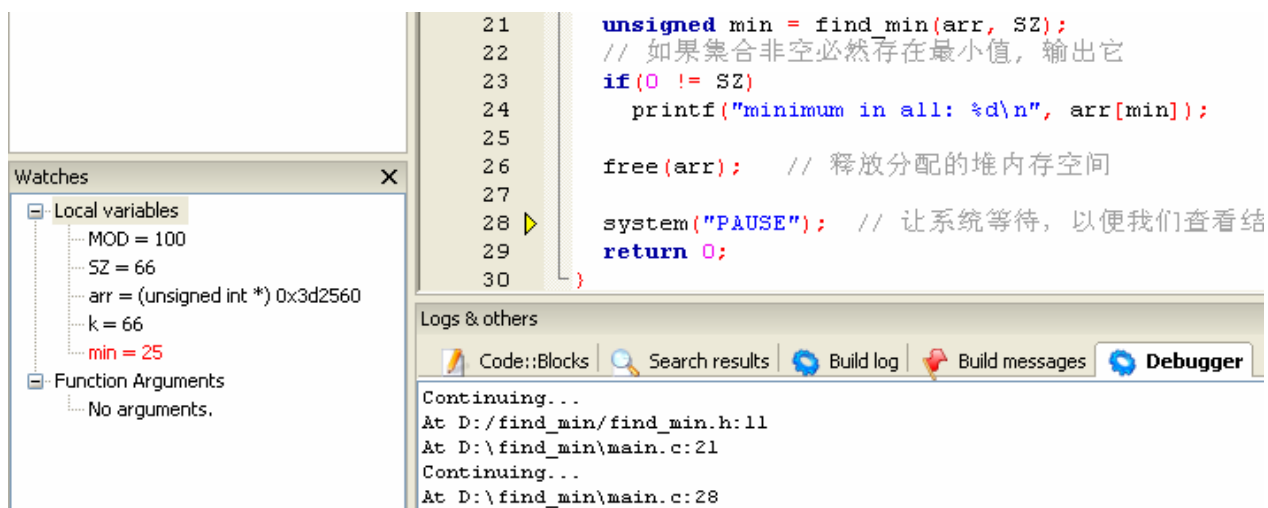
点击Step out  按钮，则跳出`find_min`函数体，重新进入`main`函数，见下图。



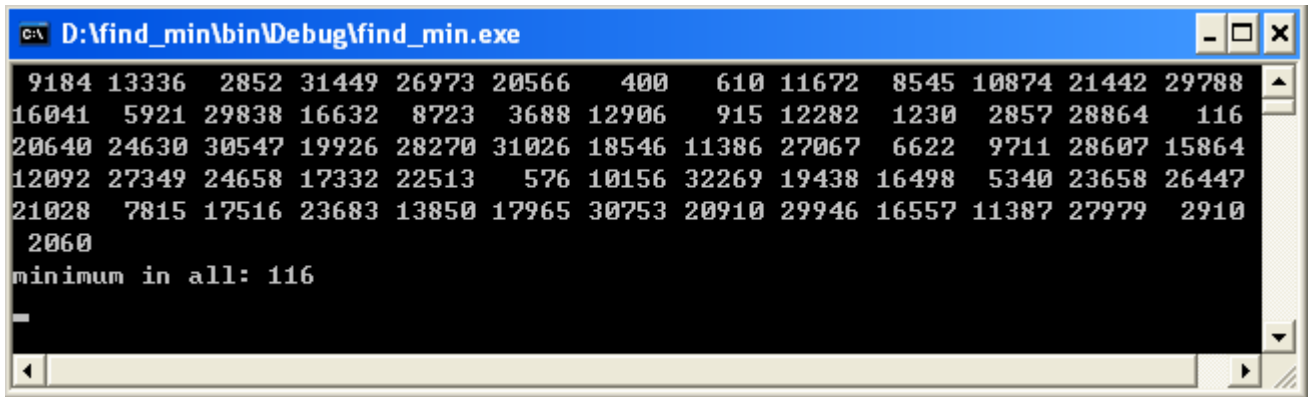


(8) 输出最小值


把光标置于第28行代码前面，点击Run to Cursor 按钮，可以看到Watches窗口已经显示局部变量min的值为25，见下图。

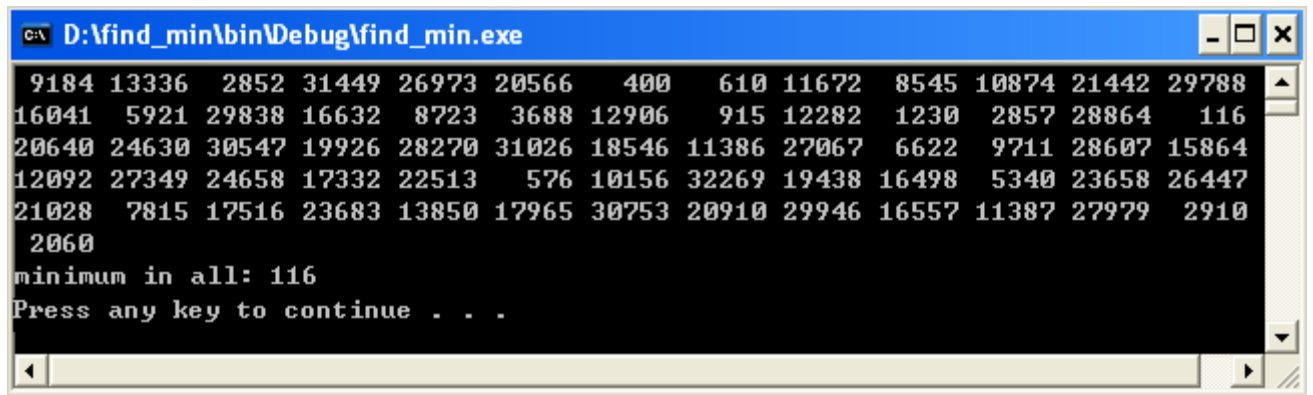


由于main函数同时输出信息窗口也显示了最小值，见下图。




```
C:\ D:\find_min\bin\Debug\find_min.exe
9184 13336 2852 31449 26973 20566 400 610 11672 8545 10874 21442 29788
16041 5921 29838 16632 8723 3688 12906 915 12282 1230 2857 28864 116
20640 24630 30547 19926 28270 31026 18546 11386 27067 6622 9711 28607 15864
12092 27349 24658 17332 22513 576 10156 32269 19438 16498 5340 23658 26447
21028 7815 17516 23683 13850 17965 30753 20910 29946 16557 11387 27979 2910
2060
minimum in all: 116
```

用鼠标点击Continue  按钮继续运行，则输出如下信息，见下图(如果您使用的中文版操作系统，最下一行信息可能是：按任意键继续. . .)。



```
C:\ D:\find_min\bin\Debug\find_min.exe
9184 13336 2852 31449 26973 20566 400 610 11672 8545 10874 21442 29788
16041 5921 29838 16632 8723 3688 12906 915 12282 1230 2857 28864 116
20640 24630 30547 19926 28270 31026 18546 11386 27067 6622 9711 28607 15864
12092 27349 24658 17332 22513 576 10156 32269 19438 16498 5340 23658 26447
21028 7815 17516 23683 13850 17965 30753 20910 29946 16557 11387 27979 2910
2060
minimum in all: 116
Press any key to continue . . .
```

程序运行完毕，用鼠标点击Stop debugger  按钮关闭调试器即可。以上程序中也可以不用在堆上动态分配空间(见main函数第14行代码)，由于需要的空间不太大，可以使用数组在栈上分配空间即可。

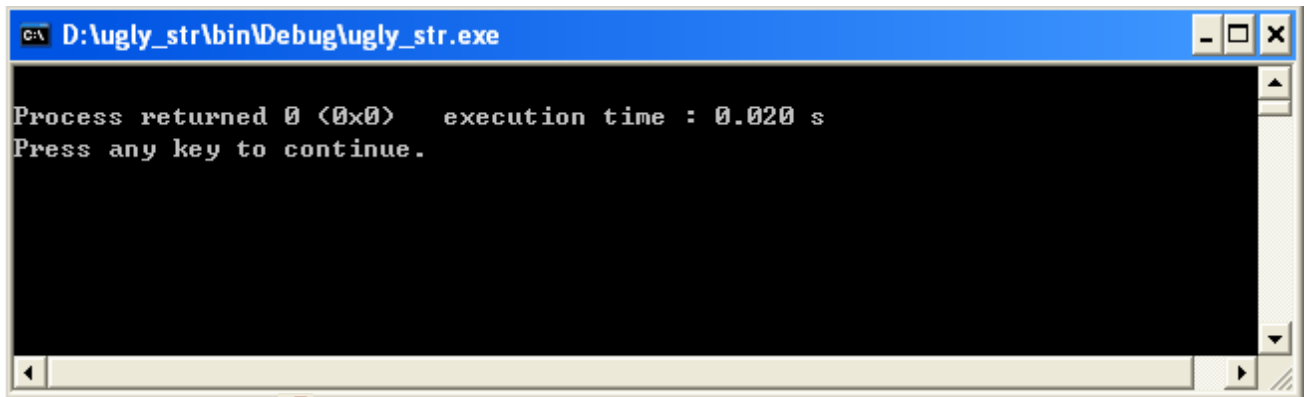
本小结例子中的程序编译运行结果正确，没有逻辑错误，这样的程序无需调试，只不过为了说明如何使用gdb调试器而进行调试的。


3.5.2 面向对象风格的程序

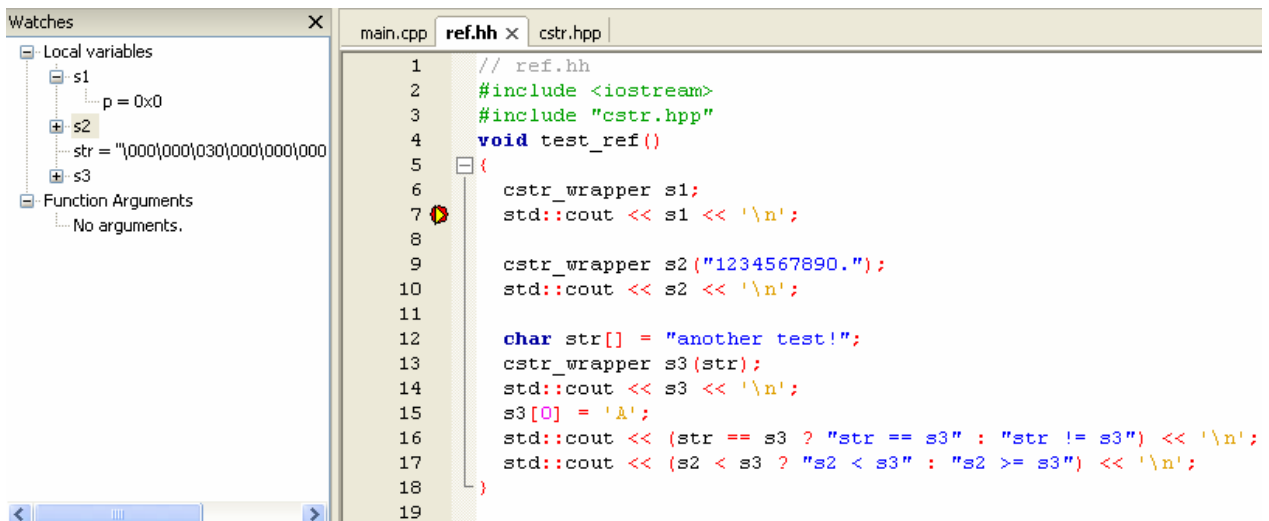
以下有一个cstr_wrapper的类，用来封装C风格的字符串，仅仅对C风格的字符串进行引用(不做深层复制)，提供了一些必要的成员函数。本程序中不止一处有逻辑错误，现在使用gdb调试器来找出错误的位置和原因。


建立一个工程，取名为ugly_str(取这个名字的原因很简单，因为这样设计封装C风格字符串很愚蠢)，该工程中有四个文件，main.cpp, cstr.hpp, ref.hh, copy.hh。其中cstr.hpp定义了cstr_wrapper，对C风格字符串进行封装，ref.hh和copy.hh是两个测试函数，分别测试引用和复制字符串时是否工作正常。建立的这个工程跟上小节3.5.1中共用一个Workspace(工作空间名字随便取)，编译或者调试这个工程前需要激活它(让系统知道你对它操作，而不是对原来的find_min工程操作)，激活的方法很简单，选中工程名ugly_str，按下鼠标右键弹出的菜单中选择Activate project，则该工程此时就处于激活状态，原来激活状态的find_min工程进入休眠。

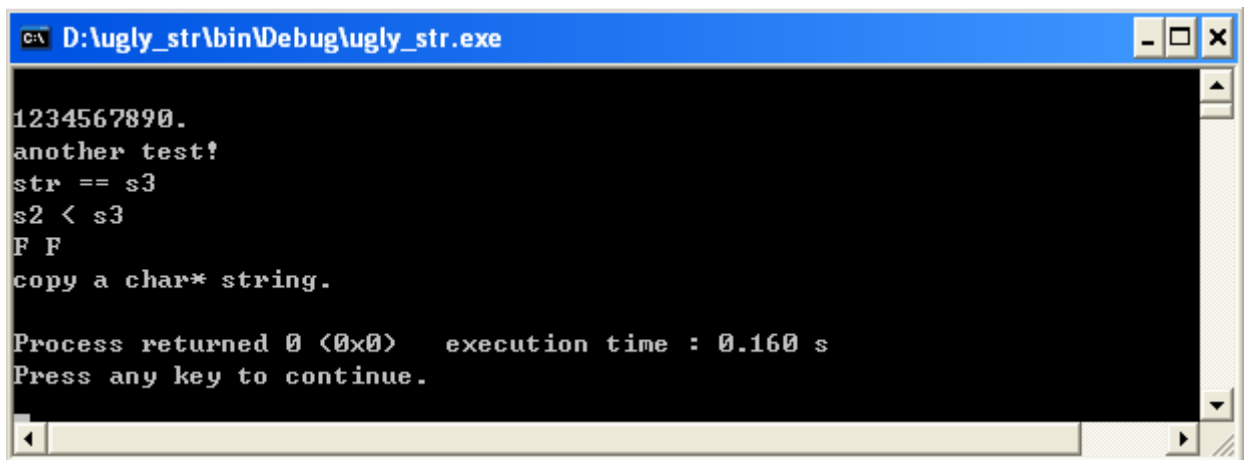
编译并运行之，见下面弹出的窗口，居然没有显示任何信息！



设置一个断点，点击按钮运行之，打开Watches窗口，见下图。



从上图中可以看出，已经建立对象s1，但是它的值(指针)为0，不指向字符串。在32位Windows系统上，操作系统占用低地址，地址为0的内存禁止用户程序访问，访问系统占用内存是非法操作，如果您懂gcc汇编的话，打开汇编窗口，或者用鼠标点击Next instruction按钮就可以看到汇编代码，更容易看出问题。查看cstr_wrapper的构造函数发现，有这样一个函数头cstr_wrapper(const char* str = NULL); 显然这里有问题，空串应该是""，而不是NULL，改正过来重新编译运行，结果如下。



