4

Desenvolvimento de programas em UNIX

Sumário:

Fases de desenvolvimento de programas

Há três fases principais de desenvolvimento de programas:

- **Edição (*Editing*)**.
  Faz-se a edição de ficheiros em código fonte (ficheiro1.c, ficheiro2.c, etc) através de um editor de texto ASCII.
- **Compilação (*Compiling or Translating*)**.
  Faz-se a compilação de cada um dos ficheiros em código fonte de forma a obter cada um dos respectivos ficheiros em código objecto (ficheiro1.o, ficheiro2.o, etc)
- **Unificação (*Linking*)**.
  Faz-se a junção dos ficheiros em código objecto num único ficheiro. (binary file)
  Este ficheiro poderá ser um ficheiro executável (a.out, a.exe,..) (chamado *executable* ou application) ou um ficheiro de biblioteca (.dll, lib, .a .so etc.) (chamado *code library* ou *assembly*)

Notas:
Por defeito o compilador (Linux) criará um ficheiro executável com nome a.out.
A opção de mudar o nome do ouput é o flag -o

Compilação dum programa c/ 1 ficheiro

Exemplo 4.1:
Considere-se o seguinte ficheiro fonte sum.c:

A compilação e execução deste programa através do interpretador de comandos UNIX são feitas do seguinte modo:

```
$ cc sum.c
$ a.out
Enter a number: 4
Sum of 1 to 4: 10
```

Notas:
O programa executável criado por defeito tem o nome a.out
Podemos alterar o nome do output do compilador usando a opção -o cc sum.c -o sum.exe

sum.c

```
#include <stdio.h>

int somatorio(int n)
{
  int soma = 0;
  for ( ; n>=1; n-- )
        soma += n;
  return soma;
}

main()
{
  int n;
  printf("Enter a number: ");
  scanf("%d", &n);
  printf("Sum 1 to %d\n",n,somatorio(n));
}
```

Compilação dum programa da linguagem C por fases

A compilação dum programa escrito na linguagem C geralmente poderá ser considerada a ser constituída por quatro fases, senda elas Pré-Processamento, Compilação para Código Assembler, Compilação de Código Assembler para Código Objecto e Unificação (linking) para criar um ficheiro final. Muitas vezes o compilador de Linux (gnu cc) chame de facto 4 programas distintas – diz-se que o "cc" é um "Compiler Driver".

Fase	Comando para cc/gcc	Output	Programa do Sistema Chamada
Pre-processor	cc –E sum.c -o sum.i	sum.i	cpp
Compiler	cc –S sum.c	sum.s	cc1
Assembler	cc –c sum.c	sum.o	as
Linker	cc sum.o	a.out	ld

Standard Libraries

O compilador procurará os ficheiros necessário, de inclusão na fase de pre-procesamento e as bibliotecas compilados na fase de linkagem, necessárias para compilação, nas pastas especificados por defeito no seu sistema e/ou configuração pessoal. Exemplos típicos (Linux 32 bits) são dados em baixo :

Pre-processor:	Header Files (.h)	Localização /usr/inlcude
Linker	Static and Dynamic Library Files (.a, .so)	Localização /usr/lib

Ficheiros de código objecto

Um ficheiro objecto contém dados e instruções codificados em linguagem máquina (0´s e 1´s) que o unificador (linker) usa para criar um programa executável ou uma biblioteca. Incluído também é uma tabela de identificadores ou símbolos chamado symbol table. Cada identificador (*symbol name*) identifica um subprograma (função) ou um dado (variável ou constante). Há dois tipos de identificadores:
-Locais. Estes identificadores só estão acessíveis dentro do ficheiro objecto onde estão definidos. Não é possível aceder-lhes a partir doutros ficheiros objecto. Por exemplo, todos os identificadores de variáveis e funções declarados como **static** num programa em C são identificadores locais ou variáveis declaradas dentro dum bloco {}.
-Globais : Todos os identificadores que são acessíveis a partir doutros ficheiros. Por exemplo uma função declarada por defeito ou uma variável ou constante declarada fora dum bloco de código. Referenciado num outro ficheiro com a palavra chave **extern**

Referências externas (external reference)

Uma referência externa é qualquer referência a um identificador pertencente a outro ficheiro. Se, porventura, na fase de unificação esta referência externa (p.ex., o identificador duma função) não encontra a sua definição (o código da função), o unificador **ld** debita para o dispositivo de saída a informação de que o identificador não pode ser resolvido.

