第一课　开发环境

TIOBE(世界计算机语言排名)
1 - C
2 - Java
3 - Objective C
4 - C++

C/C++/数据结构和算法 - 平台无关，算法逻辑
UC/Win32/Android/iOS - 平台相关，系统调用
嵌入式/驱动程序/移植 - 硬件相关，硬件接口

一、课程内容

开发环境 - 1天 -+
内存管理 - 1天　|
文件系统 - 2天　|
进程管理 - 1天　|
信号处理 - 1天　+- 共10天
进程通信 - 1天　|
网络通信 - 1天　|
线程管理 - 1天　|
线程同步 - 1天 -+

二、Unix操作系统

1. 简介

美国AT&T公司贝尔实验室，
1971年，
肯.汤普逊、丹尼斯.里奇。

PDP-11，多用户、多任务、支持多种处理器架构。

高安全性、高可靠性，高稳定性。

既可构建大型关键业务系统的商业服务器应用，
也可构建面向移动终端、手持设备等的嵌入式应用。

图示：pdp-11.jpg

2. 三大派生版本

1) System V

AIX: IBM，银行
Solaris: SUN->Oracle，电信
HP-UX
IRIX

2) Berkley

FreeBSD
NetBSD
OpenBSD
Mac OS X

3) Hybrid

Minix: 迷你版的类Unix操作系统。

Linux: GPL，免费开源，商用服务器(RedHat)、
桌面(Ubuntu)、嵌入式(Android)。

3. Unix族谱

图示：unix_history.png

三、Linux操作系统

1. 简介

类Unix操作系统，免费开源。

不同发行版本使用相同内核。

手机、平板电脑、路由器、视频游戏控制台、台式计算机、
大型计算机、超级计算机。

严格意义上的Linux仅指操作系统内核。

隶属于GNU工程。

发明人Linus Torvalds。

图示：linus.jpg

2. 标志

Tux (Tuxedo，一只企鹅)

图示：tux.png

3. 相关知识

1) Minix操作系统

荷兰阿姆斯特丹Vrije大学，
数学与计算机科学系，
Andrew S. Tanenbaum，
ACM和IEEE的资深会员。

2) GNU工程

Richard Stallman发起于1984年，
由自由软件基金会(FSF)提供支持。

GNU的基本原则就是共享，
其主旨在于发展一个有别于一切商业Unix系统的，
免费且完整的类Unix系统——GNU Not Unix。

3) POSIX标准

Portable Operating System Interface for
Computing Systems，
统一的系统编程接口规范。

由IEEE和ISO/IEC开发。

保证应用程序源代码级的可移植性。

Linux完全遵循POSIX标准。

4) GPL

通用公共许可证。

允许对某成果及其派生成果的重用、修改和复制，
对所有人都是自由的，但不能声明做了原始工作，
或声明由他人所做。

4. 版本

1) 早期版本：0.01, 0.02, ..., 0.99, 1.0

2) 旧计划：介于1.0和2.6之间，A.B.C

A: 主版本号，内核大幅更新。
B: 次版本号，内核重大修改，奇数测试版，偶数稳定版。
C: 补丁序号，内核轻微修订。

3) 2003年12月发布2.6.0以后：缩短发布周期，A.B.C-D.E

D: 构建次数，反映极微小的更新。
E: 描述信息。
   rc/r - 候选版本，其后的数字表示第几个候选版本，
          越大越接近正式版本
   smp - 对称多处理器
   pp   - Red Hat Linux的测试版本
   EL   - Red Hat Linux的企业版本
   mm   - 测试新技术或新功能
   fc   - Red Hat Linux的Fedora Core版本

如:

cat /proc/version

Linux version 3.6.11-4.fc16.i686

cat /proc/version
Linux version 3.2.0-39-generic-pae

5. 特点

1) 遵循GNU/GPL
2) 开放性
3) 多用户
4) 多任务
5) 设备独立性
6) 丰富的网络功能
7) 可靠的系统安全
8) 良好的可移植性

6. 发行版本

1) 大众的Ubuntu
2) 优雅的Linux Mint
3) 锐意的Fedora
4) 华丽的openSUSE
5) 自由的Debian
6) 简洁的Slackware
7) 老牌的RedHat