从上图可以看出ref.hh函数测试已经通过，但是copy.hh测试输出的第一个字符串就错了，结合源代

码，先注释掉test_ref();再对test_copy();进行跟踪。

程序代码见下面的贴图。

```
1 // main.cpp
2 #include "copy.hh"
3 #include "ref.hh"
4
5 int main()
6 {
7     test_ref();
8     test_copy();
9     return 0;
10 }
11
```

```
1 // copy.hh
2 #include "cstr.hpp"
3 cstr_wrapper test_copy1()
4 {
5     char str[] = "copy a char[] string.";
6     return cstr_wrapper(str);
7 }
8
9 cstr_wrapper test_copy2()
10 {
11     char* str = "copy a char* string.";
12     return cstr_wrapper(str);
13 }
14
15 #include <iostream>
16 void test_copy()
17 {
18     cstr_wrapper str1 = test_copy1();
19     std::cout << str1 << ' ';
20     str1[0] = 'C';
21     std::cout << str1 << '\n';
22
23     cstr_wrapper str2 = test_copy2();
24     std::cout << str2 << '\n';
25 }
26
```

```
1 // cstr.hpp
2 #ifndef CSTR_WRAPPER_HPP_
3 #define CSTR_WRAPPER_HPP_
4
5 #include <cstring>
6 #include <iostream>
7 #include <stdexcept>
8
9 class cstr_wrapper
10 {
11 public:
12     cstr_wrapper(const char* str = NULL);
13     cstr_wrapper(const cstr_wrapper&);
14
15     unsigned size() const; // 字符串中的字符个数，不包括结束符
16     bool empty() const; // 判断是否为空串
17     char& operator [] (unsigned); // 引用某个字符，不检查下标
18     const char& operator [] (unsigned) const; // 引用某个字符，检查下标
19     const char& at(unsigned) const; // 引用某个字符，检查下标
20     const char* c_str() const; // 转换成c风格的字符串
21     // 其它成员函数，略...
22     // 字符串比较，重载<和==
23     friend bool operator < (const cstr_wrapper&, const cstr_wrapper&);
24     friend bool operator == (const cstr_wrapper&, const cstr_wrapper&);
25     // 输出字符串，重载<<
26     friend std::ostream& operator << (std::ostream&, const cstr_wrapper&);
27
28 private:
29     char* p;
30 };

```



```

31
32 // 构造函数
33 cstr_wrapper::cstr_wrapper(const char* str) : p(const_cast<char*>(str)) {}
34 cstr_wrapper::cstr_wrapper(const cstr_wrapper& s) : p(s.p) {}
35 // 字符串大小, 以及取某个字符 (不检查下标)
36 unsigned cstr_wrapper::size() const { return std::strlen(p); }
37 bool cstr_wrapper::empty() const { return 0 == std::strcmp(p, ""); }
38 const char& cstr_wrapper::operator [] (unsigned i) const { return p[i]; }
39 char& cstr_wrapper::operator [] (unsigned i) { return p[i]; }
40 // 取某个字符时, 对下标进行检查
41 const char& cstr_wrapper::at(unsigned i) const
42 {
43     if(i < std::strlen(p))
44         return p[i];
45     else
46         throw std::out_of_range("subscript error");
47 }
48 // 转换成c风格的字符串
49 const char* cstr_wrapper::c_str() const { return p; }
50 // 判断字符串大小
51 bool operator < (const cstr_wrapper& s1, const cstr_wrapper& s2)
52 {
53     return std::strcmp(s1.p, s2.p) < 0;
54 }
55 // 判断字符串是否相等
56 bool operator == (const cstr_wrapper& s1, const cstr_wrapper& s2)
57 {
58     return 0 == std::strcmp(s1.p, s2.p);
59 }
60 // 输出一个字符串
61 std::ostream& operator << (std::ostream& os, const cstr_wrapper& s)
62 {
63     return os << s.p;
64 }
65
66 // ref.hh
67 #include <iostream>
68 #include "cstr.hpp"
69 void test_ref()
70 {
71     cstr_wrapper s1;
72     std::cout << s1 << '\n';
73
74     cstr_wrapper s2("1234567890.");
75     std::cout << s2 << '\n';
76
77     char str[] = "another test!";
78     cstr_wrapper s3(str);
79     std::cout << s3 << '\n';
80     s3[0] = 'A';
81     std::cout << (str == s3 ? "str == s3" : "str != s3") << '\n';
82     std::cout << (s2 < s3 ? "s2 < s3" : "s2 >= s3") << '\n';
83 }
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

```



1 // main.cpp
2 #include "copy.hh"
3 #include "ref.hh"
4
5 int main()
6 {
7     //test_ref();
8     test_copy();
9     return 0;
10 }
11

```

```

15 #include <iostream>
16 void test_copy()
17 {
18     cstr_wrapper str1 = test_copy1();
19     std::cout << str1 << ' ';
20     str1[0] = 'C';
21     std::cout << str1 << '\n';
22
23     cstr_wrapper str2 = test_copy2();
24     std::cout << str2 << '\n';
25 }

```

设置好断点，点击  按钮启动调试器，见上图，打开Watches窗口，进行跟踪。点击Step into  按钮，进入test_copy1的函数体，见下图。

```

1 // copy.hh
2 #include "cstr.hpp"
3 cstr_wrapper test_copy1()
4 {
5     char str[] = "copy a char[] string.";
6     return cstr_wrapper(str);
7 }

```

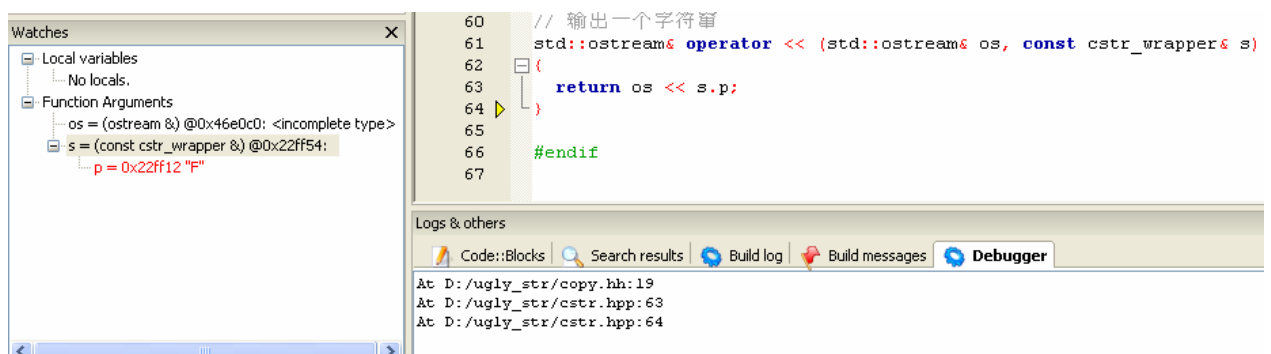
继续Step into  跟踪对象的创建过程，见下图。

```

32 // 构造函数
33 cstr_wrapper::cstr_wrapper(const char* str) : p(const_cast<char*>(str)) {}
34 cstr_wrapper::cstr_wrapper(const cstr_wrapper& s) : p(s.p) {}
35 //字符串大小，以及取某个字符(不检查下标)
36 unsigned cstr_wrapper::size() const { return std::strlen(p); }
37 bool cstr_wrapper::empty() const { return 0 == std::strcmp(p, ""); }
38 const char& cstr_wrapper::operator [] (unsigned i) const { return p[i]; }
39 char& cstr_wrapper::operator [] (unsigned i) { return p[i]; }

```

从上面的截图可以看出，调用cstr_wrapper(const char*);创建对象，再结合test_copy1函数的代码可以发现虽然返回一个临时对象的拷贝没有任何问题，但cstr_wrapper的定义中构造函数仅仅拷贝了指针，待test_copy1函数体内的str过期被销毁的时候，新创建的对象(仅仅一个指针)将指向何处？继续跟踪到调用重载operator<<的友元函数输出时，从Watches可以发现此时创建的对象值(指针)，已经指向一个莫名其妙的字符串"F"(在您的电脑上可能是其他字符串内容)。



The screenshot shows the Visual Studio IDE. On the left, the 'Watches' window is open, displaying the state of variables during a debug session. It shows 's = (const cstr_wrapper &) @0x22ff54' and its internal pointer 'p = 0x22ff12 'F''. On the right, the code editor shows the implementation of the test_copy function, with a breakpoint set at line 63. The code includes the output of the string 'copy a char[] string.' followed by a space and then the modified string 'copy a char[] string.C'.

由于cstr_wrapper本身定义就是复制一个指针，没有办法对它进行大的改动(除非重新设计和编码)，

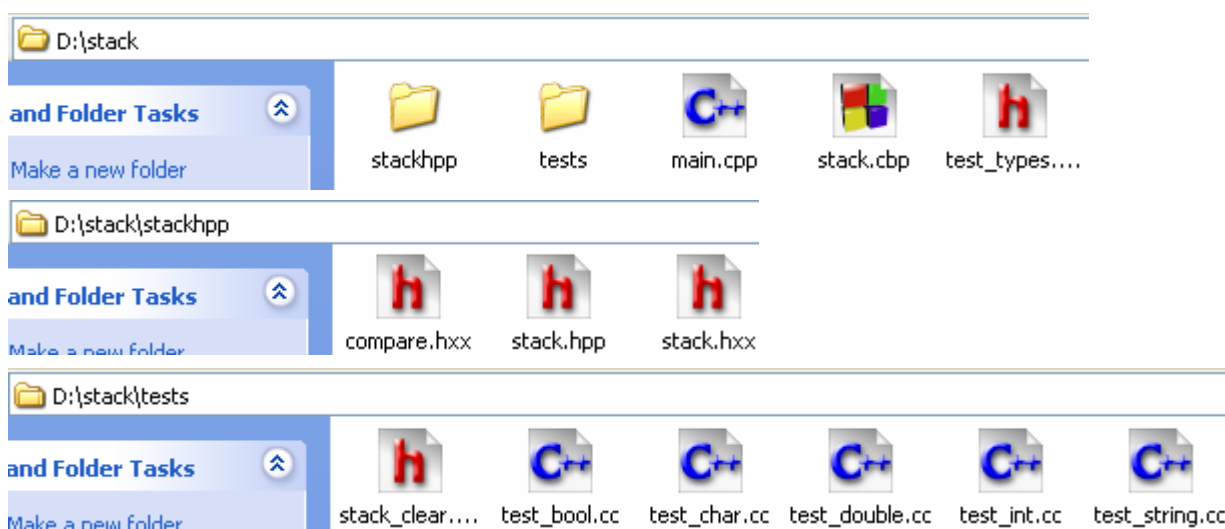
只能遗弃不用`cstr_wrapper`。

也许有的读者还想不明白为什么`test_copy2`函数能正常工作。虽然这个函数也是返回一个拷贝的对象(指针)，但是它的函数体内有一条语句`char* str = "copy a char* string."`；由于C++本身设计为了兼容C的语法，而C语言使用这种格式定义字符串，在程序停止运行前不会被销毁，程序运行结束后由操作系统回收占用的资源，因此拷贝指向这个字符串的指针在程序运行的生命周期内始终不会成为野指针，自然就能正常输出了。

3.5.3 泛型程序

泛型程序调试更复杂一点，因为一个函数或者类往往由于初始化类型的不同，生成不同种类的对象，而且函数和运算符重载过程也比一般的函数和运算符重载复杂的多。

以下是一个简易栈的实现以及几个运算符重载，为了确保该栈实现的健壮性，也对它进行一些必要的测试。下面是各文件目录层次关系。



下面是代码贴图。

```
1 // compare.hxx
2 #ifndef COMPARE_HXX_
3 #define COMPARE_HXX_
4
5 //比较任何可比的简单类型或者复杂类型(包括容器类型)大小
6 template <typename T>
7 bool operator != (const T& x, const T& y) { return !(x == y); }
8
9 template <typename T>
10 bool operator <= (const T& x, const T& y) { return !(y < x); }
11
12 template <typename T>
13 bool operator > (const T& x, const T& y) { return y < x; }
14
15 template <typename T>
16 bool operator >= (const T& x, const T& y) { return !(x < y); }
17
18 #endif
19
```

```

1 // main.cpp
2 #include "test_types.hpp"
3 int main()
4 {
5     test_types();
6     return 0;
7 }
8

```

```

1 // stack.hpp
2 #ifndef STACK_HPP_
3 #define STACK_HPP_
4
5 #include "compare.hxx"
6 #include "stack.hxx"
7
8 #endif
9

```

```

1 // stack.hxx
2 #ifndef STACK_HXX_
3 #define STACK_HXX_
4
5 #include <iostream>
6 #include <stdexcept>
7 #include <cstdlib>
8 #include <cstring>
9
10 template <class T> // T必须是POD类型
11 class Stack
12 {
13 public:
14     typedef unsigned long size_type;
15     typedef T& reference;
16     typedef const T& const_reference;
17 public:
18     Stack() : cap(SZ), sz(0) { data = get(cap); }
19     Stack(unsigned n, const T& d = T()) : cap(SZ), sz(n)
20     {
21         while(cap < sz) cap <<= 1;
22         data = get(cap);
23         std::memset(data, d, tsize() * sz);
24     }
25     Stack(const Stack& st) : cap(st.cap), sz(st.sz), data(get(cap))
26     {
27         std::memcpy(data, st.data, sz * tsize());
28     }
29     ~Stack()
30     {
31         delete [] data;
32         sz = cap = 0;
33     }
34
35     Stack& operator = (const Stack& s) // 可以赋值
36     {
37         if(this != &s)
38         {
39             delete [] data;
40             cap = s.cap;
41             data = get(cap);
42             sz = s.sz;
43             std::memcpy(data, s.data, sz * tsize());
44         }
45         return *this;
46     }

```

```

47 unsigned size() const { return sz; } // 栈中元素个数
48 bool empty() const { return 0 == sz; } // 栈是否空
49 const T& top() const // 栈顶元素
50 {
51     if (sz < 1) Underflow_Error();
52     return data[sz - 1];
53 }
54 T& top()
55 {
56     if (sz < 1) Underflow_Error();
57     return data[sz - 1];
58 }
59 void push(const T& d) // 入栈
60 {
61     if (sz + 1 > cap)
62         rebuild();
63     data[sz++] = d;
64 }
65 void pop() // 删除栈顶元素
66 {
67     if (sz < 1) Underflow_Error();
68     --sz;
69 }
70
71 bool operator == (const Stack& st) const
72 {
73     return 0 == std::memcmp(data, st.data, sz * tsize());
74 }
75
76 bool operator < (const Stack& st) const
77 {
78     return std::memcmp(data, st.data, sz * tsize()) < 0;
79 }
80 void swap(Stack& s) // 交换两个栈
81 {
82     unsigned t;
83     t = cap, cap = s.cap, s.cap = t;
84     t = sz, sz = s.sz, s.sz = t;
85     T* d;
86     d = data, data = s.data, s.data = d;
87 }
88 protected:
89 T* get(unsigned n) // 分配内存
90 {
91     T* p;
92     try { p = new T[n]; }
93     catch(const std::bad_alloc& ba)
94     {
95         std::cerr << ba.what();
96         throw;
97     }
98     return p;
99 }

```

```

100
101     void rebuild()                                // 栈扩大为原来的2倍
102     {
103         cap <<= 1;
104         T* p = get(cap);
105         std::memcpy(p, data, sz * tsize());
106         delete [] data;
107         data = p;
108     }
109
110 private:
111     void Underflow_Error() { throw std::underflow_error("Stackunderflowed"); }
112     unsigned tsize() const { return sizeof(T); }
113     static const unsigned SZ = 1;
114     unsigned cap, sz;
115     T* data;
116 };
117
118 #endif
119
120 // test_types.hpp
121 #include <cstdlib>
122 #include <ctime>
123
124 #include "tests/test_int.cc"
125 #include "tests/test_double.cc"
126 #include "tests/test_char.cc"
127 #include "tests/test_bool.cc"
128 #include "tests/test_string.cc"
129
130 static void init_rand() { std::srand(std::time(0)); }
131 void test_types()
132 {
133     init_rand();
134
135     test_int();
136     test_double();
137     test_char();
138     test_bool();
139     test_string();
140 }
141
142 // stack_clear.hpp
143 #ifndef STACK_CLEAR_HPP_
144 #define STACK_CLEAR_HPP_
145
146 #include "../stackhpp/stack.hpp"
147
148 // 清空一个栈
149 template <typename T>
150 void stack_clear(Stack<T>& s)
151 {
152     if (!s.empty())
153     {
154         Stack<T> st; // 建立一个临时空栈
155         st.swap(s); // s存储st的元素
156     } // st已经被销毁
157 }
158
159 #endif
160
161
162
163
164
165
166
167
168
169

```

