Instituto Superior de Engenharia do Porto
Departamento de Engenharia Informática

SISTEMAS OPERATIVOS I

Parte III

Abril de 2006

Nuno Malheiro
Maria João Viamonte
Berta Batista
Luis Lino Ferreira

Sugestões e participações de erros para:
ntm@dei.isep.ipp.pt

BIBLIOGRAFIA

"An Introduction to GCC" de Brian J. Gough, ISBN: 0-9541617-9-3

ÍNDICE

1 INTRODUÇÃO

O objectivo deste documento é o fornecimento de elementos para a melhor utilização do compilador de C, estando focalizado no que é disponibilizado pela GNU (GCC). Recorde-se que não é um objectivo deste documento o ensino da linguagem em si mas antes da globalidade do seu processo de compilação, incluindo pré-processamento e as várias opções de compilação no que diz respeito, por exemplo, a optimização e *debugging*.

O autor inicial do compilador de C da GNU (GCC) foi Richard Stallman, o fundador do Projecto GNU.
A primeira vez que o GCC foi disponibilizado foi em 1987, tendo-se tornado desde então uma ferramenta importante no desenvolvimento de software.
Ao longo dos anos, o GCC foi incorporando linguagens adicionais, incluindo Fortan, ADA, Java e Objective-C. Dada esta expansão, a sigla GCC alterou-se de "GNU C Compiler" para "GNU Compiler Collection". Neste momento, a sua utilização é bastante abrangente.

O GCC é um compilador cuja característica é a portabilidade através da maioria das múltiplas plataformas existentes actualmente. Pode produzir código executável em muitos tipos de processador, incluindo ainda alguns microcontroladores.
O GCC pode adicionalmente produzir código de compilação que não seja o nativo da máquina que o está a executar (*cross-compilation*), permitindo assim a compilação de programas complexos para sistemas pequenos, que não possuam capacidade de compilação (ex. sistemas embebidos). Um único ambiente de desenvolvimento instalado numa única plataforma com uma única arquitectura poderá produzir código compilado para essa mesma plataforma e, além disso, produzir código compilado para plataformas tão distintas como um microcontrolador ou ainda um supercomputador.

A linguagem C permite acesso directo à memória do computador. Devido a este facto, tem sido maioritariamente usada para o desenvolvimento de software de baixo nível, onde é necessária grande eficiência e onde o controlo sobre os recursos a utilizar é considerado crítico. Esta é, no entanto, uma possibilidade que tem a desvantagem de necessitar de uma correcção muito maior do programa, para evitar a corrupção da informação. Alguns dos erros potenciais poderão ser detectados em compilação. No entanto não existe nenhuma garantia da eliminação de todos os potenciais problemas.

2 A compilação de um programa em C

Os programas podem ser compilados, tendo como fonte um único ficheiro ou vários. Poderão ainda utilizar *header files* (ficheiros com extensão ".h") e bibliotecas do sistema.
A compilação refere o processo de conversão de um programa sob a forma textual, estruturada sintacticamente utilizando uma linguagem de programação, para código máquina. O código máquina é guardado num ficheiro referido como executável ou binário.

2.1 A compilação de um pequeno programa em C

O exemplo mais usado para mostrar a funcionalidade mais básica da linguagem é o "Olá Mundo". Esta é a nossa versão desse programa:

```
#include <stdio.h>

int main (void)
{
  printf ("Ola Mundo!\n");
  return 0;
}
```

Assumimos que o código fonte se encontra guardado num ficheiro chamado “ola.c”. Para compilar o ficheiro “ola.c” utilizando o gcc, usa-se o seguinte comando:

```
$ gcc -Wall ola.c -o ola
```

Este comando compila o ficheiro fonte para código máquina, guardando-o em seguida num ficheiro executável chamado “olá”. O nome do ficheiro de saída foi estabelecido através do uso da opção –o. Esta opção é normalmente a última do comando de compilação e, na sua ausência, o nome do executável utilizado pelo compilador é “a.out”.
A opção –Wall activa todos os avisos do compilador. Esta opção é importante dado que os avisos gerados pelo compilador podem ajudar a detectar situações potenciais de erro num programa em C.
No caso do “Olá Mundo”, o compilador não produz quaisquer avisos dado que o programa é completamente válido. O código fonte que não produz qualquer aviso é designado como limpo em compilação.
Para executar o programa deve utilizar-se:

```
$ ./ola
Ola Mundo!
```

Esta acção carrega o ficheiro executável (existente no directório corrente) para a memória e provoca um início da execução das instruções nele contidas.

2.2 Encontrar erros num programa simples

Como já foi ditto, os avisos do compilador são uma ajuda essecial para a programação em C. Para demonstrar isto apresenta-se o seguinte programa que contém um erro subtil: usa a função **`printf`** de forma incorrecta, especificando um formato de vírgula flutuante “%f” para um valor inteiro.

```
#include <stdio.h>

int main (void)
{
  printf ("Dois mais dois é %f\n", 4);
  return 0;
}
```

Este erro não é óbvio a uma primeira vista. Todavia pode ser detectado pelo compilador se a opção –Wall estiver activa. Compilar o programa acima “mau.c” com a opção –Wall activa origina o seguinte:

```
$ gcc -Wall mau.c -o mau
mau.c: In function 'main':
mau.c:6: warning: format '%f' expects type 'double', but argument 2 has
type 'int'
```

Esta mensagem indica que a *string* de formato foi utilizada incorrectamente no ficheiro “mau.c” na linha 6. As mensagens produzidas têm sempre o formato **`ficheiro:número_de_linha:mensagem`**. O compilador distingue as mensagens de erro que impedem a compilação, das mensagens de aviso que indicam problemas possíveis mas não impedem o processo de compilação.

Neste caso, a especificação correcta de formato deveria ser “%d” para o argumento inteiro. Sem a opção –Wall o resultado parecerá correcto mas produzirá resultados erráticos, alterando em cada execução:

```
$ gcc mau.c -o mau
$ ./mau
Dois mais dois é -0.111085
```

A especificação incorrecta provoca uma corrupção da saída dado que a função **`printf`** recebe um inteiro ao invés de um número em vírgula flutuante. Estes tipos são guardado em formatos diferente e ocupam um número distinto de bytes, provocando um resultado errático que pode inclusivamente variar de plataforma para plataforma. A execução apresentada é meramente representativa de uma possível execução. Outras execuções originaram resultados alternativos mas sempre errados.
Torna-se então claramente perigoso desenvolver um programa sem a devida verificação dos avisos de compilação. As funções que não estiverem a ser usadas correctamente poderão ser terminadas ou produzir resultados incorrectos. A activação de **`-Wall`** localizará a maioria dos erros comuns na programação em C.

