杨亮

GNU

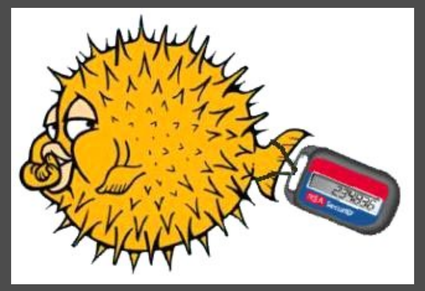
Topics

- GNU & Linux
- Linux下编程的流程
- GCC
- GDB
- Makefile
- 文件读写
- 文件操作

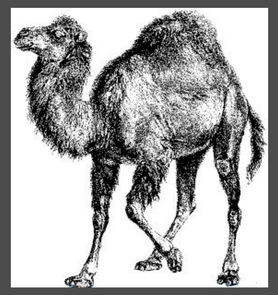
Richard Stallman

GNU/GNU

James Gosling

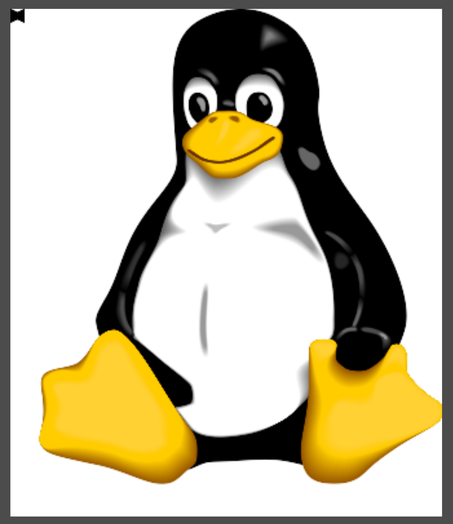


BlowFish/ openssh



Perl

Daemon/BSD



Tux/Linux

Duke/Java

Dino/Mozilla

Thunderbird

Richard Stallman

GNU's Not Unix

GNU

创建一套完全自由的操作系统 (1983.9.27)

GNU Guh-NOO /gəˈnuː/ 革奴

英汉翻译

gnu [nju:]

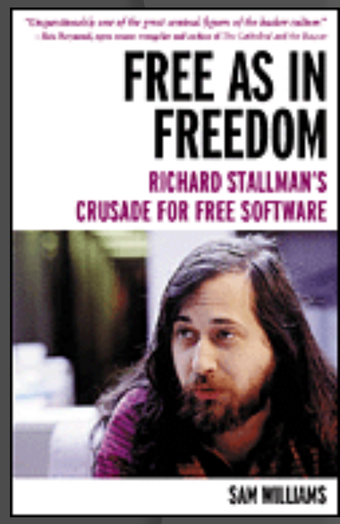
n. 角马；牛羚

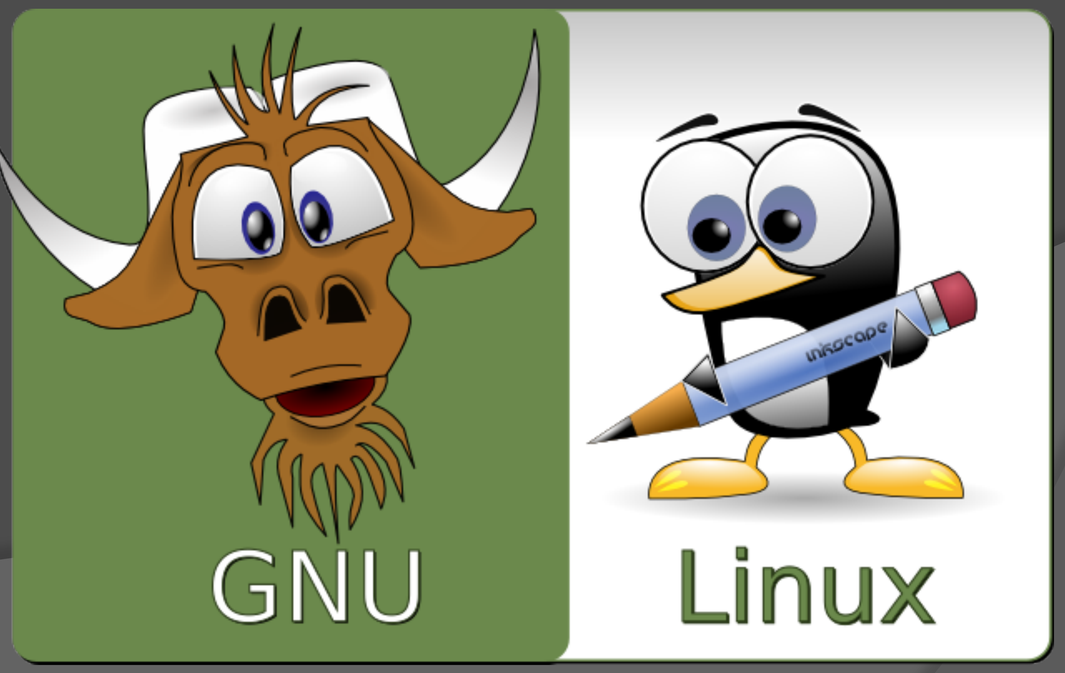
[复数gnus或gun]

GPL

- 为保证GNU软件可以自由地“使用、复制、修改和发布”，所有GNU软件都有一份在禁止其他人添加任何限制的情况下授权所有权利给任何人的协议条款
- GNU通用公共许可证（GNU General Public License，GPL）。
- 这个就是被称为“反版权”(或称Copyleft)的概念。

