

第2章 Linux编程环境



第2讲 Linux编程环境

- 2.1 gcc编译器
- 2.2 gdb调试器
- 2.3 make工具

2.1 GCC

- gcc的全称是GNU Compiler Collection, 即GNU编译器套件.
- 从名称可以看出来,gcc产出于GNU项目,它的初衷是为了给GNU操作系统专门写一款编译器,以解决不同GNU系统间编译器混乱的问题.
- 现在,它已经可以编译众多语言,例如C, C++, Objective-C, Fortran, Ada, Go.并且成为了C, C++编译器的首选

2.1.1 gcc简介

- GCC是Linux平台下最常用的编译程序，是Linux平台编译器的事实标准。
- GCC支持的体系结构有40余种，常见的有x86系列、Arm、PowerPC等。同时，GCC还能运行在不同的操作系统上，如Linux、Solaris、Windows等。
- GCC除了支持C语言外，还支持多种其他语言，例如C++、Ada、Java、Objective-C、Fortran、Pascal等。

编译过程



2.1.2 gcc 常用选项：预处理控制

- ➡ (1)-E选项。该选项指示gcc编译器仅对输入文件进行预处理，同时将预处理器的输出被送到标准输出而不是文件。
- ➡ (2) -D name选项。预定义名称为name的宏，其内容为1。



```
#include <stdio.h>
```

```
void main()
```

```
{
```

```
    int arr[10],i=0;
```

```
    for (i=0;i<10;i++)
```

```
    {
```

```
        arr[i]=i;
```

```
        if (DEBUG) //使用了一个名为DEBUG的宏，  
                    该宏在编译 的时候定义。
```


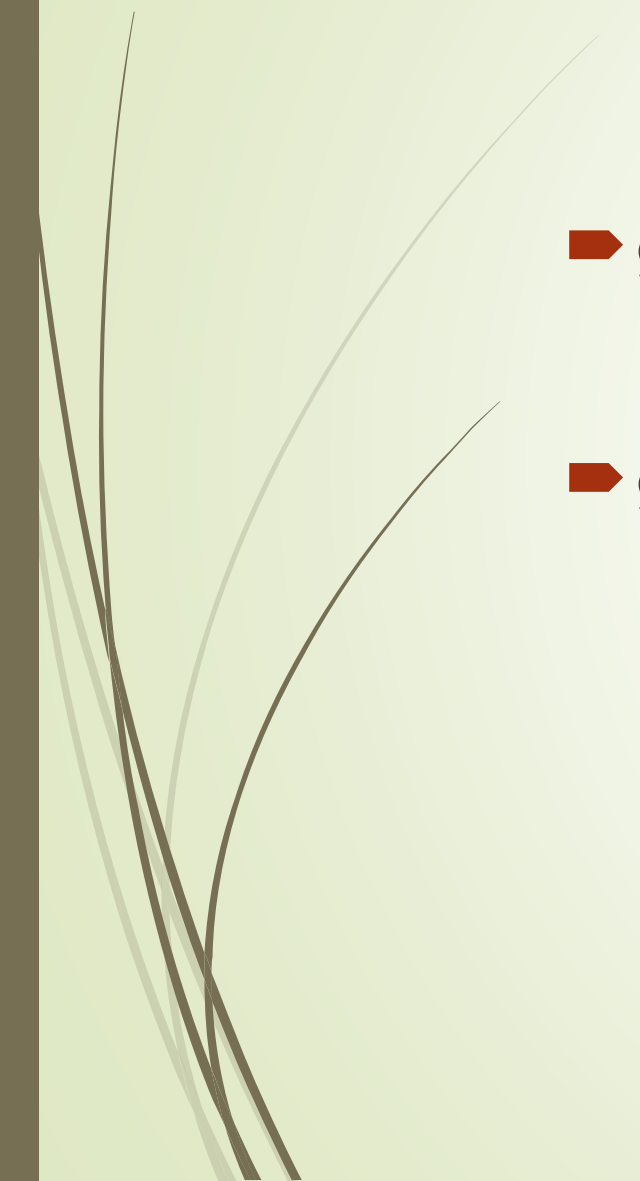
```
        {
```

```
            printf("arr[%d]=%d\n",i,arr[i]);
```

```
        }
```


```
    }
```

```
}
```