2.3 Compilação de múltiplos ficheiros fonte

Como é conhecido, um programa em C pode ser dividido em vários ficheiros para facilitar a sua edição e compreensão, especialmente em grandes programas. Adicionalmente a compilação independente das várias partes individuais é permitida.
No próximo exemplo o "Olá Mundo" vai ser dividido em três ficheiros: "main.c", "ola_fn.c" e "ola.h". Este será o "main.c":

```
#include "ola.h"

int main (void)
{
  ola ("Mundo");
  return 0;
}
```

A invocação inicial da função printf foi substituida por uma chamada a uma função "ola" externa, que foi declarada em "ola.h" e definida em "ola_fn.c".
O programa principal inclui o *header* "ola.h" que contém a declaração da função "ola". A declaração é usada para assegurar que os tipos dos argumentos e o valor de retorno são compatíveis entre a chamada da função e a sua definição. Não é necessário incluir o *header* de sistema "stdio.h" no ficheiro "main.c" dado que não é usada directamente nenhuma função lá definida.
A declaração em "ola.h" é uma linha que especifica o protótipo da função "ola":

```
void ola (const char * nome);
```

A definição da função "ola" contida no ficheiro "ola_fn.c" é:

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

void ola (const char * nome)
{
  printf ("Ola, %s!\n", nome);
}
```

Esta função imprime uma mensagem parametrizada pela variável "nome", imprimindo "Olá <nome>!".
A diferença entre as duas formas de inclusão de ficheiros *header* é de que **`#include "Ficheiro.h"`** procura o ficheiro no directório corrente antes de o procurar nos directórios de ficheiros *header* do sistema. Por outro lado, **`#include <Ficheiro.h>`** procura os directórios de ficheiros *header* do sistema mas não pesquisa o directório corrente.

Para compilar estes ficheiros utilizando o GCC, usa-se o commando:

```
$ gcc -Wall main.c ola_fn.c -o novo_ola
```

Neste caso, usa-se a opção –o para especificar um ficheiro de saída cujo nome nada tem a ver com as entradas. Note-se ainda que o ficheiro "ola.h" não é referido, sendo acedido automaticamente pelo compilador ao encontrar a directiva **`#include "ola.h".`**
Para executar o programa anterior, usa-se o nome do executável:

```
$ ./novo_ola
Ola Mundo!
```

Todas as partes do programa foram combinadas num só ficheiro executável, produzindo em execução o mesmo resultado que o ficheiro simples usado anteriormente.

2.4 Compilação independente

Se um programa for armazenado num só ficheiro, então qualquer alteração a uma só função requer uma compilação de todo o programa, produzindo um novo executável. A recompilação de grande ficheiros fonte pode demorar bastante tempo.
Se os programas forem guardados em vários ficheiros independentes, só precisam de ser recompilados os ficheiros que tiverem sido alterados. Nesta abordagem, os ficheiros fonte são compilados separadamente e após esse passo são ligados (*linked*) entre si. Neste processo de dois passos, existe um primeiro passo que compila um ficheiro sem criar um executável. O resultado dessa compilação é guardado num ficheiro com a extensão ".o" que é referido como um ficheiro objecto. No segundo passo os ficheiros objecto são associados por um outro programa chamado *linker*. O *linker* combina todos os ficheiros objecto para produzir um único executável. Um ficheiro objecto contém código máquina onde todas as referências a endereços de memória de funções ou variáveis são deixados por definir. Isto permite a sua compilação sem referências directas entre si. O *linker* preenche estas referênciascom os valores em falta quando produz o executável.

2.5 Criação de ficheiros objecto

A opção –c é usada para efectuar a compilação de um ficheiro fonte num ficheiro objecto. Por exemplo, o seguinte comando compila o ficheiro fonte "main.c" num ficheiro objecto

```
$ gcc -Wall -c main.c
```

Este comando produz um ficheiro objecto chamado "main.o", que contém o código máquina da função main. Contém ainda uma referência externa à função "ola", todavia o endereço de memória que lhe corresponde encontra-se propositadamente indefinido neste passo.
O commando correspondente à compilação da função "ola" é:

```
$ gcc -Wall -c ola_fn.c
```

O commando anterior produz o ficheiro objecto "ola_fn.o".
Note-se que não há necessidade de usar a opção –o para especificar o nome de saída dado que a compilação com a opção –c cria automaticamente um ficheiro com mesmo nome do ficheiro fonte ".c" mas com a extensão ".o".

2.6 Criação de executáveis a partir de ficheiros objecto

O passo final na criação de um ficheiro executável é a utilização do gcc para ligar (*linkar*) os ficheiros objecto e preencher os endereços de memória das funções externas, que ainda se encontram indefinidos. Neste documento usam-se indiferentemente os termos "ligar" e "linkar" para simbolizar essa fase de compilação. Para linkar os ficheiros objecto, usa-se a linha de comando:

```
$ gcc main.o ola_fn.o -o ola
```

Esta é uma das poucas ocasiões em que não é necessária a opção –Wall dado que os ficheiros individuais se encontram todos compilados. A *linkagem* é um processo sem qualquer ambiguidade que ou tem sucesso ou falha se alguma das referências não puder ser resolvida, ou seja, só existe sucesso se todas as referências necessárias se encontrarem em algum dos ficheiros objecto definidos na linha de comando. Para efectuar esta operação o gcc recorre a um programa separado chamado ld.
O resultado desta operação pode ser executado usando a seguinte linha de comando:

```
$ ./ola
Ola Mundo!
```

Note-se que produz exactamente o mesmo resultado que a versão que utiliza um único ficheiro fonte.

2.7 Recompilação e Religação (Relinkagem)

Para mostrar como os ficheiros podem ser compilados de forma independente, editamos o ficheiro “main.c” e modificamos a instrução de impressão:

```
#include "ola.h"

int main (void)
{
  ola ("a Todos");  /* alterado de "Mundo" */
  return 0;
}
```

O ficheiro “main.c” actualizado pode ser recompilado usando o commando:

```
$ gcc -Wall -c main.c
```

O commando produz um novo ficheiro “main.o”. O ficheiro objecto “ola_fn.o”, correspondente ao ficheiro fonte “ola_fn.c”, encontra-se ainda actualizado dado que não sofreu nenhum tipo de alteração no seu código ou no *header* “ola.h” que utiliza.
O novo ficheiro objecto pode ser religado usando o comando:

```
$ gcc main.o ola_fn.o -o ola
```

O resultado produz o seguite resultado:

```
$ ./ola
Ola a Todos!
```

Só o ficheiro “main.c” foi recompilado e religado aos ficheiros objecto existentes. Se só o ficheiro “ola_fn” tivesse sido alterado, poderiamos ter recompilado e religado só este, ou seja, existiu uma poupança de tempo de compilação que pode não ser desprezável em grandes aplicações. O processo de (re)compilação pode ser automatizado recorrendo ao comando **`make`** do Unix.

2.8 Um makefile simples

Esta secção propõe um *makefile* simples para introduzir o seu conceito. Note-se que este comando é bastante elaborado e pode ser encontrado em todos os sistemas Unix.
O comando make lê a descrição de um projecto de um makefile (por omissão chamado “makefile” no directório corrente). Um *makefile* especifica um conjunto de regras de compilação quer em termos dos seus

objectivos (tais como os executáveis) quer das suas dependências (tais como ficheiros fonte e objecto) no seguinte formato:

```
objectivo: dependências
           comando
```

Para cada objectivo, o comando make verifica o tempo de alteração das dependências correspondentes, para determinar se o objectivo precisa de reconstrução. As linhas de comando devem ser indentadas usando o TAB ao invés dos espaços.
O utilitário make da GNU contém algumas regras por omissão que são ditas regras implícitas. Este facto simplifica a construção dos makefiles. Estas regras especificam, por exemplo, que os ficheiros ".o" podem ser obtidos através dos ficheiros ".c" por compilação e que um executável é criado através da ligação de ficheiros ".o".
As regras implícitas são definidas através de variáveis com valores pré-preenchidos tais como:

- CC (o compilador de C)
- CFLAGS (as opções de compilação de programas em C)

Às variáveis podem ser (re)atribuídos valores sob a forma:

```
VARIÁVEL=VALOR
```

Um *makefile* para o projecto anterior pode ser escrito da seguinte forma:

```
CC=gcc
CFLAGS=-Wall
FICHEIROS = main.o ola_fn.o
EXECUTAVEL = main

$(EXECUTAVEL): $(FICHEIROS)

clean:
      rm -f $(EXECUTAVEL) $(FICHEIROS)
```

O ficheiro pode ser interpretado da seguinte forma: usando o compilador de C gcc com a opção –Wall, construa o objecto executável main à custa dos objectos "main.o" e "ola_fn.o" (estes serão construídos através de regras implícitas a partir dos respectivos ficheiros fonte). O objectivo "clean" não tem dependências e simplemente possui o comando que remove todos os ficheiros criados por compliação.
Para utilizar o *makefile*, invoca-se o comando **make** seguido do objectivo. Sem argumentos o *makefile* produz o primeiro objectivo do ficheiro (que neste caso é o main):

```
$ make
gcc -Wall   -c -o main.o main.c
gcc -Wall   -c -o ola_fn.o ola_fn.c
gcc   main.o ola_fn.o   -o main
$ ./main
Ola Mundo!
```

Para reconstruir o executável após a alteração a qualquer dos ficheiros, simplesmente se chama novamente o make. Por verificação dos tempos de criação e alteração, o comando make identifica as reconstruções a efectuar:

```
$ joe main.c                              (edição do ficheiro)
$ make
gcc -Wall   -c -o main.o main.c
gcc    main.o hello_fn.o   -o main
$ ./main
Ola a Todos!
```

Finalmente, a limpeza de todos os ficheiros é facilitada pela possibilidade de utilização da opção “clean”:

```
$ make clean
rm -f main main.o ola_fn.o
```

São usados ficheiros mais sofisticados para aplicações maiores. A maioria inclui também regras para teste de consistência e instalação da aplicação.

2.9 Ligação a bibliotecas estáticas externas

Uma biblioteca é uma colecção de ficheiros objecto pré-compilados que podem ser ligados a outros programas. A utilização mais comum consiste no uso de bibliotecas que disponibilizam funções do sistema tais como por exemplo a função **tan** que devolve a tangente de um número (ângulo) e se encontra na biblioteca de matemática da linguagem C.
Em Unix, as bibliotecas encontram-se em ficheiros especiais com a extensão “.a” e são chamadas bibliotecas estáticas. São criadas a partir de ficheiros objecto usando a ferramenta de arquivo da GNU chamada **ar**. Quando se efectua a linkagem, as referências às funções da biblioteca são resolvidas usando o código objecto que lá se encontra.
As bibliotecas de sistema encontram-se normalmente em “/usr/lib” e “/lib”. Em sistemas tipo Unix, a biblioteca de matemática, por exemplo, encontra-se quase sempre no ficheiro “/usr/lib/libm.a”. As declarações de protótipos das funções incluídas nesta biblioteca são disponibilizadas no *header* “/usr/include/math.h”. A biblioteca *standard* de C está localizada em “/usr/lib/libc.a” e contém as funções especificadas na norma ANSI/ISSO C (tais como o `printf`, por exemplo). A biblioteca *standard* de C é, por omissão, linkada com qualquer ficheiro fonte compilado,
O exemplo seguinte chamado “calc.c” usa a função **tan** da biblioteca de matemática “libm.a”:

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

int main (void)
{
  float  x = tan (3.141592/4.0);
  printf ("A tangente de pi/4 e' %f\n", x);
  return 0;
}
```

A tentativa de criar um executável a partir somente deste ficheiro fonte causa um erro de compilação na fase de ligação:

```
$ gcc -Wall calc.c -o calc
/tmp/cc5Fj8Ai.o(.text+0x26): In function `main':
: undefined reference to `tan'
collect2: ld returned 1 exit status
```

O problema encontra-se na referência à função tan que não consegue ser encontrada (resolvida) sem a biblioteca de matemática do sistema “libm.a”. O compilador aborta o passo de *linkagem* gerando uma mensagem de erro sobre a sua incapacidade de encontrar a função tan. Este problema resolve-se incluindo no comando de compilação informação sobre a resolução desta função. O nome “/tmp/cc5Fj8Ai.o” refere um ficheiro temporário utilizado internamente pelo gcc.