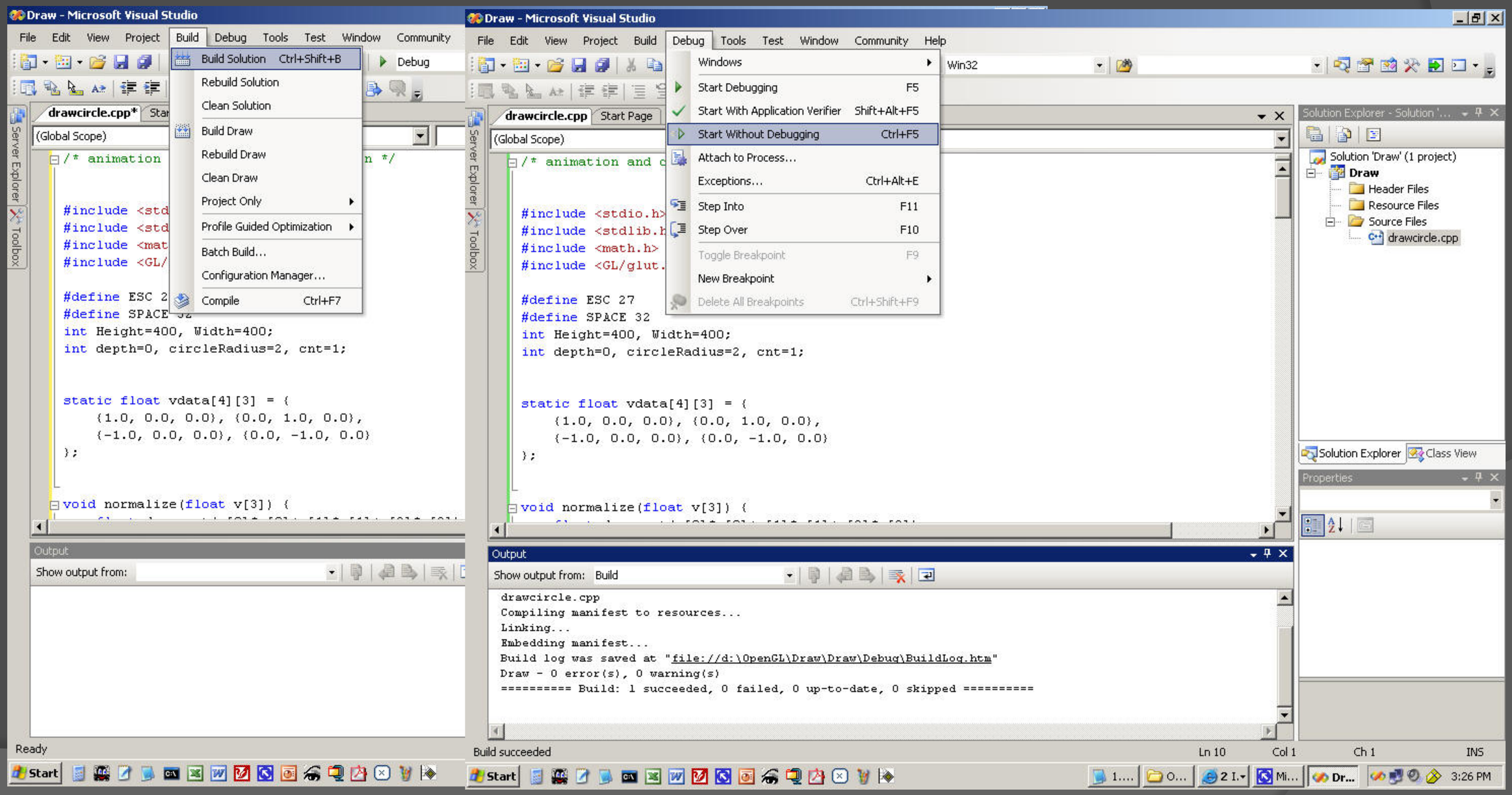
1983.9 GNU计划
1985 自由软件基金会
1990 GNU包括Emacs 、GCC、Shell、Unix工具，但是没有完成内核HURD
1991 Linux以GPL协议发布
1992 Linux与其他GNU软件结合