```

1 // test_string.cc
2 #include <cstdlib>
3 #include <string>
4 #include <iostream>
5 #include "stack_clear.hpp"
6
7 void rand_str(std::string& str) // 产生随机字符串，仅仅是字母和数字的组合
8 {
9     const std::string
10     carr("0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz");
11     for(std::string::size_type i = std::rand() % 75; i > 0; --i)
12         str.push_back(carr[std::rand() % carr.size()]);
13 }
14 void test_string() // 测试Stack<std::string>各个成员函数以及比较运算符
15 {
16     typedef Stack<std::string>::size_type size_type;
17
18     std::cout << "test strings:\n";
19     //std::srand(std::time(0));
20     const size_type M = 11, SZ1 = rand() % M, SZ2 = std::rand() % M;
21     Stack<std::string> sts1, sts2;
22     std::string str;
23     // 用随机字符串填充Stack<std::string> sts1
24     for(unsigned i = 0; i < SZ1; ++i)
25     {
26         rand_str(str);
27         sts1.push(str);
28         str.clear();
29     }
30
31     for(unsigned i = 0; i < SZ2; ++i)
32     {
33         rand_str(str);
34         sts2.push(str);
35         str.clear();
36     }
37
38     std::cout << "size of Stack: " << sts1.size() << ' ' << sts2.size() << '\n';
39     std::cout << "top element: " << (sts1.empty() ? "" : sts1.top()) << '\n';
40     std::cout << "top element: " << (sts2.empty() ? "" : sts2.top()) << '\n';
41     std::cout << "comparing Stack sts1 and sts2:\n";
42     std::cout << (sts1 != sts2 ? "sts1 != sts2" : "sts1 == sts2") << '\n';
43     std::cout << (sts1 > sts2 ? "sts1 > sts2" : "sts1 <= sts2") << '\n';
44     std::cout << (sts1 >= sts2 ? "sts1 >= sts2" : "sts1 < sts2") << '\n';
45     sts2.swap(sts1);
46     std::cout << (sts1 <= sts2 ? "sts1 <= sts2" : "sts1 > sts2") << '\n';
47
48     stack_clear(sts1);
49     stack_clear(sts2);
50     std::cout << "strings test ok!\n\n";
51 }
52

```



```

1 // test_int.cc
2 #include <cstdlib>
3 #include <iostream>
4 #include "stack_clear.hpp"
5
6 void test_int() // 测试stack<int>各个成员函数以及比较运算符
7 {
8     typedef Stack<int>::size_type size_type;
9
10    std::cout << "test integers:\n";
11    //std::srand(std::time(0));
12    const size_type M = 11, SZ1 = std::rand() % M, SZ2 = std::rand() % M;
13    Stack<int> st1, st2;
14    // 用随机数填充Stack<int> st1
15    for(size_type i = 0; i < SZ1; ++i)
16        st1.push(std::rand());
17    for(size_type i = 0; i < SZ2; ++i)
18        st2.push(std::rand());
19
20    std::cout << "size of Stack: " << st1.size() << ' ' << st2.size() << '\n';
21    std::cout << "top element: ";
22    if(!st1.empty())
23        std::cout << st1.top() << ' ';
24    if(!st2.empty())
25        std::cout << st2.top() << '\n';
26    std::cout << "comparing Stack st1 and st2:\n";
27    std::cout << (st1 != st2 ? "st1 != st2" : "st1 == st2") << '\n';
28    std::cout << (st1 > st2 ? "st1 > st2" : "st1 <= st2") << '\n';
29    std::cout << (st1 >= st2 ? "st1 >= st2" : "st1 < st2") << '\n';
30    st2.swap(st1);
31    std::cout << (st1 <= st2 ? "st1 <= st2" : "st1 > st2") << '\n';
32
33    stack_clear(st1);
34    stack_clear(st2);
35    std::cout << "integers test ok!\n\n";
36 }
37

```

```

1 // test_bool.cc
2 #include <cstdlib>
3 #include <iostream>
4 #include "stack_clear.hpp"
5
6 void test_bool() // 测试stack<bool>成员函数以及比较运算符
7 {
8     typedef Stack<bool>::size_type size_type;
9
10    std::cout << "test booleans:\n";
11    const size_type M = 11, SZ1 = std::rand() % M, SZ2 = std::rand() % M;
12    Stack<bool> stb1, stb2;
13    // 用bool值true或者false填充stb1
14    for(size_type i = 0; i < SZ1; ++i)
15        stb1.push(std::rand() & 1);
16    for(size_type i = 0; i < SZ2; ++i)
17        stb2.push(std::rand() & 1);
18

```

```

18
19     std::cout << "size of Stack: " << stb1.size() << ' ' << stb2.size() << '\n';
20     std::cout << "top element: ";
21     std::cout << (stb1.empty() ? "" : (stb1.top() == true ? "true" : "false")) << ' ';
22     std::cout << (stb2.empty() ? "" : (stb2.top() == true ? "true" : "false")) << '\n';
23     std::cout << "comparing Stack stb1 and stb2:\n";
24     std::cout << (stb1 != stb2 ? "stb1 != stb2" : "stb1 == stb2") << '\n';
25     std::cout << (stb1 > stb2 ? "stb1 > stb2" : "stb1 <= stb2") << '\n';
26     std::cout << (stb1 >= stb2 ? "stb1 >= stb2" : "stb1 < stb2") << '\n';
27     stb2.swap(stb1);
28     std::cout << (stb1 <= stb2 ? "stb1 <= stb2" : "stb1 > stb2") << '\n';
29
30     stack_clear(stb1);
31     stack_clear(stb2);
32     std::cout << "booleans test ok!\n\n\n";
33 }
34

```

```

1 // test_double.cc
2 #include <cstdlib>
3 #include <iostream>
4 #include "stack_clear.hpp"
5
6 void test_double() // 测试stack<double>各个成员函数以及比较运算符
7 {
8     typedef Stack<double>::size_type size_type;
9
10    std::cout << "test real numbers:\n";
11    //std::srand(std::time(0));
12    const size_type M = 11, SZ1 = std::rand() % M, SZ2 = std::rand() % M;
13    Stack<double> std1, std2;
14    // 用随机浮点数填充Stack<double> std1
15    for(size_type i = 0; i < SZ1; ++i)
16        std1.push(static_cast<double>(std::rand()) / RAND_MAX);
17    for(size_type i = 0; i < SZ2; ++i)
18        std2.push(static_cast<double>(std::rand()) / RAND_MAX);
19
20    std::cout << "size of Stack: " << std1.size() << ' ' << std2.size() << '\n';
21    std::cout << "top element: ";
22    if(!std1.empty())
23        std::cout << std1.top() << ' ';
24    if(!std2.empty())
25        std::cout << std2.top() << '\n';
26    std::cout << "comparing Stack std1 and std2:\n";
27    std::cout << (std1 != std2 ? "std1 != std2" : "std1 == std2") << '\n';
28    std::cout << (std1 > std2 ? "std1 > std2" : "std1 <= std2") << '\n';
29    std::cout << (std1 >= std2 ? "std1 >= std2" : "std1 < std2") << '\n';
30    std2.swap(std1);
31    std::cout << (std1 <= std2 ? "std1 <= std2" : "std1 > std2") << '\n';
32
33    stack_clear(std1);
34    stack_clear(std2);
35    std::cout << "floating points test ok!\n\n";
36 }
37

