

# 第三讲

---

## Linux下的C和C++编程

# 主要内容

---

- C语言编译器GCC
- 程序维护工具make

# 引言

---

- 需要 C 语言吗？

- **Shell** 够用吗？

- 脚本语言是一种解释性语言，用户输入只有当执行脚本后才被识别和执行。解释性语言在每次执行脚本时必须重新解释，效率低下，也不适合于直接操作计算机的**RAM**和**I/O**设备。

- **Linux**和**C**语言关联？

- **Shell**命令和**Linux**内核都是用**C**和**C++**编写而成的。

# C语言编译器GCC

---

- ❑ GCC 的安装
- ❑ 使用 GCC 编译器编译程序
- ❑ 函数库
- ❑ 调试器

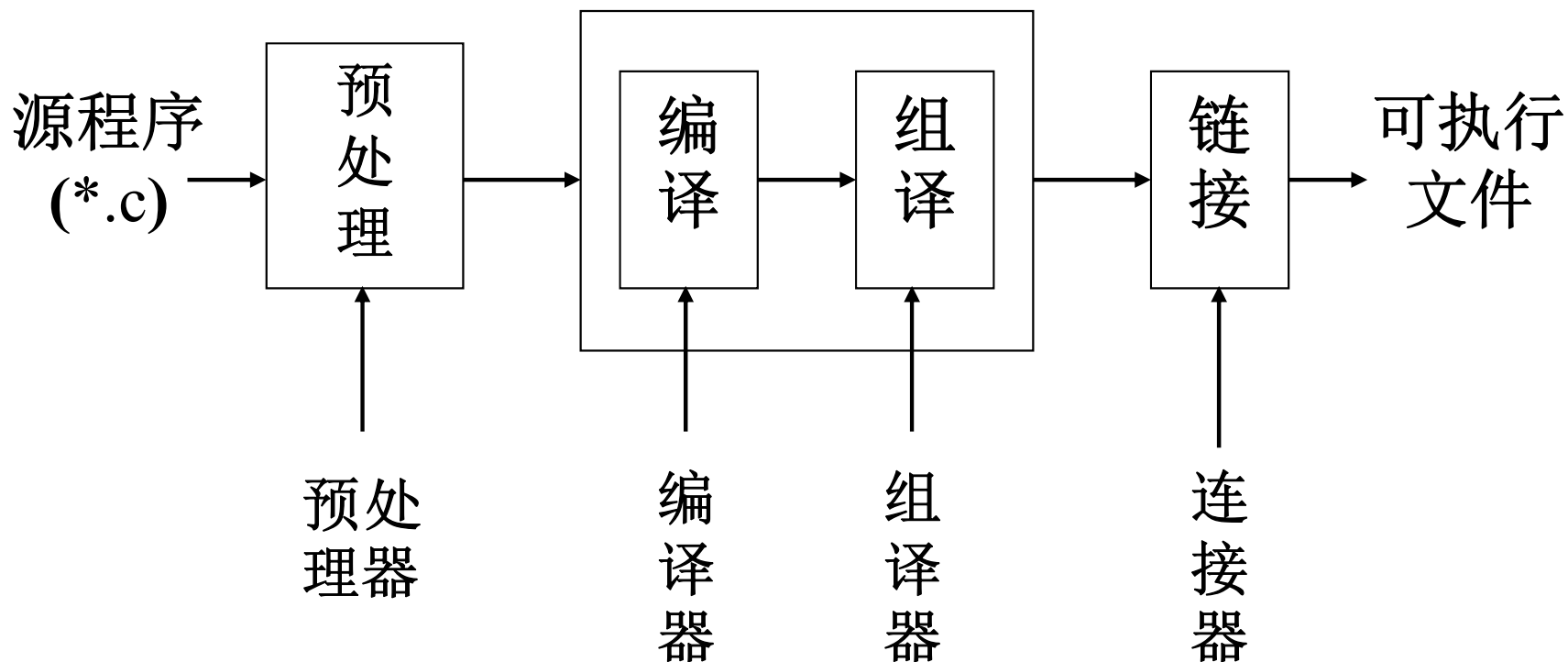
# C语言编译器GCC

---

- 在 **Linux** 开发环境下，最常用的 **C/C++** 语言编译器 **GCC**(**GNU C Compile**) 的缩写，它是 **GNU/Linux** 系统下的标准 **C** 编译器。
- 目前，**GCC**可以支持多种高级语言，如
  - C/C++
  - Object C
  - JAVA
  - Fortran
  - PASCAL
  - ADA等
  - 还可根据需要选择安装支持的语言。