Para permitir ao compilador o acesso à função tan, poder-se-ia especificá-la como objecto usando:

```
$ gcc -Wall calc.c /usr/lib/libm.a -o calc
```

A biblioteca “libm.a” contém os ficheiros objecto necessários para a função necessária (e outras tais como sin, cos, log, etc.). O linker procura e encontra a implementação da função tan, conseguindo produzir um executável:

```
$ ./calc
A tangente de pi/4 e' 1.000000
```

No ficheiro executável está incluído o código máquina da função main e o da função tan, copiado do ficheiro objecto correspondente da biblioteca “libm.a”.
Para evitar a necessidade de especificar linhas longas de comando, o compilador disponibiliza a opção –l. O comando seguinte é equivalente à linha de compilação que usa o nome completo “/usr/lib/libm.a”.

```
$ gcc -Wall calc.c –lm -o calc
```

Em termos mais gerais, a opção –lNOME tentará procurar bibliotecas nos directórios de sistema chamadas “libNOME.a” para poder *linkar* os ficheiros objecto no seu interior que sejam necessários à compilação. Poder-se-à ainda adicionar directórios para pesquisa de outras bibliotecas. Um programa com alguma dimensão utilizará várias vezes a opção –lNOME para outras tantas bibliotecas de que necessite tais como matemática, gráficos, operações de rede, etc.

Em termos da ordenação das bibliotecas, o comportamento tradicional dos *linkers* é procurar funções externas da esquerda para a direita das bibliotecas especificadas na linha de comando. Isto significa que uma biblioteca que contenha a definição de uma função deverá aparecer depois de todos os ficheiros ou bibliotecas que contenham utilização dessa função. Ou seja, não deve existir mais à direita um ficheiro ou biblioteca que use funções dos ficheiros ou bibliotecas mais à esquerda. No entanto os compiladores mais recentes já detectam estas situações, apesar de as “boas prácticas” ditarem que se mantenha a ordem lógica.

2.10 Utilização de ficheiros header de bibliotecas do sistema

Para utilizar uma biblioteca é essencial a inclusão do ficheiro de *header* apropriado para que a declaração das funções, argumentos e valores de retorno tenham os tipos adequados. Sem estas declarações, os argumentos de uma função poderiam ser passados sem os tipos correctos, causando corrupção dos resultados.
O exemplo seguinte mostra um outro programa que efectua uma chamada a uma função da biblioteca matemática do C. Neste caso, a função **pow** é usada para calcular o cubo do número dois (ou dois ao cubo):

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

int main (void)
{
  double x = pow (2.0, 3.0);
  printf ("Dois ao cubo e' %f\n", x);
  return 0;
}
```

3 Opções de compilação

Esta secção descreve as opções de compilação mais usadas do GCC. Estas opções controlam características tais como directórios de pesquisa de bibliotecas e ficheiros *header*, o uso adicional de avisos e diagnóstico e macros de pré-processador.

3.1 Opções sobre directórios de pesquisa

Um problema comum na compilação de um programa em C que use um ficheiro *header* de uma biblioteca é:

```
FICHEIRO.h: Arquivo ou diretório não encontrado
```

Este erro ocorre se o ficheiro não se encontra presente nos directórios de pesquisa de ficheiros *header* usados pelo GCC. Um problema similar pode ocorrer para as próprias bibliotecas:

```
/usr/bin/ld: cannot find -lBIBLIOTECA
```

As mensagens de erro dependem obviamente da versão do GCC instalada e das suas opções. As mensagens que se mostram neste documento são simplesmente representativas.
O erro mostrado acima acontece porque uma das bibliotecas necessárias à compilação não se encontra presente nos directórios normais de pesquisa.
Por omissão os directórios de pesquisa de ficheiros *header* são:

```
/usr/local/include/
/usr/include/
```

Por omissão os directórios de pesquisa de bibliotecas do sistema são:

```
/usr/local/lib/
/usr/lib/
```

A lista de directórios contendo ficheiros *header* é normalmente designada por *include path*. A lista de directórios para pesquisa de bibliotecas é vulgarmente designada por *library search path* ou *link path*.
Os directórios são pesquisados pela ordem em que aparecem nessas listas. Se um ficheiro existe em mais do que um directório, a versão utilizada é a do directório com maior precedência, ou seja, aquele que se encontra primeiro na lista de directórios. O directório "/usr/local/include/" tem precedência sobre o directório "/usr/include/", por exemplo, porque "/usr/local/include/" aparece primeiro na lista de directórios. De forma similar, "/usr/local/lib/" tem precedência sobre "/usr/lib/".
Quando se necessita de directórios de pesquisa adicionais para efectuar a inclusão de outros ficheiros *header* ou bibliotecas, as opções a usar são respectivamente **–I** e **–L**.

3.1.1 Exemplo sobre directórios de pesquisa

O programa seguinte exemplifica a utilização de uma biblioteca que contém várias funções úteis de baixo nível. A sua utilização permite poupar bastante tempo aos utilizadores destas estruturas de informação. No exemplo a usar vamos utilizar uma lista ligada e imprimir os seus elementos no terminal. O ficheiro chama-se "lista.c" e contém o seguinte:

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

void  myfunc(char *i, char **udata)
{
   printf("%s ; %s : %s\n", udata[0], udata[1], i);
}

int main(void)
{
  GList *list = NULL;
  char *args[] = {"Parametro 1", "Parametro 2"};

  list = g_list_append(list, "Elemento 1");
```

```
    list = g_list_append(list, "Elemento 2");

    g_list_foreach(list, (GFunc) myfunc, args);
    return 0;
}
```

Este programa usa o *header* "glib.h" e a biblioteca "glib-2.0". Se a biblioteca tivesse sido instalada em "/usr/local/lib" e o(s) ficheiro(s) de header em "/usr/local/include", então o programa poderia ser compilado usando a seguinte instrução:

```
$ gcc -Wall lista.c –lglib-2.0
```

Se os directórios necessários fizessem parte do conjunto de directórios de procura, então não haveria problema. No entanto, se a localização for outra, o erro que aparece será:

```
$ gcc -Wall lista.c –lglib-2.0
lista.h : 1:18: error: glib.h: Arquivo ou diretório não encontrado
```