四、GNU编译工具GCC
---

1. 支持多种编程语言

C、C++、Objective-C、Java、Fortran、Pascal、Ada

2. 支持多种平台

Unix、Linux、Windows。

3. 构建(Build)过程

编辑 -> 预编译 -> 编译 -> 汇编 -> 链接

```
1) 编辑:   vi hello.c                -> hello.c
2) 预编译: gcc -E hello.c -o hello.i -> hello.i -+
3) 编译:   gcc -S hello.i            -> hello.s  | GCC
4) 汇编:   gcc -c hello.s            -> hello.o  | 工具链
5) 链接:   gcc hello.o -o hello      -> hello    -+
```

范例：hello.c

4. 查看版本

```
gcc -v
```

5. 文件后缀

```
.h  - C语言源代码头文件
.c  - 预处理前的C语言源代码文件
.i  - 预处理后的C语言源代码文件
.s  - 汇编语言文件
.o  - 目标文件
.a  - 静态库文件
.so - 共享库(动态库)文件
```

6. 编译单个源程序

```
gcc [选项参数] 文件

-c        - 只编译不链接
-o        - 指定输出文件
-E        - 预编译
-S        - 产生汇编文件
-pedantic - 对不符合ANSI/ISO C语言标准的
            扩展语法产生警告
-Wall     - 产生尽可能多的警告。
            范例：gcc -Wall wall.c
-Werror   - 将警告作为错误处理。
            范例：gcc -Werror werror.c
-x        - 指定源代码的语言。
            范例：gcc -x c++ cpp.c -lstdc++
-g        - 生成调试信息
-O1/O2/O3 - 优化等级
```

7. 编译多个源程序

```
gcc [选项参数] 文件1 文件2 ...
```

思考：头文件的作用是什么?

1) 声明外部变量、函数和类。
2) 定义宏、类型别名和自定义类型。
3) 包含其它头文件。
4) 借助头文件卫士，防止因同一个头文件被多次包含，
   而引发重定义错。

包含头文件时需要注意：

1) gcc的-I选项

指定头文件附加搜索路径。

2) #include <...>

先找-I指定的目录，再找系统目录。

3) #include "..."

先找-I指定的目录，再找当前目录，最后找系统目录。

4) 头文件的系统目录

/usr/include
/usr/local/include
/usr/lib/gcc/i686-linux-gnu/4.6.3/include
/usr/include/c++/4.6.3 (C++编译器优先查找此目录)

范例：calc.h、calc.c、math.c

math.c中不包含calc.h，输出0.000000。
参数和返回值均按int处理。

math.c中包含calc.h，输出30.000000。
参数和返回值均按double处理。

8. 预处理指令

```
#include      // 将指定文件的内容插至此指令处
#include_next // 与#include一样，
              // 但从当前目录之后的目录查找，极少用
#define       // 定义宏
#undef        // 删除宏
#if           // 判定
#ifdef        // 判定宏是否已定义
#ifndef       // 判定宏是否未定义
#else         // 与#if、#ifdef、#ifndef结合使用
#elif         // else if多选分支
#endif        // 结束判定
##            // 连接宏内两个连续的字符串
#             // 将宏参数扩展成字符串字面值

#error   // 产生错误，结束预处理
#warning // 产生警告
```

范例：error.c

```
# gcc error.c -DVERSION=2
error.c:4:3: error: #error "Version too low !"

# gcc error.c -DVERSION=3

# gcc error.c -DVERSION=4
error.c:6:3: warning: #warning "Version too high !" [-Wcpp]
```

#line // 指定行号

范例：line.c

#pragma // 提供额外信息的标准方法，可用于指定平台

```
#pragma GCC dependency <文件> // 若<文件>比此文件新
                              // 则产生警告
#pragma GCC poison <标识>     // 若出现<标识>
                              // 则产生错误
#pragma pack(1/2/4/8)         // 按1/2/4/8字节
                              // 对齐补齐
```

范例：pragma.c

9. 预定义宏

```
__BASE_FILE__     // 正在编译的源文件名
__FILE__          // 所在文件名
__LINE__          // 行号
__FUNCTION__      // 函数名
__func__          // 同__FUNCTION__
__DATE__          // 日期
__TIME__          // 时间
__INCLUDE_LEVEL__ // 包含层数，从0开始
__cplusplus       // C++编译器将其定义为1，
                  // C编译器不定义该宏
```