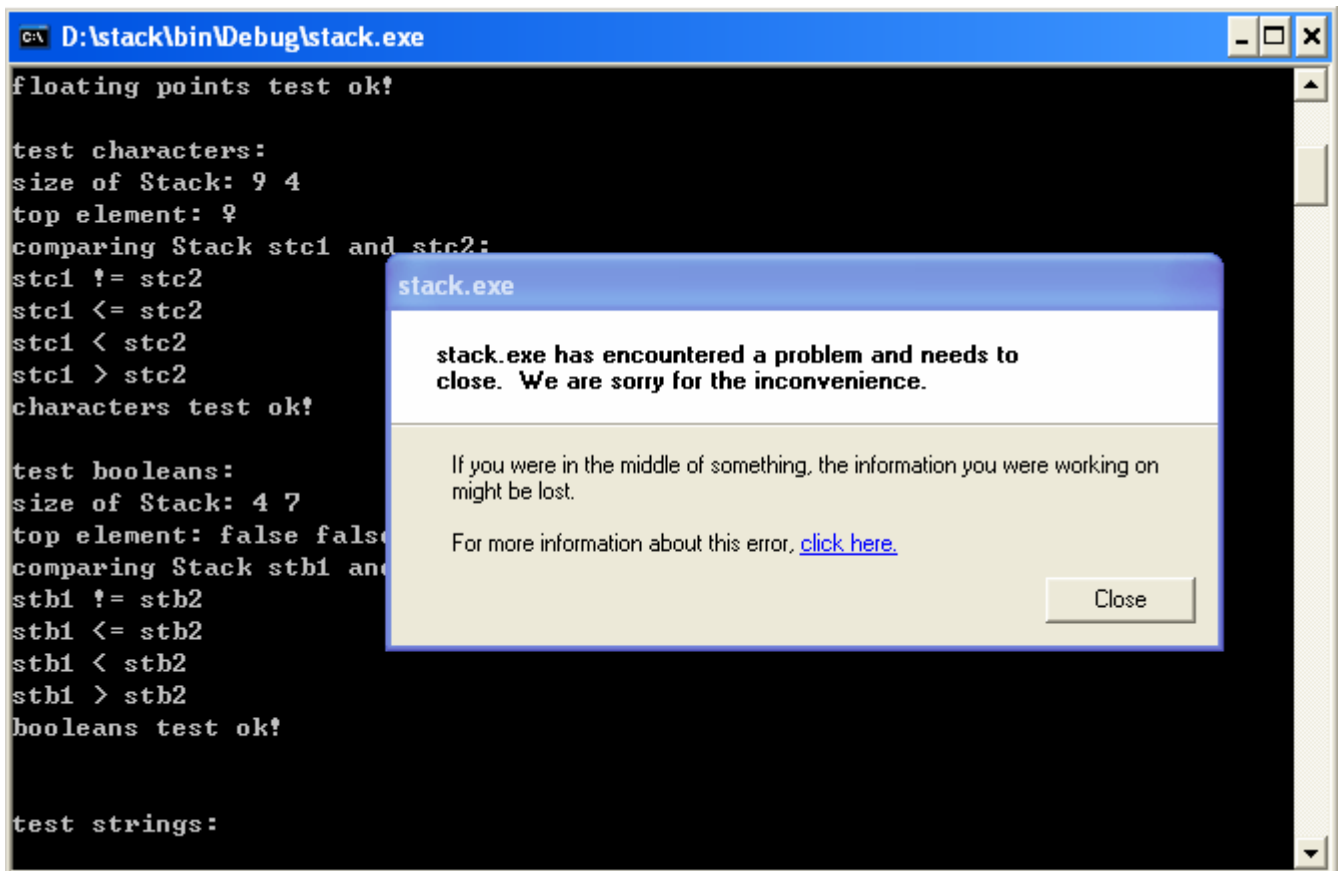
```

```

1 // test_char.cc
2 #include <cstdlib>
3 #include <iostream>
4 #include "stack_clear.hpp"
5
6 void test_char()
7 {
8     typedef Stack<char>::size_type size_type;
9
10    std::cout << "test characters:\n";
11    //std::srand(std::time(0));
12    const size_type M = 11, SZ1 = std::rand() % M, SZ2 = std::rand() % M;
13    Stack<char> stc1, stc2;
14    // 用随机字符填充Stack<char> stc1
15    for(size_type i = 0; i < SZ1; ++i)
16        stc1.push(static_cast<char>(std::rand()));
17    for(size_type i = 0; i < SZ2; ++i)
18        stc2.push(static_cast<char>(std::rand()));
19
20    std::cout << "size of Stack: " << stc1.size() << ' ' << stc2.size() << '\n';
21    std::cout << "top element: ";
22    std::cout << (stc1.empty() ? '\0' : stc1.top()) << ' ';
23    std::cout << (stc2.empty() ? '\0' : stc2.top()) << '\n';
24    std::cout << "comparing Stack stc1 and stc2:\n";
25    std::cout << (stc1 != stc2 ? "stc1 != stc2" : "stc1 == stc2") << '\n';
26    std::cout << (stc1 > stc2 ? "stc1 > stc2" : "stc1 <= stc2") << '\n';
27    std::cout << (stc1 >= stc2 ? "stc1 >= stc2" : "stc1 < stc2") << '\n';
28    stc2.swap(stc1);
29    std::cout << (stc1 <= stc2 ? "stc1 <= stc2" : "stc1 > stc2") << '\n';
30
31    stack_clear(stc1);
32    stack_clear(stc2);
33    std::cout << "characters test ok!\n\n";
34 }
35

```

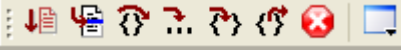
编译并运行该程序，运行时失败，弹出错误信息见下图。从显示结果看，是string测试出了问题。当然，很多人可能早就知道很多C的库函数(例如memcpy, memset等)，不能对对象操作，否则程序很容易崩溃，但是知道为什么吗？如果您并不清楚为什么，那就通过一个简单的例子，来看一看到底究竟怎么回事。

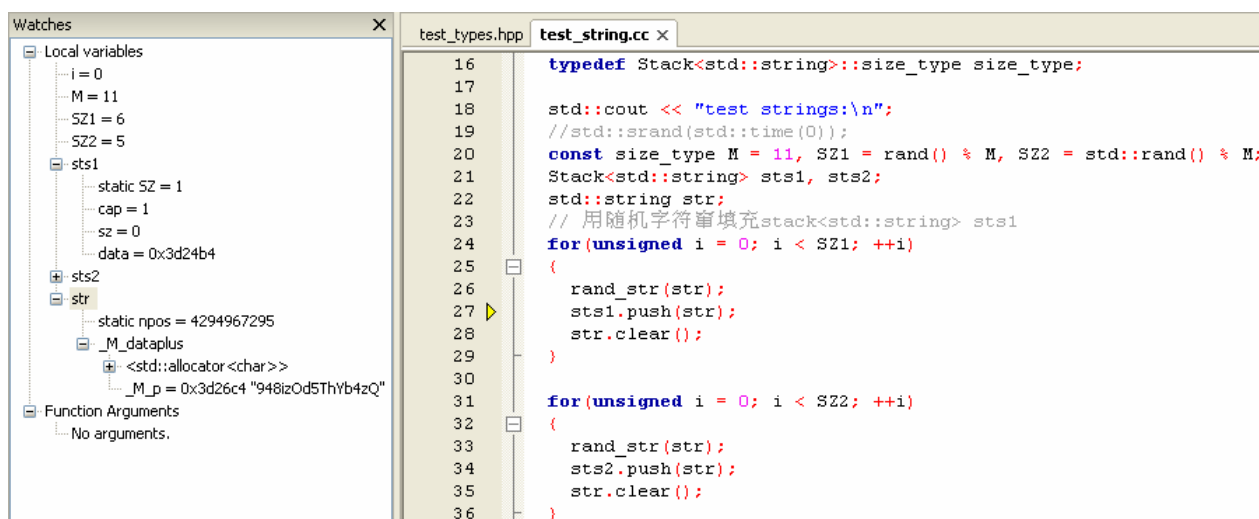



首先，注释掉test_types.hpp中跟test_string();无关的代码，因为其它几个数据类型测试已经通过，不必再跟踪。设置断点，然后启动调试器，见下面贴图。

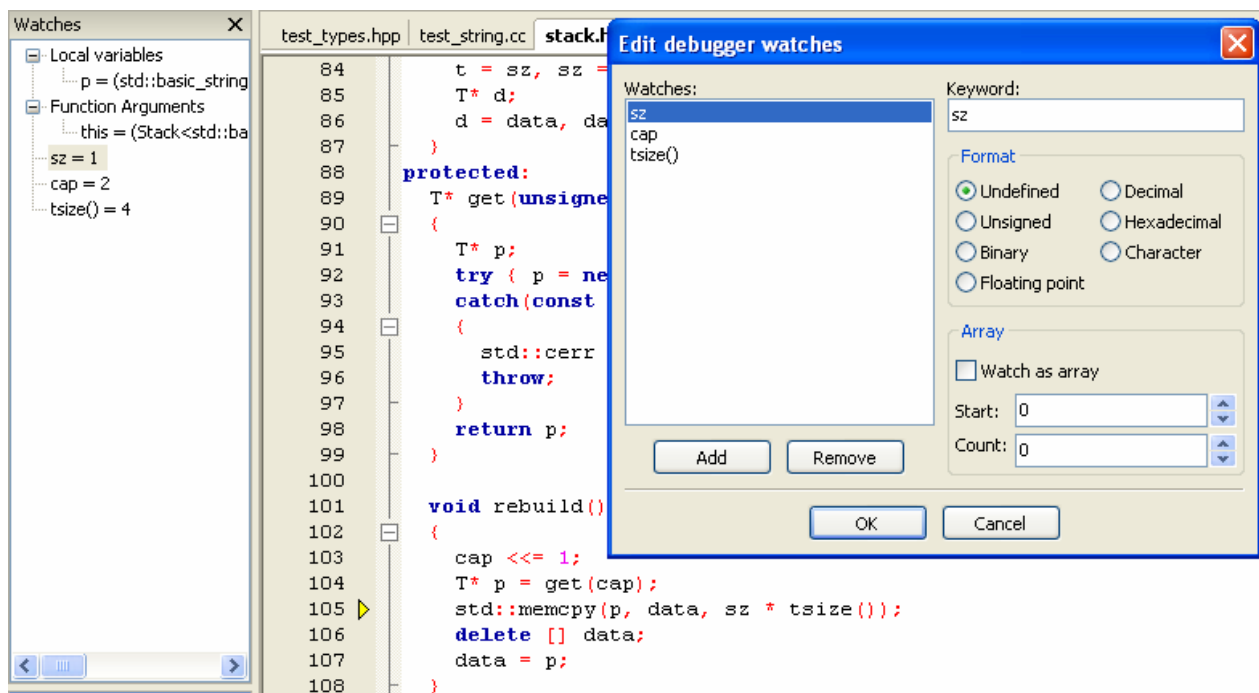
```
1 // test_types.hpp
2 #include <cstdlib>
3 #include <ctime>
4
5 // #include "tests/test_int.cc"
6 // #include "tests/test_double.cc"
7 // #include "tests/test_char.cc"
8 // #include "tests/test_bool.cc"
9 #include "tests/test_string.cc"
10
11 static void init_rand() { std::srand(std::time(0)); }
12 void test_types()
13 {
14     init_rand();
15
16     // test_int();
17     // test_double();
18     // test_char();
19     // test_bool();
20     test_string();
21 }
22
```

打开watches窗口，接下来专门针对string类型的Stack建立过程一路跟踪下去(注意巧妙的用好调试程

序的快捷菜单上的几个按钮), 已经跟踪到了test_string.cc第27行代码前, 仍然没有问题, 见下图。

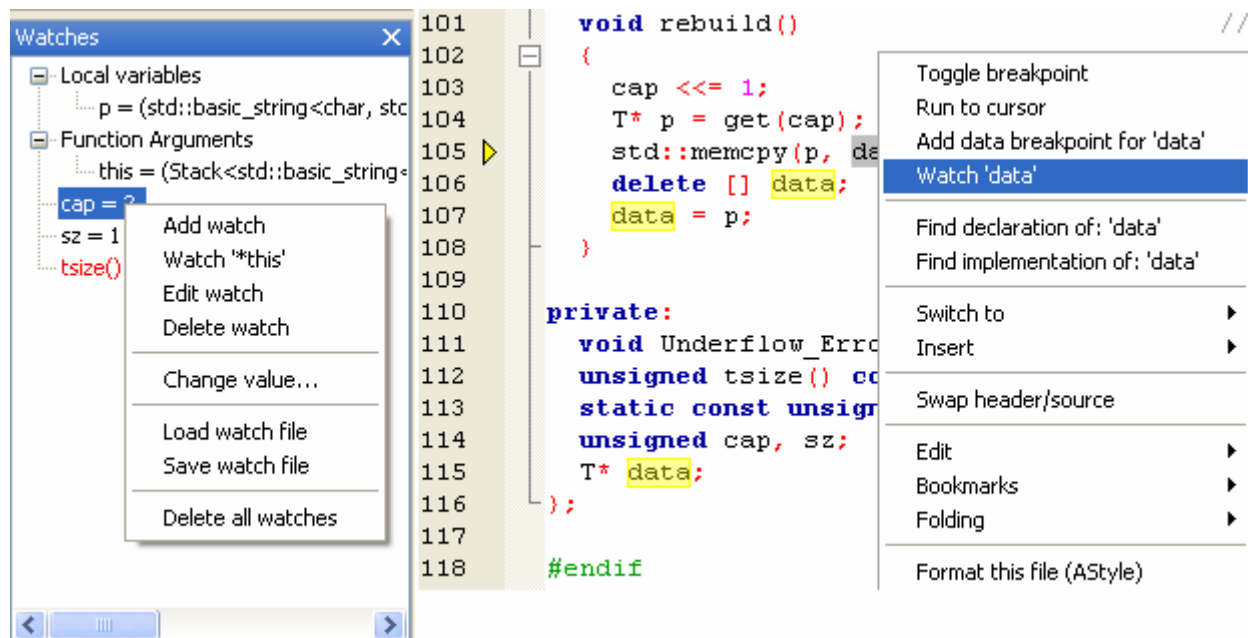


继续Step into, 进入栈的push成员函数体内, 压入栈第一个string对象没有发生任何问题, 继续跟踪把第二个string对象压入栈的过程, 此时栈空间不足, 需要重新分配空间, 然后把原来栈中的string对象复制到新开辟的空间再销毁原来的空间。这个过程中开辟空间没有问题, 因为new足够智能, 它知道该开辟多大的空间(因为new能调用构造函数创建一个string数组, T被初始化为new以后, 再调用就成了p = new std::string[n]), push对象到栈中, 这个复制过程也不会产生问题(因为std::string已经重载了=, 可以正常复制std::string对象), 但是由于栈空间不足, 调用rebuild函数时, 我们从Debug下拉菜单选择Edit watches...增加几个变量, 监测它们的变化, 见下面的贴图。



编辑Watches窗口变量的方法很多, 也可以把鼠标指示移动到Watches窗口内, 按下鼠标右键会弹出

一个快捷菜单，有几个功能供我们选择。Add watch用于手工添加变量到Watches窗口内，Edit watch用来编辑Watches窗口内当前选中的变量，Delete watch则用来把选中的变量从Watches窗口内删除，见左下图。



如果想添加一个变量到Watches窗口，还有更直接的方法。例如，想添加data到Watches窗口，可以用鼠标选中data(选中后显示灰色)，按下鼠标右键，在弹出的快捷菜单中选择'Watch 'data''，见右上图则data会自动添加到Watches窗口内。

tsize()成员函数值居然为4，有点不可思议，难道产生的每一个string对象不会超过4个字符的长度？赶紧看看产生的字符串str的值(把data添加到Watches窗口就可以了，data指向的字符串就是当前str的值)，是"948izOd5ThYb4zQ"，显然它远远超过4个字符了。接下来的std::memcpy(p, data, sz * tsize());就仅仅复制sz * tsize()个字符到新空间中，此时的sz为1，也就是说仅仅复制1 * 4个字符到新空间中，tsize()的实现很容易找到(选中tsize，按下鼠标右键，从弹出的快捷菜单中选择'Find implementation of: 'tsize''，则自动追踪到Sack内成员函数tsize()的实现处)，它是**unsigned tsize() const { return sizeof(T); }**，也就是说sizeof(std::string)的值为4，你敢相信这个结果吗？继续跟踪下去，当企图再往栈中放入第三个std::string对象时，由于空间不足，再次分配更多空间，此时栈数据成员cap的值已经为4，sz已经为2，又进入rebuild函数体，运行语句std::memcpy(p, data, sz * tsize());时程序崩溃了。以上调试过程中如果同时打开栈状态窗口和汇编窗口可能更容易看出问题出在什么地方，当然啦，那需要您能看懂gcc汇编。

或许你已经明白是sizeof惹的祸，其实还不止sizeof，memcpy和memset也功不可没，都有份。如果想改进这个栈的实现就需要替换这两个函数，STL的algorithm函数头下有个copy，是个泛型算法，知道怎么复制简单类型(例如int, char, bool, double等)和复杂类型(例如本例的std::string或者自定义类型)，当然，此刻的sizeof已经不必用了。这里不再介绍怎么改进本例中的栈实现，而是把它留给读者作为练习。

3.5.4 混合风格的程序

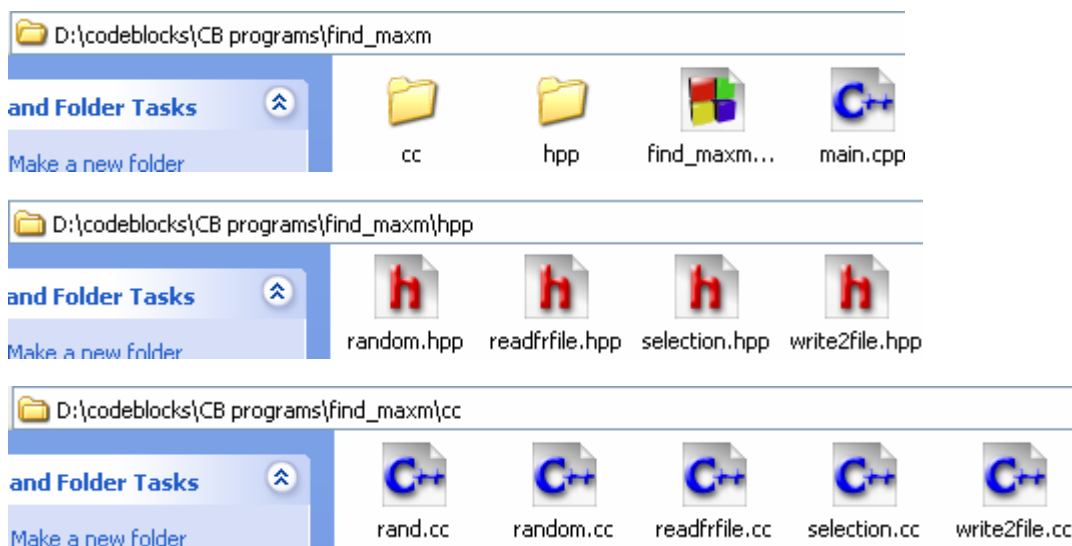
其实很多时候，我们写的程序，尤其复杂的C++程序，并非一种编程风格，而是以上几种编程风格

的混用，而且我们编程经常使用库函数和类。下面给出一个简单例子。

网络上数据非常丰富，有时我们需要从这些海量数据中筛选出我们感兴趣的一些信息。我们在此把这个复杂的问题简单化，假设我们有10G个正整数存放在一个磁盘文件里面，我们需要从这些数中找出1百万个(假设仅仅512MB内存，但是硬盘容量充足)最大的。

为了模拟以上过程，并节约时间，我们处理数据量小一点。首先用随机数产生器产生1千万个正整数，把这些随机数写到硬盘上，放在一个文件里，要求从这些数中找出1K个最大的，在32位系统上使用额外4MB缓存运行速度就基本可以接受。可以这样做：每次读取1M个数放入内存，建造一个堆，筛选出1K个最大的，这样需要读取10次，然后建堆筛选10次，共产生10K个较大的数，再从这10K个中选出1K个最大的。为了加快存取速度，设置读写缓冲区4MB(sizeof(size_t) * 1M)。

这里并不打算再讲述一步步怎么调试程序，因为调试以下程序并不难，但没有给出测试代码，希望读者动手自己补上并调试运行使之得到正确结果。下面是源代码文件层次图和源代码的贴图。




```

1  #include <iostream>
2  #include <fstream>
3  #include <vector>
4  #include <string>
5  #include <ctime>
6  #include "hpp/write2file.hpp"
7  #include "hpp/selection.hpp"
8
9  int main()
10 {
11     std::string tofile;
12     std::cout << "input a file name to hold random numbers: ";
13     std::getline(std::cin, tofile);
14     std::ofstream fout(tofile.c_str());
15     const size_t SZ = 10000000; // 产生sz个随机数
16     const size_t M = 1 << 10; // 找出M个最大的
17     const size_t BSZ = 1 << 20; // 缓存大小为BSZ个元素空间
18     std::clock_t time = std::clock();
19     write2file(fout, SZ, BSZ); // 把sz个随机数写到磁盘上
20     std::cout << "generating " << SZ << " numbers and writing them into "
21               << tofile << " used: " << std::clock() - time << "ms\n";
22
23     std::clock_t t = std::clock();
24     std::ifstream fin(tofile.c_str());
25     std::vector<size_t> v;
26     v.reserve(SZ / BSZ * (M + 1)); // 设置缓冲区大小
27     maxinbuff(fin, M, BSZ, v); // 找到的最大M个随机数暂存缓冲区
28     std::cout << "finding " << M << " maximum numbers in " << tofile
29               << " used: " << std::clock() - t << "ms\n";
30
31     std::cout << "input a file name to hold maximum numbers: ";
32     std::getline(std::cin, tofile);
33     fout.open(tofile.c_str());
34     buff2file(fout, v, M); // 最大的M个随机数写到磁盘上
35 }
36
37 --
38 #ifndef WRITE2FILE_HPP_
39 #define WRITE2FILE_HPP_
40
41 #include <fstream>
42 #include <vector>
43 #include "../cc/write2file.cc"
44
45 void buff2file(std::ofstream&, const std::vector<size_t>&, size_t);
46 void write2file(std::ofstream&, size_t, size_t);
47
48 #endif
49
50 // selection.hpp
51 #ifndef COLLECTION_HPP_
52 #define COLLECTION_HPP_
53
54 #include <fstream>
55 #include <vector>
56 #include "../cc/selection.cc"
57
58 void maxinbuff(std::ifstream&, size_t, size_t, std::vector<size_t>&);
59
60 #endif

```

```

1 // readfrfile.hpp
2 #ifndef READFRFILE_HPP_
3 #define READFRFILE_HPP_
4
5 #include <fstream>
6 #include <vector>
7 #include "../cc/readfrfile.cc"
8
9 void readfrfile(std::ifstream&, std::vector<size_t>&);
10
11 #endif
12
13 // random.hpp
14 #ifndef RANDOM_HPP_
15 #define RANDOM_HPP_
16
17 #include <vector>
18 #include "../cc/random.cc"
19
20 // 产生随机数并写入缓冲区中
21 void gen_rand(std::vector<size_t>&, size_t);
22
23 #endif
24
25 // rand.cc
26 #include <ctime>
27
28 class random
29 {
30 public:
31     explicit random(size_t s = 0) : seed(s)
32     {
33         if (0 == seed) seed = std::time(0);
34         randomize();
35     }
36     void reset(size_t s)
37     {
38         seed = s;
39         if (0 == seed) seed = std::time(0);
40         randomize();
41     }
42     size_t rand32() //returns a random integer in the range [0, -1UL)
43     {
44         randomize();
45         return seed;
46     }
47     double real() //returns a random real number in the range [0.0, 1.0)
48     {
49         randomize();
50         return (double)(seed) / -1UL;
51     }
52 private:
53     size_t seed;
54     void randomize() { seed = 1103515245UL * seed + 12345UL; }
55 };
56
57
58
59
60
61
62

```

```

33 class rand_help
34 {
35     static random r;
36 public:
37     rand_help() {}
38     void operator()(size_t s) { r.reset(s); }
39     size_t operator()() { return r.rand32(); }
40     double operator()(double) { return r.real(); }
41 };
42 random rand_help::r;
43
44 extern void srandom(size_t s) { rand_help()(s); }
45 extern size_t rand32() { return rand_help()(); }
46 extern double drand() { return rand_help()(1.0); }
47
1 // random.cc
2 #include <vector>
3 #include "rand.cc"
4
5 // 产生随机数并写入缓冲区中
6 void gen_rand(std::vector<size_t>& v, size_t n)
7 {
8     for(size_t i = 0; i < n; ++i)
9         v.push_back(rand32());
10
11 // readfrfile.cc
12 #include <fstream>
13 #include <string>
14 #include <cstring>
15
16 // 把字符串转化成无符号整数
17 inline size_t str2uint(const std::string& str)
18 {
19     size_t n = 0;
20     for(size_t i = 0; i < str.size(); ++i)
21     {
22         n *= 10;
23         n += (str[i] - '0');
24     }
25     return n;
26 }
27
28 // 把所有由字符串转化成的整数都写入缓冲区
29 void str2buff(const std::string& str, std::vector<size_t>& v)
30 {
31     for(size_t i = 0, k; i < str.size(); i = k + 1)
32     {
33         while(!std::isdigit(str[i])) { ++i; }
34         for(k = i; k < str.size() && std::isdigit(str[k]); ++k) {}
35         std::string tmp(std::string(&str[i], &str[k]));
36         v.push_back(str2uint(tmp));
37     }
38 }
39
40 // 每次从文件中读取一行，并把他们转化成整数写入缓冲区
41 void readfrfile(std::ifstream& fin, std::vector<size_t>& buff)
42 {
43     std::string str;
44     std::getline(fin, str);
45     str2buff(str, buff);
46 }
47