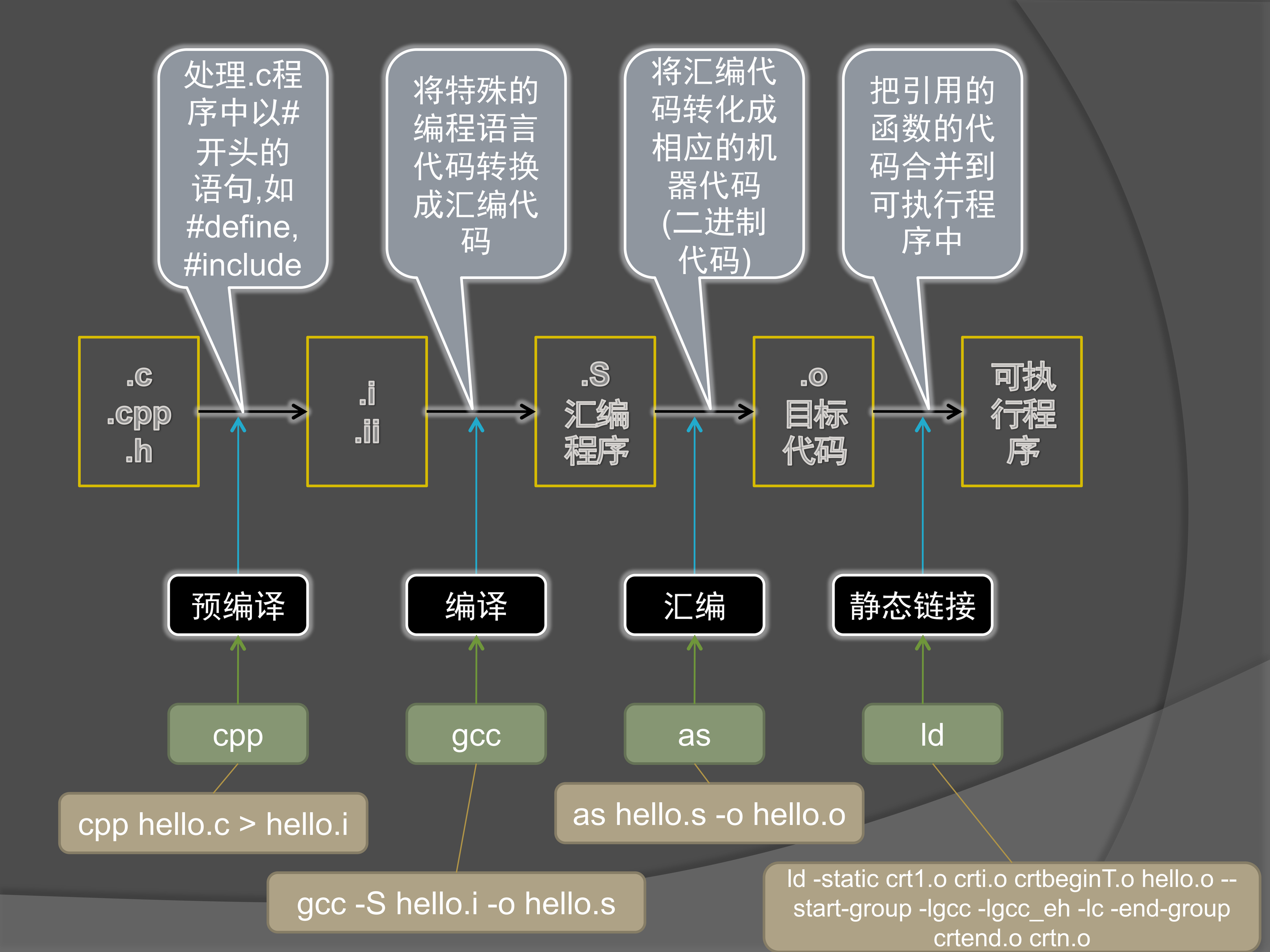


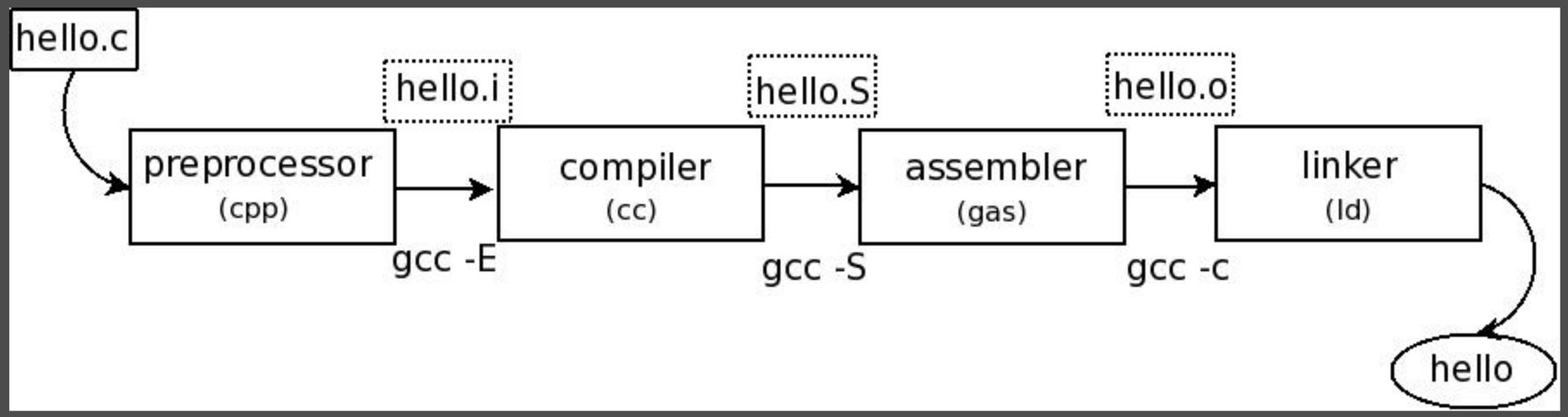


编译程序

- Visual C++中

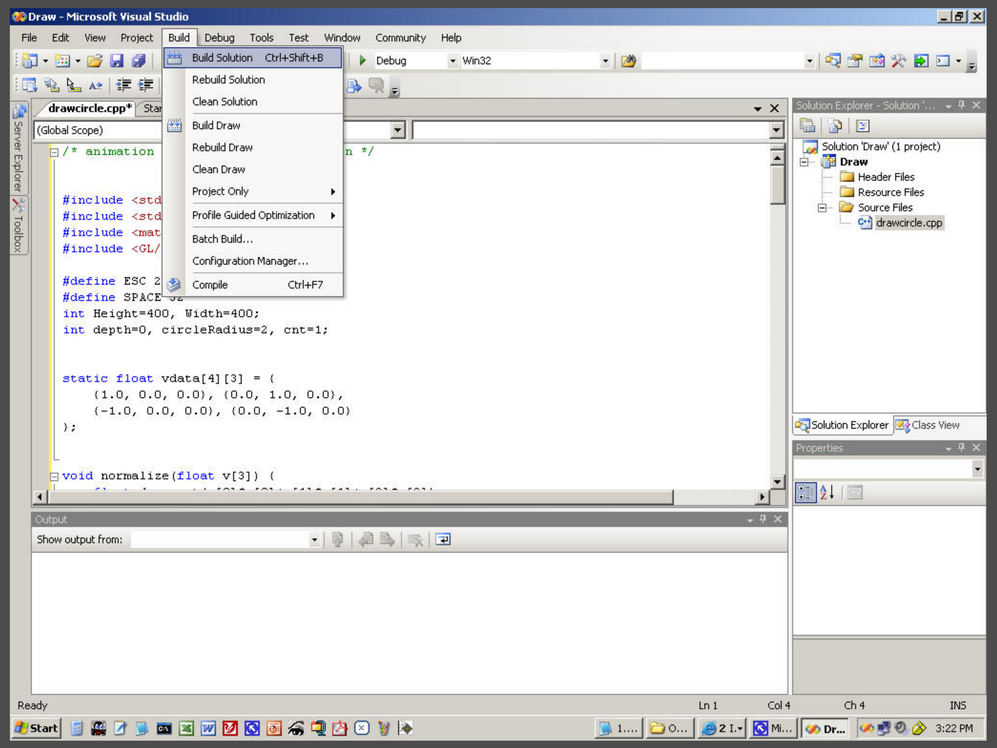


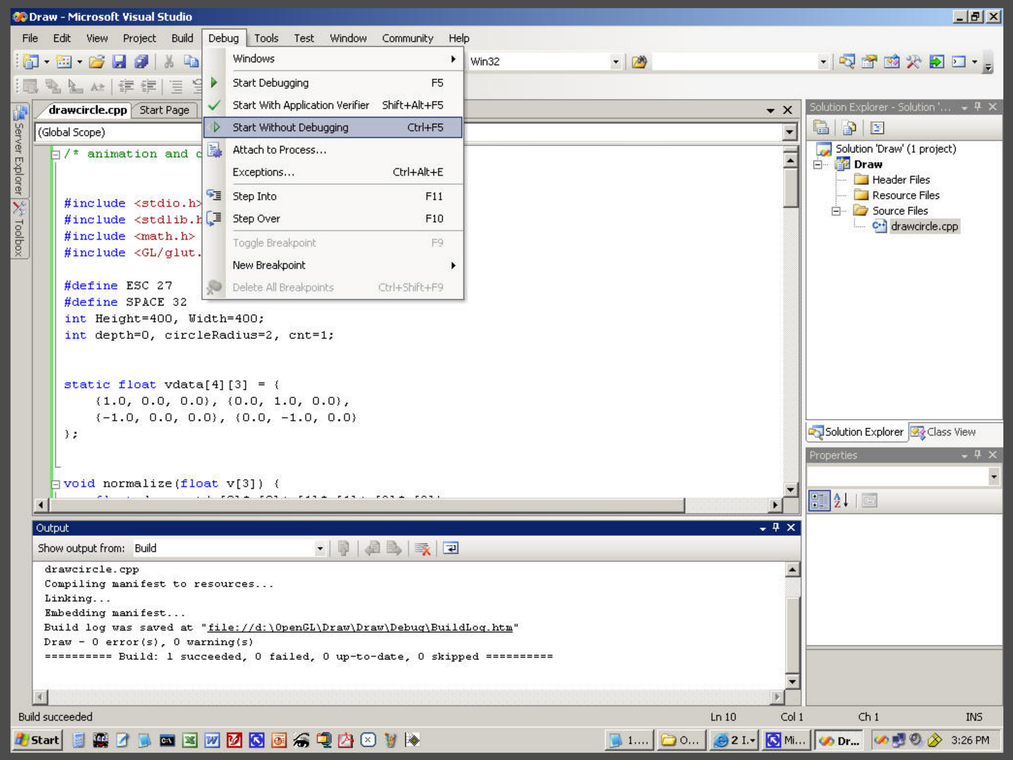

处理.c程序中以#开头的语句,如#define, #include
将特殊的编程语言代码转换成汇编代码
将汇编代码转化成相应的机器代码(二进制代码)
把引用的函数的代码合并到可执行程序中
.c
.cpp
.h
.i
.ii
.s
汇编程序
.o
目标代码
可执行程序
预编译
编译
汇编
静态链接
cpp
gcc
as
ld
cpp hello.c > hello.i
gcc -S hello.i -o hello.s
as hello.s -o hello.o
ld -static crt1.o crti.o crtbeginT.o hello.o --start-group -lgcc -lgcc_eh -lc -end-group crtend.o crtn.o


hello.c
hello.i
hello.S
hello.o
preprocessor
(cpp)
compiler
(cc)
assembler
(gas)
linker
(ld)
gcc -E
gcc -S
gcc -c
hello

怎么这么麻烦，难怪没什么人用

其实也可以很简单






```
$gcc hello.c -o hello
$./hello
```

涉及的文本文件格式

hello.c

```c
#include <stdio.h>

int main () {
        printf("hello world!\n");
        return 0;
}
```

Vi

hello.i

```c
extern char *ctermid (char *__s) __attribute__ ((__nothrow__));
# 908 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__));



extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__)) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__));
# 938 "/usr/include/stdio.h" 3 4

# 2 "hello.c" 2

int main () {
 printf("hello world!\n");
 return 0;
}
```

hello.s

```asm
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "hello world!"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    puts
