多个函数

```
//* two_func.c -- a program using two functions in one file */
#include <stdio.h>
void butler(void);      /* ANSI/ISO C function prototyping */
int main(void)
{
    printf("I will summon the butler function.\n");
    butler();
    printf("Yes. Bring me some tea and writeable DVDs.\n");

    return 0;
}

void butler(void)       /* start of function definition */
{
    printf("You rang, sir?\n");
}
```

输出：

```
I will summon the butler function.
You rang, sir?
Yes. Bring me some tea and writeable DVDs.
```

函数可以在main前面声明，在main中调用，具体实现放到main的后面。函数声明也叫做函数原型，可以在编译阶段进行语法检查，比如函数名是否存在，参数个数或参数类型是否匹配，可以提升一些效率。

同时，在表现形式上，C语言的入口是main开始，如果项目比较大，我们更倾向于打开程序尽快看到main函数。且我们完成函数实现之后并不需要经常考虑函数细节，只需要知道函数的名字和功能就可以，这样就可以快速使用函数。

调试错误

在编写程序过程中，一定会遇到各种错误，然而编译器都会给出一些相关提示信息，我们要根据提示找到错误并改正错误。

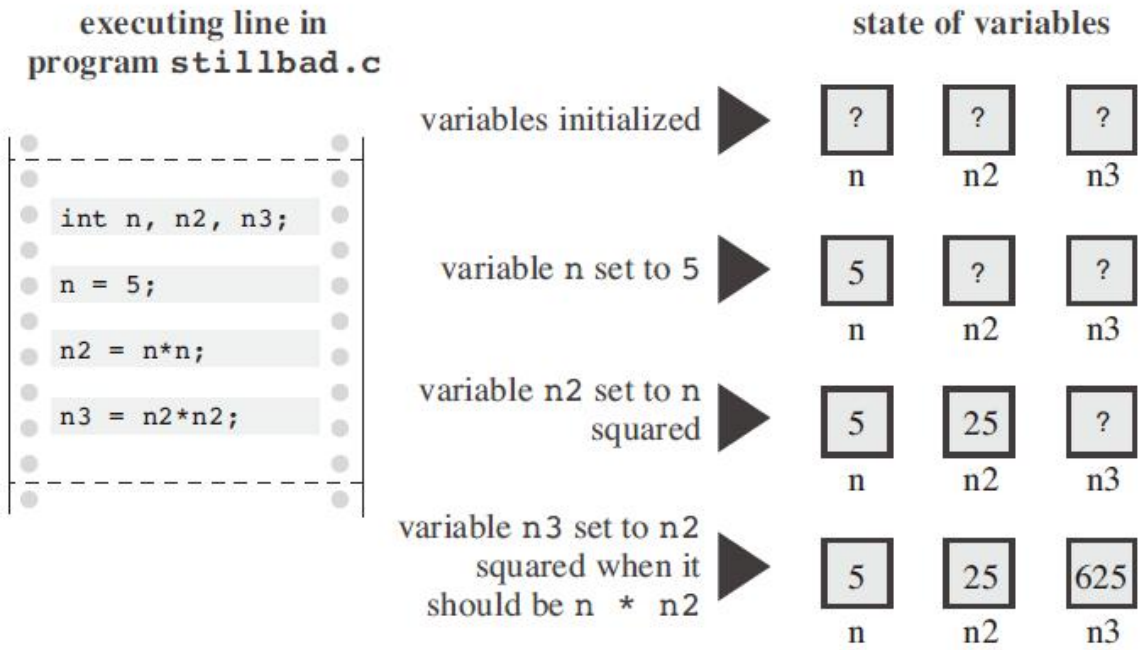
```
#include <stdio.h>
int main(void)
(
    int n, int n2, int n3;

/* this program has several errors
    n = 5;
    n2 = n * n;
    n3 = n2 * n2;
    printf("n = %d, n squared = %d, n cubed = %d\n", n, n2, n3)

    return 0;
)
```

错误类型一般分为语法错误和语义错误，语法错误就是编译器无法通过语法检查。

比如：

1. main()后面应该使用花括号{}。
2. 变量声明改为：int n,n2,n3；或者单独声明。
3. 注释末尾加上*/。
4. printf()后面加上分号";"。