Este erro significa que o ficheiro necessário "glib.h" não foi encontrado. Será necessária uma pesquisa ao sistema de ficheiros para encontrar os directórios necessários. Para facilmente pesquisar estas dependências pode ser usado o comando **pkg-config**, que retorna informação sobre a utilização de uma determinada biblioteca:

```
$ pkg-config --cflags glib-2.0
-I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include
```

Este comando devolve uma linha que pode ser usada directamente como opção **-I** de compilação. Para isso pode usar-se a forma:

```
$ gcc -Wall lista.c –lglib-2.0 -I/usr/include/glib-2.0 \
-I/usr/lib/glib-2.0/include –lglib-2.0
```

Ou, alternativamente e sempre que a biblioteca esteja registada no sistema, a forma:

```
$ gcc -Wall lista.c $(pkg-config --cflags glib-2.0) -lglib-2.0
```

Ambos os formatos declaram dois directórios adicionais para pesquisa de ficheiros *header*. Nesta situação será possível encontrar os necessários ficheiros para que a compilação se efectue até ao passo de *linkagem*. Neste passo, ou a biblioteca é encontrada ou então será necessário efectuar um acréscimo à linha de comandos para que o directório a pesquisar seja encontrado pelo GCC.

No caso de a biblioteca não existir no conjunto de directórios, o erro será:

```
$ gcc -Wall lista.c –lglib-2.0 $(pkg-config --cflags glib-2.0) –lglib-2.0
/usr/bin/ld: cannot find –lglib-2.0
```

Da mesma forma que no exemplo anterior, pode ser usado o comando **pkg-config**, que retorna informação sobre a localização de uma determinada biblioteca:

```
$ pkg-config --libs glib-2.0
-L/usr/lib/glib-2.0/libs –lglib-2.0
```

O comando global para efectuar a compilação pode, neste momento ser descrito como:

```
$ gcc -Wall lista.c –lglib-2.0 -I/usr/include/glib-2.0 \
-I/usr/lib/glib-2.0/include –lglib-2.0 -L/usr/lib/glib-2.0/libs –lglib-2.0
```

Uma outra forma de efectuar a mesma compilação seria:

```
$ gcc -Wall lista.c –lglib-2.0 $(pkg-config --cflags glib-2.0) \
  $(pkg-config --libs glib-2.0)
```

O executável produzido pode ser executado através do comando:

```
$ ./a.out
Parametro 1 ; Parametro 2 : Elemento 1
Parametro 1 ; Parametro 2 : Elemento 2
```

Note-se que não deverão ser colocados nomes absolutos nas directivas **#include** dado que este facto impedirá a sua compilação em sistemas com localizações diferentes para as dependências do projecto. As opções **–I** e **–L** são a solução ao problema de aumento dos directórios de pesquisa.

3.1.2 Variáveis de ambiente

Os directórios de pesquisa podem ainda ser controlados através de variáveis de ambiente da *shell*. As pesquisas podem ser automatizadas através da atribuição permanente de variáveis de ambiente nos ficheiros de configuração, por exemplo ".bash_profile" para a *shell* GNU Bash. Os directórios adicionais podem ser incluídos através da variável de ambiente C_INCLUDE_PATH para ficheiros *header*. Os comandos seguintes, por exemplo, acrescentam dois directórios ao conjunto de pesquisa de *headers* (em bash):

```
$ C_INCLUDE_PATH=/usr/include/glib-2.0:/usr/lib/glib-2.0/include
$ export C_INCLUDE_PATH
```

Estes directórios serão pesquisados após todos os directórios especificados na linha de comando, através da opção **–I** e exactamente antes dos directórios normais (tais como "/usr/local/include" e "/usr/include"). O comando de exportação é necessário para que as variáveis fiquem acessíveis aos programas a executar, ou seja, o compilador GCC.
A inclusão de múltiplos directórios em (quaisquer) variáveis de ambiente é efectuado através da utilização de ":" para separar cada um dos directórios. Poderão ser usados nomes absolutos ou relativos. O formato de um conjunto de directórios é o seguinte:

```
DIR1:DIR2:DIR3:...
```

De forma similar, podem ser adicionados directórios de localização de bibliotecas através da variável de ambiente LIBRARY_PATH. Os comandos seguintes permitem a pesquisa de bibliotecas em "/usr/lib/glib-2.0/libs":

```
$ LIBRARY_PATH =/usr/lib/glib-2.0/libs
$ export LIBRARY_PATH
```

Este directório será pesquisado após os directórios especificados com a opção **–L** e antes dos directórios de bibliotecas do sistema (tal como "/usr/local/lib" and "/usr/lib").

Usando as variáveis de ambiente definidas acima, o programa da secção anterior poderia ser compilado sem o uso da opções **–I** e –**L**, simplesmente usando o comando:

```
$ gcc -Wall lista.c –lglib-2.0
```

3.2 Bibliotecas estáticas e partilhadas

As bibliotecas externas podem ser de dois tipos: estáticas e partilhadas. As bibliotecas estáticas são guardadas em ficheiros ".a" como já foi visto. Quando um programa é *linkado* com uma biblioteca deste

tipo, o código máquina de quaisquer funções externas utilizadas pelo programa é copiado da biblioteca para o ficheiro executável. Desta forma todo o código necessário à execução encontra-se já no executável no final da compilação.
As bibliotecas partilhadas (normalmente com a extensão ".so" – *shared object*) são ligadas ao ficheiro executável de uma forma mais avançada. Um executável *linkado* com bibliotecas partilhadas possui unicamente uma tabela das funções que requer para a sua execução, ao invés de todo o código que a defina. Imediatamente antes do executável iniciar a sua execução, o código máquina que define as funções externas é copiado das bibliotecas partilhadas para memória. Se todas as funções externas estiverem disponíveis, o executável arranca, senão origina uma mensagem de falta de biblioteca partilhada
O processo de ligação a bibliotecas partilhadas ao executável é chamada de ligação dinâmica (*dynamic linking*) e diminui consideravelmente o tamanho do ficheiro executável e o espaço utilizado em disco, dado que a mesma biblioteca pode ser usada por múltiplas aplicações. Há ainda a vantagem de ser possível actualizar a própria biblioteca de forma independente das aplicações que a usam, desde que o interface não seja quebrado.
Dadas estas vantagens, o **gcc** tenta compilar programas para a utilização de bibliotecas partilhadas. Sempre que seja necessária a inclusão de uma biblioteca, através da opção **–l**, o compilador procurará primeiro a possibilidade partilhada e só na sua ausência usará a biblioteca estática.
No entanto, quando o executável é iniciado é necessário encontrar todas as bibliotecas partilhadas necessárias. Apesar de um programa poder ter sido correctamente compilado e *linkado*, existe ainda uma possibilidade de falha se este utilizar bibliotecas partilhadas que não estejam disponíveis nos directórios de pesquisa de bibliotecas partilhadas. A forma mais simples de acrescentar directórios de pesquisa de bibliotecas partilhadas será o preenchimento da variável de ambiente **LD_LIBRARY_PATH**.
O comando seguinte inclui o directório "/usr/lib/glib-2.0/libs" na lista de directórios nos quais se pesquisam bibliotecas partilhadas:

```
$ LIBRARY_PATH =/usr/lib/glib-2.0/libs
$ export LIBRARY_PATH
```

