数据结构与算法

DATA STRUCTURE

第八讲 debug和测试

信息管理与工程学院

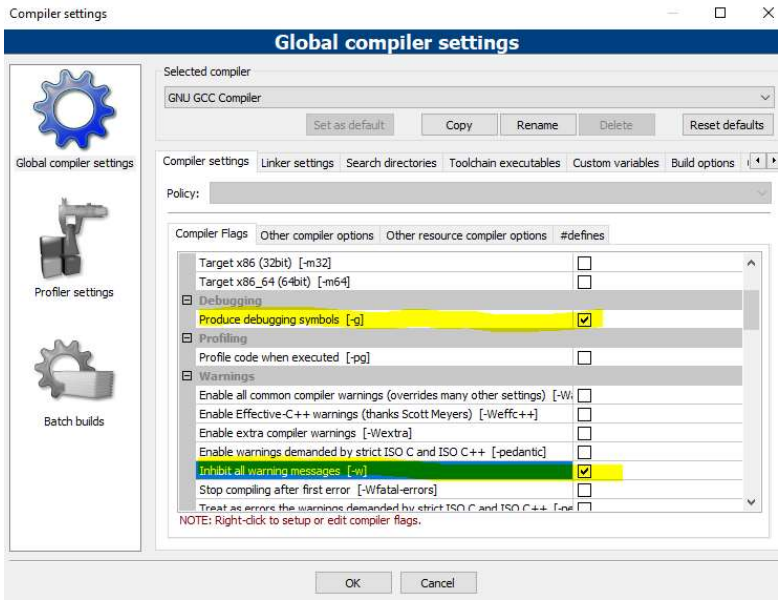
2017 - 2018 第一学期

课堂内容

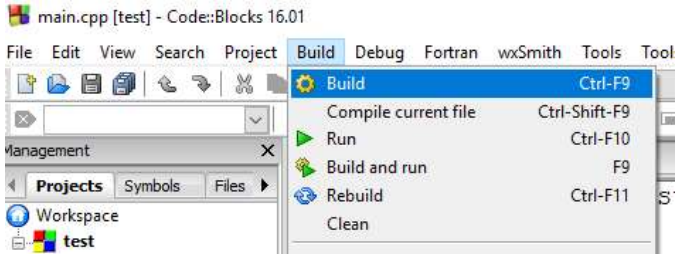
- 如何Debug和测试
- 特殊矩阵
- 字符串
- 匹配算法

codeblocks

- 编译可以debug程序
  - Codeblock-> settings->compiler



  - Codeblock-> build



codeblocks

- 编译可debug程序
  - 在cmd窗口
  - set PATH=%PATH%;C:\Program Files (x86)\CodeBlocks\MinGW\bin;

```
g++ main.cpp -o main
```

  - 需要加上-g，注意-Wall，-Werror用法

```
g++ main.cpp -g -Wall -Werror -o main
```

1 在cmd开始gdb

- 开始：**gdb main.exe**

```
E:\GIT\Works\ProgrammingLecture\projects\hilo>gdb main.exe
GNU gdb (GDB) 7.6.1
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "mingw32".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from E:\GIT\Works\ProgrammingLecture\projects\hilo\main.exe...done.
(gdb)
```

- 退出：**(gdb) q 或者quit**

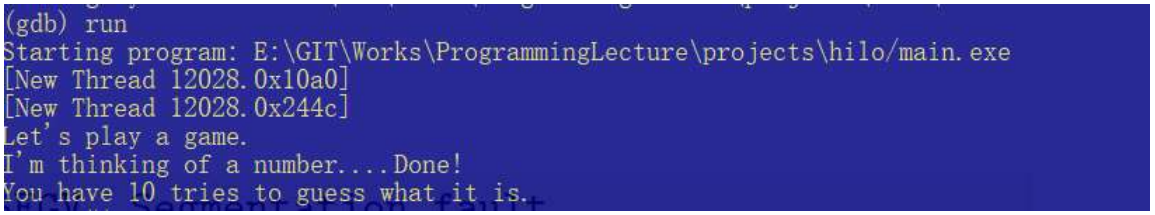
```
Reading symbols from E:\GIT\Works\ProgrammingLecture\projects\hilo\main.exe...done.
(gdb) q

E:\GIT\Works\ProgrammingLecture\projects\hilo>
```

2 运行程序

运行：**(gdb) run**

- 程序运行没问题，输出结果后结束



- 程序如果崩溃了，就会输出一些有用的信息，比如在哪个文件哪一行程序出错，还有些函数调用的参数信息

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400524 in sum_array_region (arr=0x7fffc902a270, r1=2, c1=5,
r2=4, c2=6) at sum-array-region2.c:12
```

3 设置断点breakpoint

- 如果程序出问题了，我们要在出问题的地方设置断点
  - (gdb) break main.cpp:12　　在main.cpp源文件第12行设置断点
  - (gdb) break 15　　在当前源文件第15行设置断点
  - (gdb) break Myclass::MyFunc　在函数入口设置断点
  - (gdb) break　　在下一行设断点
- 之后再运行 (gdb) run，程序就会停留在第一个断点处