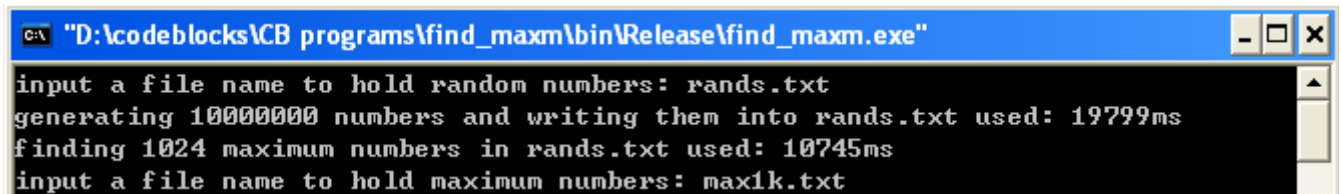
```

```

1 // selection.cc
2 #include <vector>
3 #include <algorithm>
4 #include <functional>
5 #include "readfrfile.cc"
6
7 // 使用二叉堆进行筛选，并把筛选结果暂存到缓冲区中
8 void collect(std::vector<size_t>& v, std::vector<size_t>& b, size_t m)
9 {
10     std::partial_sort(b.begin(), b.begin() + m, b.end(), std::greater<size_t>());
11     v.insert(v.end(), b.begin(), b.begin() + m); // 附加到上次存放结果的后面
12 }
13
14 // 每次从文件读取一行，转化成整数，筛出m个最大的候选值暂存起来，
15 // 待粗筛完毕，然后进行二次筛选，找出最终的m个最大值
16 void maxinbuff(std::ifstream& fin, size_t m, size_t bsz, std::vector<size_t>& v)
17 {
18     std::vector<size_t> buff;
19     buff.reserve(bsz);
20     while(!fin.eof())
21     {
22         readfrfile(fin, buff);
23         collect(v, buff, m);
24         buff.clear();
25     }
26     fin.close();
27     std::nth_element(v.begin(), v.begin() + m, v.end(), std::greater<size_t>());
28 }
29
30 // write2file.cc
31 #include <fstream>
32 #include <vector>
33 #include "../hpp/random.hpp"
34
35 // 把缓冲区中的前n个数据写入到文件中，然后写入一个换行符
36 void buff2file(std::ofstream& fout, const std::vector<size_t>& v, size_t n)
37 {
38     if(v.empty())
39         return;
40     for(size_t i = 0; i < n; ++i)
41         fout << v[i] << ' ';
42     fout << '\n';
43 }
44
45 // 把n个数据分批写入到一个文件中，每批写入bsz个
46 void write2file(std::ofstream& fout, size_t n, size_t bsz)
47 {
48     std::vector<size_t> v;
49     v.reserve(bsz);
50     for(size_t c = 0, T = n / bsz; c < T; ++c)
51     {
52         gen_rand(v, bsz);
53         buff2file(fout, v, bsz);
54     }
55     const size_t L = n % bsz;
56     if(L != 0)
57     {
58         gen_rand(v, L);
59         buff2file(fout, v, L);
60     }
61     fout.close();
62 }
63

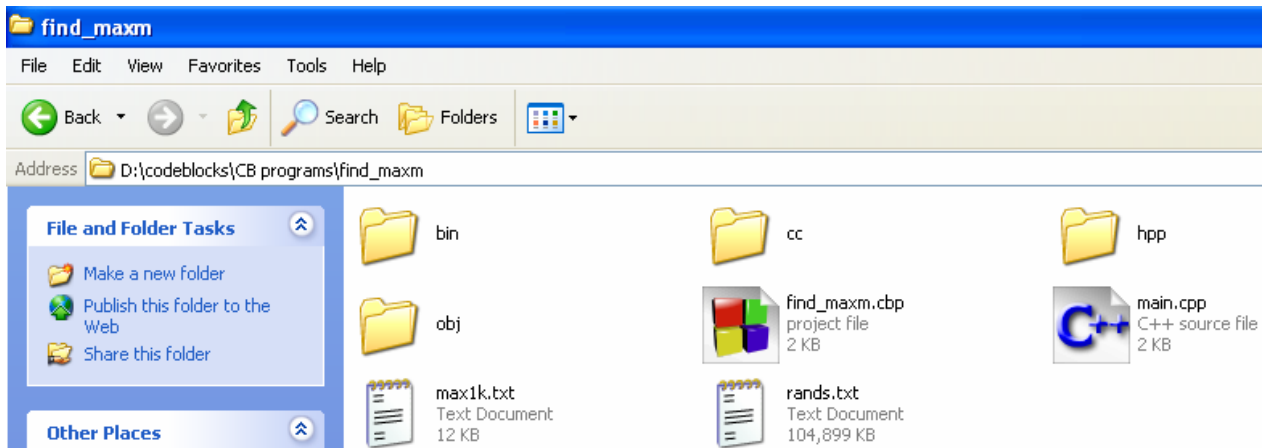
```

编译并运行以上程序，一种可能的结果如下截图：



```
"D:\codeblocks\CB programs\find_maxm\bin\Release\find_maxm.exe"
input a file name to hold random numbers: rands.txt
generating 10000000 numbers and writing them into rands.txt used: 19799ms
finding 1024 maximum numbers in rands.txt used: 10745ms
input a file name to hold maximum numbers: max1k.txt
```

假设生成的随机数存放在文件 rands.txt 中，筛选出的最大 1K 个随机数存放在文件 max1k.txt 中，程序运行完毕，打开所在目录可以看到如下：

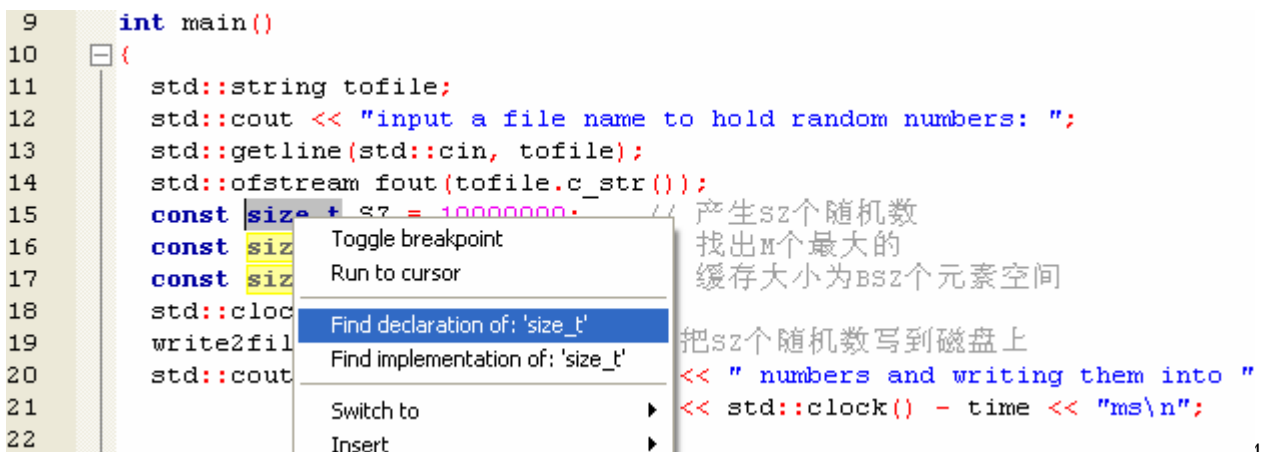


说明可以产生随机数(rands.txt 大小为 104899KB)，也能筛选出一些结果(max1k.txt 文件大小为 12KB)。至于结果是否正确，请读者编程测试验证，作为练习。

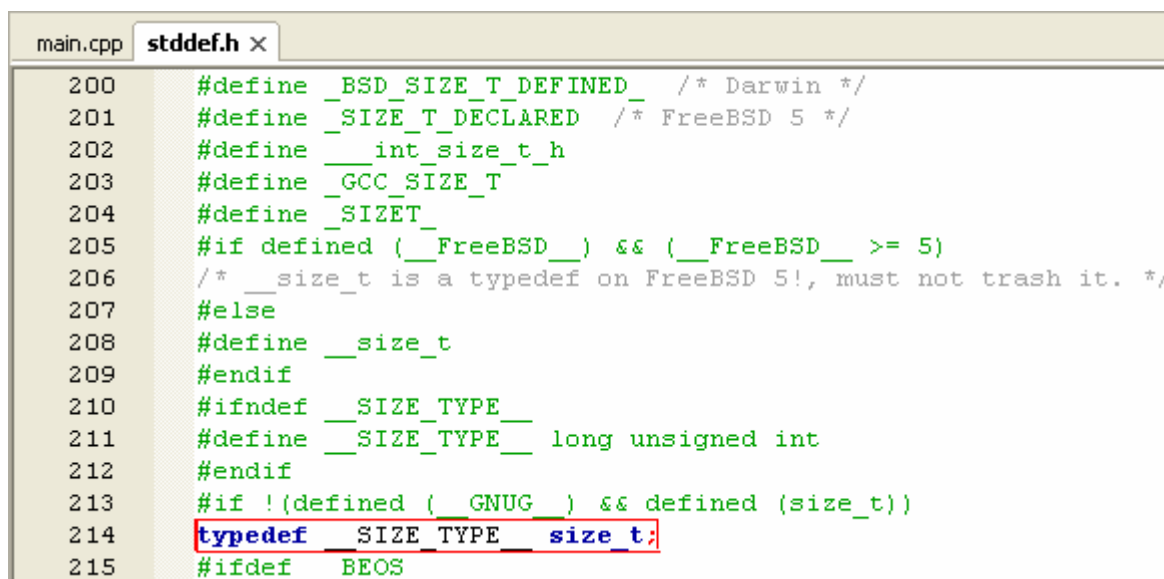
以上运行结果是在笔者的古董笔记本电脑(IBM T23)上运行测得，产生一千万个随机数并分批写入磁盘文件消耗了将近 20 秒，分批读取这些数据找出 1K 个最大的大约耗费了 10 秒钟，其实时间主要耗费在读写磁盘文件上，假如不读写磁盘的话，产生 1 千万的随机数在笔者的古董电脑上也不会超过 1 秒，从这些随机数中找出 1K 个最大的需要的时间不会超过 5 毫秒。

3.6 阅读别人编写的程序

有时为了提升自己的水平，我们往往需要阅读别人的程序消化吸收为自己所用。这里仍旧以上面的 3.5.4 小结中的程序作为示例。首先，看几个伪关键字(并非真正的关键字，而是宏定义或者 typedef 定义的别名)。main 函数里面用到了 size_t 和 clock_t，我们看一看它们究竟是什么原型。选中一个 size_t，按下鼠标右键，从弹出的快捷菜单中选择 **Find declaration of: 'size_t'**，见下图。



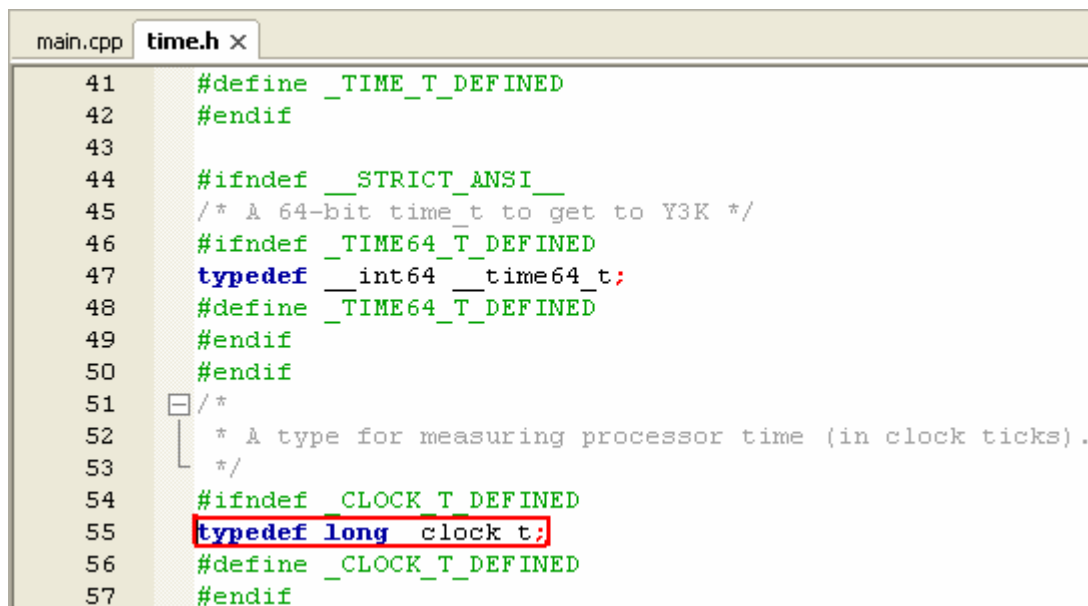
这样,我们可以找到 `size_t` 的声明位置,在 `stddef.h` 头文件里面,是一个 `typedef` 来的 `__SIZE_TYPE__`,见下面的贴图第 214 行红色方框中的部分。



```
main.cpp  stddef.h x
200     #define __BSD_SIZE_T_DEFINED_ /* Darwin */
201     #define __SIZE_T_DECLARED /* FreeBSD 5 */
202     #define __int_size_t_h
203     #define __GCC_SIZE_T
204     #define __SIZET__
205     #if defined (__FreeBSD__) && (__FreeBSD__ >= 5)
206     /* __size_t is a typedef on FreeBSD 5!, must not trash it. */
207     #else
208     #define __size_t
209     #endif
210     #ifndef __SIZE_TYPE__
211     #define __SIZE_TYPE__ long unsigned int
212     #endif
213     #if !(defined (__GNU__ ) && defined (size_t))
214     typedef __SIZE_TYPE__ size_t;
215     #ifdef __BEOS__
```

那 `__SIZE_TYPE__` 又是什么呢,按照同样的方法追踪下去..., 仍然在文件 `stddef.h` 里面,就在第 211 行,是一个 `long unsigned int` 类型,用宏定义的,仍然见上图。

那 `clock_t` 又是什么呢? 按照同样的方法去追踪,我们发现它在 `time.h` 文件里面,是 `long` 的 `typedef`,在第 55 行,见下面的贴图红色方框标出的部分。



