14ª Aula - Biblioteca Pessoal

Programação
Mestrado em Engenharia Física Tecnológica

Samuel M. Eleutério
sme@tecnico.ulisboa.pt

Departamento de Física
Instituto Superior Técnico
Universidade de Lisboa

Biblioteca Pessoal - Introdução

- Ao longo do tempo vamos escrevendo um **conjunto de funções** que podem vir a ser **reutilizadas em circunstâncias diferentes** daquelas para que foram **inicialmente escritas**.
- Por exemplo, escreveram-se os dados referentes aos **planetas do sistema solar** e **guardaram-se** esse dados em estruturas. Podemos, quando for necessário, chamar esses dados **noutros programas** que os usem.
- Escreveram-se as **funções** para a **álgebra dos números complexos** para um exercício mas, elas podem igualmente ser vir a ser usadas **noutras aplicações** que **utilizem complexos**.
- Aliás, as **funções** de **C**, utilizadas até aqui, foram escritas por alguém que as **guardou** e **juntou** em **bibliotecas**.
- O nosso objectivo hoje é esboçar o início de uma **biblioteca pessoal**, isto é, um **conjunto de funções** que podem ser **guardadas** e **posteriormente usadas** em diversas aplicações.

Biblioteca Pessoal ('Utils.c', 'Utils.h' e 'TestProg_01.c')

- Podemos começar o nosso projecto de uma **biblioteca pessoal** por um ficheiro '**Utils.c**' no qual incluímos as funções:
  - **void tsrand** (**void**);
    Executa a função **srand** que inicializa a '**seed**' do **gerador dos números aleatórios** tendo, como argumento, o **instante actual** dado pelo resultado da função **time** (**NULL**).
  - **double xrand** (**double max**);
    Esta função recebe como argumento a **escala** pretendida dos aleatórios ([**0**,**max**]) e devolve um **aleatório** do tipo '**double**' naquele intervalo.
  - **float fxrand** (**float max**);
    Função idêntica à anterior '**xrand**' mas destina-se ao tipo '**float**'.
- Uma vez **escritas estas funções**, devemos criar um ficheiro '**.h**' a ela associado, '**Utils.h**' em que incluímos os **protótipos** daquelas **funções**.

Biblioteca Pessoal ('Utils.c', 'Utils.h' e 'TestProg_01.c')

- Como se pode ver, para além dos **protótipos** das **funções**, foram incluídas em '**Utils.h**' algumas **macros** úteis:
  - É **testado** se os valores lógicos '**TRUE**' e '**FALSE**' se encontram definidos e, no caso de não estarem são definidos.
  - #**define** **QUAD**(**x**) ((**x**) * (**x**))
    '**QUAD**' é assim uma **macro** que tem como resultado o **quadrado do número** dado.
  - #**define** **DELTA**(**a**,**b**) (((**a**) == (**b**)) ? **1** : **0**)
    Que define, como **macro**, a **função delta de Kronecker**.
  - #**define** **freeNull**(**x**) {**free**(**x**); **x**=**NULL**;}
    Esta **macro** recebe como argumento um **ponteiro**, aplica-lhe a função **free** que **liberta a memória** que lhe foi atribuída e reinicializa-o em '**NULL**'.
- Criámos assim em '**Utils.c**' e em '**Utils.h**' os nossos primeiros utilitários.

Biblioteca Pessoal ('TestProg_02 e 03.c')

De um modo análogo, podemos criar **outro ficheiro**, em que guardamos as **funções** e **macros** para manipular **vectores** e **matrizes** – '**UtilVect.c**' e '**UtilVect.h**'. São elas para **vectores**:

- **void *vec_new (unsigned int num, size_t nbytes);**
  Cria um vector com '**num**' elementos cada um deles com '**nbytes**' bytes e retorna um **ponteiro** para o vector criado, ou '**NULL**' em caso de erro.
- **double vec_modulo (double *vect, unsigned int num);**
  Calcula a **módulo do vector** (**vect**) para '**double**'.
- **double vec_pint (double *vect1, double *vect2, unsigned int num);**
  Calcula o **produtor interno** dos vectores '**vect1**' e '**vect2**' para '**double**'.

Biblioteca Pessoal ('TestProg_04.c')

E para **matrizes**:

- **void **matx_new (unsigned int ncols, unsigned int nrows, size_t nbytes);**

  Cria uma matriz com '**ncols**' e '**nrows**' em que cada elemento tem '**nbytes**' bytes e retorna um **ponteiro** para a matriz criada, ou '**NULL**' em caso de erro.

- **double matx_trac (double **matriz, unsigned int num);**

  Calcula o traço da matriz '**matriz**' para '**double**'.

O ficheiro '**UtilsVect.h**' tem ainda as **macros** para **libertar memória** e **inicializar a zero** os **vectores** e as **matrizes**:

- **#define vec_free(v) {free(v); v=NULL;}**
  #define matx_free(m) {free(m[0]); free(m); m=NULL;}
- **#define vec_clean(type,v,n) memset(v,0,n*sizeof(type))**
  #define matx_clean(type,m,nl,nc) vec_clean(type,m[0],nl*nc)

Como Criar uma Biblioteca (Library)

- Quando se tem **diversos ficheiros** que se vão usar como um conjunto, como é o caso de uma **biblioteca de uso diverso**, é conveniente **criar** um **arquivador** que junta (**arquiva**) os **ficheiros compilados** num só.
- O **arquivo** assim criado deverá ser **posteriormente utilizado** pelo **compilador** na operações de '**ligação**' dos constituintes do programa ('*linkagem*').
- O comando de '**unix**' que nos permite executar estas tarefas é '**ar**'. Podemos então **juntar** os nossos ficheiros (**já compilados**):

  ar rv libPessoal.a Utils.o UtilsVect.o

  em que:

  - **r**: **insere** os **ficheiros** no **arquivo**, substituindo-os caso existam;
  - **v**: **informa** sobre a acção efectuada;

Como Criar uma Biblioteca (Library)

- Uma vez criado o arquivo '**libPessoal.a**', devemos **criar um índice** contendo todos os **símbolos** (nomes, funções, variáveis, etc.) definidos nos ficheiros do arquivo:

  ranlib libPessoal.a

  isto acelera o processo de '**linking**' da biblioteca e permitir às **funções** nela contidas **chamarem-se umas às outras** independentemente da sua posição no arquivo.
- Note-se que as **duas operações** anteriores podem ser reunidas **numa só** acrescentando a opção '**s**' aos qualificadores de '**ar**':

  ar rsv libPessoal.a Utils.o UtilsVect.o
- Para **visualizar o conteúdo** do arquivo '**libPessoal.a**', pode usar-se o comando '**nm**' (lista dos símbolos):

  nm libPessoal.a
- Para mais **informações** sobre estes programas ver '**FreeBSD Man Pages**' (em '**Bibliografia**' no **site da cadeira**).

Bibliotecas ('Library')

- Em termos genéricos uma **biblioteca** é um **ficheiro** que contém outros **ficheiros** que, por sua vez, têm **código compilado**. E, estando **indexada**, é fácil encontrar os **símbolos** nela definidos.
- Podemos **classificar** as bibliotecas em **dois grupos**:
  - **Bibliotecas com ligação estática** ('static libraries' – "**.a**"): na operação de **link** de um programa, o **código da biblioteca** fica **integrado** no ficheiro **executável**.
  - **Bibliotecas com ligação partilhada** ('shared libraries' – "**.so**"): durante o **link**, é **verificado** se todos os **símbolos** exigidos estão definidos no **código do utilizador** ou nas **bibliotecas** invocadas. Ao **executar** o programa, o **sistema verifica** as **bibliotecas** associadas e **liga-as** ao programa ('**dynamic loader**').

Bibliotecas ('Library')

- Na maioria das **utilizações**, em que existe **código comum** a diversos programas, usam-se '**shared libraries**', excepto quando, por **razões de segurança** ou **emergência**, se quer garantir o **funcionamento correcto** dos programas.
- Para criar uma **biblioteca partilhada** ('**shared library**') chamada '**libPessoal.so**' a partir dos dois ficheiros objecto ('**.o**') criados faz-se:

gcc --share -o libPessoal.so Utils.o UtilsVect.o

- Dependendo dos sistemas, poderá ou não ser feita a compilação dos ficheiros '**.c**' com a opção '**-fPIC**' (position-independent code).

Bibliotecas ('Library')

- Na **compilação** é indica a biblioteca com '**-l**' e o nome da **biblioteca** sem o '**lib**';
- Caso a **biblioteca** não esteja numa das pastas de bibliotecas do sistema (por exemplo, '**/usr/lib/**', em unix), tem de se indicar a pasta em que ela se encontra, '**-Lpasta**', assim,

```
gcc -o prog_01 prog_01.c -L./ -lPessoal -lm
```

- Para a **execução** do programa é agora necessário **incluir** a **pasta** em que está a biblioteca na lista das pasta em que o sistema irá procurar bibliotecas.
- Tal pode ser feito acrescentando a pasta em que ela está à **variável de ambiente** '**LD_LIBRARY_PATH**':

```
export LD_LIBRARY_PATH =¨LD_LIBRARY_PATH:./¨
```

- Finalmente, pode **executar-se** o programa.

Criação de uma 'Makefile'

- Por vezes a **compilação** e a '*linkagem*' são **operações** que incluem um **conjunto complexo de operações**, por isso, é extremamente conveniente **utilizar instrumentos** que **executem** correctamente essas tarefas.
- Em '**unix**' existem diversas **ferramentas** capazes de executar aquelas tarefas. Aqui iremos usar uma das mais comuns, o comando '**make**'.
- Uma '**Makefile**' é organizada por **tarefas específicas**, chamadas **comandos**, endereçadas pelas suas **etiquetas**.
- Os comandos são separados entre si por linhas vazias.

Criação de uma 'Makefile'

- Assim, cada **comando** é constituído por:
  - Uma **etiqueta** ('**label**'): **sequência de caracteres** que começa no **início da linha** e termina em '**:**'.
  - Depois dos '**:**' devem ser colocadas as **sequências de comandos** que o devem **anteceder**, ou que **decidem** da sua execução;
  - As **tarefas a executar** seguem-se, **sem** linhas em branco, **uma** por linha e **iniciadas** por um '**TAB**'.
- Para além dos **comandos** podem declarar-se **variáveis** que permitem definir os **comandos** de um modo conveniente. Essas **variáveis** definem-se através do sinal '**=**'. Exemplos:

  CC = gcc
  LINK_FLAGS = -lm
- A utilização das variáveis faz-se do seguinte modo:

  $(CC) $(CFLAGS) -c UtilsVect.c
- É sempre uma **boa regra** definir uma **variável** associada a cada um dos **comandos de unix** a utilizar.

Criação de uma 'Makefile'

- O comando '**make**' (sem mais argumentos) lê o **ficheiro** de nome '**Makefile**' e **executa** o **conjunto de instruções** associadas à etiqueta ('label') '**all:**'.
- Para ler outro ficheiro faz-se:

  make -f nomeFicheiro

- Para executar uma tarefa (**etiqueta**) dar-se o comando

  make [**-f nomeFicheiro**] **nomeEtiqueta**

  em que os parêntesis rectos indicam que é opcional.