```
(gdb) break 14
Breakpoint 1 at 0x401374: file main.cpp, line 14.
(gdb) run
Starting program: E:\GIT\Works\ProgrammingLecture\projects\hilo/main.exe
[New Thread 4220.0x30e8]
[New Thread 4220.0x2bd4]

Breakpoint 1, main () at main.cpp:15
15          rand();
```

```
#include "stdafx.h"
#include <iostream>

void PrintValue(int nValue)
{
    std::cout << nValue;
}

int main()
{
    PrintValue(5);
    return 0;
}
```

3 设置条件断点

- 如果一个函数被调用很多次（递归），这里的断点会hit很多次
- 但是可能某次断点出了问题
- 这时候可以使用有条件中断

```
(gdb) break file1.c:6 if i >= ARRAYSIZE
```

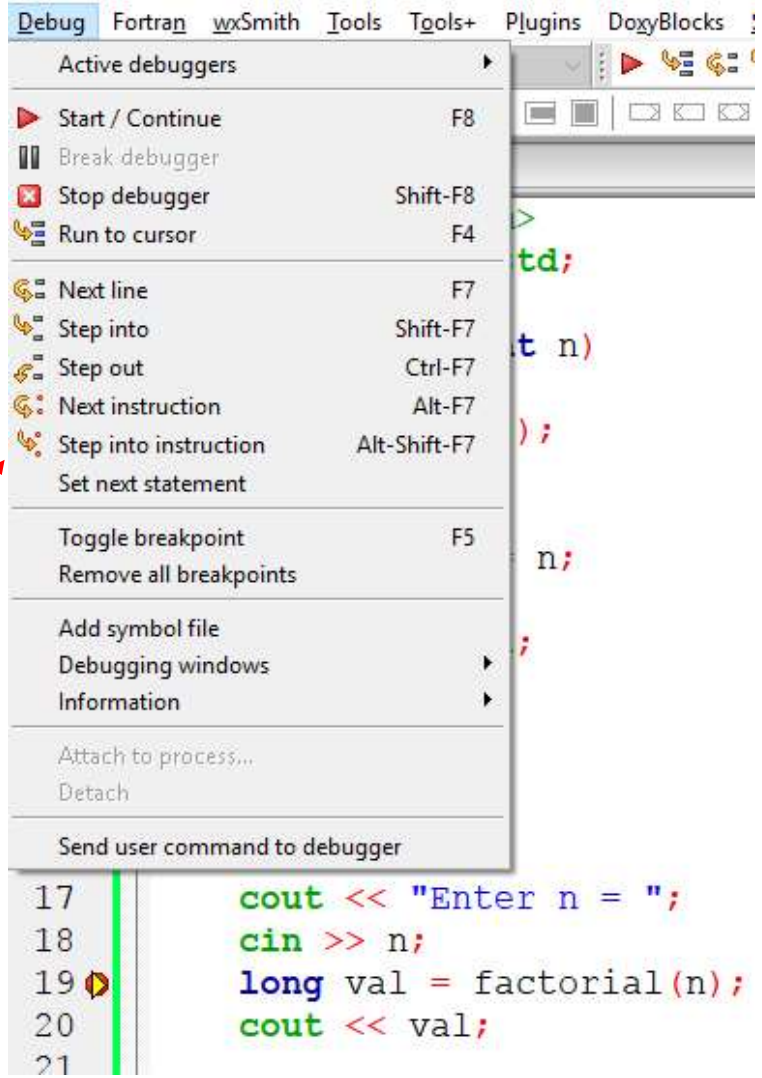
4 到了断点后，下一步

1. 运行(gdb) run，程序又会从头重新开始，不是很有用
2. 运行 (gdb) continue，程序就会停留在下一个断点处，如果有的话
3. 运行 (gdb) step，程序会进入下一个指令，可能进入函数体里面
4. 运行 (gdb) stepi，程序会进入下一个机器指令，可能进入函数体里面
5. 运行 (gdb) next，程序会到达下一行，如果当前行是函数调用也是跳过
6. 运行 (gdb) nexti，程序到达下一个机器指令，跳过当前函数调用
7. 运行 (gdb) finish，程序会跳出当前函数，除非中途有别的断点

注意如果要重复上一个命令，直接enter
Ctrl+C 强行中断程序，由gdb控制

4 下一步stepping

- Step over （next line） == （gdb） next
- Step into == （gdb） step
- Step out == （gdb） finish
- Next Instruction == （gdb） nexti
- Step into instruction == （gdb） stepi
- Run to cursor == （gdb） until #line



5 检查call stack

- 运行(gdb) backtrace，显示整个函数调用call stack
- 运行(gdb) backtrace n，显示前n个函数调用call stack
- 运行(gdb) frame，显示当前栈帧stack frame
- 运行(gdb) frame n，切换到第n个栈帧stack frame
- 运行(gdb) up，切换上一个栈帧frame
- 运行(gdb) down，切换下一个栈帧frame

6 显示源代码

- 当程序停在断点时，会显示当前一行源代码。为了显示更多，
  - (gdb) list main.cpp:12　　显示main.cpp第12行附近源代码
  - (gdb) list 15　　显示当前源文件第15行附近源代码
  - (gdb) list Myclass::MyFunc　　显示函数入口附近源代码
  - (gdb) list　　显示当前行附近源代码

7 然后查看变量

1. 运行(gdb) print my_var，显示变量的值，按原来的类型
2. 运行 (gdb) print /x my_var，显示变量的16进制值（强制转换）
3. 运行 (gdb) watch my_var，一旦变量值改变，会停下来显示原来的值和更新后的值
4. 运行(gdb) set my_var = 4，改变变量的值，设为4

参数 print /x, /d, /u, /o, /t, /a, /c, /f, /s,/r
类似的显示命令还有display，output，等

8 查看内存和寄存器

1. 运行(gdb) x address，显示内存地址指向的值
2. 运行 (gdb) disassemble /m，显示当前附近汇编指令（混合）
3. 运行 (gdb) info registers，显示当前寄存器的值
4. 运行 (gdb) p /x $pc，显示pc寄存器的值
5. 运行 (gdb) x /i $pc，显示下一个机器指令
6. 运行 (gdb) set $sp += 4，改变sp寄存器的值加4

$pc: 程序计数寄存器，下一个指令地址
$sp: 当前堆栈顶部指针寄存器
$fp: 当前堆栈帧指针寄存器
$eax: 一般保存函数返回值

9 显示自定义数据类型值