        movl    $0, %eax
        leave
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.4.4 20100630 (Red Hat 4.4.4-10)"
        .section        .note.GNU-stack,"",@progbits
~
~
```

```
ELF GCC: (GNU) 4.4.4 20100630 (Red Hat 4.4.4-10) .symtab .strtab .shstrtab .rel.text .data .bss .rodata .comment .note.GNU-stack
hello world!
hello.c main puts
```

vi hello.o

```
[yangliang@fedora ~]$ objdump hello.o  -s -d

hello.o:     file format elf32-i386

Contents of section .text:
 0000 5589e583 e4f083ec 10c70424 00000000  U..........$....
 0010 e8fcffff ffb80000 0000c9c3           ............
Contents of section .rodata:
 0000 68656c6c 6f20776f 726c6421 00        hello world!.
Contents of section .comment:
 0000 00474343 3a202847 4e552920 342e342e  .GCC: (GNU) 4.4.
 0010 34203230 31303036 33302028 52656420  4 20100630 (Red
 0020 48617420 342e342e 342d3130 2900      Hat 4.4.4-10).

Disassembly of section .text:

00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 e4 f0                and    $0xfffffff0,%esp
   6:   83 ec 10                sub    $0x10,%esp
   9:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  10:   e8 fc ff ff ff          call   11 <main+0x11>
  15:   b8 00 00 00 00          mov    $0x0,%eax
  1a:   c9                      leave
  1b:   c3                      ret
```

