第四章 函数

第48课 GDB调试

《信息学奥赛一本通·编程启蒙 C++版》

一、GDB 简介

GDB（GNU symbolic debugger）简单地说就是一个调试工具。它是一个受通用公共许可证即 GPL 保护的自由软件。

像所有的调试器一样，GDB 可以让你调试一个程序，包括让程序在你希望的地方停下，此时你可以查看变量、寄存器、内存及堆栈。更进一步你可以修改变量及内存值。

DEV-cpp 已经集成了 GDB 调试工具，使我们初学者有了图形化界面，使用更方便，但它们仅是 GDB 的一层外壳。

命令	缩写	说明
run	r	运行带参数的可执行文件：r 后面接参数
break	b	break/b xxx 在某行打断点
continue	c	继续运行
next	n	单步运行
step	s	如果有函数则进入函数执行
backtrace	bt	查看栈帧中各级函数调用及参数
up	Up	跳到下一个编号的栈帧
down	down	跳到上一个编号的栈帧
finish		跳出当前的函数
jump	j	跳转到指定行/地址后继续执行，因此如果在跳转的目标行上如果没有设置断点，会继续往下执行
stop		停止运行
quit	ctrl+d	退出 GDB
print	p	print/p var 打印变量的值
		print/p &var 打印变量地址
		printf/p *addr 打印地址的值
		printf/p /x var 用 16 进制显示数据

二、在 dev-cpp 中使用 GDB

在 dev-cpp 中进行调试时，一般的步骤为：

（1）先根据系统，选择“Debug”模式然后进行编译。

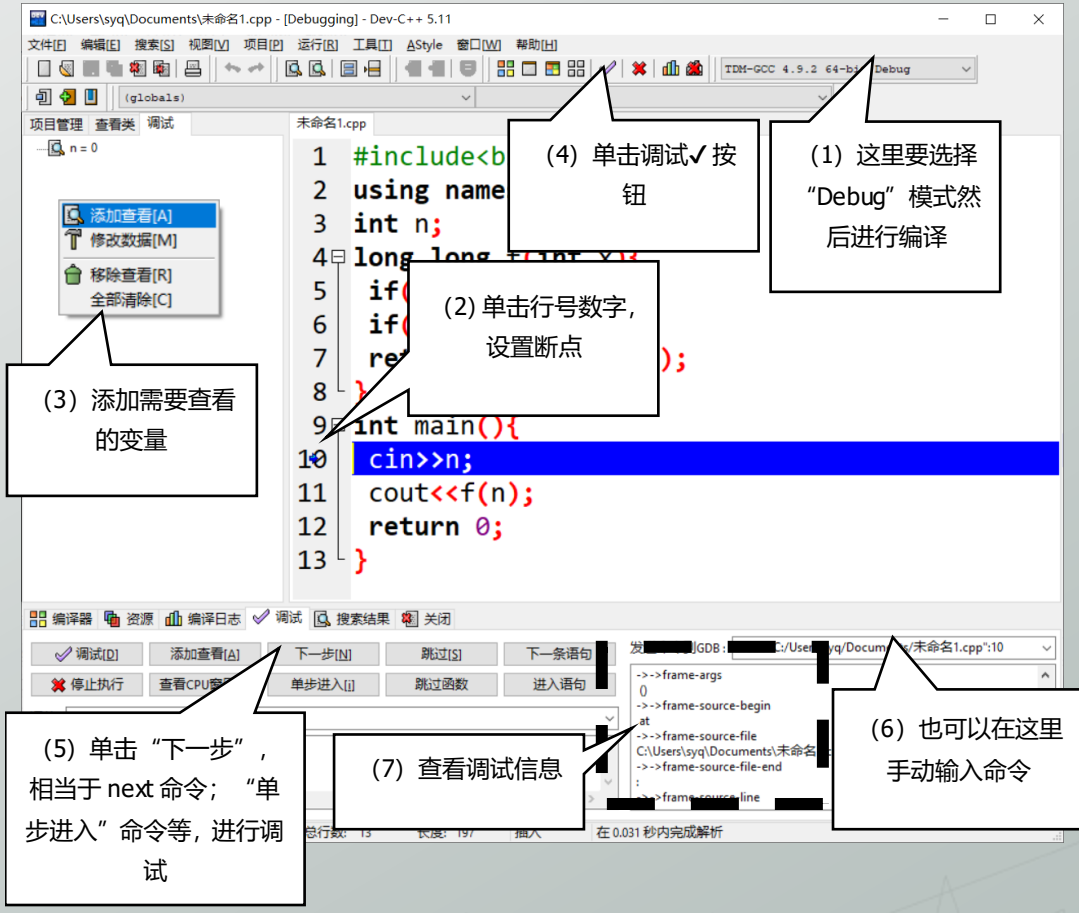
（2）单击行号数字，在该行设置断点，相当于命令“break/b xxx 在某行打断点”。