```
main.cpp  time.h x
41     #define __TIME_T_DEFINED
42     #endif
43
44     #ifndef __STRICT_ANSI__
45     /* A 64-bit time_t to get to Y3K */
46     #ifndef __TIME64_T_DEFINED
47     typedef __int64 __time64_t;
48     #define __TIME64_T_DEFINED
49     #endif
50     #endif
51     /*
52     * A type for measuring processor time (in clock ticks).
53     */
54     #ifndef __CLOCK_T_DEFINED
55     typedef long clock_t;
56     #define __CLOCK_T_DEFINED
57     #endif
```

如果您不明白任何函数名或者定义的变量名均可以采用类似的方法去追踪。假如您想看看某个函数的实现,可以采用类似的方法。例如您想知道 `write2file` 的实现,就选中这个名称,按下鼠标右键,从弹出的快捷菜单中选择 `Find implementation of: 'write2file'`,见下面的贴图。

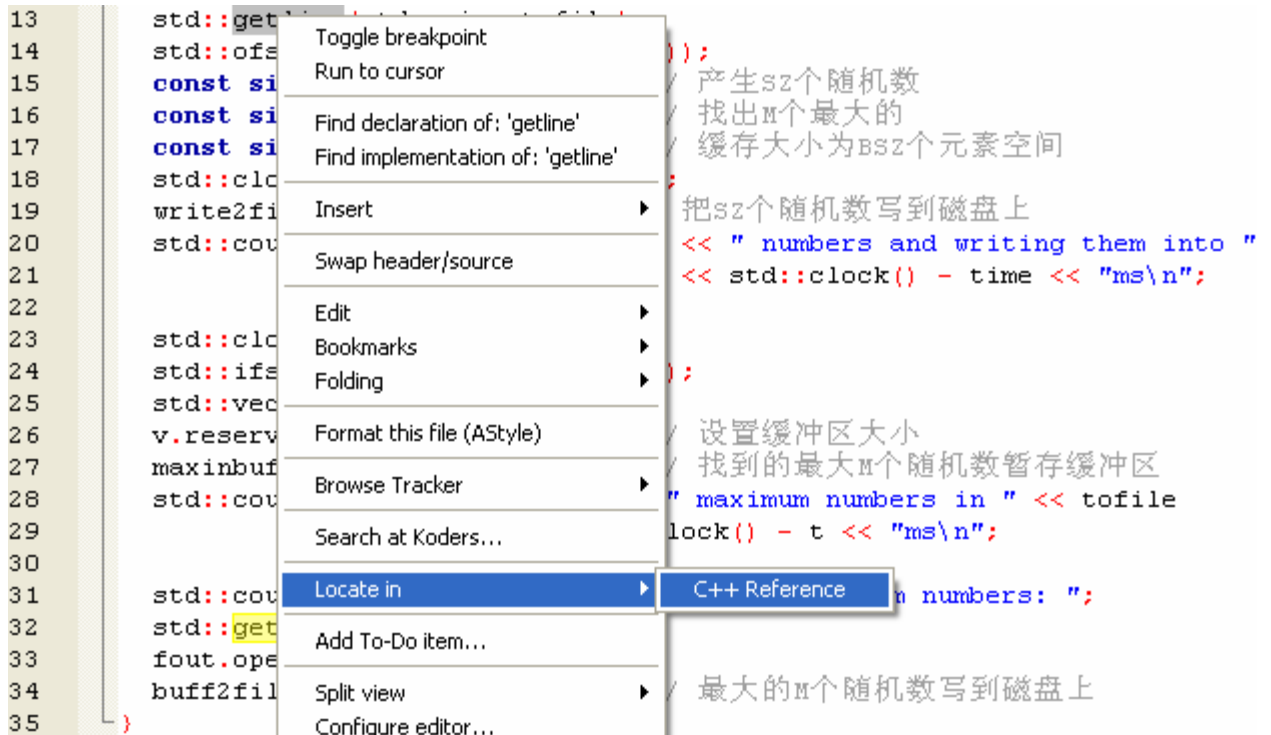
```
main.cpp x
7  #include "hpp/selection.hpp"
8
9  int main()
10 {
11     std::string tofile;
12     std::cout << "input a file name to hold random numbers: ";
13     std::getline(std::cin, tofile);
14     std::ofstream fout(tofile.c_str());
15     const size_t SZ = 10000000; // 产生sz个随机数
16     const size_t M = 1 << 10; // 找出M个最大的
17     const size_t BSZ = 1 << 20; // 缓存大小为BSZ个元素空间
18     std::clock_t time = std::clock();
19     write2file(fout, SZ, BSZ); // 把sz个随机数写到磁盘上
20     std::cout << " numbers and writing them into "
21     << std::clock() - time << "ms\n";
22
23     std::clo
24     std::ifs
```

此时会自动找到这个函数的实现，它在文件 write2file.cc 文件里面，光标停留在该函数头前面，见下面的贴图中红色方框标出的部分。如果该实现中用到了其它的函数您不清楚的地方，可以采用此方法继续追踪下去...

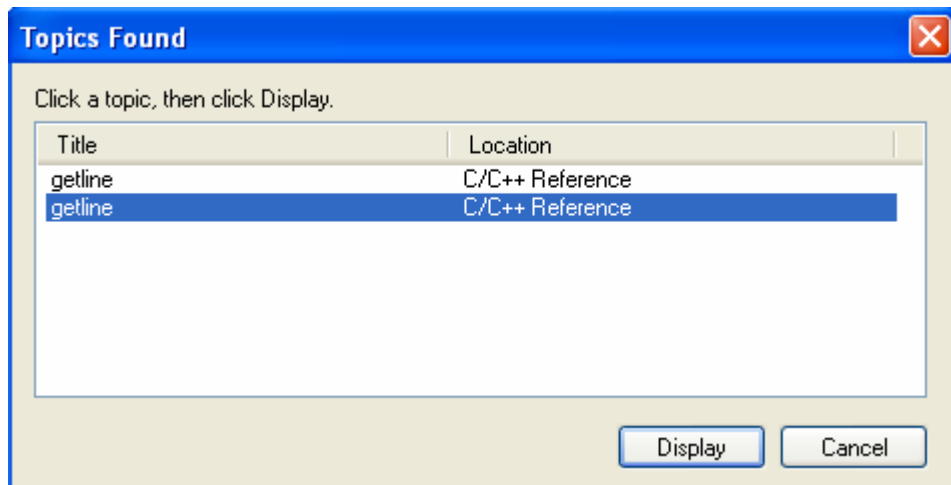
```
16 // 把n个数据分批写入到一个文件中，每批写入bsz个
17 void write2file(std::ofstream& fout, size_t n, size_t bsz)
18 {
19     std::vector<size_t> v;
20     v.reserve(bsz);
21     for(size_t c = 0, T = n / bsz; c < T; ++c)
22     {
23         gen_rand(v, bsz);
24         buff2file(fout, v, bsz);
25     }
26     const size_t L = n % bsz;
27     if(0 != L)
28     {
29         gen_rand(v, L);
30         buff2file(fout, v, L);
31     }
32     fout.close();
33 }
34
```

以上是我们自己定义的函数，如果要是调用的系统的函数怎么办？采用同样的方法固然可以，但是重名函数很多，实现也各不相同，有些函数实现十分复杂，我们很难短时间内看明白。为此我们可以在文档中查询该函数的用法就可以了，可能您不会忘记我们前面有关章节介绍按下 F1 快捷键可以在我们已经事先设置好的 cppreference.chm 里面查询和网上查询。那是一种方法，还有更快捷的方法，我们无需键入函数名就可以自动查找。

例如，假如我们想查找函数 `getline` 的相关使用方法，选中 `getline` 这个名称，按下鼠标右键，在弹出的快捷菜单中选择 **Locate in** **C++ Reference**，这样仍然可以在 `cppreference.chm` 文件中查找，见下图。



当选择 **C++ Reference** 的时候，会弹出一个窗口，在帮助文件中找到两个函数名为 `getline` 的函数，第二个是我们需要的，选中第二个，并点击 **Display** 按钮，见下面的贴图。



点击 **Display** 后，会自动搜索 `cppreference.chm` 文件中有关 `getline` 函数的相关用法介绍，并切换到该函数的用法界面部分，见下面的贴图。

C/C++ Reference

Hide Back Forward Home Options

Contents Index Search

Type in the keyword to find:

getline

getline

Syntax:

```
#include <string>
istream& getline( istream& is, string& s, char delimiter = '\n' );
```

The C++ string class defines the global function `getline()` to read strings from and I/O stream. The `getline()` function, which is not part of the string class, reads a line from *is* and stores it into *s*. If a character *delimiter* is specified, then `getline()` will use *delimiter* to decide when to stop reading data.

For example, the following code reads a line of text from **STDIN** and displays it to **STDOUT**:

```
string s;
getline( cin, s );
cout << "You entered " << s << endl;
```

After getting a line of data in a string, you may find that [string streams](#) are useful in extracting data from that string. For example, the following code reads numbers from standard input, ignoring any "commented" lines that begin with double slashes:

```
// expects either space-delimited numbers or lines that start with
// two forward slashes (//)
string s;
while( getline(cin,s) ) {
    if( s.size() >= 2 && s[0] == '/' && s[1] == '/' ) {
        cout << " ignoring comment: " << s << endl;
    } else {
        istringstream ss(s);
        double d;
        while( ss >> d ) {
            cout << " got a number: " << d << endl;
        }
    }
}
```

通过上述介绍的几个简单菜单按钮，我们就可以比较方便的阅读别人编写的源程序了。

4. 附录

附录有几个部分，分别讲述在不同的操作系统上安装 Code::Blocks，配置其他库等。

4.1 在 Linux 下安装 Code::Blocks

Linux 版本众多，不同版本有些差异，这里以在 ubuntu 和 fedora 上安装 Code::Blocks 为例。

4.1.1 Ubuntu

- (1) 在 `/etc/apt/sources.list` 下添加如下内容。

以 **root** 身份登陆，打开一个图形界面的编辑器，在终端上粘贴下面一行内容。

```
gksu gedit /etc/apt/sources.list
```

在文件末尾贴上以下内容。

```
# codeblocks
```

```
deb http://lgp203.free.fr/ubuntu/ gutsy universe
```

```
# wxwidgets
```

```
deb http://apt.wxwidgets.org/ gutsy-wx main
```

注意：在第(1)步您也可能需要使用 **feisty** 而非 **gutsy**，这跟你使用的 **ubuntu** 版本有关。

- (2) 确保您的 **package system** 信任这些源就需要添加他们的 **key**。

在终端键入以下内容。

```
wget -q http://lgp203.free.fr/public.key -O- | sudo apt-key add -
```

```
wget -q http://apt.wxwidgets.org/key.asc -O- | sudo apt-key add -
```

在终端键入以下内容更新这些包。

```
sudo apt-get update
```

```
sudo apt-get upgrade
```

- (3) 安装 Code::Blocks

在终端键入下行内容。

```
sudo apt-get install libcodeblocks0 codeblocks libwxsmithlib0 codeblocks-contrib
```

想得到最新的 **nightly build** 就重复一次第(3)步。

现在您应该可以在程序语言列表中见到 Code::Blocks 了。

4.1.2 Fedora

- (1) 以 **root** 身份登陆，在控制台窗口执行下面的命令。

```
yum install codeblocks
```

- (2) 运行 Code::Blocks

在应用程序的下拉菜单中找到程序设计选择 Code::Blocks IDE。

或者打开一个控制台窗口执行下面的命令。