objdump -sd hello.o

代码段

二进制代码

可显示的文本

注释段，用于调试使用

对代码段的反汇编

你们学了C可能不知道的事情

为什么要有头文件？

头文件里应该放些什么

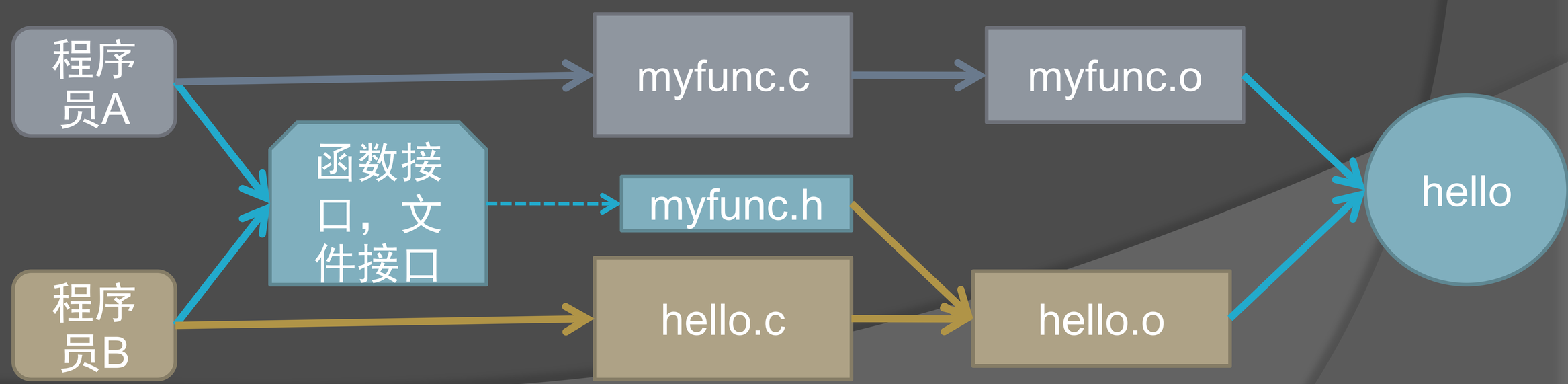
为什么要有声明？

声明

```
int myfunc(int num)
```

定义

```
int myfunc(int num) {
    printf("hello world");
}
```

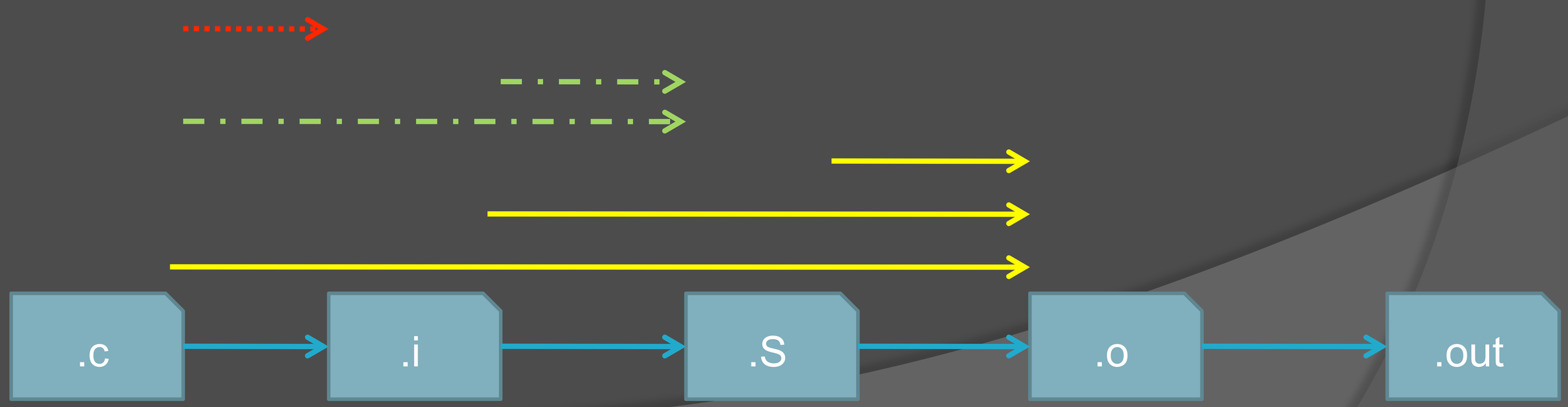


GCC

如果不增加之三个参数则可以直接生成可执行文件

- GNU project C and C++ compiler
- 基本选项
  - -E 预编译 （从.c到.i）
  - -S 编译（可从.c或.i编译为.s）
  - -c 整体编译或者汇编 （可从.c .i .s生成.o）

如果要制定输处文件名 用 -o（小o）选项



一些常用选项

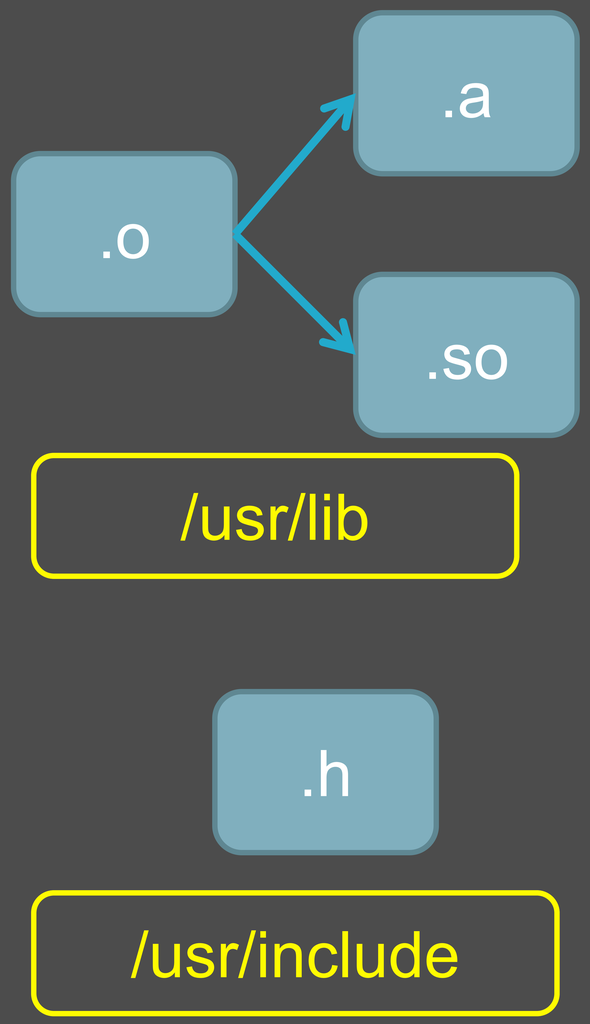
这些不算多，而且是应该记住的，不信man gcc一下，一共有13794行帮助信息

- -o 输出到相应的文件
- -I *dir* 把dir增加到搜索头文件的目录中 （include）
- -L *dir* 把dir增加达到搜索库文件的路径中 （library）
- -l*library* 链接名为liblibrary的库，例如lm
- -g*level*生成标准的调试信息
- -ggdb*level*生成转为gdb有用的调试信息
- -p 增加对程序性能进行监控的profile内容
- -O*level* 编译器进行优化的级别 O0(欧零)不进行优化
- -w 关闭所有警告 (警告也是一种找出错误的方法)
- -W打印一些额外的警告信息
- -Wall 输出一些很有用的警告
- -Werror 把所有警告当做错误

```
NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [-c|-S|-E] [-std=standard]
           [-g] [-pg] [-Olevel]
           [-Wwarn...] [-pedantic]
           [-Idir...] [-Ldir...]
           [-Dmacro[=defn]...] [-Umacro]
           [-foption...] [-mmachine-option...]
           [-o outfile] [@file] infile...
```

- 如果是自己的库或者是头文件可能不在这两个相应的文件夹下，那么在链接的时候就要制定
- 有些动态链接库并不是默认加载的，例如math数学库libm.so，这个时候我们就要使用 -lm




```
libFLAC++.so.6                libjasper.so.1.0.0
libFLAC++.so.6.2.0            libjpeg.so.62
libFLAC.so.8                  libjpeg.so.62.0.0
libFLAC.so.8.2.0              libjson-glib-1.0.so.0
libfontconfig.so.1            libjson-glib-1.0.so.0.1001.3
libfontconfig.so.1.4.4        libkadm5clnt.so.6
libfontenc.so.1               libkadm5clnt.so.6.0
libfontenc.so.1.0.0           libkadm5srv.so.6
libform.so.5                  libkadm5srv.so.6.0
libform.so.5.7                libkate.so.1
libformw.so.5                 libkate.so.1.2.0
libformw.so.5.7               libkdb5.so.4
libfprint.so.0                libkdb5.so.4.0
libfprint.so.0.0.0            libkms.so.1
```

```
libio.h     netash      obstack.h   rpc         stdio_ext.h
limits.h    netatalk    paths.h     rpcsvc      stdio.h
link.h      netax25     poll.h      sched.h     stdlib.h
linux       netdb.h     printf.h    scsi        strata_client.h
locale.h    neteconet   protocols   search.h    string.h
malloc.h    netinet     pthread.h   semaphore.h strings.h
math.h      netipx      pty.h       setjmp.h    sys
mcheck.h    netiucv     pwd.h       sgtty.h     syscall.h
memory.h    netpacket   python2.6   shadow.h    sysexits.h
mntent.h    netrom      rdma        signal.h    syslog.h
monetary.h  netrose     re_comp.h   sound       tar.h
```

```
gcc myapp.c -L/home/fred/lib -I/home/fred/include -lnew -o myapp
```