# C语言编译器GCC

- **gcc** 可以使程序员灵活地控制编译过程。编译过程一般可以分为下面四个阶段，每个阶段分别调用不同的工具进行处理，如下图所示。



# C语言编译器GCC

---

## ■ GCC编译分为四个步骤：

### ■ 预处理

**GCC**调用**cpp**程序进行预处理，即分析像**#include**、**#define**之类的命令。

### ■ 编译

**GCC**是调用**ccl**程序进行编译的，它根据源代码生成汇编语言。

### ■ 汇编

**GCC**调用**as**程序将上一步的结果生成扩展名为**.o**的目标文件

### ■ 连接

**GCC**调用**ld**程序将目标文件进行连接，最后生成可执行文件。

# C语言编译器GCC

---

## ■ gcc 的版本信息

一般来说，系统安装后就已经安装和设定好了gcc。在 shell 的提示符下键入 `gcc -v`，屏幕上就会显示出目前正在使用的 gcc 的版本。

## ■ GCC 的安装

- `sudo apt-get install build-essential`



# C语言编译器GCC

---

## ■ 使用 GCC 编译器

- 通常后跟一些选项和文件名来使用 GCC 编译器，gcc 命令的基本用法如下：
  - gcc [ option | filename ]
  - g++ [ option | filename ]
- 其中 option 为 gcc 使用时的选项，而 filename 为 gcc 要处理的文件

# C程序: test1.c

---

```
#include <stdio.h>  
#define MIN(A,B) ((A)>(B)?(B):(A))  
int main(void){  
int a;  
a=MIN(1,2);  
printf("The result is:%d\n",a);  
return 0;  
}
```

# C++程序: test2.cxx

---

```
#include <iostream>  
#define MIN(A,B) ((A)>(B)?(B):(A))  
int main(void){  
int a;  
a=MIN(1,2);  
std::cout <<"The result is:"<<a<< "\n";  
return 0;  
}
```

# 使用GCC控制编译过程

---

- 预处理：该步骤完成宏和**include**的处理。

**gcc -E -o test1.pre.c test1.c**

打开**test1.pre.c**文件，可以发现处理在开头加入了许多函数声明外，**MIN (A,B)**宏在代码中也被展开。

- 生成汇编代码：该步骤将预处理生成的代码进行处理，并根据编程参数进行优化，最后生成汇编语言。

**gcc -S test1.c cat test1.s**

- 生成目标代码：该步骤把中间代码变成特定机器上的指令代码。

**gcc -c test1.c file test1.o**

- 链接生成可执行代码。

**gcc -o test1 test1.c**

# C语言编译器GCC

---

- 整个程序由两个文件**area.c** 和**circle.c**组成，编译命令：

**gcc -o myprog area.c circle.c;**

**gcc -c area.c; gcc -c circle.c;**

**gcc -o myprog area.o circle.o**

# C语言编译器GCC

---

- 生成可执行程序的最后一步是链接，也就是将分散的目标文件和库文件组合起来。通常在**Linux**系统上，这些库文件可以在**/lib**与**/usr/lib**目录中找到。
- 当用户使用的是静态的函数库时，链接器会找到程序需要的模组，将它们物理复制到可执行文件中。共享函数库会在执行文件时留下一个记号，指明程序执行时，首先必须加载这个函数库。
- 共享函数库使执行文件更小，**Linux**默认的行为是链接共享函数库。
- 静态函数库（**.a**），共享函数库（**.so.x.x.x**）

# C语言编译器GCC

---

## ■ 函数库

- 用户可用 “ldd” 命令来查程序需要的共享函数库：
  - **ldd test1**
  - 上例说明 **test1** 依赖**libc.so.6**与**linux-gate.so.1**的存在
- **Linux** 缺省的行为是连接共享函数库。

# C语言编译器GCC

---

## ■ 调试程序

- **GCC**包含完整的出错检查和警告提示功能，可以帮助**Linux**程序员写出更加专业的代码。

- 代码：**illcode.c**程序

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

```
void main(void){
```

```
long long int var=1;
```

```
printf("It is not standard C code!\n");
```

```
}
```



# C语言编译器GCC

---

## ■ 调试程序

- 存在以下问题：
- **Main**函数的返回值被声明为**void**，但实际上应该是**int**
- 使用了“**long long**”来声明整数，不符合**ANSI/ISO C**语言规范
- **Main**函数在终止前没有调用**return**语句

# C语言编译器GCC

---

## ■ 调试程序

- **-pedantic**参数检查源代码是否符合**ANSI/ISO C语言规范**

**gcc -pedantic -o illcode illcode.c**

- **-Wall**或**-W**参数能够使**GCC**产生尽可能多的警告信息。

# C语言编译器GCC

---

## ■ 调试器

- 在 **Linux** 环境下最普及的调试工具是 **GDB** 和 **DDD**，此处仅对 **GDB** 作简单介绍。
- 调试器的功能就是能够观察一个程序在执行时的内部活动，或程序出错时发生了什么。**GNU 的调试器称为 GDB（GNU debugger）**，该程序是一个交互式工具，工作在字符模式，可用于源代码级调试，以及跟踪没有源代码的程序或检查某个终止的程序留下的核心文件。

