Programação modular

Quando começamos a programar projectos com alguma dimensão, é usual dividir o código do mesmo por diversos ficheiros a que chamamos módulos. Um módulo é uma colecção de funções que realizam tarefas relacionadas. Podemos dividir um módulo em duas partes:

- **Parte pública do módulo:** Definição de estruturas de dados e funções que devem ser acedidas fora do módulo. Estas definições estão no ``header file'' por convenção. Os "header files" têm extensão .h.
- **Parte privada do módulo:** Tudo o que é interno ao módulo (não visivel pelo mundo exterior). Ficheiro com extensão .c.

A ideia é dividir um problema em secções diferentes e escrever um ``header file'' para cada um dos ficheiros C com as definições de todas as funções e variáveis globais desse ficheiro que devem ser usadas por outros módulos.

Esta abordagem tem diversas vantagens:

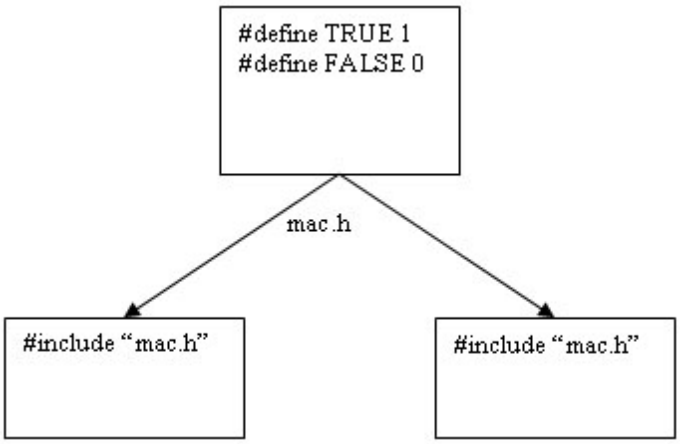
- Estrutura do programa fica mais clara ao agrupar funções e variáveis relacionadas num mesmo ficheiro.
- Possibilidade de compilar cada um dos módulos separadamente, poupando tempo.
- A reutilização das funções é facilitada.

Divisão de um programa em módulos

Quando dividimos o código de um programa em diversos módulos, levantam-se algumas questões:

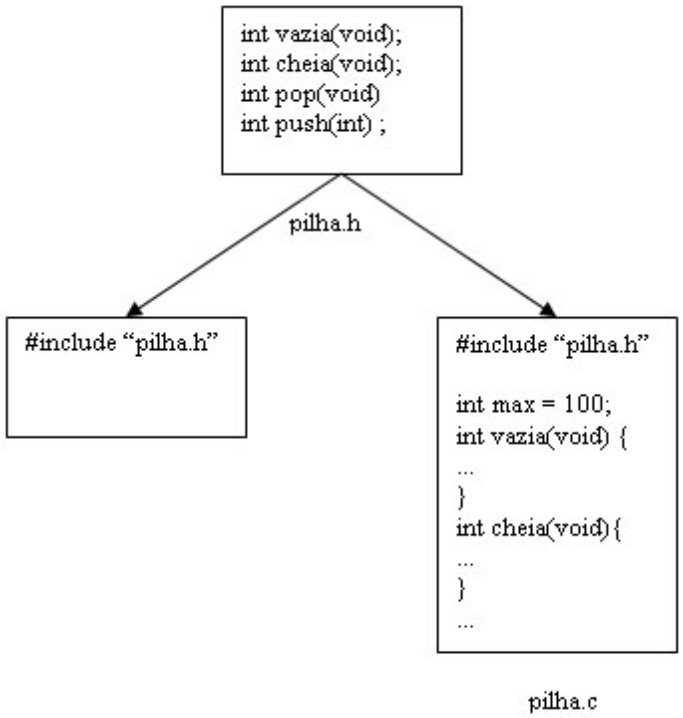
- Como é que uma função de um módulo acede a uma outra de outro módulo?
- Como é que uma função de um módulo acede a uma variável de outro módulo?
- Como é que uma macro é partilhada por diversos módulos?

A resposta está na directiva *include* do pré-processador do C. A imagem seguinte ilustra a inclusão do ficheiro ("header file") *mac.h* noutros dois:



Aqui já se vê uma vantagem do uso de módulos, as definições comuns aos dois ficheiros estão armazenadas no "header file" *mac.h* , quaisquer alterações neste são automaticamente propagadas aos ficheiros que o incluem.