- 
- 
- `gcc -DDEBUG -o debug debug.c`
 - `gcc -DDEBUG=0 -o debug debug.c`

2. 1. 3 gcc选项：编译及警告信息控制

常用选项	说 明
-o	表示要求编译器生成指定文件名的可执行文件
-c	表示只要求编译器进行编译生成.o的目标文件，而不要进行链接
-g	要求编译器在编译的时候提供以后对程序进行调试的信息
-E	表示编译器对源文件只进行预处理就停止，而不做编译、汇编和链接
-S	表示编译器只进行编译，而不做汇编和链接
-O	表示编译器优化生成可执行文件
-Wall	生成所有的警告信息



例如, `$ gcc -o hello hello.c`

`$. /hello`

`gcc -c hello.c`

`./hello` 无法执行

`gcc -o test first.c second.c third.c`

2.1.4 语言控制和程序调试及优化

- (1)-ansi选项，它等价于-std=c89。该选项指定源程序使用ISO C90标准
- (2)-std=选项，它确定源程序中所使用的C语言标准。

优化选项：

- (1) -O选项，编译器设法减小代码长度及执行时间，但不会进行花费大量编译时间的优化。
- (2) -O1选项，优化编译功能需要更多时间及大量内存。
- (3) -O2选项，该选项表示进一步优化。
- (4) -O0

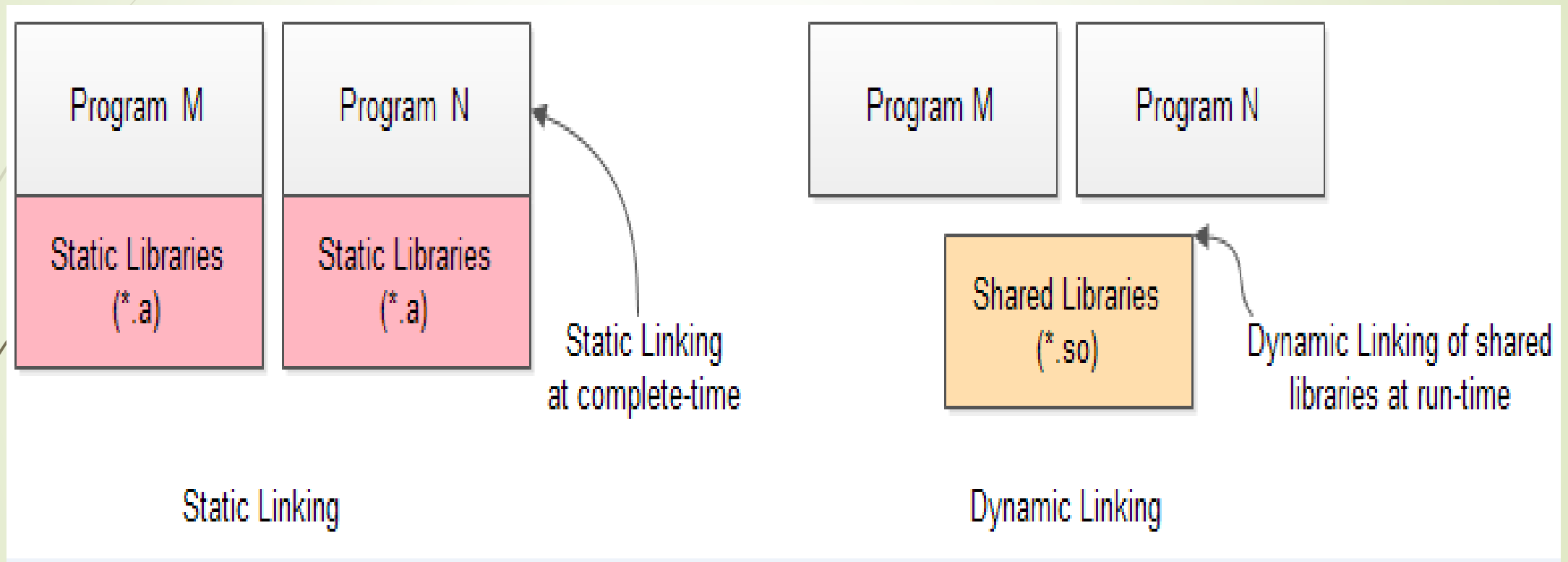
2.1.5 搜索路径控制和gcc链接选项

- `$ gcc test.c -I../inc -o test`
- 此命令告诉GCC包含文件存放在../inc 目录下，在当前目录的上一级。可使用 **多个-I** 来指定多个目录。
- -L dir选项。该选项将dir添加到库文件搜索路径中
- -l library或者-l library选项，指定需要使用的库名称
- -static选项，该选项表示在编译时强制使用对应的静态链接库。
- -shared选项，该选项创建共享库。它所创建的动态库文件以.so后缀结尾。