Por exemplo, após a compilação do ficheiro teste.c, segue-se a unificação do identificador **pritf** com o seu código existente na biblioteca **libc** (por defeito). Só que o identificador pritf não pode ser resolvido, visto que não existe esta função na biblioteca **libc**.

```
$ cc teste.c
ld:
Unresolved:
pritf
$
```

teste.c

```
#include <stdio.h>
int main()
{
   pritf("A vida corre bem!")
   return 0;
}
```

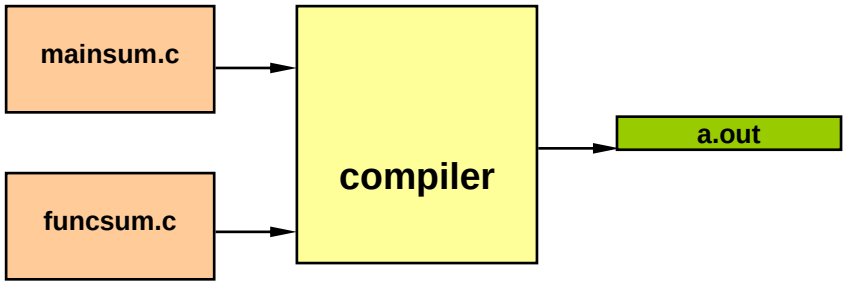
A resolução das referências externas só é feita na fase de unificação. Isto é assim porque na fase de compilação só é compilado um ficheiro de cada vez. Portanto, se uma função estiver implementada num ficheiro e a ser usada noutro ficheiro é impossível fazer a resolução ou "associação" entre o identificador da função e o seu código.
A mesma situação ocorre na utilização de variáveis declaradas **extern** num ficheiro mas são definidas num outro. A criação da tabela de símbolos em cada ficheiro objecto serve, pois, para ser possível efectuar a resolução de símbolos na fase de unificação.

```
cc -c teste2.c
cc -o teste2 teste2.o
teste2.o: In function `main':
teste2.c:(.text+0x6): undefined reference to `x'
collect2: error: ld returned 1 exit status          (linker
ld returns error)
```

teste2.c

```
#include <stdio.h>
extern int x;
int main()
{
   printf("Life the universe and .. %d\n",x);
   return 0;
}
```

Compilação dum programa c/ vários ficheiros



Exemplo 4.2:

Considere-se os seguintes ficheiros fonte, mainsum.c e funcsum.c:

A compilação e execução deste programa através do interpretador de comandos UNIX são feitas do seguinte modo:

```
$ cc mainsum.c funcsum.c
$ a.out
Enter a number: 4
Sum of 1 to 4: 10
```

mainsum.c

```
#include <stdio.h>

int somatorio(int n);

main()
{
   int n;
   printf("Enter a number: ");
   scanf("%d", &n);
   printf("Sum 1 to %d: %d\n",n,somatorio(n));
}
```

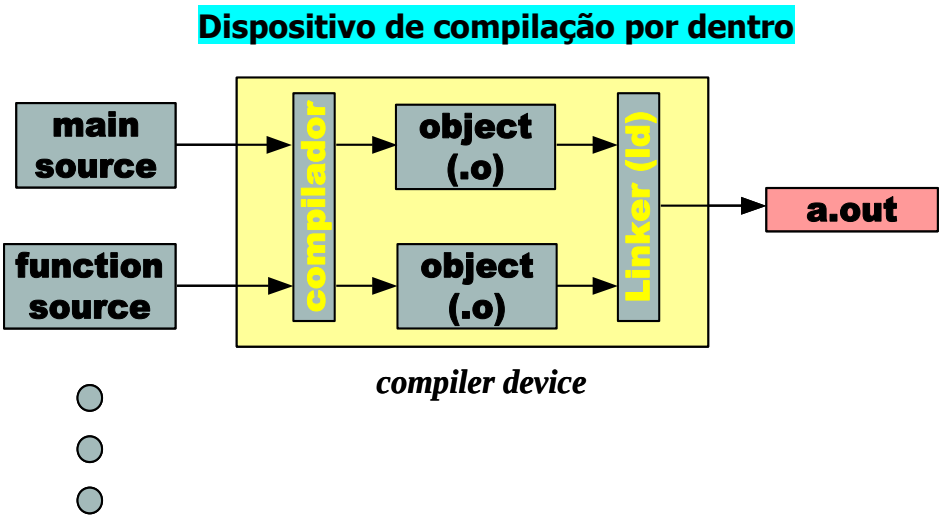
funcsum.c

```
int somatorio(int n)
{
   int soma = 0;
   for (; n>=1; n--)
         soma += n;
   return soma;
}
```

Exemplo 4.3

Os ficheiros fontes podem ser compilado separadamente para ficheiros objectos.
A seguir podem ser "unidos" (linked) num único ficheiro executável.

```
$ cc –c funcsum.c
$ cc -c mainsum.c
$ cc –o myprog funcsum.o mainsum.o
$ myprog
Enter a number: 4
Sum of 1 to 4: 10
```




```
$ cc -v mainsum.c funcsum.c
mainsum.c:
/lib/cpp mainsum.c /tmp/ctmAAAa01061 -D__hp9000s800 -D__hpux D__unix
cc: Entering Preprocessor
/lib/ccom /tmp/ctmAAAa01061 main.o -00 -v -Ac
cc: Entering Scanner and Parser.
cc: Entering Semantic Analyzer.
cc: Entering Allocator.
cc: Entering Code Generator.
cc: Entering Code Optimizer.
funcsum.c:
/lib/cpp funcsum.c /tmp/ctmAAAa01061 -D__hp9000s800 -D__hpux D__unix
cc: Entering Preprocessor
/lib/ccom /tmp/ctmAAAa01061 funcsum.o -00 -v -Ac
cc: Entering Scanner and Parser.
cc: Entering Semantic Analyzer.
cc: Entering Allocator.
cc: Entering Code Generator.
cc: Entering Code Optimizer.
/bin/ld /lib/crt0.o -u main mainsum.o funcsum.o -lc
cc: Entering Link editor.
```