No caso dum módulo necessitar de aceder a funções definidas noutro, os protótipos respectivos devem estar no "header file" correspondente. Por exemplo, posso programar um módulo constituído por dois ficheiros, *pilha.h* e *pilha.c*, referentes à implementação de uma pilha. O *pilha.c* tem a implementação das funções de manipulação da pilha e o *pilha.h* tem os protótipos das funções implementadas em *pilha.c*. Agora implemento outro módulo, para usar a pilha, neste tenho que incluir os protótipos das funções que vou usar. A imagem seguinte ilustra esta ideia:



Extern

Partilhar variáveis é parecido com partilhar funções. As funções são definidas num ficheiro e um protótipo é partilhado por outros módulos. As variáveis também são definidas algures, por exemplo:

```
int i;
```

e partilhadas por outros módulos com a declaração:

```
extern int i;
```

O *extern* indica que uma variável está defina fora do módulo corrente (não a define!!!). Podemos então distinguir os seguintes casos:

- *extern variável*: definida noutro ficheiro.
- "*nenhum*" *variável:* pública.
- *static variável:* local ao ficheiro (privada).

Static

Para variáveis/ficheiros declarados globalmente, significa privado para o ficheiro local. No caso de estar declarado localmente significa que a variável está declarada em memória estática em vez de estar na pilha.

Um exemplo simples

O ficheiro *main.c* inclui funções definidas em *escreve_string.c* e declaradas no seu ``header file'' *escreve_string.h*.

main.c:

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

char *OutraString = "Ola a todos!";

main(){

  printf("\n Começo....\n");
  escreveString(A_STRING);
  printf("\n Fim....\n");
}
```

escreve_string.h:

```
#define A_STRING "Olá mundo"
void escreveString(char *);
```

O escreve_string.c:

```
extern char *OutraString;

void escreveString(char *estaString){

  printf("\n%s\n",estaString);
  printf("\n A variavel global: %s declarada no módulo e definida no
main",OutraString);

}
```

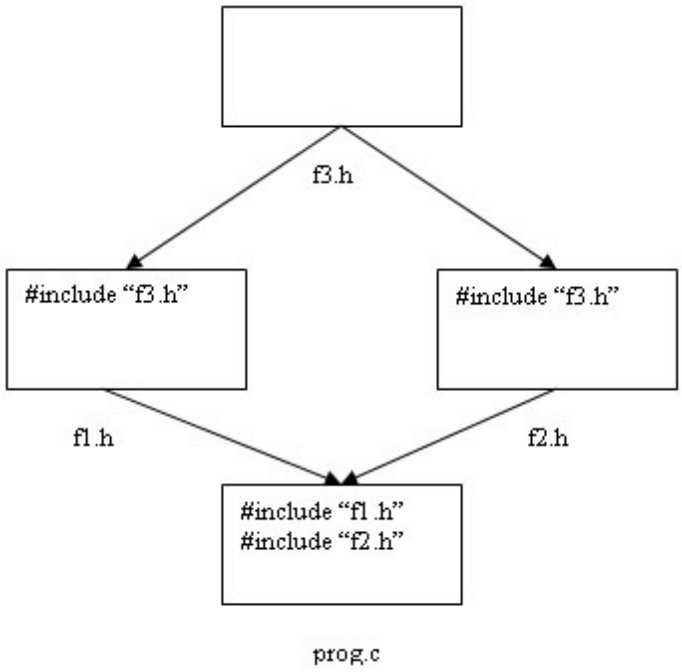
Compilar o programa

A compilação dos diferentes ficheiros num só executável faz-se da seguinte forma:

```
gcc -o mod main.c escreve_string.c
```

Protecção de "header files"

A inclusão de um mesmo "header file" mais do que uma vez origina problemas. No entanto esta situação não é muito rara. Por exemplo, se f1.h incluir f3.h, f2.h incluir f3.h e prog.c incluir f1.h e f2.h, f3.h será incluído duas vezes o que origina um erro de compilação. A imagem seguinte ilustra esta ideia:



Para resolver este problema recorremos mais uma vez ao pré-processador. Por exemplo, se quisermos proteger o "header file" do exemplo anterior escrevemos o seguinte:

```
#ifndef ESCREVE_STRING_H
#define ESCREVE_STRING_H

#define A_STRING "Olá mundo"
void escreveString(char *);