范例：print.h、predef.h、predef.c

```
# gcc predef.c

__BASE_FILE__     : predef.c
__FILE__          : print.h
__LINE__          : 9
__FUNCTION__      : print
__func__          : print
__DATE__          : May 25 2013
__TIME__          : 07:31:39
__INCLUDE_LEVEL__ : 2

# g++ predef.c

__BASE_FILE__     : predef.c
__FILE__          : print.h
__LINE__          : 9
__FUNCTION__      : print
__func__          : print
__DATE__          : May 25 2013
__TIME__          : 07:32:33
__INCLUDE_LEVEL__ : 2
__cplusplus       : 1
```

10. 环境变量

```
C_INCLUDE_PATH     - C头文件的附加搜索路径，
                     相当于gcc的-I选项
CPATH              - 同C_INCLUDE_PATH
CPLUS_INCLUDE_PATH - C++头文件的附加搜索路径
LIBRARY_PATH       - 链接时查找静态库/共享库的路径
```

LD_LIBRARY_PATH   - 运行时查找共享库的路径

范例：calc.h、calc.c、cpath.c

```
# gcc calc.c cpath.c
cpath.c:2:17: fatal error: calc.h: No such file or directory
```

通过gcc的-I选项指定C/C++头文件的附加搜索路径：

```
# gcc calc.c cpath.c -I.
```

将当前目录作为C头文件附加搜索路径，
添加到CPATH环境变量中：

```
# export CPATH=$CPATH:. // export保证当前shell的
                        // 子进程继承此环境变量
# echo $CPATH
# env | grep CPATH
```

也可以在~/.bashrc或~/.bash_profile
配置文件中写环境变量，持久有效：

```
export CPATH=$CPATH:.
```

执行
```
# source ~/.bashrc
```
或
```
# source ~/.bash_profile
```
生效。以后每次登录自动生效。

头文件的三种定位方式：

1) #include "目录/xxx.h"   - 头文件路径发生变化，
                                需要修改源程序
2) C_INCLUDE_PATH/CPATH=目录 - 同时构建多个工程，
                                可能引发冲突
3) gcc -I目录   - 既不用改程序，
                                也不会有冲突

五、库
------

1. 合久必分——增量编译——易于维护。
   分久必合——库——易于使用。

2. 链接静态库是将库中的被调用代码复制到调用模块中，
   而链接共享库则只是在调用模块中，
   嵌入被调用代码在库中的(相对)地址。

3. 静态库占用空间非常大，不易修改但执行效率高。
   共享库占用空间小，易于修改但执行效率略低。

4. 静态库的缺省扩展名是.a，共享库的缺省扩展名是.so。

六、静态库
----

1. 创建静态库

1) 编辑源程序：.c/.h
2) 编译成目标文件：gcc -c xxx.c -> xxx.o
3) 打包成静态库文件：ar -r libxxx.a xxx.o ...

```
# gcc -c calc.c
# gcc -c show.c
# ar -r libmath.a calc.o show.o
```

ar指令：ar [选项] 静态库文件名 目标文件列表
-r - 将目标文件插入到静态库中，已存在则更新
-q - 将目标文件追加到静态库尾
-d - 从静态库中删除目标文件
-t - 列表显示静态库中的目标文件
-x - 将静态库展开为目标文件

注意：提供静态库的同时也需要提供头文件。

2. 调用静态库

```
# gcc main.c libmath.a (直接法)
```

或通过LIBRARY_PATH环境变量指定库路径：

```
# export LIBRARY_PATH=$LIBRARY_PATH:.
# gcc main.c -lmath (环境法)
```

或通过gcc的-L选项指定库路径：

```
# unset LIBRARY_PATH
# gcc main.c -lmath -L. (参数法)
```

一般化的方法：gcc .c/.o -l<库名> -L<库路径>

3. 运行

```
# ./a.out
```

在可执行程序的链接阶段，已将所调用的函数的二进制代码，
复制到可执行程序中，因此运行时不需要依赖静态库。

范例：static/

七、共享库
---

1. 创建共享库

1) 编辑源程序：.c/.h
2) 编译成目标文件：gcc -c -fpic xxx.c -> xxx.o

3) 链接成共享库文件：gcc -shared xxx.o ... -o libxxx.so

```
# gcc -c -fpic calc.c
# gcc -c -fpic show.c
# gcc -shared calc.o show.o -o libmath.so
```