然而，尽管现在程序可以正确运行，但是依然在逻辑上有问题，这就是语义错误。最终计算体积应该是n的三次方，然而当前程序的运行后得到的体积是n的4次方。



可以更改为 n3 = n * n2;

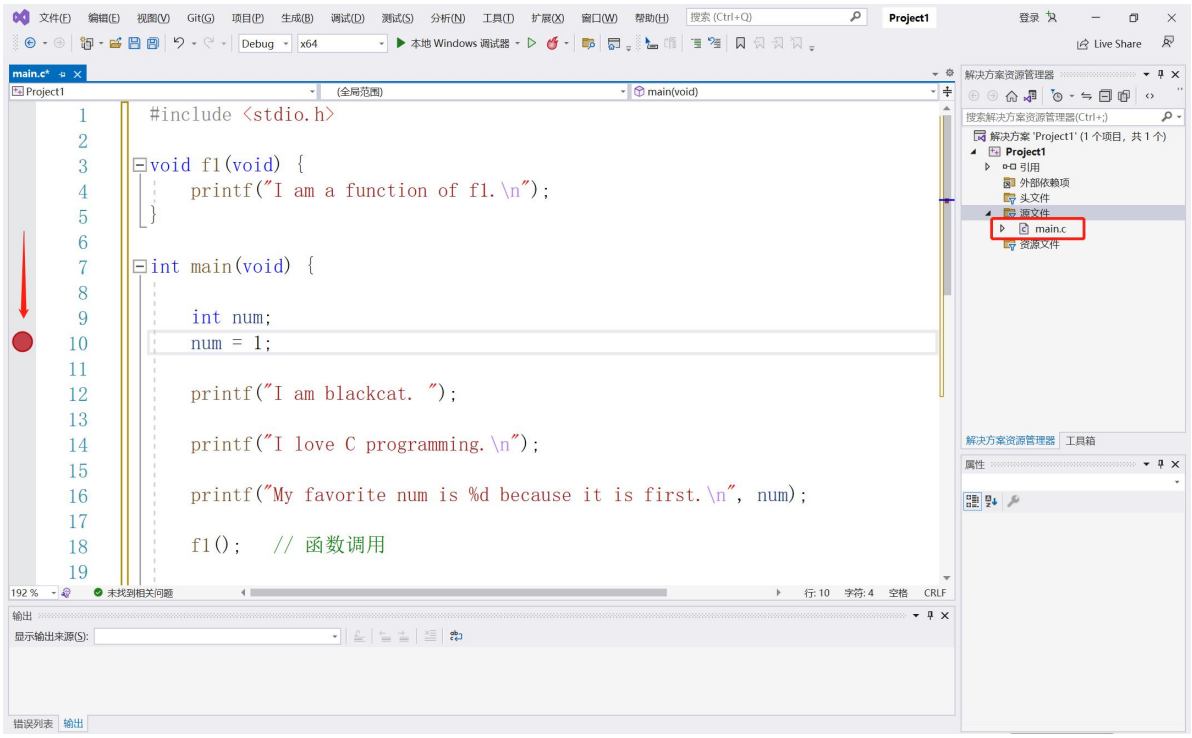
程序状态

编程过程中，要非常清楚自己的程序运行流程，跟踪记录每个变量，如果不明确程序的结果，不应该胡乱尝试。可以借助于printf打印变量值，和自己预计的结果进行对比，分析错误原因。