2.1.6 利用gcc创建库文件

- 开发过程中，使用外部或者其它模块提供的功能，该功能以库文件的形式存在
- 主要分为静态库及动态(或共享)库两种形式。

静态库和动态库的区别



2.2 GDB

GDB调试器简介

Linux系统中包含了GNU 调试程序gdb，用来调试C和 C++程序的调试器。gdb 提供如下功能：

- 运行程序，设置所有的能影响程序运行的参数和环境。
- 控制程序在指定的条件下停止运行。
- 当程序停止时，可以检查程序的状态。
- 修改程序的错误，并重新运行程序。
- 动态监视程序中变量的值。
- 可以单步执行代码，观察程序的运行状态。

准备工作

- ▶ 在开启gdb调试之前，需要在编译源程序的时候加上-g选项，并将程序的崩溃信息转储的core文件
 - ▶ `gcc -g test.c func.c -o test.out //编译加上调试信息`
- ▶ 使用ulimit -c命令可查看core文件的生成开关。若结果为0，则表示关闭了此功能，不会生成core文件。
- ▶ `ulimit -c unlimited` //让程序在崩溃时产生core文件
- ▶ 使用ulimit -c filesize命令，可以限制core文件的大小（filesize的单位为kbyte）
- ▶ core文件生成路径：输入可执行文件运行命令的同一路径下。若系统生成的core文件不带其它任何扩展名称，则全部命名为core。新的core文件生成将覆盖原来的core文件。
- ▶ 输入命令：`gdb test.out test.out-core` 进行调试

gdb的启动

有两种方法运行gdb，即在终端窗口的命令行中直接输入gdb命令或gdb filename命令运行gdb。

方法1:

先启动gdb后执行file filename命令。即

```
gdb
```

```
file filename
```

执行上述两条命令就可启动gdb，并装入可执行的程序filename

方法2:

启动gdb的同时装入可执行的程序。即

```
gdb filename
```

其中，filename是要调试的可执行文件。

启动gdb后，就可以使用gdb的命令调试程序。



log

- 如果你想把GDB命令输出到一个文件有，有几种方法控制
- `set logging on`
- `set logging off`
- `set logging file <filename>`
- `set logging overwrite [on | off]` //默认会追加到logfile里
- `set logging redirect [on | off]` //默认GDB输出会在terminal和logfile里显示，用redirect让它只在logfile里显示
- `show logging`

让被调试程序运行

- 包含三个命令：
- run : 开始运行
 - 后面可以跟args
- start : 运行并停在main函数上
 - 后面可以跟args
- continue : 继续运行

checkpoint

- gdb可以在程序执行的过程中保留快照(状态)信息，称之为checkpoint，可以在进来返回到该处再次查看当时的信息，比如内存、寄存器以及部分系统状态。通过设置checkpoint，万一调试的过程中错误发生了但是已经跳过了错误发生的地方，就可以快速返回checkpoint再开始调试，而不用重启程序重新来过。
- **checkpoint**
- **info checkpoints**
- **restart checkpoint-id**
- **delete checkpoint checkpoint-id**
- 此外，checkpoint的作用还在于断点、观测点不是什么情况下都可用的情况下，因为Linux系统有时候为了安全考虑，会随机化新进程的地址空间，这样重启调试程序会导致之前使用绝对地址设置的断点、观测点不可用。

断点

- 在一个位置上设置断点，可以对应多个位置，gdb要自动在需要的位置插入断点。在动态库里也可以设置断点，不过其地址在加载后才能解析。断点的设置有几种方法
- `break`
- `b`
- `break [Function Name]` //函数名
- `break [File Name]:[Line Number]` //文件名的第几行
- `break [Line Number]` //第几行
- `break *[Address]` // 想在地址0x4007d9 上设定断点 eg: `break *0x4007d9`

- // 用下面命令得到相应行的地址 i line gdbprog.cc:14
- `break [...] if [Condition]`
- `break [...] thread [Thread-id]`
- `b [...]`

条件断点

- 大约有以下几种形式
 - `break main if argc > 1`
 - `break 180 if (string == NULL && i < 0)`
 - `break test.c:34 if (x & y) == 1`
 - `break myfunc if i % (j+3) != 0`
 - `break 44 if strlen(mystring) == 0`
 - `b 10 if ((int)$gdb_strcmp(a,"chinaunix") == 0)`
 - `b 10 if ((int)aa.find("dd",0) == 0)`
- 在if和后面的（之间没放空格另外注意条件表达式的返回值类型是int)