警告和出错消息

编译时的错误是最容易修改的错误，因为你知道可能哪里错了

```c
#include <stdio.h>

void main() {
        long long int i,j;
        if (i=10)
                printf("This is a non-conforming C program\n");
}
```

```
[yangliang@fedora ~]$ gcc -Wall pedant.c -o pedant
pedant.c:3: 警告：‘main’的返回类型不是‘int’
pedant.c: 在函数‘main’中：
pedant.c:5: 警告：建议在用作真值的赋值语句前后加上括号
pedant.c:4: 警告：未使用的变量‘j’
```

优化和调试选项

在调试程序的时候不要进行优化，因为优化会改变原来的代码的结构，当调试没有问题了在加上优化选项

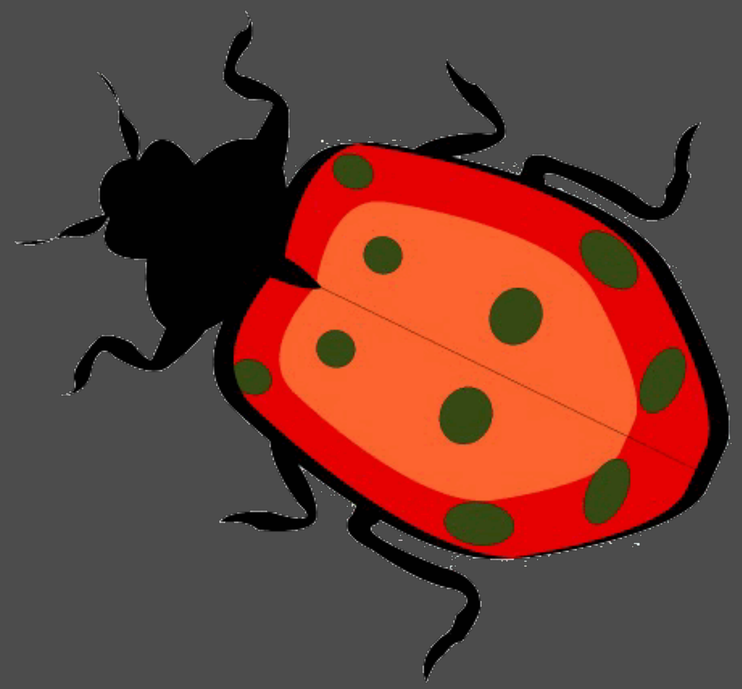
- 优化选项
  - -O1
  - -O2
  - -O3

作业：
写程序说明两个1000*1000的矩阵相乘，采用不用的优化参数的效果

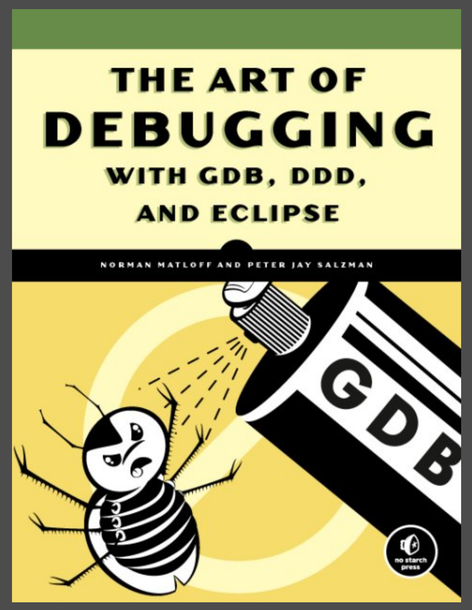
- 调试选项
  - -g*level*，-g = -g2
  - 包含符号表（函数名、变量名），行号信息，局部和外部变量名
  - -p 增加profile信息，用于找出系统的瓶颈