O dispositivo de compilação ou compilador compreende três fases fundamentais:

- **Pre-processamento –** interpretação dos diretivas # (#include, #define , #if #ifdef..)
- **Compilação** propriamente dita incluindo **Assemblagem**
- **unificação** (*linking*) de código objecto.

A geração de código objecto é feita na fase de compilação - pela operação de converter o código Assembler em Instruções maquinas, a qual compreende 6 subfases, desde o pre-processamento (`Entering Preprocessor`) até ao optimizador de código (`Entering Code Optimizer`).
A fase de unificação análise e junte todos os ficheiros de código objecto criados na primeira fase, juntando-lhe ainda (no exemplo apresentado) o ficheiro /lib/crt0.o, sem o qual neste sistema não é possível correr um programa.
Nota que optimizações podem ser feitas em múltiplas e varias fases de operação e não apenas ao nível de Assembler ou geração das Instruções Maquinas.

O compilador coloca simplesmente as referências externas nas tabelas de símbolos dos vários ficheiros objecto, deixando para o unificador (*linker*) o acasalamento das referências externas com as definições globais.
Além do acasalamento das referências externas com as definições globais existentes nos ficheiros objecto, o unificador (ld) acasala referências externas com as definições globais existentes nas bibliotecas (*libraries*).
Isto acontece porque uma biblioteca é um ficheiro objecto que contém subprogramas e dados que podem ser usados por vários programas.

Visualização da tabela de símbolos

A visualização dos identificadores existentes na tabela de símbolos dum ficheiro objecto é feita através do comando UNIX **nm**:

```
$ nm -p funcsum.o
1073741824  d  $THIS_DATA$      Other symbols created from compiling
1073741824  d  $THIS_SHORTDATA$
1073741824  b  $THIS_BSS$
1073741824  d  $THIS_SHORTBSS$
0000000000  T  somatorio        Global definition of somatorio
$ nm -p mainsum.o
0000000000  U  $global$         Other symbols created from compiling
1073741824  d  $THIS_DATA$
1073741824  d  $THIS_SHORTDATA$
1073741824  b  $THIS_BSS$
1073741824  d  $THIS_SHORTBSS$
0000000000  T  main             Global definition of main
0000000000  U  printf           External reference to printf
0000000000  U  scanf            External reference to scanf
0000000000  U  somatorio        External reference to somatorio
```

```
cc -o soma -v funcsum.c mainsum.c

Thread model: posix
gcc version 4.9.2 20150212 (Red Hat 4.9.2-6) (GCC)

COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'
 /usr/libexec/gcc/x86_64-redhat-linux/4.9.2/cc1 funcsum.c .. -o /tmp/ccE5lPZL.s
as -v --64 -o /tmp/ccguJPUg.o /tmp/ccE5lPZL.s

/usr/libexec/gcc/x86_64-redhat-linux/4.9.2/cc1  mainsum.c  -o /tmp/ccE5lPZL.s
as -v --64 -o /tmp/ccmQ5JVL.o /tmp/ccE5lPZL.s

/usr/libexec/gcc/x86_64-redhat-linux/4.9.2/collect2 .... -o soma
/usr/lib/gcc/x86_64-redhat-linux/4.9.2/../../../../lib64/crt1.o …..

Nota: collect2 calls the linker (ld)
```

Código Assembler

Podemos embutir código assembler nos nossos programas.

https://www.codeproject.com/Articles/15971/Using-Inline-Assembly-in-C-C

Experimente este programa que é baseado no link em cima !

```
#include <stdio.h>

int main() {
    /* Add 10 and 20 and store result into register %eax */
    asm ( "movl $10, %eax;"
          "movl $20, %ebx;"
          "addl %ebx, %eax;"
    );

    /* Subtract 20 from 10 and store result into register %eax */
    asm ( "movl $10, %eax;"
          "movl $20, %ebx;"
          "subl %ebx, %eax;"
    );

    /* Multiply 10 and 20 and store result into register %eax */
    asm ( "movl $10, %eax;"
          "movl $20, %ebx;"
          "imull %ebx, %eax;"
    );

    return 0 ;
}
```

Podem imprimir o valor do registo %eax com o codigo seguinte

```
int eax_value;

// Inline assembly to move the value of eax register to eax_value variable
asm("movl %%eax, %0;" : "=r" (eax_value));

// Printing the value of eax using printf
printf("Value of %%eax register: %d\n", eax_value);
```

Mais uma Refrence
http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html