```
codeblocks
```

4.2 在 Mac OS X 下安装 Code::Blocks

4.2.1 准备工作

(1) 确认X11已经安装

看看X11是否在您的系统中已经安装了(在应用程序的菜单里),如果没有安装就拿出 Mac OS的盘装上吧。安装方法很简单,选中备选安装包的installer,打开X11前面打勾安装上。如果您使用的是10.1至10.3的版本,可以从Apple网上下载X11。

(2) 安装开发工具 (Mac OS X 10.4以上版本不需要这一步)

打开xterm,试试你在终端上不能使用Code::Blocks。xterm是X11的一部分,这也就是为何要安装X11的原因。

运行gcc命令,如果出错信息提示没有输入文件,说明gcc可用,如果出错信息说找不到gcc,那您需要安装开发工具。

如果需要安装开发工具的话,您可以从Apple网站下载或从Mac OS盘上安装,找到Xcode installer装上就是了,运行这个installer,除了文档可以不用安装外,其他全部选中安装上。

Mac OS X 10.3,需要安装1.2以上版本的Xcode工具, Mac OS X 10.4需要安装2.2以上版本的Xcode工具, Mac OS X 10.5需要安装3.1以上版本的Xcode工具。

4.2.2 安装Code::Blocks


解压zip安装包把CodeBlocks.app放到您想放的位置(建议放在/Developer/Applications或者~/Applications下)。

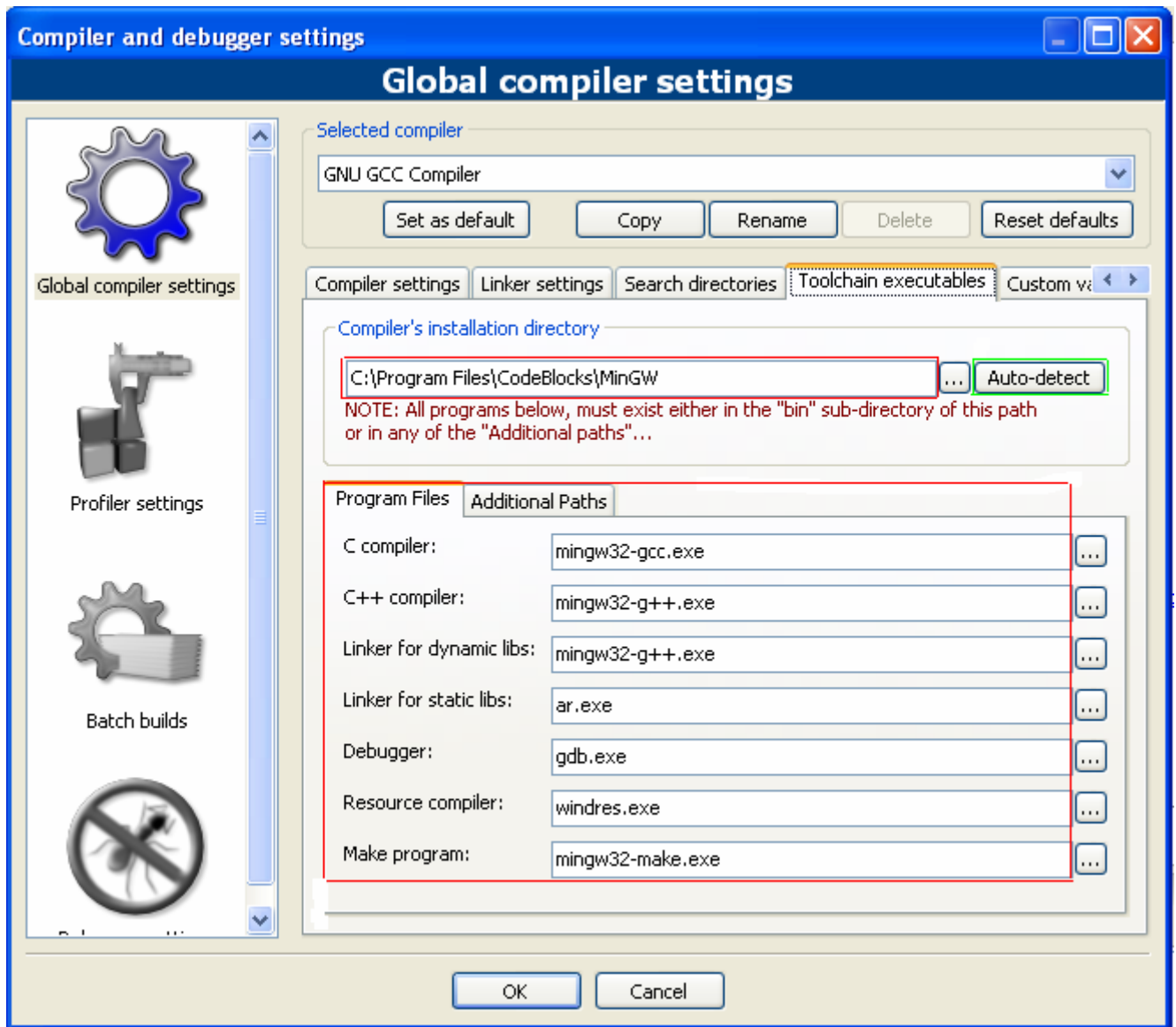
4.3 Code::Blocks 搭配高版本 gcc 编译器

Code::Blocks 搭配较高版本的 gcc 编译器有两种方法,如果你用 Windows 系统,可以升级和配置高版本的 MinGW(内嵌 gcc)从而使用高版本 gcc。在所有系统上都可以升级和搭配高版本的 gcc 编译器。

4.3.1 升级 MinGW


目前最新稳定的 MinGW 版本内嵌了 gcc3.4.5 编译器,测试版则内嵌了 gcc4.3.3,升级 MinGW 前需要先卸载原来古老版本的 MinGW 或者把新版本的 MinGW 安装到不同的位置。

笔者建议您首先卸载古老版本的 MinGW,然后到 <http://www.tdragon.net/recentgcc/> 下载最新测试版 MinGW 的安装器(installer)并执行之,把 MinGW 安装到 Code::Blocks 文件夹下,启动 Code::Blocks,如果不能自动检测到新安装的 MinGW 的话,就在 Settings 下拉菜单中选择 Compiler and debugger...按钮,此时会弹出一个界面,选择左边带有标签 Global compiler settings 的图标,然后选择 Toolchain executables,见下图。用鼠标点击 Auto-detect 按钮(绿色框内),如果能识别最好,不能识别择需要点击按钮,手工配置路径(注意红色框内的配置),配置完毕后,点击 OK 按钮退出就可以了。

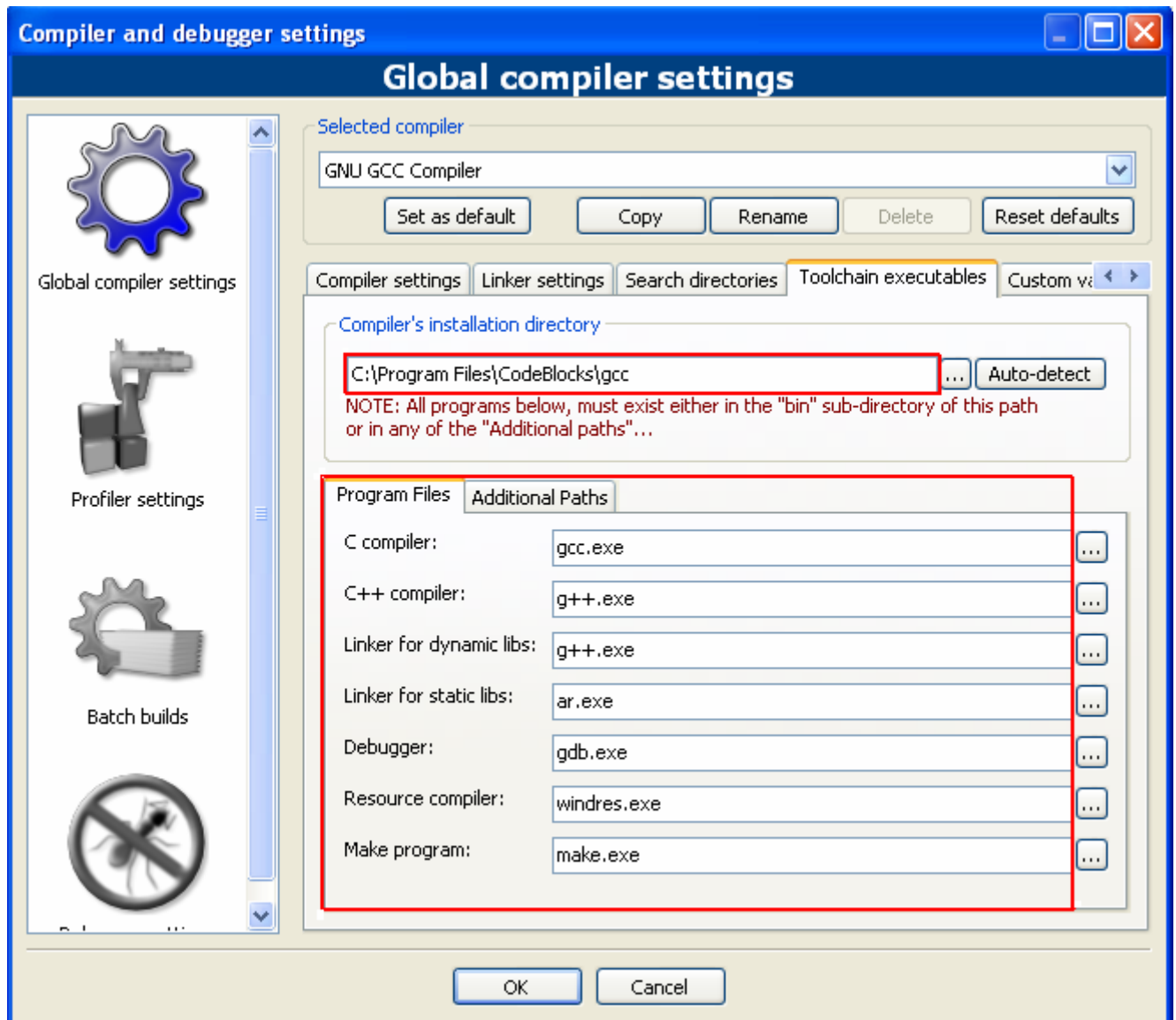


4.3.2 升级并配置 gcc 编译器

如果能找到 gcc 的二进制代码当然最好，如果没有二进制安装包就需要自己编译 gcc，编译 gcc 的方法这里不再赘述，请参阅相关文档。

假设您手头已经有了高版本的 gcc 二进制安装包，Windows 版本的 gcc 二进制文件可以登陆 <http://www.equation.com/servlet/equation.cmd?call=fortran> 下载。下载完毕后，笔者建议您把它安装到 Code::Blocks 的目录下，然后配置编译器选项(跟上面配置 MinGW 的类似)，从 Settings 下拉菜单中选择 Compiler and debugger...按钮，此时会弹出一个界面，选择左边带有标签 Global compiler settings 的图标，然后选择 Toolchain executables，点击按钮  逐一配置好路径，见下图(注意粗线红色框内的文件配置)。

配置完毕后，编译运行一个小程序测试一下，如果编译成功则说明配置正确，失败就按照上面的配置说明好好检查一下，看看哪里有问题改正。



4.4 安装配置 boost

安装 boost 有点麻烦，因为需要下载源代码和编译文件，编译完 boost，然后再配置，配置完毕后还要测试安装是否正确。以下以 Windows XP 上安装 boost 为例进行介绍。

(1) 下载 boost

登陆 <http://www.boost.org> 下载最新的 boost 源代码包和编译文件。写本文的时候最新的 boost 源代码包是 1.38.0 版，最新的编译文件 boost jam 3.1.17，下载完毕后解压。

注意最好下载二进制的 boost jam，这样就可以直接使用它编译 boost 了。在 Windows 上安装 boost 需要下载 boost-jam-3.1.17-1-ntx86.zip，在 linux 上安装 boost 就下载 boost-jam-3.1.17-1-linuxx86.tgz，在 Free BSD Unix 上安装 boost 就下载 boost-jam-3.1.17-1-freebsdxx86.tgz，Mac OS X 上安装 boost 就下载 boost-jam-3.1.17-1-macosxx86.tgz。

(2) 编译 boost

首先需要解压 boost 1.38.0 的包，再解压 boost-jam-3.1.17-1-ntx86.zip，解压完毕 boost-jam-3.1.17-1-ntx86.zip 后，只有一个可执行文件 bjam.exe，把这个文件复制到解压后的 boost 安装包下(在笔者电脑上为 D:\TDDOWNLOAD\boost_1_38_0\boost_1_38_0)。

接下来需要设计 gcc 编译器环境变量，假如你已经成功安装了 GCC，或者 MinGW，全局环境变量就已经自动配置好了，可以进行命令行编译，否则需要设置 GCC 编译器路径。笔者电脑上 gcc.exe 的路径是 C:\Program Files\CodeBlocks\MinGW\bin 把它加到全局变量路径就可以了。按照这样次序去找，My Computer(我的电脑) -> Properties(属性) -> Advanced(高级) -> Environment Variables(环境变量) -> Path -> Edit(编辑) -> [然后把 gcc.exe 文件的路径(C:\Program Files\CodeBlocks\MinGW\bin)复制到环境变量 Path 其它环境变量的后面]保存就可以了。这是一种一劳永逸的方法，如果您并不经常使用命令行编译文件，可以采用即设即用的方式。打开控制台键入 set path=C:\ProgramFiles\CodeBlocks\MinGW\bin 并按下回车键，则 gcc 编译器本次可用命令行编译，关闭控制台即失效。

在控制台上找到 bjam.exe 的路径(已经复制到 D:\TDDOWNLOAD\boost_1_38_0\boost_1_38_0 下面)键入 bjam -toolset=gcc -prefix=C:\boost install 进行编译，此命令会把编译后的 boost 文件安装到 C:\boost 下。在控制台上用 gcc 编译器以命令行形式编译 boost 见下图。

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Chipset>set path=C:\Program Files\CodeBlocks\MinGW\bin

C:\Documents and Settings\Chipset>gcc
gcc: no input files

C:\Documents and Settings\Chipset>d:

D:\>cd D:\TDDOWNLOAD\boost_1_38_0\boost_1_38_0

D:\TDDOWNLOAD\boost_1_38_0\boost_1_38_0>bjam --toolset=gcc --prefix=C:\boost ins
tall_
```

即设即用gcc编译器

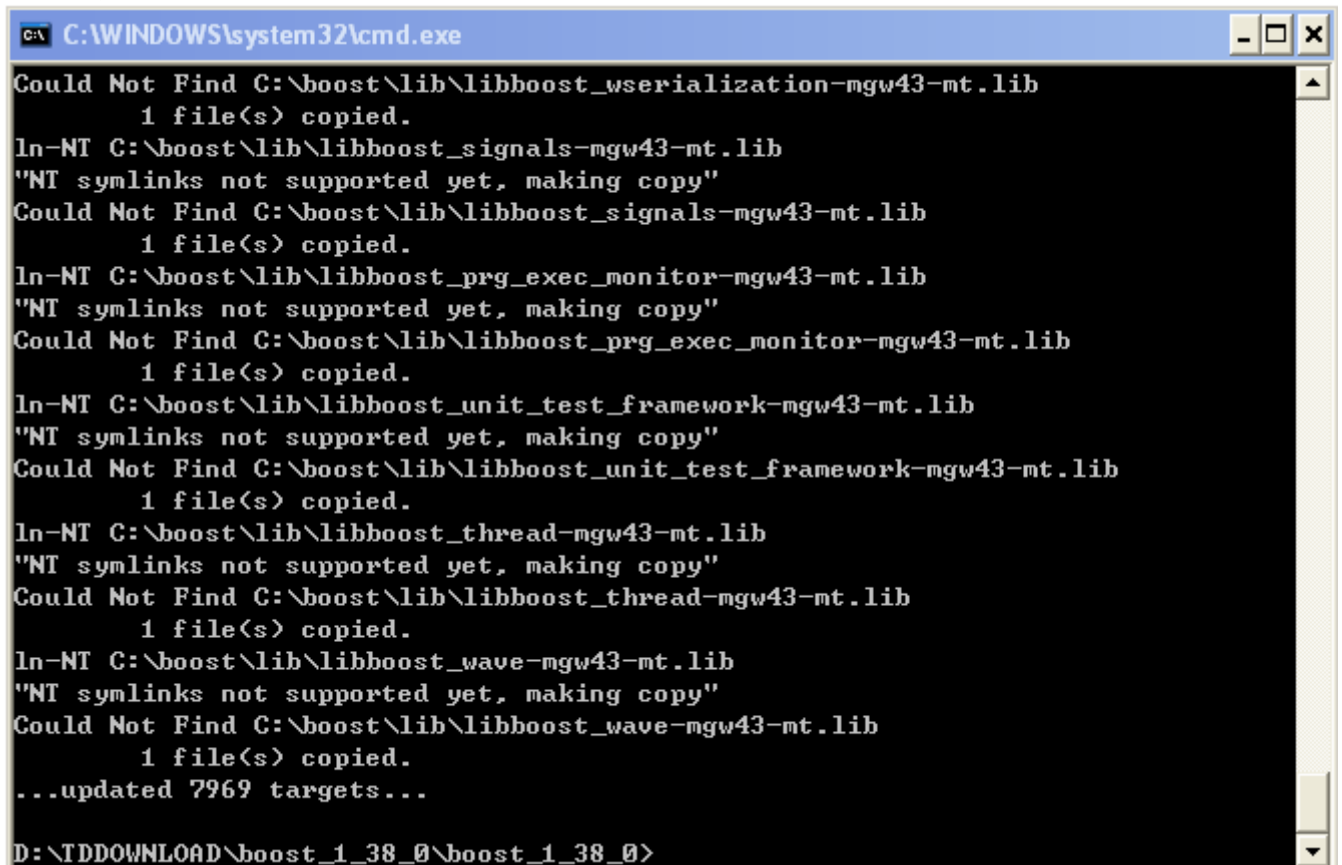
测试gcc编译器是否可用命令行编译

命令行编译boost

需要特别注意：命令行中=前后不可以有空格，文件夹名字随便取，但是一个文件夹名字只能是一个单词，不能是两个或者两个以上中间用空格隔开的单词，否则编译失败。

编译过程将会是比较枯燥等待，好在时间不会太长，编译过程中 CPU、内存、硬盘负荷都很重，在

笔者的古董笔记本电脑(WinXP SP2, PIII 1.13GHz CPU, 512MB 内存 80GB 硬盘)上, 纯编译时间大约用了一个小时。编译过程中往往会提示很多信息(包括一些警告信息), 除非失败, 否则一律不用理会它们, 编译完毕后最终显示结果见下图。



```
C:\WINDOWS\system32\cmd.exe
Could Not Find C:\boost\lib\libboost_wserialization-mgw43-mt.lib
1 file(s) copied.
ln-NT C:\boost\lib\libboost_signals-mgw43-mt.lib
"NT symlinks not supported yet, making copy"
Could Not Find C:\boost\lib\libboost_signals-mgw43-mt.lib
1 file(s) copied.
ln-NT C:\boost\lib\libboost_prg_exec_monitor-mgw43-mt.lib
"NT symlinks not supported yet, making copy"
Could Not Find C:\boost\lib\libboost_prg_exec_monitor-mgw43-mt.lib
1 file(s) copied.
ln-NT C:\boost\lib\libboost_unit_test_framework-mgw43-mt.lib
"NT symlinks not supported yet, making copy"
Could Not Find C:\boost\lib\libboost_unit_test_framework-mgw43-mt.lib
1 file(s) copied.
ln-NT C:\boost\lib\libboost_thread-mgw43-mt.lib
"NT symlinks not supported yet, making copy"
Could Not Find C:\boost\lib\libboost_thread-mgw43-mt.lib
1 file(s) copied.
ln-NT C:\boost\lib\libboost_wave-mgw43-mt.lib
"NT symlinks not supported yet, making copy"
Could Not Find C:\boost\lib\libboost_wave-mgw43-mt.lib
1 file(s) copied.
...updated 7969 targets...
D:\TDOWNLOAD\boost_1_38_0\boost_1_38_0>
```

从上图中的文字可以看出, boost1.38.0 成功安装。一般而言, 编译成功所有的文件并不容易, 根据笔者经验, 用 gcc3.4.5 和 gcc4.3.2 编译 boost1.37.0 从未完全编译成功所有的文件, 这次使用 gcc4.3.3 编译 boost1.38.0 编译成功的文件数见上图中最后的提示信息(...updated 7969 targets...).