3.3 Quadro de opções

```
Sintaxe: gcc [opções] programa-fonte
```

Opções:

-include <ficheiro>	Inclui o conteúdo de <ficheiro> antes de outros ficheiros.
-I <dir>	Adiciona <dir> ao final do caminho de inclusão.
-L <dir>	Adiciona <dir> ao final do caminho de inclusão de bibliotecas
-o	Ficheiro de *output*. Se não for indicado assume ***a.out***.
-lang-c	Assume que o ficheiro de input é em C.
-w	Inibe mensagens de *warning*.
-Werror	Trata as mensagens de *warning* como erros.
-M	Mostra as dependências de make.
-MM	Como –M, mas ignora os *system headers*.
-MD	Como –M, mas escreve o output num ficheiro .d.
-v	Mostra a versão do GCC
-H	Mostra o nome dos header files à medida que são usados.
-E	Só executa o pré-processador.
-S	Executa pré-processador e compilador.
-c	Executa pré-processador, compilador e assembler.
-x	Permite operar com outros tipos de ficheiro origem – algumas das opções possíveis são **cpp-output** e **assembler**.

--help	Mostra o ficheiro de *help*.
-g	Inclui no código executável informação de *debbuging*.
-**save-temps**	Executa pré-processador, compilador e assembler e guarda os respectivos ficheiros

4 Utilização do Pré-processador

Este capítulo descreve a forma de utilização do pré-processador **cpp**, que é um programa que integra o GCC. O pré-processador expande macros em ficheiros fonte antes de iniciar o processo de compilação. A sua chamada é automática sempre que o GCC processa um programa em C.

4.1 Definição de macros

Para demonstrar o uso do pré-processsador da linguagem C apresenta-se um pequeno exemplo de utilização da instrução condicional **#ifdef** para verificar se uma determinada macro está definida:

```
#include <stdio.h>

int main (void)
{
#ifdef TESTE
  printf ("Modo de teste\n");
#endif
  printf ("Execução...\n");
  return 0;
}
```

Quando a macro está definida, o pré-processador inclui o código no interior do bloco condicional (de **#ifdef** até **#endif**). Neste exemplo a macro chamada TESTE, conjugada com a parte condicional do código pode, no mesmo programa, conter dois programas: um que é usado em modo de teste, imprimindo mensagens correspondentes a esse modo e um outro correspondendo à execução normal. Note-se que se não se estiver em modo de teste, as instruções no bloco condicional são desprezadas, não sendo sequer compiladas.
A opção **–DNOME** define uma macro de pré-processador chamada NOME a partir da linha de comando. Se o programa anterior for compilado com a opção **–DTESTE**, a macro TESTE ficará definida e o resultado será:

```
$ gcc -Wall -DTESTE dteste.c
$ ./a.out
Modo de teste
Execução...\
```

Se o mesmo programa for compilado sem a opção **–D**, a mensagem de “Modo de teste” é omitida do código fonte após o pré-processamento, sendo que este código não será utilizado:

```
$ gcc -Wall dteste.c
$ ./a.out
Execução...\
```

As macros são geralmente indefinidas a não ser que se especifique a opção de compilação **–D** ou, alternativamente, se use o comando **#define** num ficheiro fonte ou *header*. No entanto algumas macros são definidas automaticamente pelo compilador. Estas últimas tipicamente usam um *namespace* reservado, iniciando com o prefixo de duplo *underscore* “__”.
O conjunto completo de macros pré-definidas poder ser listada usando o pré-processador **cpp** com a opção **–dM** num ficheiro vazio:

```
cpp -dM /dev/null
#define __DBL_MIN_EXP__ (-1021)
#define __FLT_MIN__ 1.17549435e-38F
#define __CHAR_BIT__ 8
#define __WCHAR_MAX__ 2147483647
#define __DBL_DENORM_MIN__ 4.9406564584124654e-324
#define __FLT_EVAL_METHOD__ 2
#define __DBL_MIN_10_EXP__ (-307)
#define __FINITE_MATH_ONLY__ 0
#define __linux 1
#define __unix 1
#define __DBL_MAX__ 1.7976931348623157e+308
#define __DBL_HAS_INFINITY__ 1
#define __VERSION__ "4.0.1 (4.0.1-5mdk for Mandriva Linux release 2006.0)"
#define i386 1
.......
```

Esta lista inclui um pequeno número de macros de sistema que não têm o prefixo de duplo *underscore* "__". Estas macros não pertencem à norma oficial ANSI/ISO para a linguagem C e podem ser retiradas através da utilização da opção **-ansi** do **gcc.**

4.2 Macros com valores

Pode ainda ser atribuído às macros um determinado valor. Este valor é inserido no código fonte em cada ponto onde a macro ocorre. O programa seguinte usa a macro NUM para representar um número que será mostrado no terminal:

```
#include <stdio.h>

int main (void)
{
  printf ("Valor de NUM e' %d\n", NUM);
  return 0;
}
```

Note-se que as macros não são expandidas dentro das cadeias de caracteres. Só a ocorrência de NUM fora da *string* é substituída pelo pré-processador. Para definir uma macro com um valor, a linha de comando deve usar a opção **-D** na forma **-DNOME=VALOR**. A linha de comando seguinte, por exemplo, define NUM como sendo 100 antes de efectuar a compilação:

```
$ gcc -Wall -DNUM=100 dtestvalor.c
$ ./a.out
Valor de NUM e' 100
```