查看断点

- 可以用info breakpoints来查看相应断点信息



显示检查数据

- print
- display
- set
- watch
- catch
- tracepoint

print

- print接受表达式和计算它的值。任何该语言支持常值，变量和操作符都可以使用，像条件表达式，函数调用，类型转换，字符常量。GDB还支持数组常量，语法是{element, element...}, 比如print {1,2,3}.GDB还支持下面操作符
- @ 二进制操作符， 可以把memory当成数组。
- 如：int *array = (int*) malloc(len * sizeof(int));
 - 可以使用下面命令来打印它的值：
 - (gdb) p *array@len
 - (gdb) p/x (short[2])0x12345678
 - \$1 = {0x1234, 0x5678}



display

- 如果你发现你经常要打印某个表达式，你可以把它加入到“automatic display list”。每次程序停止时，都会显示
- `display expr`
- `display/fmt expr`
- `display/fmt addr`
- `undisplay <dnums>`
- `delete display dnums`
- `disable display dnums`
- `enable display dnums`

watch

- 监视点是监视特定内存位置、特定表达式的值是否改变而触发程序暂停执行，而不用去关心该值到底在代码的哪个位置被修改的。监视的表达式可以是：某个变量的引用、强制地址解析(比如(int)0x12345678，你无法watch一个地址，因为地址是永远也不会改变的)、合理的表达式(比如a-b+c/d，gdb会检测其中引用的各个变量)。
- watch [-l|-location] expr [thread thread-id] [mask maskvalue]
- -location会让gdb计算expr的表达式，并将计算的结果作为地址，并探测该地址上的值
- watch命令还存在两个变体：rwatch当expr被程序读的时候触发中断；awatch会在程序读取或者写入expr的时候被中断。rwatch和awatch只支持硬件模式的检测点



Stack命令

- 主要包含命令
 - bt
 - frame
 - info frame
- 



bt

- 可以用负数, -1是第一层, -2是第二层
- (gdb) bt
- #0 level0 () at recursion.cpp:5
- #1 0x08048462 in test (level=0) at recursion.cpp:17
- #2 0x0804845b in test (level=1) at recursion.cpp:14
- #3 0x0804845b in test (level=2) at recursion.cpp:14
- #4 0x0804845b in test (level=3) at recursion.cpp:14
- #5 0x0804845b in test (level=4) at recursion.cpp:14
- #6 0x0804845b in test (level=5) at recursion.cpp:14
- #7 0x08048479 in main () at recursion.cpp:22
- (gdb) bt -2
- #6 0x0804845b in test (level=5) at recursion.cpp:14
- #7 0x08048479 in main () at recursion.cpp:22

frame

- frame是非常有用的命令, 它可以用来显示当前帧的信息 基本语法是
 - frame
 - frame [Frame number]
 - f
- 如果没有参数, 就是当前行的信息

```
(gdb) frame
#0 level0 () at recursion.cpp:5
5 printf("Reached level 0\n");
(gdb) info args
No arguments.
(gdb) frame 1
#1 0x08048462 in test (level=0) at recursion.cpp:17
17 level0();
(gdb) info args
level = 0
(gdb) frame 2
#2 0x0804845b in test (level=1) at recursion.cpp:14
14 test(prevLevel);
(gdb) info args
level = 1
```

info frame

- ➡ 相比直接的 frame, 这个命令输出更详细的 stack frame 信息, 包括

```
(gdb) i frame
Stack level 0, frame at 0x7fffffffef250:
rip = 0x4009e0 in PrintArray (gdbprog.cc:40); saved rip 0x400a3f
called by frame at 0x7fffffffef280
source language c++.
Arglist at 0x7fffffffef240, args:
Locals at 0x7fffffffef240, Previous frame's sp is 0x7fffffffef250
Saved registers:
  rbp at 0x7fffffffef240, rip at 0x7fffffffef248
```

```
(gdb) bt
#0  PrintArray () at gdbprog.cc:40
#1  0x0000000000400a3f in main () at gdbprog.cc:57
(gdb) i f 0x0000000000400a3f
Stack frame at 0x400a3f:
rip = 0x0; saved rip 0x7ffff7ff9720
called by frame at 0x7fffffffef240
Arglist at 0x7fffffffef228, args:
Locals at 0x7fffffffef228, Previous frame's sp is 0x7fffffffef238
Saved registers:
  rip at 0x7fffffffef230
```

程序信息

- info proc
- info variables
- info functions
- info source
- info sources
- info sharedlibrary
- info files

(gdb) info functions

All defined functions:

File gdbprog.cc:

```
int DoOperation(int**);
```

```
void InitArrays(int*);
```

```
void PrintArray();
```

```
int main();
```

```
static void __static_initialization_and_destruction_0(int, int);
```

```
static void __tcf_0(void*);
```

```
static void global constructors keyed to iArray();
```

Non-debugging symbols:

```
0x0000000000400660 _init
```

```
0x0000000000400688 std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
```

```
0x0000000000400688 _ZNSolsEi@plt
```

```
0x0000000000400698 std::ios_base::Init::Init()
```

源文件

➤ list

➤ info line

➤ 查看当前运行
源代码 语法
如右：

```
list <linenum>
list <first>, <last>
list <function>
list
list -
list *<addr>
```

```
(gdb) list *$pc
0x4009e0 is in PrintArray() (gdbprog.cc:40).
35
36     void PrintArray()
37     {
38         int i;
39
40         for(i = 0;i < 10;i++)
41         {
42             if (iArray[i] == 6)
43                 cout << "This is an error\n";
44             else
```

变量查看

- 产生当前运行进程的一份memory image和它的process status
 - gcore [file]
- **info registers**
- **info variables**
 - 可以用 i var 来查看某个全局或者静态变量在哪个文件定义的
- **info locals**
 - 显示当前帧的函数的参数， 配合frame, up和down来使用
- **info args**
 - 显示当前帧的函数的参数， 配合frame, up和down来使用

总结：（1）工作环境相关命令

命令格式↵	含 义↵
set args ↵	指定运行时参数，如：set args 2↵
show args↵	查看设置好的运行参数↵
path dir↵	设定程序的运行路径↵
show paths↵	查看程序的运行路径↵
set environment var [=value]↵	设置环境变量↵
show environment [var]↵	查看环境变量↵
cd dir↵	进入到 dir 目录，相当于 shell 中的 cd 命令↵
pwd↵	显示当前工作目录↵
shell command↵	运行 shell 的 command 命令↵

总结：(2) 设置断点与恢复命令

命令格式	含 义
info b	查看所设断点
break 行号或函数名 <条件表达式>	设置断点
tbreak 行号或函数名 <条件表达式>	设置临时断点，到达后被自动删除
delete [断点号]	删除指定断点，其断点号为" info b" 中的第一栏。 若缺省断点号则删除所有断点
disable [断点号]	停止指定断点，使用" info b" 仍能查看此断点。 同 delete 一样，省断点号则停止所有断点
enable [断点号]	激活指定断点，即激活被 disable 停止的断点
condition [断点号] <条件表达式>	修改对应断点的条件
ignore [断点号] <num>	在程序执行中，忽略对应断点 num 次
step	单步恢复程序运行，且进入函数调用
next	单步恢复程序运行，但不进入函数调用
finish	运行程序，直到当前函数完成返回
c	继续执行函数，直到函数结束或遇到新的断点

总结：（3）gdb中源码查看相关命令

命令格式	含 义
list <行号> <函数名>	查看指定位置代码
file [文件名]	加载指定文件
forward-search 正则表达式	源代码前向搜索
reverse-search 正则表达式	源代码后向搜索
dir dir	停止路径名
show directories	显示定义了的源文件搜索路径
info line	显示加载到 gdb 内存中的代码

总结：（4）gdb中查看运行数据命令

命令格式	含义
print 表达式 变量	查看程序运行时对应表达式和变量的值
x <n/f/u>	查看内存变量内容。其中 n 为整数表示显示内存的长度，f 表示显示的格式，u 表示从当前地址往后请求显示的字节数
display 表达式	设定在单步运行或其他情况中，自动显示的对应表达式的内容