如果不增加调试选项，普通的程序是没有符号表和行号这些内容的，因为对于可执行程序这些都没有意义

GDB



- Bug
- Debug
- GDB=GNU Debugger



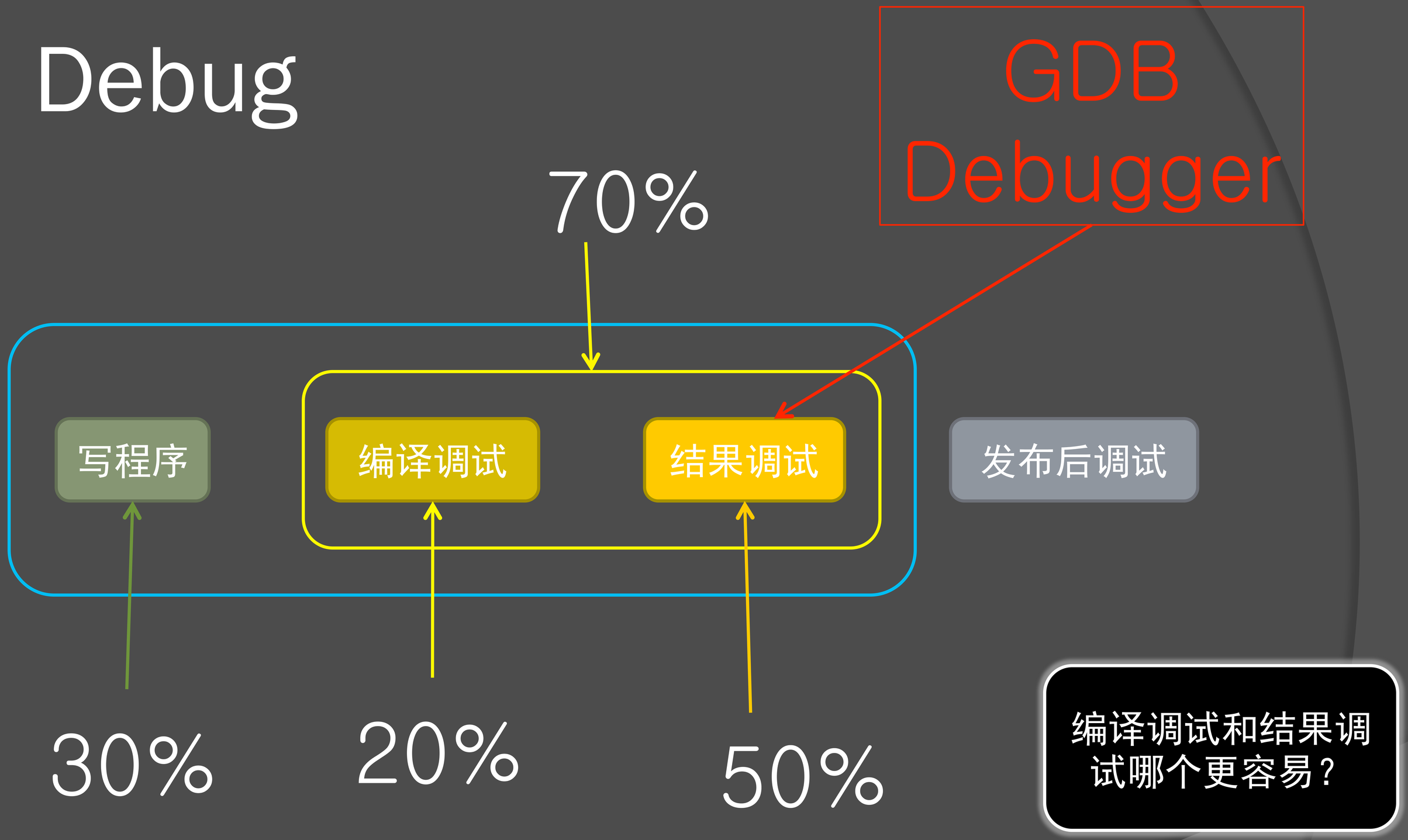
Baidu的要求

Baidu 百度

Php 每千行<5个

C++ 每千行<2.5个

Debug



调试的基本原则

为什么程序会有错误？

我们总总是会没有依据的做出假设，有些事正确的，有些事错误的

我认为
假设

事实上
结果

惊讶是件好事

确认原则

printf

增加、编译、测试、去除。。

珍爱声明，原理printf

用debugger，编译一次，监控不同变量值的改变、程序的流程、函数返回值，指针问题，找到一个或者多个错误

调试的其他原则

- 从简单工作开始
- 用调试工具确定段错误的地方
- 通过发生中断的确定无限循环的位置
- 使用二分搜索

结果调试

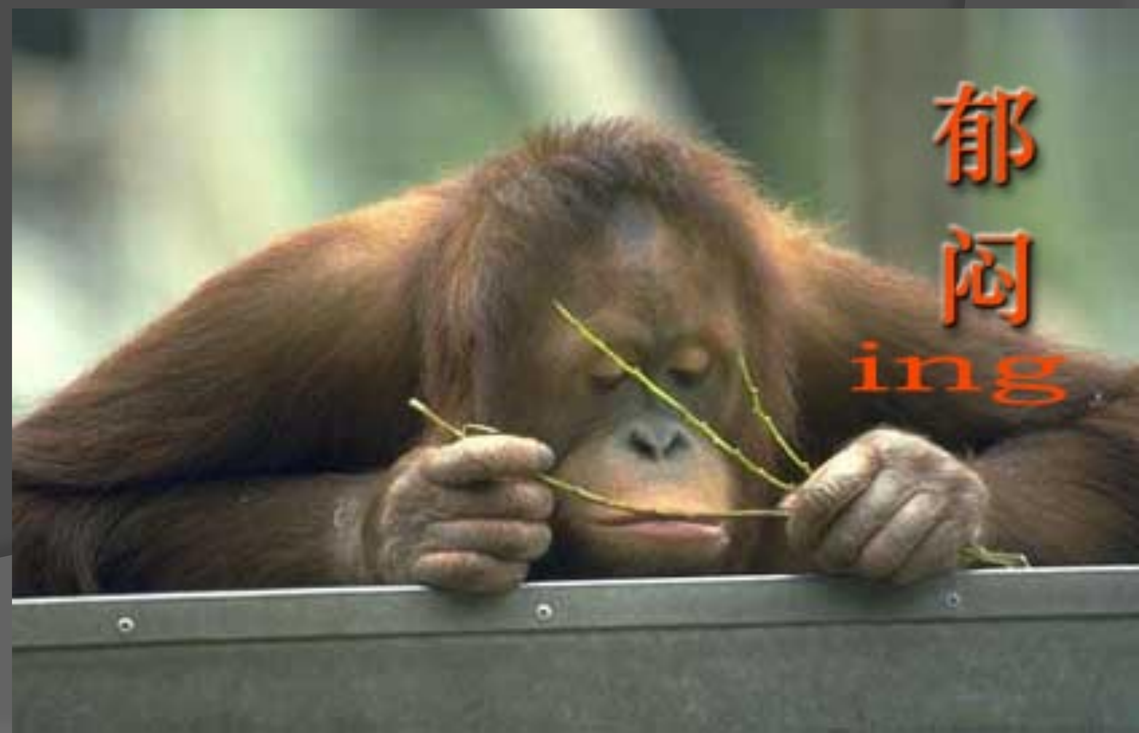
程序逻辑有问题，运算结果不对

业务逻辑或者算法可能有问题

程序莫名奇妙的死掉，连输出都没有

系统出Core了
系统又down了

你有没有遇到过程序无缘无故的死掉？



我们来看个程序

Core Dump 在程序异常终止的时候，操作系统就会把程序down 时的内存内容 dump 出来（现在通常是写在一个叫 core 的 file 里面），让 我们或是 debugger 做为参考。这个动作就叫作 core dump。

```c
#include <stdio.h>
int a() {
        char *p = NULL;
        printf("%d\n",*p);
        return 0;
}

int main() {
        a();
        return 0;
}
```

在 C/C++语言中，最常发生错误的地方就是指 针有问题

```
[yangliang@fedora testcore]$ gcc test.c -g -o test
[yangliang@fedora testcore]$ ./test
段错误 (core dumped)
```

```
[yangliang@fedora testcore]$ ls -l
总用量 80
-rw-------. 1 yangliang yangliang 151552  8月 19 08:34 core.2099
-rwxrwxr-x. 1 yangliang yangliang   5950  8月 19 08:34 test
-rw-rw-r--. 1 yangliang yangliang    113  8月 19 08:28 test.c
```