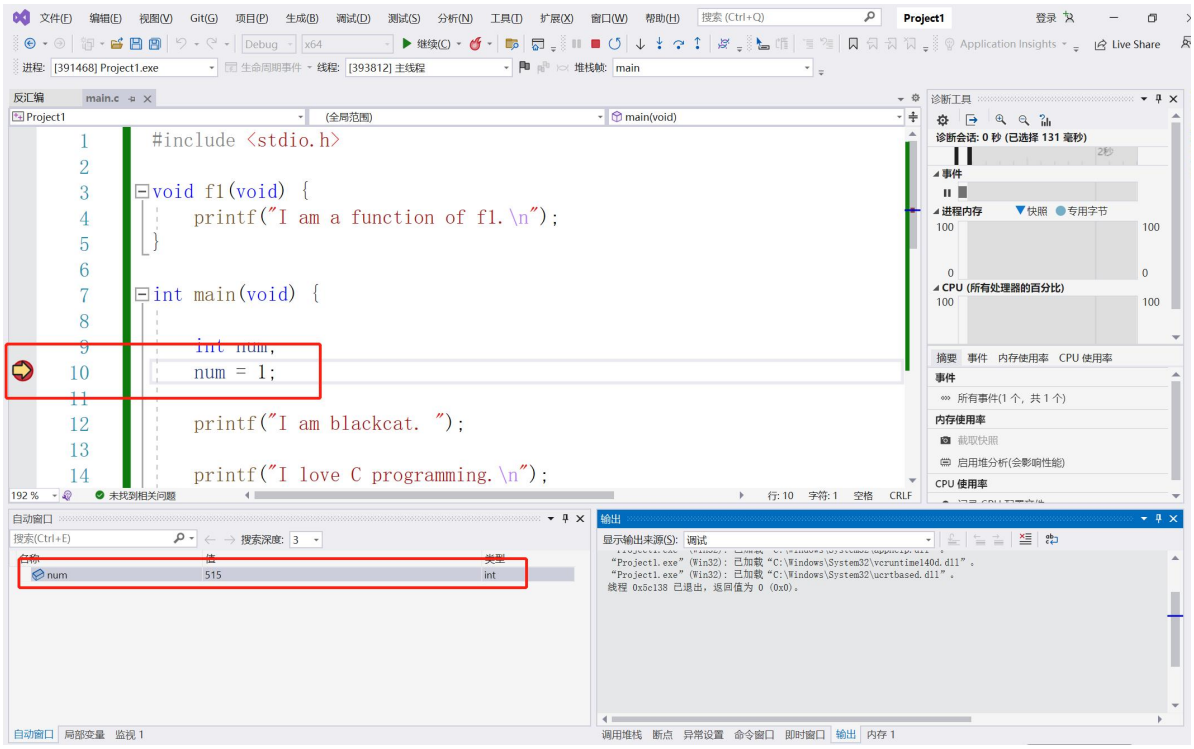
程序状态就是相当于程序当前状态的一个快照，但是程序一旦运行很快就结束了，我们无法窥测其中的具体过程。因此，也可以借助于带有调试功能的编译器来单步执行程序，更加有利于错误分析。