总结： （5）其他gdb命令

run命令：执行当前被调试的程序。

kill命令：停止正在调试的应用程序。

watch命令：设置监视点，监视表达式的变化。

awatch命令：设置读写监视点。当要监视的表达式被读或写时将应用程序挂起。它的语法与watch命令相同。

rwatch命令：设置读监视点，当监视表达式被读时将程序挂起，等待调试。此命令的语法与watch相同。



info break命令：显示当前断点列表，包括每个断点到达的次数。

info files命令：显示调试文件的信息。

info func命令：显示所有的函数名。

info local命令：显示当前函数的所有局部变量的信息。

info prog命令：显示调试程序的执行状态。

Shell命令：执行Linux Shell命令。

make命令：不退出gdb而重新编译生成可执行文件。

quit命令：退出gdb。

(6) gdb中修改运行参数相关命令

`gdb`可修改运行时的参数，并使该变量按照用户当前输入的值继续运行。

方法为：在单步执行的过程中，键入命令：

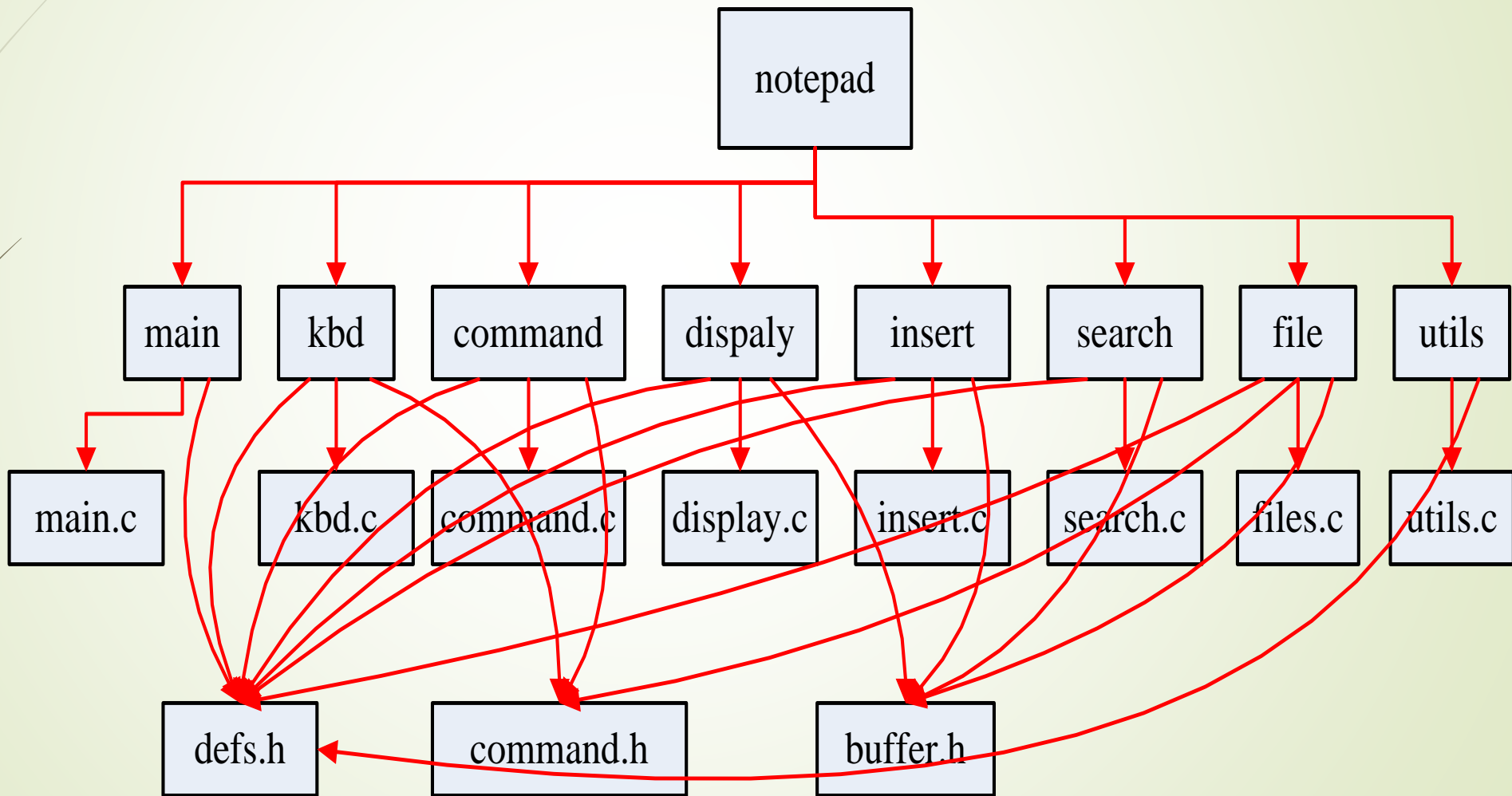
`set 变量=设定值`

程序就按照该设定的值运行。

特别注意，在gcc编译选项中一定要加入“-g”。只有在代码处于“运行”或“暂停”状态时才能查看变量值，设置断点后程序在指定行之前停止。

2.3 make工具

记录依赖关系并识别局部更新



make工具

make是一个自动化的程序自动维护工具。

它根据Makefile所描述的“依赖关系”自动决定项目的那些部分需要重新编译。

基本原理：

如果某个源程序文件被修改，那么依赖这个源程序文件的所有目标文件，都需要重新编译。

包括：

- 1) 如果仅修改了某几个 源文件，则只重新编译这几个源文件；
- 2) 如果某个头文件被修改了，则重新编译所有包含该头文件的源文件。

2.3.1 make工作原理

GNU的make工作时的执行步骤如下：

1. 读入所有的Makefile。
2. 读入被include的其它Makefile。
3. 初始化文件中的变量。
4. 推导隐式规则，并分析所有规则。
5. 为所有的目标文件创建依赖关系链。
6. 根据依赖关系，决定哪些目标要重新生成。
7. 执行生成命令。