# C语言编译器GCC

---

- **GDB 主要有以下这些功能：**

- 能跟踪程序中变量的值。
- 能够设置断点使程序在指定的代码行上停下来。
- 能够一行行地执行源代码。
- 修正某个 **bug** 引起的问题，然后继续查找另一个 **bug**。

# C语言编译器GCC

---

- 一般来说 GDB 主要调试的是 C/C++ 的程序。

要调试 C/C++ 的程序，首先在编译时，必须把调试信息加到可执行文件中。使用编译器（gcc/g++）的 **-g 参数** 可以做到这一点。如：

- gcc -g test.c -o test
  - g++ -g test.cpp -o test
  - 如果没有 -g，在调试时将见不到程序的函数名、变量名，所代替的全是运行时的内存地址。
- 启动 GDB 的方法是从命令行键入 “gdb filename”，按回车键就可以运行 GDB 来调试可执行文件。

# C语言编译器GCC

---

## ■ 基本 GDB 命令：

- **file** 装入想要调试的可执行文件
- **kill** 终止正在调试的程序
- **list** 列出产生执行文件的源代码的一部分
- **next** 执行一行源代码但不进入函数内部
- **step** 执行一行源代码而且进入函数内部
- **run** 执行当前被调试的程序
- **quit** 终止 gdb
- **watch** 监视一个变量的值而不管它何时被改变
- **break** 在代码里设置断点, 这将使程序执行到这里时被挂起
- **shell** 不离开 gdb 就执行 UNIX shell 命令

# C语言编译器GCC

---

## ■ GDB使用举例

### ■ 源代码如下

```
// bug.c
#include <stdio.h>
#include <stdlib.h>
static char buff[256];
static char* string;

int main()
{
    printf("input a string:");
    gets(string);

    printf("\n Your string is:%s\n",string);
    return 0;
}
```

编译:  
gcc -o bug bug.c

# C语言编译器GCC

## ■ 编译并运行

```
[root@donger gcctest]# ls
add.c  bug.c      minus.c  mytest.c  test.c  ts
add.o  gcctest.c  minus.o  test      test.o
[root@donger gcctest]# gcc -o bug bug.c
/tmp/ccgab33e.o(.text+0x36): In function `main':
bug.c: warning: the `gets' function is dangerous
and should not be used.
[root@donger gcctest]# ls
add.c  bug.c      minus.o  test.c
add.o  gcctest.c  mytest.c  test.o
bug    minus.c    test     ts
[root@donger gcctest]# ./bug
input a string:hello
段错误
```

编译

????

■



# C语言编译器GCC

## ■ 使用gdb调试bug

```
[root@donger gcctest]# gdb bug
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) run
Starting program: /home/donger/gcctest/bug
Reading symbols from shared object read from target memory...(no debugging sy
mbols found)...done.
Loaded system supplied DSO at 0xeac000
(no debugging symbols found)
(no debugging symbols found)

Program received signal SIGSEGV, Segmentation fault.
0x002647e0 in gets () from /lib/libc.so.6
(gdb) where
#0 0x002647e0 in gets () from /lib/libc.so.6
#1 0x080483ea in main ()
(gdb) █
```

运行bug

输入字符串

出错位置

能不能看到源代码呢？

# C语言编译器GCC

- 使用gcc的-g参数
  - **gcc -g -o bug bug.c**
  - 重新调试

```
Program received signal SIGSEGV, Segmentation fault.  
0x002647e0 in gets () from /lib/libc.so.6
```

```
(gdb) where
```

```
#0  0x002647e0 in gets () from /lib/libc.so.6
```

```
#1  0x080483ea in main () at bug.c:10
```

```
(gdb) list
```

```
12             gets(string);
```

```
13
```

```
14             printf("\n Your string is:%s\n",string);
```

```
15             return 0;
```

```
16         }
```

```
(gdb) █
```

源代码

# C语言编译器GCC

---

```
(gdb) break 12
```

```
Breakpoint 2 at 0x80483dc: file bug.c, line 12.
```

```
(gdb) next
```

```
Single stepping until exit from function gets,  
which has no line number information.
```

```
Program terminated with signal SIGSEGV, Segmentation fault.  
The program no longer exists.
```

```
(gdb) print string
```

```
$1 = 0x0
```

```
(gdb) █
```

# 程序维护工具make

---

## ■ 引言

- 随着软件复杂度的提高，人们提出了模块化的概念，即将复杂的软件分解为很多细小的功能模块，于是，软件代码按功能模块分散到各个不同的文件。
- 这使得编译这些文件成为问题，有时候只是改动某个文件就不得不重新编译整个工程。对于大型项目而言，重新编译所有代码往往需要很长时间。
- 使用**make**对源代码进行管理后，将只编译改动的代码文件，而不用完全编译。