个人编程习惯比较喜欢VS系列，本样例使用 VS 2022 Community 版本。

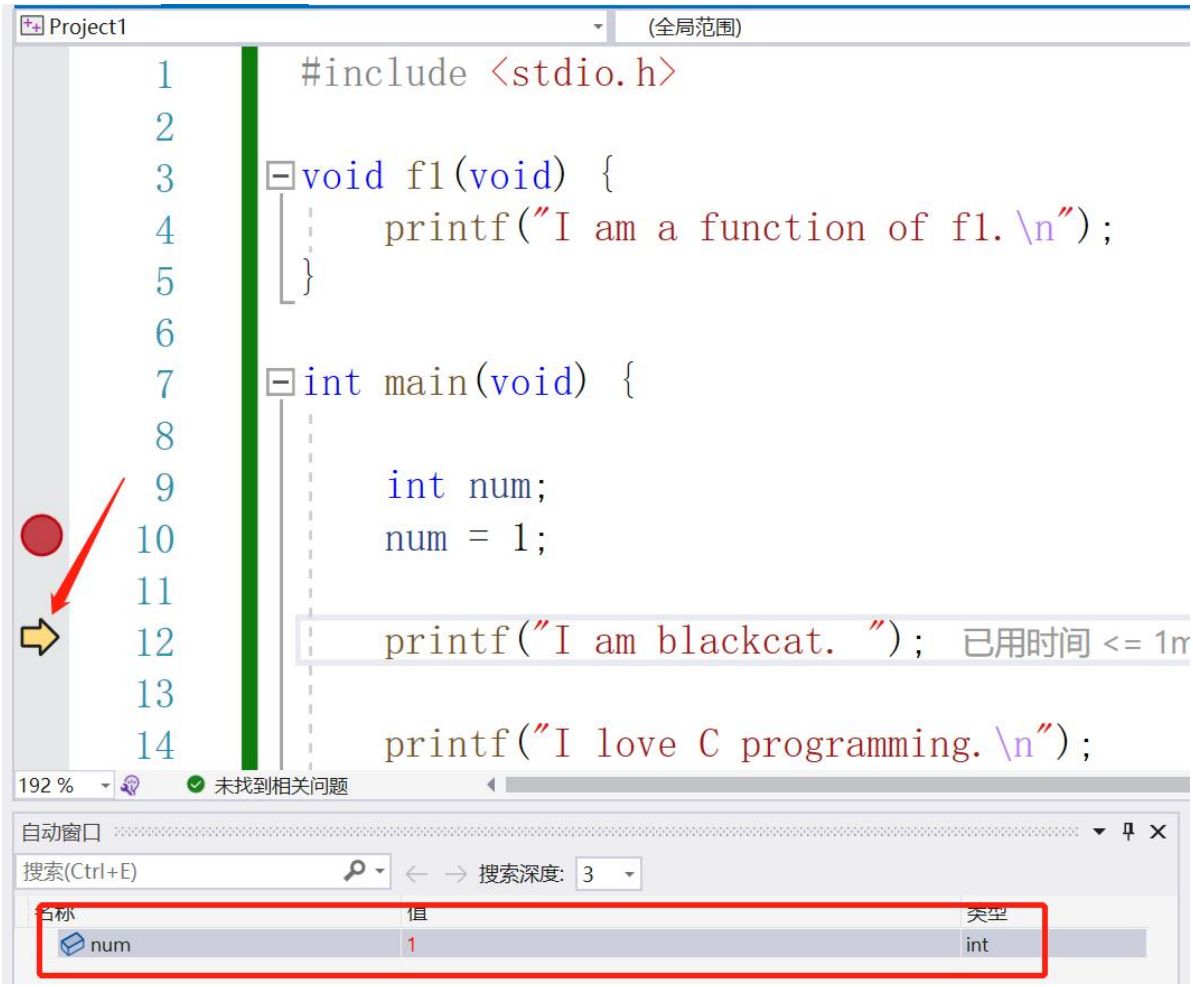
F9下断点，调试时，会停留在断点位置：



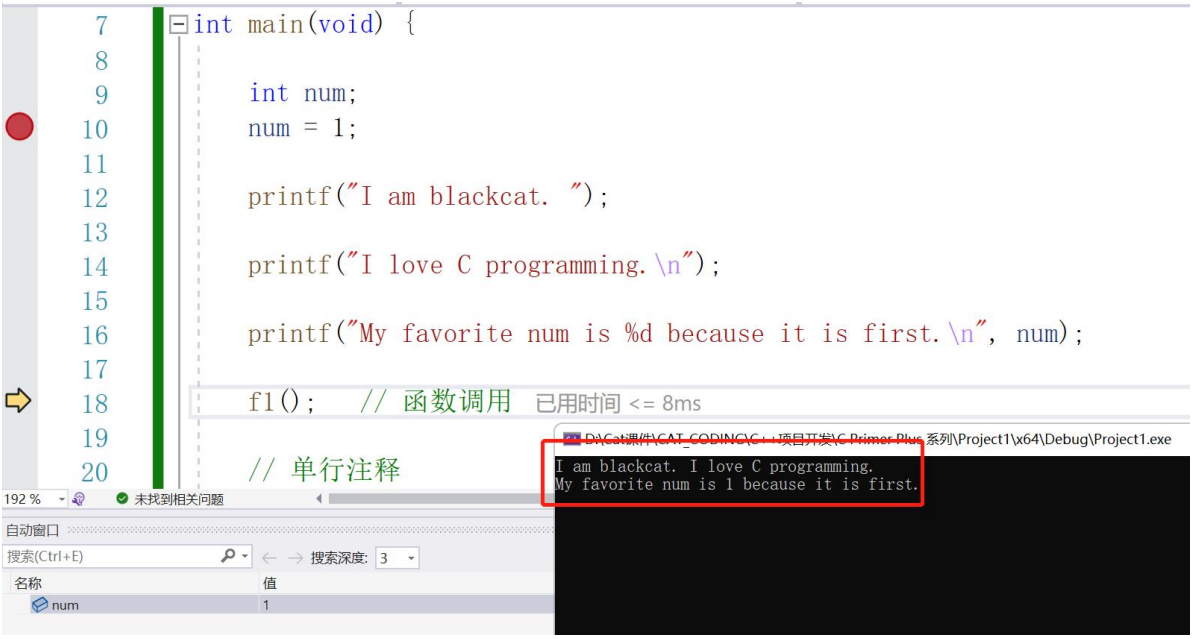
当前num未赋值，是随机值，F11逐语句执行：



箭头指向为当前待执行行，num已经被赋值为1。

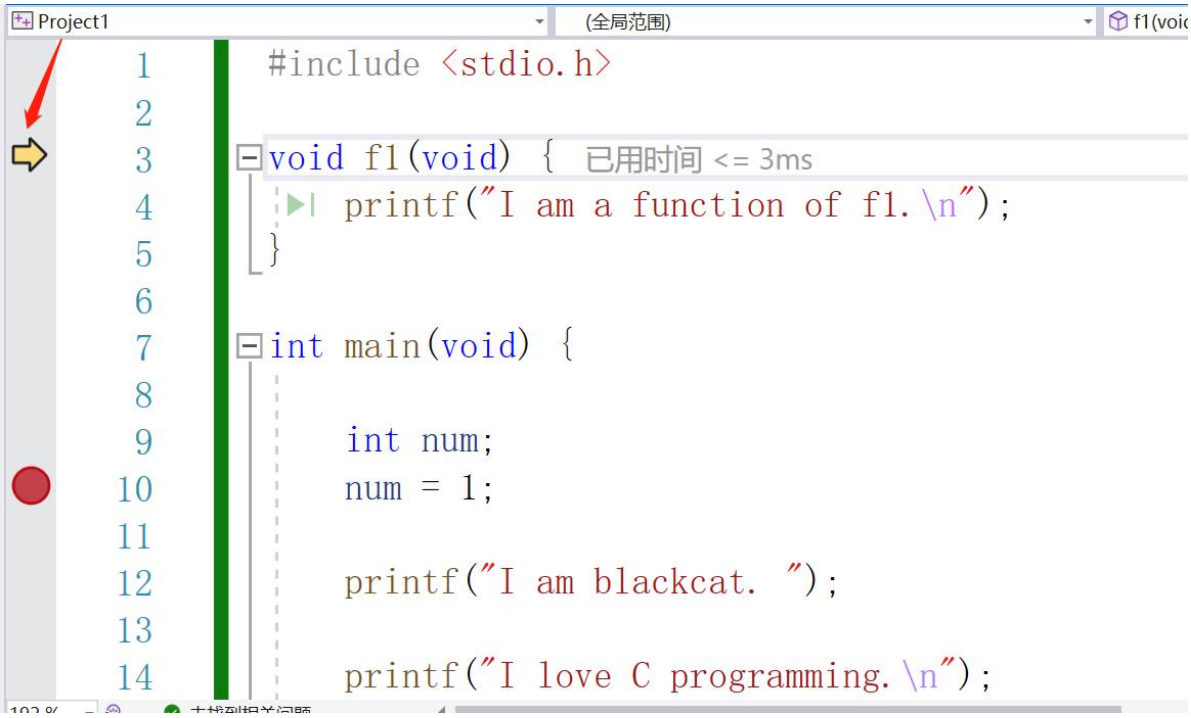


现在待执行f1()，控制台已经输出两段信息：



如果F11逐语句执行则会进入到f1内部，如果F10逐过程执行会执行完f1。

我这里按下F11：



然后可以逐过程也可以逐语句，也可以Shift+F11跳出当前函数。

单步调试是复杂程序调试时常用的方法，如果是为了学习算法，前期可以使用单步调试，更要学会通过printf语句来分析问题，在其他语言中也经常要用类似的输出语句调试，因为在很多算法竞赛中，调试功能是被禁止使用的。

关键字和保留标识符

关键字是C语言的词汇，比如main、return，具有特殊意义，不能用作标识符，比如变量名、函数名等。

ISO C Keywords			
auto	extern	short	while
break	float	**signed**	***_Alignas***
case	for	sizeof	***_Alignof***
char	goto	static	***_Bool***
const	if	struct	***_Complex***
continue	*inline*	switch	***_Generic***
default	int	typedef	***_Imaginary***

do	long	union	***_Noreturn***
double	register	unsigned	***_Static_assert***
else	restrict	void	***#_Thread_local***
enum	return	**volatile**	