O exemplo anterior usa um número. No entanto uma macro pode tomar qualquer forma. Qualquer que seja o valor da macro, este é introduzido directamente no código fonte no local da macro. O próximo exemplo usa um valor de "2+2" para a macro NUM:

```
$ gcc -Wall -DNUM="2+2" dtestvalor.c
$ ./a.out
Valor de NUM e' 4
```

Após a substituição pelo pré-processador da macro NUM como "2+2", o código fonte transforma-se em:

```
#include <stdio.h>

int main (void)
{
```

```
  printf ("Valor de NUM e' %d\n", 2+2);
  return 0;
}
```

É uma boa prática rodear uma macro de parênteses, sempre que esta seja uma parte de uma expressão. O programa seguinte, por exemplo, usa parênteses para garantir o correcto funcionamento da expressão 10*NUM:

```
#include <stdio.h>

int main (void)
{
  printf ("10 vezes NUM e' %d\n", 10*(NUM) );
  return 0;
}
```

Com os parênteses, o resultado produzido é o esperado quando se executa o código:

```
$ gcc -Wall -DNUM="2+2" dtestmul10.c
$ ./a.out
10 vezes NUM e' 40
```

Sem os parenteses, o programa produziria como resultado o valor 22 vindo da expressão 10*2+2 ao invés da utilização da expressão desejada 10*(2+2).
Quando uma macro for definida com a opção **–D**, o **gcc** usa por omissão o valor 1. A compilação do programa "dtestvalor.c" apenas com a opção **–DNUM** gera:

```
$ gcc -Wall -DNUM dtestvalor.c
$ ./a.out
Valor de NUM e' 1
```

Uma macro pode ainda ser definida como vazia através da utilização da *string* vazia na opção **–DNOME="".** Esta macro é tratada em condicionais tais como **#ifdef** mas expande-se na *string* vazia.
Para utilizar uma string como opção de compilação, teriam que se usar caracteres da shell que permitam que as aspas não sejam interpretadas. Um exemplo será:

```
$ gcc -Wall –DMENSAGEM=' "Olá Mundo" ' olá_mundo_macro.c
```

4.3 Pré-processamento de ficheiros fonte

É possível ver o efeito do pré-processador directamente nos ficheiros fonte utilizando a opção **–E** do **gcc**. O ficheiro abaixo, por exemplo, define e usa a macro TESTE:

```
#define TESTE "Ola Mundo"
const char str[] = TESTE;
```

Se o ficheiro se chamar "teste.c", o efeito do pré-processador pode ser visto com a seguinte linha de comando:

```
$ gcc -E teste.c
# 1 "teste.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "teste.c"

const char str[] = "Ola Mundo";
```

A opção **–E** provoca a escrita no terminal do código fonte pré-processado, saindo sem ser efectuada uma compilação. O valor da macro TESTE é substituída directamente na saída, produzindo a sequência de caracteres **`const char str[] = "Ola Mundo";.`**

O pré-processador introduz ainda algumas linhas iniciadas por **#** para auxiliar no *debug* do programa. Estas linhas não afectam o próprio programa.

A vantagem de se ver o código pré-processado é a sua utilidade para examinar *headers* e declarações de funções do sistema. O programa seguinte inclui o header “stdio.h” para obter a declaração da função printf:

```
#include <stdio.h>

int main (void)
{
  printf ("Ola Mundo!\n");
  return 0;
}
```

É possível ver o código pré-processado usando:

```
$ gcc -E ola.c
```

Num sistema unix usando o gcc, o código gerado seria:

```
# 1 "ola.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "ola.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 28 "/usr/include/stdio.h" 3 4
. . . . .
extern int printf (__const char *__restrict __format, ...);
. . . . .
# 2 "ola.c" 2
int main (void)
{
   printf ("Olá Mundo!\n");
     return 0;
}
```

O código pré-procesado é normalmente bastante grande e pode ser redireccionado para um ficheiro ou, alternativamente, guardado com a opção **–save-temps**:

```
$ gcc -c -save-temps ola.c
```

Após esta linha de comando, é criado um ficheiro “ola.i” com o código pré-processado, alem de “ola.s” com o ficheiro em código assembly e o ficheiro objecto “ola.o”.

5 Compilação para Debug

Um ficheiro executável normalmente não contém referências ao programa fonte inicial que lhe deu origem, tais como nome de variáveis ou números de linha. O executável é simplesmente uma sequência de instruções em código máquina produzida pelo compilador. Esta informação é insuficiente para efectuar o *debug* do programa, dado que dificulta a localização dos eventuais erros. O GCC tem a opção de *debug* **–g** que permite guardar informação adicional de debug no ficheiro executável, possibilitando informações tais como a associação entre determinada instrução em código máquina e a linha do ficheiro fonte que a

produziu. A execução de um programa poderá, então ser efectuada no âmbito de um *debugger* tal como o GNU Debugger **gdb**. A sua utilização permite uma depuração efectiva dos erros do código.

6 Compilação para Optimização

Para controlar o compromisso entre o tempo de compilação e a memória utilizada pelo compilador e a velocidade de execução e o espaço ocupado do executável resultante, o GCC disponibiliza um leque de níveis de optimização, numerados de zero a três.

Um nível de optimização é escolhido através da opção **–ONIVEL**, onde o NIVEL é um numero inteiro entre 0 e 3. Os efeitos das diferentes optimização são descritas abaixo:

Opção -O0 ou nenhuma opção -O (por omissão) :

Neste nível de optimização o GCC não efectua nenhuma optimização, compilando a fonte da forma mais simples possível. Cada comando no código fonte é convertido directamente nas correspondentes instruções dentro do ficheiro executável, sem qualquer re-arranjo. Esta é a melhor opção quando ainda em desenvolvimento ou já em *debug*. Esta é a opção por omissão.

Opção -O1 ou -O :

Neste nível são ligadas as formas mais comuns de optimização, que não requeiram quaisquer compromissos entre velocidade e espaço. Com esta opção, os executáveis deverão ser mais pequenos e mais rápidos do que com a anterior. Curiosamente esta opção normalmente reduz o tempo de compilação devido à redução do volume de informação a tratar.