# 程序维护工具make

---

## ■ make工具

- **Make**通过读入配置好的文本文件，并根据文本文件中预先定义的规则和步骤，完成代码的编译和链接工作，最终生成所需要的项目文件。这个文本文件在缺失情况下为**makefile**或**Makefile**。
- **Make**在对项目文件进行编译时，会判断文件的修改和生成时间。如果某源代码文件在上次编译后再次被修改，则**make**将只编译该文件，而不会对整个重新编译。

# Makefile

---

## ■ An example,

- there are two files (a.c, b.c) to produce a program called **test**.
- #一个简单的Makefile例子( # 表示为注释行)  
**test: a.c b.c**  
**<TAB> gcc a.c b.c -o test**
- 执行**make**产生“**test**”可执行文件。

## ■ Example:

- **=== makefile 开始 ===**  
**myprog : foo.o bar.o**  
**gcc foo.o bar.o -o myprog**  
  
**foo.o : foo.c foo.h bar.h**  
**gcc -c foo.c -o foo.o**  
  
**bar.o : bar.c bar.h**  
**gcc -c bar.c -o bar.o**  
**=== makefile 结束 ===**

# Makefile的变量

■ **makefile** 里的变量就像一个环境变量，定义可以贮存一个文件名列表、存储编译器名、贮存编译器选择项等。如：**CC** (**C**编译器名称)，**CFLAGS** (**C**编译器选择项)，等等。

■ **An example:**

■ **=== makefile 开始 ===**

**OBJS = foo.o bar.o**

**CC = gcc**

**CFLAGS = -Wall -O -g**

**myprog : \$(OBJS)**

**\$(CC) \$(OBJS) -o myprog**

**foo.o : foo.c foo.h bar.h**

**\$(CC) \$(CFLAGS) -c foo.c -o foo.o**

**bar.o : bar.c bar.h**

**\$(CC) \$(CFLAGS) -c bar.c -o bar.o**

**=== makefile 结束 ===**

# Makefile的内部变量

- 还有一些设定好的内部变量，它们根据每一个规则内容定义。
- 三个比较有用的变量是 **`$@`**, **`$<`** 和 **`$^`**。
  - **`$@`** 扩展成当前规则的目标文件名，
  - **`$<`** 扩展成依靠列表中的第一个依靠文件，
  - **`$^`** 扩展成整个依靠的列表（除掉了里面所有重复的文件名）。

■ === makefile 开始 ===

**`OBJS = foo.o bar.o`**

**`CC = gcc`**

**`CFLAGS = -Wall -O -g`**

**`myprog : $(OBJS)`**

**`$(CC) $^ -o $@`**

**`foo.o : foo.c foo.h bar.h`**

**`$(CC) $(CFLAGS) -c $< -o $@`**

**`bar.o : bar.c bar.h`**

**`$(CC) $(CFLAGS) -c $< -o $@`**

■ === makefile 结束 ===



# Make的隐含规则

---

- 该规则定义如何从各种相关文件中生成特定类型的目标；如从 **.c** 文件和相关文件里产生 **.o** 文件，这是一个标准的步骤。
- 还有一些内置的规则，这些规则告诉它当你没有给出某些命令的时候，应该怎么办。
- 如果你把生成 **foo.o** 和 **bar.o** 的命令从它们的规则中删除，**make** 将会查找它的隐含规则，然后会找到一个适当的命令。
- 一个典型的隐含规则，把**.c**文件变成**.o**文件：
  - **.C.o:**
    - **\$(CC) \$(CFLAGS) \$(CPPFLAGS) -c \$< -o \$@**

# Running make

---

## ■ Make [-f makefile][options][targets]

## ■ Examples

- **make**, 建立**makefile**的第一个目标，如果没有此文件，给出错误信息；
- **make target**, 建立由**makefile**定义的特定目标；
- **make -f filename**, 进行由**filename**定义的程序编译；
- **make clean**, 建立**clean**目标，既清除目标文件；

# Example

---

- Two files area.c & circle.c

- Their makefiles:

- Makefile:

- #this is a simple makefile
- area:area.o circle.o
- gcc -o area
- circle.o area.o
- clean:
- rm \*.o

- Makefile1:

- #this is a simple makefile with macros.
- OBJS = area.o circle.o
- CC = gcc
- CFLAGS = -Wall -O -g
- myprog : \$(OBJS)
- \$(CC) \$^ -o \$@
- circle.o : circle.c
- \$(CC) \$(CFLAGS) -c \$< -
- o \$@
- area.o : area.c
- \$(CC) \$(CFLAGS) -c \$< -
- o \$@