或一次完成编译和链接：

```
# gcc -shared -fpic calc.c show.c -o libmath.so
```

PIC (Position Independent Code)：位置无关代码。
可执行程序加载它们时，可将其映射到其地址空间的
任何位置。

-fPIC : 大模式，生成代码比较大，运行速度比较慢，
所有平台都支持。

-fpic : 小模式，生成代码比较小，运行速度比较快，
仅部分平台支持。

注意：提供共享库的同时也需要提供头文件。

2. 调用共享库

```
# gcc main.c libmath.so (直接法)
```

或通过LIBRARY_PATH环境变量指定库路径：

```
# export LIBRARY_PATH=$LIBRARY_PATH:.
# gcc main.c -lmath (环境法)
```

或通过gcc的-L选项指定库路径：

```
# unset LIBRARY_PATH
# gcc main.c -lmath -L. (参数法)
```

一般化的方法：gcc .c/.o -l<库名> -L<库路径>

3. 运行

运行时需要保证LD_LIBRARY_PATH
环境变量中包含共享库所在的路径：

```
# export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
# ./a.out
```

在可执行程序的链接阶段，
并不将所调用函数的二进制代码复制到可执行程序中，
而只是将该函数在共享库中的地址嵌入到可执行程序中，
因此运行时需要依赖共享库。

范例：shared/

gcc缺省链接共享库，可通过-static选项强制链接静态库。

如：gcc -static hello.c

八、动态加载共享库

#include <dlfcn.h>

1. 加载共享库

```
void* dlopen (
    const char* filename, // 共享库路径，
                          // 若只给文件名，
                          // 则根据LD_LIBRARY_PATH
                          // 环境变量搜索
    int         flag      // 加载方式
);
```

成功返回共享库句柄，失败返回NULL。

flag取值：

RTLD_LAZY - 延迟加载，使用共享库中的符号
(如调用函数)时才加载。

RTLD_NOW - 立即加载。

2. 获取函数地址

```
void* dlsym (
    void*       handle, // 共享库句柄
    const char* symbol  // 函数名
);
```

成功返回函数地址，失败返回NULL。

3. 卸载共享库

```
int dlclose (
    void* handle // 共享库句柄
);
```

成功返回0，失败返回非零。

4. 获取错误信息

```
char* dlerror (void);
```

有错误发生则返回错误信息字符串指针，否则返回NULL。

范例：load.c

注意：链接时不再需要-lmath，但需要-ldl。

九、辅助工具

nm: 查看目标文件、可执行文件、静态库、
共享库中的符号列表。

ldd: 查看可执行文件和共享库的动态依赖。

ldconfig: 共享库管理。

事先将共享库的路径信息写入/etc/ld.so.conf配置文件中，
ldconfig根据该配置文件生成/etc/ld.so.cache缓冲文件，
并将该缓冲文件载入内存，借以提高共享库的加载效率。

系统启动时自动执行ldconfig，但若修改了共享库配置，
则需要手动执行该程序。

strip: 减肥。去除目标文件、可执行文件、
静态库和共享库中的符号列表、调试信息等。

objdump: 显示二进制模块的反汇编信息。

```
# objdump -S a.out

指令地址     机器指令                  汇编指令
--------     --------------------      --------------------
8048514:     55                        push %ebp
8048515:     89 e5                     mov  %esp,%ebp
8048517:     83 e4 f0                  and  $0xfffffff0,%esp
804851a:     83 ec 20                  sub  $0x20,%esp
804851d:     c7 44 24 04 02 00 00      movl $0x2,0x4(%esp)
```

作业：编写一个函数diamond()，打印一个菱形，其高度、
宽度、实心或者空心以及图案字符，均可通过参数设置。
分别封装为静态库libdiamond_static.a和
动态库libdiamond_shared.so，并调用之。

代码：diamond/