```
struct entry {
  int key;
  char *name;
  float price;
  long serial_number;
};
struct entry * e1 = <something>; struc
```

- 显示e1的值，或者说指向entry的指针

```
(gdb) print e1
```

- 显示e1指向的entry里面的值

```
(gdb) print *e1
```

- 显示e1指向的entry里面某个数据的值

```
(gdb) print e1->key
(gdb) print e1->name
(gdb) print e1->price
(gdb) print e1->serial_number
```

- 显示指针的指针的指针指向的某个数据的值

```
(gdb) print list_prt->next->next->next->data
```

10 一些有用的命令

- backtrace （bt） – 显示函数调用call stack
- where – 类似backtrace
- finish – 运行到函数结束step out
- delete – 删除断点
- info breakpoints – 显示所有断点
- info thread – 显示所有线程和其id
- thread #id– 切换到第几个thread

- help <command>
- Enter Tab可以显示更多命令
- 命令可以直接用首字母缩写，如果没有歧义的话
- 参考https://www.gnu.org/software/gdb/

例子1：逻辑错误

```
#include<iostream>
using namespace std;

int factorial(int n)
{
   int result(1);
   while(n-- > 0)
   {
      result *= n;
   }
   return result;
}

int main()
{
   int val = factorial(3);
   cout << "n! = " << val << " ";
}
```

例子2：栈溢出

```
#include <iostream>
using namespace std;

int factorial(int n)
{
   return factorial(n-1) * n;
}

int main()
{
   cout << "n! = " << factorial(3) << " ";
}
```

例子3：数组越界

```
#include <iostream>
using namespace std;

void ReadNum(int * array, int size)
{
   for (int i = 0; i < size; i++)
   {
      cout << "\nEnter value [" << i << "] ";
      cin >> array[i];
   }
}

int readInArrayAndSum(int *array, int size)
{
   ReadNum(array, size);

   int sum = 0;
   for ( int i = 0; i <= size; i++ )
   {
      sum += array[i];
   }
   return sum;
}
```

```
int main()
{
   for( ; ; )
   {
      int size = 0;
      cout << "\nEnter the size of the array to sum:";
      cin >> size;
      if (size <= 0)
      {
         cout << "\nInvalid array size: " << size << endl;
         break;
      }

      int * array;
      int result = readInArrayAndSum(array, size);
      cout << "\nsum of [";
      for (int i = 0; i < size; i++)
      {
         cout << array[i] << ", ";
      }

      cout << "\b\b] is " << result << endl;
   }

   return 0;
}
```

例子4：无限循环

```
#include <iostream>
using namespace std;

int main()
{
   const char * szBuffer = "!Error";

   int length = sizeof(szBuffer);

   while (length > 0);
   {
      cout << szBuffer[--length] << " ";
   }
}
```

- 注意sizeof是可以用来计算数组大小的。尤其如果是字符数组，那么sizeof就是字符数组的长度，因为字符是一个字节。
- 这里如果改成
  const char szBuffer[] = “!Error”;
  就可以通过测试
- 或者使用strlen(szBuffer)
- 这里sizeof(szBuffer)其实是指针类型占用的字节数：4

例子五

- 在gdb里观察IntArray和SortedIntArray的内存分布

Q&A

Thanks!