(3) 配置 boost

打开 code::blocks, 在 Settings 的下拉菜单中选择 **Compiler and debugger...** 然后在 Search directory 栏目下用 Add 添加 boost 的路径(在笔者电脑上 `C:\boost\include\boost-1_38`)保存退出。此外, 再设置一下全局变量 (在 Setting 下拉菜单下 **Global variables...** 中的几个值, 笔者的设置是 **Current Set:** boost, **Current Variable:** boost, **base** `C:\boost\include\boost-1_38`)。

(4) 测试 boost

我们仅仅需要测试 boost 是否安装配置正确, 而并非测试 boost 是否完整或者有没有 bug, 测试 boost 的完整性和是否有 bug 不是一件很容易的事情, boost 团队中有很多人一直进行着这项伟大的工作。


```

1 //简单测试，仅仅测试是否安装正确，而不是测试boost完整性
2 //以下程序的功能是从终端接受一个整数序列，然后乘以3再输出到终端
3 //test1.cpp
4 #include <iostream>
5 #include <iterator>
6 #include <algorithm>
7 #include <boost/lambda/lambda.hpp>
8
9 int main()
10 {
11     using namespace boost::lambda;
12     typedef std::istream_iterator<int> in;
13     std::for_each(in(std::cin), in(), std::cout << (_1 * 3) << " ");
14 }
15

```

```

1 //以下程序在终端输出[2] (2,8)
2 //test2.cpp
3 #include <boost/numeric/ublas/vector.hpp>
4 #include <boost/numeric/ublas/matrix.hpp>
5 #include <boost/numeric/ublas/io.hpp>
6 #include <iostream>
7
8 using namespace boost::numeric::ublas;
9
10 int main ()
11 {
12     vector<double> x (2);
13     x(0) = 1; x(1) = 2;
14     matrix<double> A(2, 2);
15     A(0, 0) = 0; A(0, 1) = 1;
16     A(1, 0) = 2; A(1, 1) = 3;
17     vector<double> y = prod(A, x);
18     std::cout << y << std::endl;
19     return 0;
20 }
21

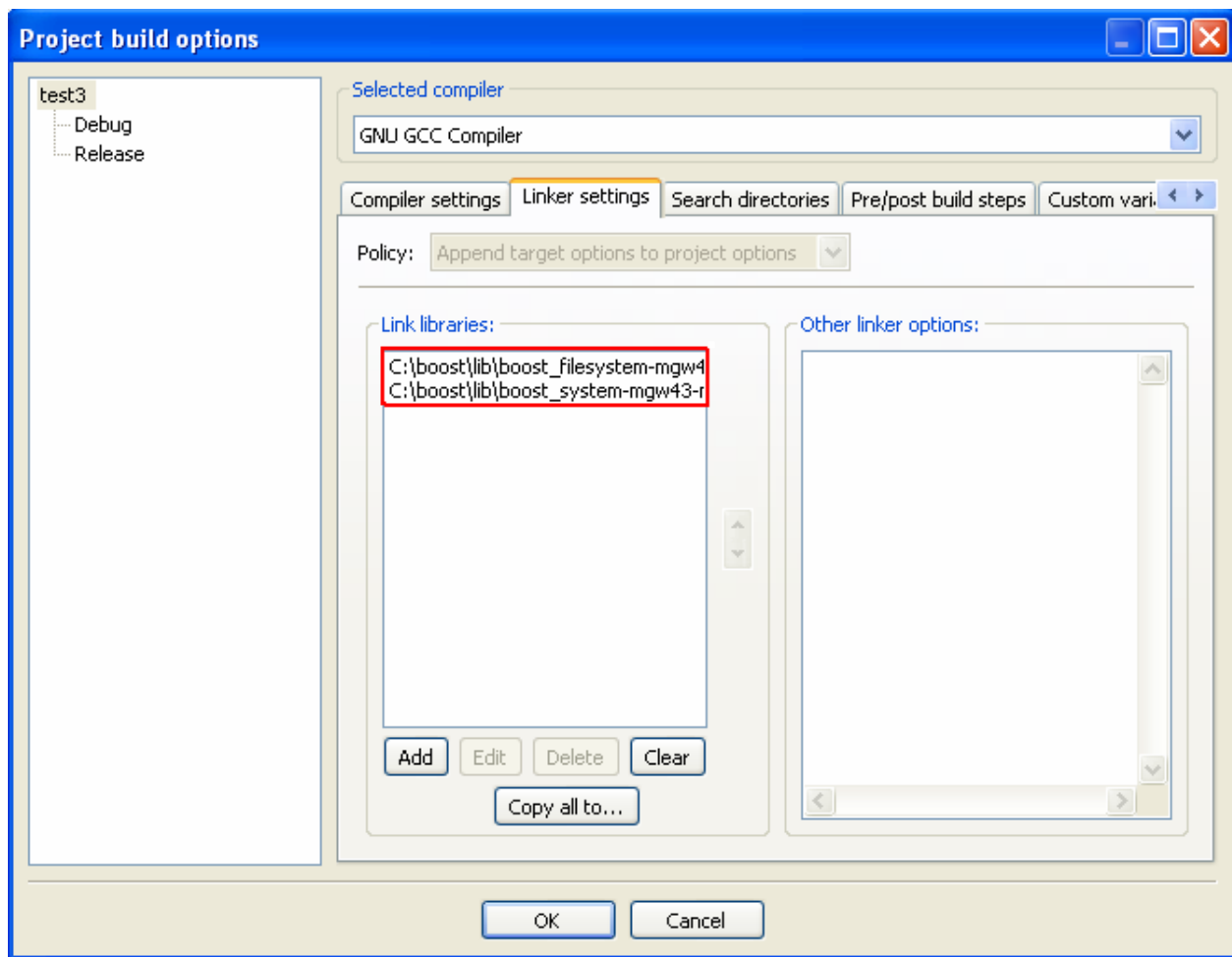
```

```

1 /*****
2 test3.cpp
3 注意编译运行时需要连接两个库文件
4 boost_system-mgw43-mt-1_38.dll和
5 boost_filesystem-mgw43-mt-1_38.dll
6 *****/
7 #include <iostream>
8 #include <boost/filesystem/operations.hpp>
9
10 namespace bfs = boost::filesystem;
11 int main()
12 {
13     bfs::path p("test3.cpp");
14     if(bfs::exists(p))
15         std::cout << p.leaf() << std::endl;
16 }
17

```

编译和运行 test3.cpp 都需要连接两个库文件，编译前需要在本工程中添加上这两个文件的路径，见下图红色框框起来的两个文件(首先在 Project 下拉菜单中选择 Build options...，然后可以参考下图用 Add 按钮添加，见红色框内文字)。



编译成功后，运行时还需要能找到这两个文件。

4.5 安装配置 Qt

C/C++ GUI 库多如牛毛，商用的免费的都很多。其中名气最大的 C/C++ GUI 库可能就是 Qt 了，Qt 非常强大而且版本升级很快，既有商用版也有免费版。

(1) 下载

如果希望在 Code::Blocks 下开发 Qt 的应用，首先需要到 Qt 的官方网站根据您使用的系统下载相应 Qt 的最新版本 <http://www.qtsoftware.com/downloads>，然后进行安装。如果您希望在 Windows 下用 Qt 编写应用程序，一般可以选用两种版本 MinGW 的或者 MSVC 的，这里以 MinGW 的为例进行简单介绍安装配置。

(2) 安装

下载完毕,启动安装文件进行安装,安装过程中可选是否安装 MinGW,因为我们已经安装了 MinGW 因此不需该项(去掉前面的勾), 笔者建议您把 Qt 安装在 CodeBlocks 目录下。Qt 的库相对比较大, 安装时间可能比较长, 请耐心等待。

(3) 设置系统环境变量

安装完 Qt, 为了便于搜索需要加载的文件, 需要设置系统环境变量, 操作系统的环境变量设置要特别小心, 否则很难说会出现什么问题。以笔者电脑安装的 Qt4.5 为例, 拖动鼠标到我的电脑图标上, 按下右键弹出菜单选择属性, 再从弹出菜单中选择高级, 然后选择环境变量, 填入以下这些项目。


QtDir=C:\Program Files\CodeBlocks\Qt\qt

Include=C:\Program Files\CodeBlocks\Qt\qt\include;C:\Program Files\CodeBlocks\MinGW\include;

Lib=C:\Program Files\CodeBlocks\Qt\qt\lib;C:\Program Files\CodeBlocks\MinGW\lib;

Path 中加入.;C:\Program Files\CodeBlocks\Qt\qt\bin;C:\Program Files\CodeBlocks\MinGW\bin;

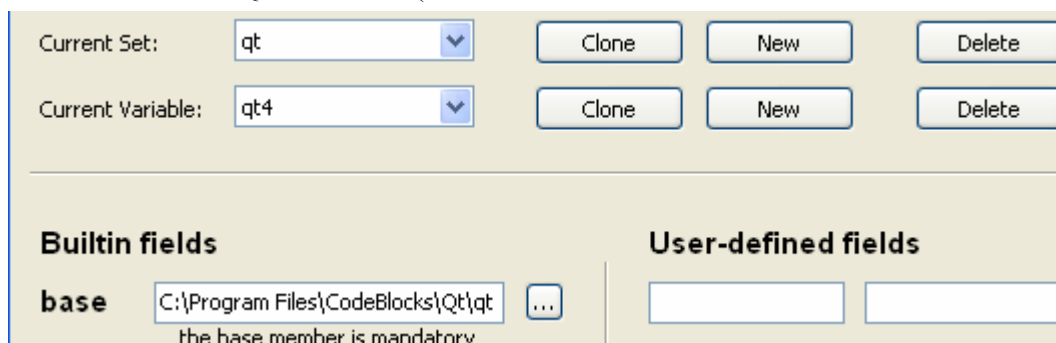
(4) 设置编译器搜索路径

Setting 下拉菜单下选择  然后在 Search directory 栏目下用 Add 添加 Qt 路径(笔者电脑上 C:\Program Files\CodeBlocks\Qt\qt), 见下面截图红色框内文字。



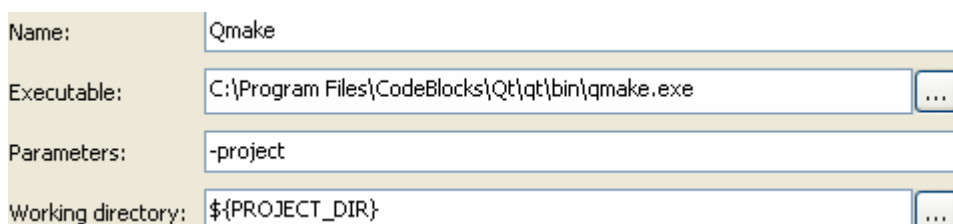
(5) 设置 Code::Blocks 全局变量

Setting 下拉菜单下选择  在弹出的对话框中设置 Qt 的全局环境变量。其中有一项 base 栏目必填, 该项是 Qt 的安装路径(笔者电脑上是  C:\Program Files\CodeBlocks\Qt\qt), 见下图。



(6) 添加 Qmake

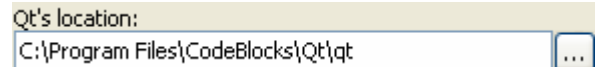
打开 Code::Blocks 的 Tools 下拉菜单, 选择  添加一个 Qmake, 填入选项见下面贴图。



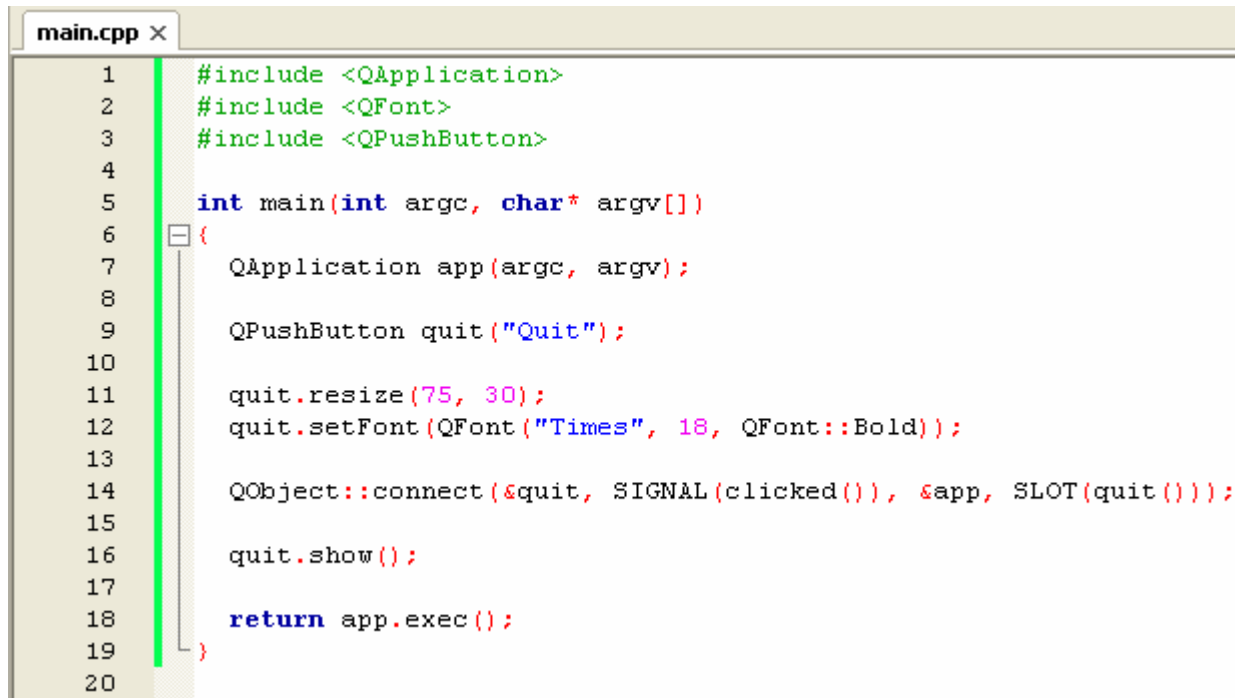
再添加一个 Qmake，但是不用写参数(Parameters:后面的框不填-project，空着)。

(7) 测试

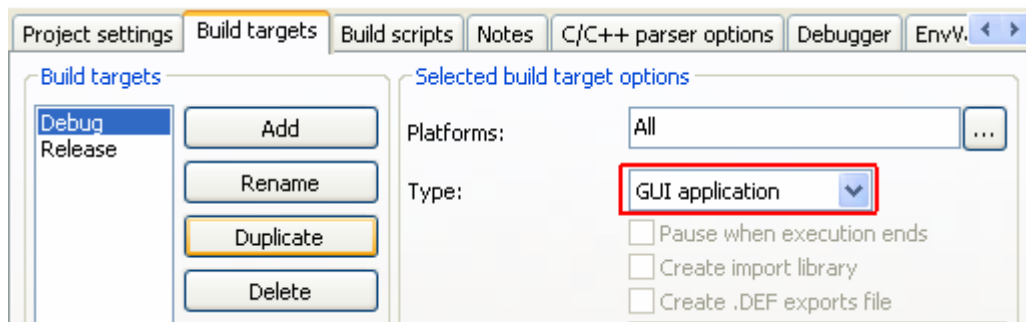
使用 Code::Blocks 的 Wizard 创建一个 Qt 工程。工程名任意，接下来有一个选项需要填入 Qt 路径，这个需要特别注意，否则编译器搜索不到需要加载的文件(在笔者电脑上的路径是 C:\Program Files\CodeBlocks\Qt\qt)见右面的截图。



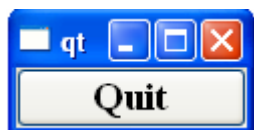
编译器选择 MinGW，最后产生的代码见下面贴图。



配置好该工程的编译选项(Project 下拉菜单选择 Build options...设置适当编译选项)和属性(注意 Project 下拉菜单 Properties...下 Build targets 类型选择 GUI application，见下面截图红色框内文字)。



编译之，除了几个警告外一般不会出现什么问题。假如您以上各个环节配置正确，运行编译后的二进制文件结果见下图(左)。



特别说明

笔者编译和调试本书中所有的程序都使用测试版 MinGW(内嵌了 gcc4.3.3), 如果您想编译本书中的任何程序, 笔者也建议您使用较新版本的编译器(例如 g++4.3.3 或者 g++4.4)和调试器(例如 gdb6.8.3), 此外, 编辑部分程序使用 Code::Blocks build 5456 (2009 年 02 月 14 日的 nightly build 版), 编辑部分程序使用 Code::blocks8.02 官方版。

作者对本书中的任何程序代码不提供任何形式的担保, 如果您希望在与本书无关的地方使用部分或者全部代码, 为了不致于给您、我以及他人带来不便, 代码前请务必加上以下文字信息。

```

/*****
Copyright (C) 2009 Chipset

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU Affero General Public License as
published by the Free Software Foundation, either version 3 of the
License, or (at your option) any later version.

but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.
*****/
```

如果您阅读本书过程中发现错误不妥之处请给我写信 Email: chipset0418@yahoo.com.cn 或者登陆我的博客 <http://www.cppblog.com/chipset> 给我留言。如果您对本书有什么建议, Chipset 洗耳恭听。

如果您在使用 Code::Blocks 的过程中发现本书能给您带来一点心得或收获, 那将是本书作者 Chipset 最大的荣幸。

谢谢!