（3）添加需要查看的变量，相当于命令“print/p var 打印变量的值”。

（4）单击调试✓按钮，相当于命令“run”。

（5）程序将执行到断点前，接下来可以单击“下一步”，相当于 next 命令；“单步进入”命令，相当于 step 命令等，进行调试，观察需要监测的变量的变化。

（6）如果程序员，就软件提供的按钮命令不够用的话，也可以手动输入命令，并查看具体的信息。



三、调用栈

调用栈描述的是函数之间的调用关系，它有多个栈帧（stack frame）组成，每个栈帧对应着一个未运行完的函数，栈帧中保存了该函数的返回地址和局部变量，这样就保证了不同函数间的局部变量互不想干，不同的函数可以有相同的变量，因为不同函数所对应的栈帧不同。在 gdb 中可以用 backtrace（bt 命令）打印所有栈帧信息。下面我们通过一个递归程序的例子，查看一下栈帧。

【例 48.1】 斐波那契数列

【题目描述】

输出斐波那契数列第 n 项。1，1，2，3，5，8，13……

【输入格式】

一个正整数 n,表示第 n 项。

【输出格式】

第 n 项是多少。

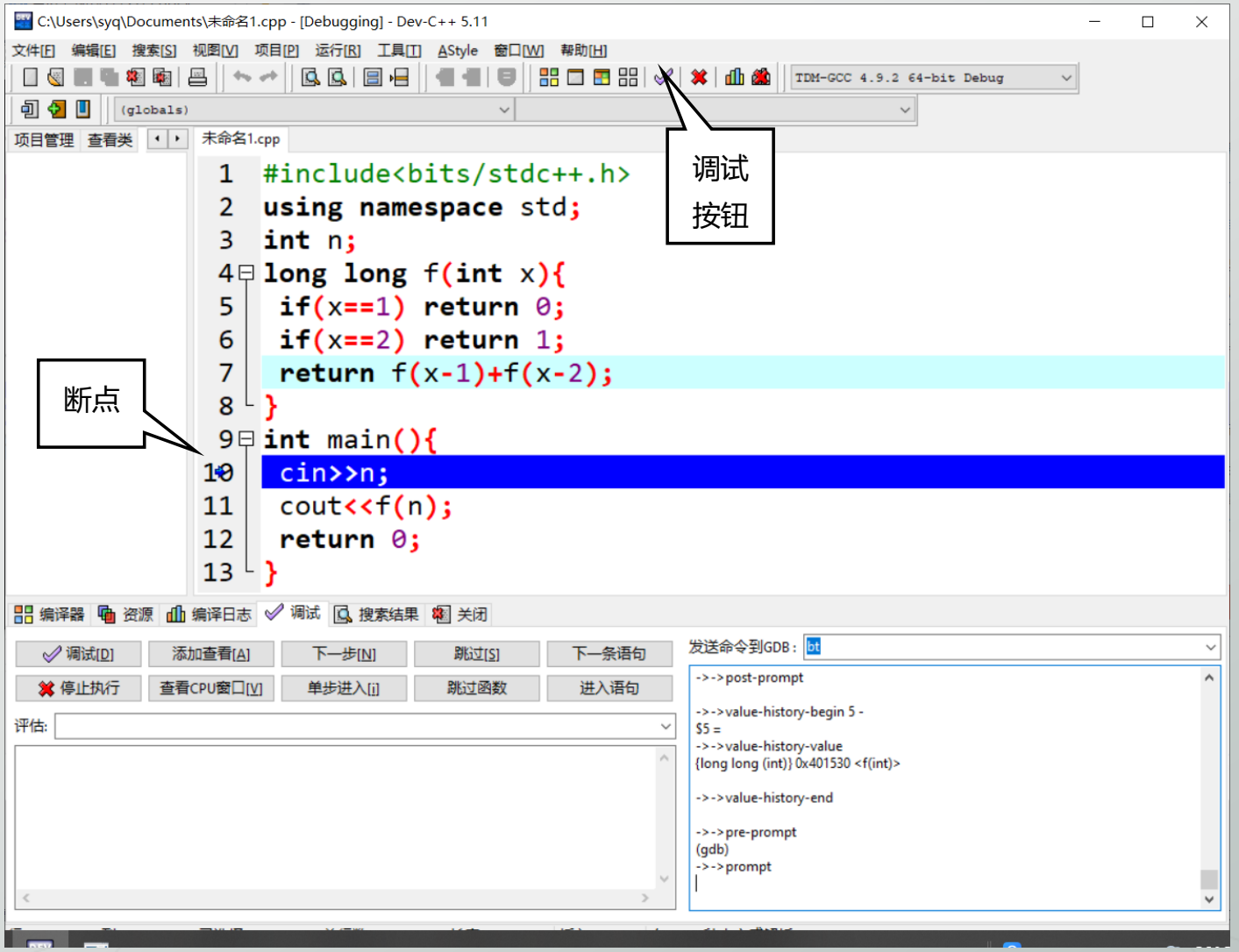
【样例输入】

3

【样例输出】

2

```cpp
#include<bits/stdc++.h>
using namespace std;
int n;
long long f(int x){
  if(x==1) return 1;
  if(x==2) return 1;
  return f(x-1)+f(x-2);}
int main(){
  cin>>n;
  cout<<f(n);
  return 0;
}
```



（1）先在第 10 行设置断点

（2）单击调试按钮

（1）发送 bt 命令，查看栈帧情况，因为刚开始执行 main，只有一个#0 栈帧，信息如下：

```
->->frame-begin 0 0x40158f
#0
->->frame-function-name
main
```

说明：0x40158f 这个栈帧的地址，由于每个读者的电脑不一样，呈现出来的地址也不一样，下面出现的地址亦是如此。

（4）输入 3，我们打算计算一下第三项斐波那契的值，然后发送 n 命令，程序执行到第 11 行