如果你的程序Down掉了，但是没有core文件，那么可以尝试一下的命令ulimit -c unlimited

我们可以用gdb知道是哪里出的问题

```
[yangliang@fedora testcore]$ gdb test core.2126
GNU gdb (GDB) Fedora (7.1-32.fc13)
Copyright (C) 2010 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 "i686-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/yangliang/testcore/test...done.
[New Thread 2126]
Missing separate debuginfo for
Try: yum --disablerepo='*' --enablerepo='*-debuginfo' install /usr/lib/debug/.build-id/9f/ffb5bf01028a997d2ab7
Reading symbols from /lib/libc.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib/ld-linux.so.2
Core was generated by `./test'.
Program terminated with signal 11, Segmentation fault.
#0  0x080483d4 in a () at test.c:4
4               printf("%d\n",*p);
Missing separate debuginfos, use: debuginfo-install glibc-2.12-3.i686
```

一些关于调试的概念

- 断点、条件断点
- 监视点
- 单步调试

断点

- break 设置断点
  - break filename:linenum if expr
  - break filename:funcname if expr
  - filename: 可以省略
  - 省略if expr是普通断点
  - if expr是条件断点，只有满足条件的时候才停止
- info breakpoints 查看断点列表
- 把设置断点中的break 换成clear则可以删除相应的断点
- 每个断点有一个唯一的id，用delete id_list也可以删除相应的断点
- disable id_list 禁用断点
- enable [once] id_list 启用断点

监视点

- watch var
- watch express
- 当var或者express的值改变的时候停止

gdb

- gdb prog core
  - run 运行程序直到遇到断点或者发生中断
  - next执行下一条语句，单步调试，执行函数，把整个函数作为一条语句
  - next n 连续执行n步
  - step执行下一条语句，单步进入，执行函数内的第一条
  - continue 仅需执行
  - continue n 运行时跳过n个断点
  - until 完成循环执行并停止
  - until …. 与break的参数一致，只是运行到那个位置
  - list 列出程序停止位置前后的代码

变量访问

- whatis varname(funcname) 显示变量（函数）的类型
- ptype structvar 如果变量时struct，则显示struct
- print varname 查看变量的值
  - $i 被查数值的历史记录
  - print $i-1 可以对$i进行运算
  - print ary[i]@j 数据从第i个元素开始的j个元素值
- set varname = value 修改变量的值
- set args arg1 g2ar…. 设置程序所需参数

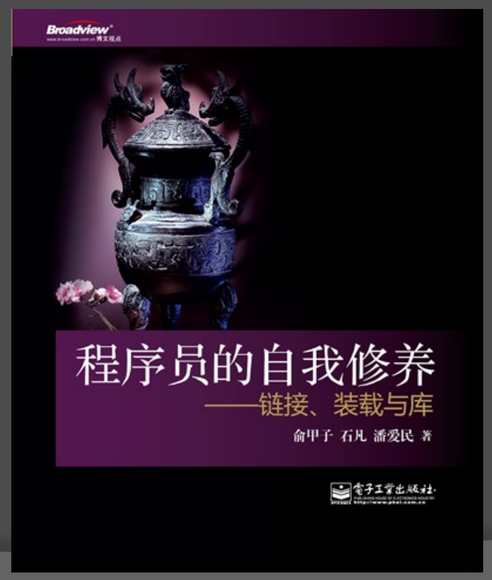
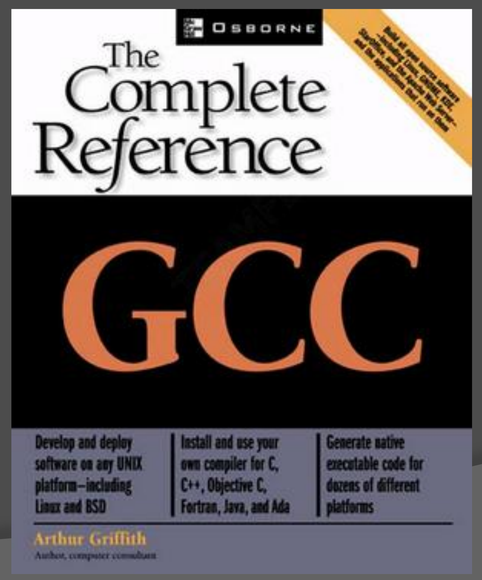
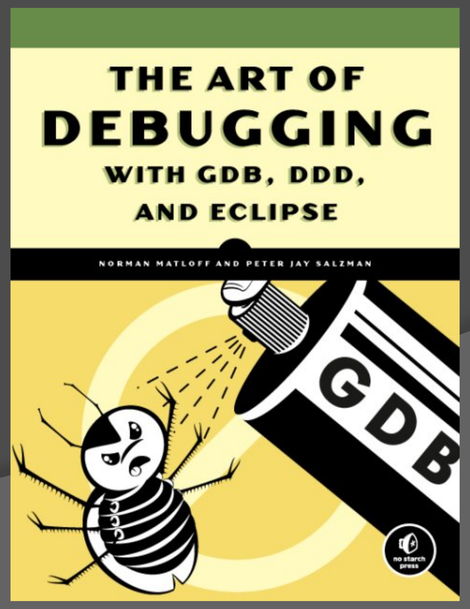
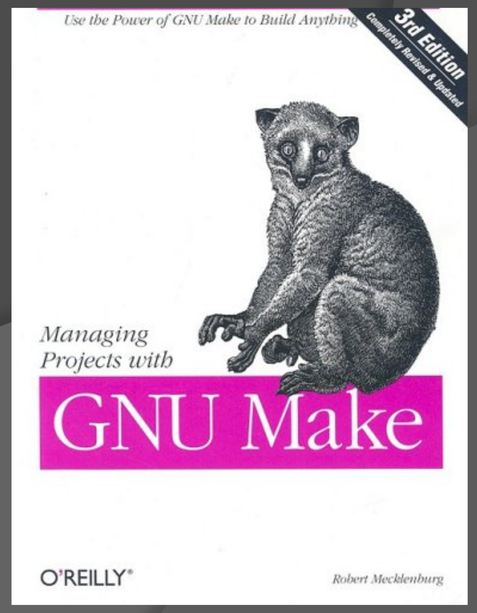
断点命令

- 达到同一断点是总要进行某些固定的操作
- commands breakpoint-num
  command
  ……
  end
- 例 break fibonacci
  command 1
  > printf “fibonacci was passed %d ”,n
  >

- 函数
  - call funcname(args) 调用某一个函数
  - finish 运行完此函数，忽略其中的断点
  - return value 不在执行此函数，但是把value作为函数的返回值返回
  - backtrace列出函数调用堆栈

其他调试工具

makefile

Reference

- GNU/linux 编程指南（第二版）
- 程序员自我修养
- **GCC – The Complete Reference**
- **The Art of Debugging with GDB, DDD, and Eclipse**
- Managing Projects with GNU make, 3rd Edition