#endif
```

Este "header file" passa a ser identificado por ESCREVE_STRING_H e só será incluído uma vez.

O utilitário make

O **make** é um utilitário que permite automatizar o processo de compilação de múltiplos ficheiros em simultâneo. Conforme o número de ficheiros aumenta, mais argumentos o programador tem que escrever no momento da compilação. Mudanças num só ficheiro levam à recompilação de todo o projecto e consequente perda de tempo, com o **make** só os ficheiros alterados e os que dependem deles são recompilados. O ficheiro *Makefile* inclui uma série de secções onde são definidas macros e regras para a compilação.

Exemplo

Se temos os ficheiros: *ola.c*, *ola.obj* e *ola.exe* e for feita uma alteração no *ola.c*, o **make** reconhece essa alteração e cria novamente os outros ficheiros. Se estivermos a tratar de vários ficheiros ``obj'' de diversas fontes que constituem um único executável e for feita uma alteração em algum ficheiro de código fonte, o **make** trata de compilar somente o ``obj'' correspondente e o executável.

Regras

O *MakeFile* é constituído por regras de dependência e de construção. Uma *regra de dependência* tem 2 partes:

lado esquerdo (alvo) : lado direito (ficheiros de cujo alvo depende)

Se o alvo está desactualizado no que diz respeito aos seus constituintes então as regras de construção que seguem as regras de dependência são verificadas.

Execução do utilitário make

Quando se executa o **make** as seguintes tarefas são executadas:

- O *Makefile* é lido. O *Makefile* indica que objectos e bibliotecas necessitam de ser ``linkados'' e quais ``headers'' e ``sources'' necessitam ser compilados para criar cada objecto.
- É verificada a data e hora de cada ficheiro ``obj'' e é comparada com a data e hora de cada ``source'' e ``header'' do qual depende. Recompila caso necessário.
- A data e hora de cada objecto é comparada com cada executável. Recompila caso necessário.

Criação de um Makefile

```
#
# Makefile
#

CC = gcc
CFLAGS =# -g

all: mod

mod: main.o escreve_string.o
	$(CC) $(CFLAGS) -o mod main.o escreve_string.o
```

```
main.o: main.c escreve_string.h
  $(CC) $(CFLAGS) -c main.c

escreve_string.o: escreve_string.c
  $(CC) $(CFLAGS) -c escreve_string.c
clean:
    rm -rf mod main.o escreve_string.o
```

Um exemplo mais completo envolvendo estes conceitos pode ser obtido aqui.

Apêndice A: Pré-processador

O pré-processador é a primeira fase de compilação de um programa em C. O pré-processador manipula apenas o texto do programa fonte servindo-se de uma linguagem própria bastante poderosa que pode vir a tornar-se muito útil para o programador. Todas as directivas do pré-processador começam com o caractere #.

Exemplo:

```
#define begin {
#define end }
```

#define

Usado para definir constantes ou substituições mais poderosas denominadas *macros*.

```
#define nome texto_de_definição
```

Exemplo:

```
#define TRUE 1
#define FALSE 0
#define max(A,B) ((A) > (B) ? (A) : (B))
```

#undef

Sempre que quisermos redefinir um nome com outra directiva *#define* , temos que o indefinir primeiro com *#undef*.

#include

Lê um arquivo de texto e inclui o seu conteúdo no local da directiva. Duas formas possíveis:

```
#include <nome_arquivo>
#include "nome_arquivo"
```

O uso de <> indica que o pré-processador deve procurar o ficheiro na directoria de bibliotecas do sistema. O uso de "" indica que o pré-processador deve procurar o ficheiro no directório actual em primeiro lugar. No contexto do programação em módulos, usa-se sempre "" para incluir ficheiros do projecto em causa.

O #if - inclusão condicional

O #if avalia uma expressão inteira e executa uma acção baseada no resultado (interpretado como TRUE ou FALSE). Esta directiva é sempre terminada com a palavra *#endif*. Entre *#if* e *#endif* poderão estar as directivas *#else* e *#elif*. Outras formas da directiva *#if* são:

- *#ifdef nome* - verifica se o nome está definido com uma directiva *#define*.
- *#ifndef nome* - verifica se o nome não está definido com uma directiva *#define*.

Exemplos:

```
#ifdef TURBOC
     #define INT_SIZE 16
#else
      #define INT_SIZE 32
#endif

#if SYSTEM == MSDOS
  #include <msdos.h>
#else
  #include<default.h>
#endif
```

Jorge Coelho - ISEP 2002