Opção -O2 :

Esta opção usa uma optimização de escalonamento de instruções, para além das optimizações anteriores. O sequenciamento de instruções torna-se mais adaptado ao *pipeline* do processador. O executável não deverá ter um tamanho superior ao obtido usando as anteriores opções, no entanto o tempo de compilação deverá ser superior e, além disso, o programa deverá consumir maiores recursos de memória. Esta é geralmente a melhor opção para a criação de uma versão final do programa. É esta a versão de compilação utilizada para compilar os vários programas que compõe o GCC.

Opção -O3 :

Esta opção activa as optimizações mais consumistas tais como *function inlining*, em adição às optimizações anteriores. Este nível pode aumentar a rapidez de execução mas certamente aumentará também o seu tamanho. Em algumas circunstâncias estas optimizações podem não ser favoráveis.

A título de exemplo, considere-se o programa “teste.c”:

```
#include <stdio.h>

double potencian (double d, unsigned n)
{
  double x = 1.0;
  unsigned j;

  for (j = 1; j <= n; j++)    x *= d;

  return x;
}

int main (void)
```

```
{
  double soma = 0.0;
  unsigned i;

  for (i = 1; i <= 100000000; i++)   soma += potencian (i, i % 5);

  printf ("soma = %g\n", soma);
  return 0;
}
```

O programa principal contém um ciclo de chamada da função **potencian**. Esta função calcula a n-ésima potência de um número em vírgula flutuante por sucessivas multiplicações. O tempo de execução do comando pode ser visto usando o comando **time** da *bash* que retorna, para um determinado comando, o tempo do utilizador, o real e o do sistema.

A sua execução provoca:

```
$ gcc -Wall -O0 teste_opt.c -lm
$ time ./a.out
soma = 4e+38
8.703u 0.000s 0:08.70 100.0%    0+0k 0+0io 0pf+0w

$ gcc -Wall -O1 teste_opt.c -lm
$ time ./a.out
soma = 4e+38
3.653u 0.002s 0:03.65 100.0%    0+0k 0+0io 0pf+0w

$ gcc -Wall -O2 teste_opt.c -lm
$ time ./a.out
soma = 4e+38
3.877u 0.000s 0:03.87 100.0%    0+0k 0+0io 0pf+0w

$ gcc -Wall -O3 teste_opt.c -lm
$ time ./a.out
soma = 4e+38
3.041u 0.000s 0:03.04 100.0%    0+0k 0+0io 0pf+0w
```

Note-se a grande diferença do tempo do processo (utilizador) que inicia em 8.703 segundos para a opção -O0 e consegue uma melhoria de quase 3 vezes no seu tempo de execução para 3.041 segundos com a opção –O3.

7 Funcionamento do compilador GCC

7.1 O processo de compilação

A sequência de comandos executada por uma simples evocação do GCC consiste nos seguintes passos:

- Pré-processamento (para expansão de macros)
- Compilação (tradução do código fonte em linguagem assembly)
- Assembly (tradução da linguagem assembly em código máquina)
- Linking (Resolução das várias partes para produção de um executável)

Como exemplo vamos analisar o programa do tipo Olá Mundo contido no ficheiro “ola.c”:

```
#include <stdio.h>

int main (void)
{
  printf ("Ola Mundo!\n");
  return 0;
}
```

Note-se que não é necessário o uso dos comandos individuais desta secção para compilar um programa, Todos estes comandos são executados de forma transparente pelo GCC e podem ser vistos utilizando a opção **–v** . O objectivo é transmitir conhecimento sobre o processo de compilação e não sobre a sua utilização.
Apesar da sua simplicidade, este exemplo usa um header e biblioteca externos, logo inclui todos os principais passos de compilação.

7.2 O Pré-processador

O primeiro passo de compilação executado é o do pré-processador, para expansão de macros e inclusão de *headers*. Para executar este passo, o GCC usa:

```
$ cpp ola.c > ola.i
```

O resultado é um ficheiro "ola.i" que contém o código fonte com todas as macros expandidas. Por convenção os ficheiros pré-processados têm a extensão ".i". Na prática os ficheiros pré-processados não são guardados em disco a não ser que seja especificado pela opção **–save-temps**.

7.3 O Compilador

O passo seguinte é o processo de compilação propriamente dito, executado para traduzir código pré-processado para linguagem *assembly* de um processador específico.
A opção **–S** comanda o **gcc** para a conversão en assembly, sem criar um ficheiro objecto:

```
gcc -Wall -S ola.i
```

O resultado é guardado em "ola.s". O resultado para um processador Intel X86 é o seguinte:

```
        .file   "ola.c"
        .section        .rodata
.LC0:
        .string "Ola Mundo!\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        subl    %eax, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    printf
```

```
            addl    $16, %esp
            movl    $0, %eax
            leave
            ret
            .size   main, .-main
            .section        .note.GNU-stack,"",@progbits
            .ident  "GCC: (GNU) 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)"
```

Note-se que o código *assembly* tem uma chamada a uma função externa printf.

7.4 O Assembler

O objectivo do assembler é a conversão do código *assembly* em linguagem máquina, gerando um ficheiro objecto. Quando existam chamadas a funções exteriores na fonte *assembly*, o assembler deixa indefinidos os endereços das funções, para que sejam mais tarde preenchidos pelo *linker*. O assembler pode ser invocado da seguinte forma:

```
$ as ola.s -o ola.o
```

Tal como o GCC especificado com a opção **–o**, o resultado "ola.o" contém todas as instruções de código máquina com uma referência indefinida a printf.

7.5 O Linker

O passo final de compilação é a invocação do *linker* para a criação de um executável. Um executável pode requerer muitas funções de várias bibliotecas do sistema, logo o comando que invoca o linker é algo complexo. Para o "Olá Mundo", uma possibilidade (dependendo do gcc instalado) é:

```
$ ld -dynamic-linker /lib/ld-linux.so.2 /usr/lib/crt1.o \
 /usr/lib/crti.o /usr/lib/gcc-lib/i686/3.4.2/crtbegin.o \
 -L/usr/lib/gcc-lib/i686/3.4.2 ola.o -lgcc -lgcc_eh \
 -lc -lgcc -lgcc_eh /usr/lib/gcc-lib/i686/3.4.2/crtend.o \
 /usr/lib/crtn.o
```

Felizmente nunca será necessária a sua invocação directa dado que o comando seguinte faz exactamente o mesmo (que pode ser visto com a opção –v):

```
$ gcc ola.o
```

O ficheiro produzido chama-se, por omissão, "a.out" e pode ser invocado usando:

```
$ ./a.out
Ola Mundo!
```