2.3.2 Makefile文件

- **Makefile**文件控制**make**程序的执行
- 一个项目拥有一个或多个**Makefile**文件
- 每个**Makefile**文件由多条**rules**构成
- 每条**rule**描述了一个依赖关系，并有一系列的行为

makefile 文件的内容

- Makefile里主要包含了五个东西：
 - 显式规则
 - 隐式规则
 - 条件指令
 - 变量定义
 - 注释

Makefile文件内容

- 显式规则。显式规则说明了，如何生成一个或多个的目标文件。这是由Makefile的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。
- 隐式规则。由于make有自动推导的功能，所以隐式的规则可以让我们比较粗糙地简略地书写Makefile，这是由make所支持的。
- 变量的定义。在Makefile中我们要定义一系列的变量，变量一般都是字符串，这个有点你C语言中的宏，当Makefile被执行时，其中的变量都会被扩展到相应的引用位置上。
- 文件指示。其包括了三个部分，一个是在一个Makefile中引用另一个Makefile，就像C语言中的include一样；另一个是指根据某些情况指定Makefile中的有效部分，就像C语言中的预编译#if一样；还有就是定义一个多行的命令。
- 注释。Makefile中只有行注释，和UNIX的Shell脚本一样，其注释是用“#”字符，这个就像C/C++中的“//”一样。如果你要在你的Makefile中使用“#”字符，可以用反斜框进行转义，如：“\#”。
- 最后，还值得一提的是，在Makefile中的命令，必须要以[Tab]键开始。

Makefile文件的显式规则


Makefile规则格式:

target	:	prerequisites	依赖关系
<TAB>		command	命令

- 目标，即make最终需要创建的文件，如可执行文件和目标文件；目标也可以是要执行的动作，如clean。
- 一个或多个依赖文件（dependency）列表，通常是编译目标文件所需要的其他文件。
- 一系列命令(command)，是make执行的动作，通常是把指定的相关文件编译成目标文件的编译命令，每个命令占一行，且每个命令行起始字符必须为TAB字符。



#Makefile



```
main:main.o hello1.o hello2.o
    gcc -o main main.o hello1.o hello2.o
main.o:main.c hello1.h hello 2.h
    gcc -c main.c
hello 1.o: hello 1.c hello 1.h
    gcc -c hello1.c
hello2.o: hello2.c hello2.h
    gcc -c hello2.c
clean:
    rm main hello1.o hello2.o main.o
```


make的执行

- 执行make命令: `make target`
- `target`是Makefile文件中定义的目标之一，若省略`target`，`make`就将生成Makefile文件中定义的第一个目标。
- `make` 等价于 `make main`
- 因为`main`是Makefile文件中定义的第一个目标，`make`首先将其读入，然后从第一行开始执行，把第一个目标`test`作为它的最终目标，所有后面的目标的更新都会影响到`main`的更新



➤ 伪目标

➤ 伪目标又称假想目标，如：


➤ `clean:`

➤ `rm *.o temp`

➤ 这里并不生成“clean”这个文件。“伪目标”并不是一个文件，只是一个标签，由于“伪目标”不是文件，所以make无法生成它的依赖关系和决定它是否要执行。

➤ 可使用“`make clean`”来使用该目标。

➤ 如果你的Makefile需要生成若干个可执行文件，可把所有的目标文件都写在一个Makefile中，可声明了一个“all”的伪目标。



声明了一个“all”的伪目标

例如：

all : prog1 prog2 prog3

prog1 : prog1.o utils.o

gcc -o prog1 prog1.o utils.o

prog2 : prog2.o

gcc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o

gcc -o prog3 prog3.o sort.o utils.o

隐式规则

- 根据后缀为.c的源文件使用gcc命令自动更新或者产生同名的.o文件
- 规则中略去产生目标文件的命令。同时，目标所依赖的文件列表中的.c文件亦可省略
- hello2.o: hello2.c hello2.h
- gcc -c hello2.c
- 可以简化为：
- hello2.o:hello2.h