（5）发送 s 命令，进入函数 f（3），再发 bt 命令，查看栈帧情况，信息如下：

```
->->frame-begin 0 0x401541
#0
->->frame-function-name
f
->->frame-args
 (
->->arg-begin
x
->->arg-name-end
=
->->arg-value -
3
->->arg-end
)
```

我们发现#0 栈帧变成了 f（3），说明 f（3）在栈顶

```
->->frame-begin 1 0x4015af
#1
->->frame-address
0x00000000004015af
->->frame-address-end
 in
->->frame-function-name
main
```

我们发现 main（）函数的栈帧变成了#1，

（6）连续发送 3 条 s 命令，将进入函数 f（2）,再发 bt 命令，查看栈帧情况，我们会发现 main 函数在#2 栈帧里，f（3）在#1 栈帧里，新的 f（2）函数在#0 栈帧里。

（7）发送 p x 查看 x 的变量，由于当前是在函数 f（2）对应的#0 栈帧里，因此 x 变量的值应该为 2。

```
->->value-history-begin 15 -
$15 =
->->value-history-value
2
```

（8）如果要看上一级函数 f（3）里的 x 变量的值，对应的是#1 栈帧里的 x 的值，可以先发送 up 命令，然后再发送 p x 查看 x 的变量，此时的 x 的值应该为 3，这样就很好的解释了，不同的函数，可以使用相同名字的变量。

```
->->value-history-begin 16 -
$16 =
->->value-history-value
3
```

读者可以继续发送 s 命令，每执行一步，发送 p x 命令查看 x 的变量，以及发送 bt 命令查看栈帧的情况，直到程序结束。我们可以得出以下结论：

（1）每次调用函数，都会建立新栈帧，调用自身的递归函数和普通的调用其他的函数，从本质上看没有区别。

（2）在递归的时候，递归的层数对应的是新建栈帧的个数，所以数量较大的递归会引起栈帧太多导致的段错误。

（3）在函数体执行完毕后会删除栈帧。

谢谢！