条件指令

- 条件表达式的语法为：
 - `<conditional-directive>`
 - `<text-if-true>`
 - `endif`
 - 以及：
 - `<conditional-directive>`
 - `<text-if-true>`
 - `Else`
 - `<text-if-false>`
 - `Endif`
- 其中`<conditional-directive>`表示条件关键字，如“`ifeq`”。这个关键字有四个。

ifeq

```
ifeq (<arg1>, <arg2> )
```

```
ifeq '<arg1>' '<arg2>'
```

```
ifeq "<arg1>" "<arg2>"
```

```
ifeq "<arg1>" '<arg2>'
```

```
ifeq '<arg1>' "<arg2>"
```

比较参数 “arg1”和 “arg2”的值是否相同。

当然，参数中还可以使用make的函数。

如：

```
ifeq ($(strip $(foo)),)
```

```
<text-if-empty>
```

```
endif
```

这个示例中使用了 “strip”函数，如果这个函数的返回值是空（Empty），那么<text-if-empty>就生效。



ifneq

- `ifneq (<arg1>, <arg2>)`
- `ifneq '<arg1>' '<arg2>'`
- `ifneq "<arg1>" "<arg2>"`
- `ifneq "<arg1>" '<arg2>'`
- `ifneq '<arg1>' "<arg2>"`
- 其比较参数“arg1”和“arg2”的值是否相同，如果不同，则为真。
- 和“ifeq”类似。

ifdef

ifdef <variable-name>

如果变量<variable-name>的值非空，那到表达式为真。否则，表达式为假。当然，<variable-name>同样可以是一个函数的返回值。

注意，ifdef只是测试一个变量是否有值，其并不会把变量扩展到当前位置。

还是来看两个例子：

示例一：

```
bar =
```

```
foo = $(bar)
```

```
ifdef foo
```

```
    frobozz = yes
```

```
else
```

```
    frobozz = no
```

```
endif
```

“\$(frobozz)”值是 “yes”

ifndef

- `ifndef <variable-name>`
和 “`ifdef`” 是相反的意思。
- 在 `<conditional-directive>` 这一行上，多余的空格是被允许的，但是不能以 `[Tab]` 键做为开始（不然就被认为是命令）。
- 而注释符 “`#`” 同样也是安全的。“`else`” 和 “`endif`” 也一样，只要不是以 `[Tab]` 键开始就行了。
- 特别注意的是，`make` 是在读取 `Makefile` 时就计算条件表达式的值，并根据条件表达式的值来选择语句，所以，最好不要把自动化变量（如 “`$@`” 等）放入条件表达式中，因为自动化变量是在运行时才有的。
- 而且，为了避免混乱，`make` 不允许把整个条件语句分成两部分放在不同的文件中。

Makefile中的变量

环境变量：

- (1) 存储文件名列表。
- (2) 存储可执行文件名。
- (3) 存储编译器标识。
- (4) 存储参数列表。

使用变量的例子

- ▀ objects = main.o kbd.o command.o display.o \
- ▀ insert.o search.o files.o utils.o
- ▀ edit : \$(objects)
 - ▀ cc -o edit \$(objects)
- ▀ main.o : main.c defs.h
 - ▀ cc -c main.c
- ▀ kbd.o : kbd.c defs.h command.h
 - ▀ cc -c kbd.c
- ▀
- ▀ clean :
rm edit \$(objects)

➡ Makefile变量

内部变量：


`$@`：当前规则的目标文件名

`$<`：成依赖列表中第一个依赖文件

`$?`：比目标文件更新的以空格分隔的整个依赖的列表

`$^`：以空格分隔的所有的依赖文件

`$+`：与`$^`功能相似，但包含有重复的依赖文件

- 
- `obj= main.o hello1.o hello2.o`
 - `main: $(obj)`
 - `gcc -o main $(obj)`

自动变量简化后的Makefile

```
main:main.o hello1.o hello2.o
```

```
    gcc -o $@ $^
```

```
main.o:main.c hello1.h hello 2.h
```

```
    gcc -c $<
```

```
hello 1.o: hello 1.c hello 1.h
```


```
    gcc -c $<
```

```
hello2.o: hello2.c hello2.h
```

```
    gcc -c $<
```

```
clean:
```

```
    rm main hello1.o hello2.o main.o
```



思考题

- 简述静态链接和动态链接的相同点和不同点
- 简述GCC编译器的工作流程
- 跟用printf函数打印输出相比，采用gdb调试的优点有哪些？
- Make工具是如何知道哪些文件需要重新生成，哪些不需要的？