Modularidade em Programas C

Laboratórios de Informática III
Guião #3-1

Departamento de Informática
Universidade do Minho
Autor: Luís Ferreira
Baseado no trabalho de: Fernando Mário Martins e Vítor Francisco Fonte

Outubro de 2023

Conteúdo

1 Introdução

A construção e o desenvolvimento de programas de dimensões já consideráveis, independentemente da linguagem, envolvem a utilização de técnicas particulares que possam garantir que os projetos de software, apesar das suas dimensões, são controláveis e geríveis, quer no seu desenvolvimento, quer no seu teste e manutenção.

Os conceitos de modularidade e encapsulamento são, em Engenharia de Software, cruciais para que o desenvolvimento de software se faça de forma controlada e reutilizável, e que o código gerado seja robusto, sendo os eventuais erros de deteção fácil e de fácil correção. O conceito de modularidade foca-se na divisão do código de um programa em vários ficheiros, seguindo determinados critérios de coerência. No caso da linguagem C, cada módulo é normalmente constituído por um ficheiro que implementa as funções do módulo (e.g., *"matematica.c"*) e um ficheiro que define a interface desse módulo, ou por outras palavras, as funções implementadas pelo módulo que podem ser acedidas e utilizadas por outros módulos (e.g., *"matematica.h"*). Já o conceito de encapsulamento trata da restrição de acesso aos dados de cada módulo. Este segundo será abordado no guião 3-2.

Apresentam-se neste texto algumas das construções da linguagem C que devem ser usadas por forma a garantir que o desenvolvimento de aplicações de média e grande escala em C seja realizado à luz dos princípios da modularidade.

2 Porquê dividir o código-fonte?

Quando se começa a programar em C é usual que os pequenos programas que desenvolvemos tenham todo o seu código-fonte num único ficheiro, possivelmente grande. Edita-se o código, compila-se e executa-se e, assim sendo, para quê preocuparmo-nos em dividi-lo por diferentes ficheiros?

Esta é a mentalidade associada à designada "small-scale programming". Em "large-scale programming" tudo é muito mais complexo e a modularidade, sob a forma da divisão do código em vários ficheiros fonte, torna-se fundamental para lidar com a complexidade do código e das arquiteturas dos projetos.

Vejamos algumas dessas razões:

- Permite que num dado projeto pessoas diferentes implementem partes diferentes do mesmo, depois de definidas as Application Programming Interfaces (APIs) (ficheiros .h) de cada parte;
- É muito mais fácil isolar erros de compilação ou de execução;
- É muito mais fácil reutilizar código codificado num ficheiro ou módulo autónomo;
- Uma eventual mudança no código de um ficheiro implica apenas a recompilação desse ficheiro (cf. Makefile);
- É mais fácil editar e navegar no código-fonte em ficheiros mais pequenos e autónomos;
- Estaremos a aplicar em C o que é normal em C++, C# ou Java, ainda que sem termos hierarquias, herança e polimorfismo.

3 Módulos: implementação (*.c*) e cabeçalho (*.h*)

Consideremos a título de exemplo a criação de um programa capaz de criar uma stack de inteiros e realizar as usuais operações de inicializar, inserir e remover dados da stack. O módulo será constituído pelos seguintes ficheiros:

- `stack.c`: o código-fonte do módulo stack;
- `stack.h`: a header file do módulo stack, que define o que é invocável do exterior, i.e. visível.

Eventualmente será necessário também um ficheiro main.c que faça uso do módulo stack criado.

3.1 A importância dos ficheiros cabeçalho *.h*

Um ficheiro cabeçalho – em Inglês, *header* – é um ficheiro `.h` que contém as funções que o módulo deseja exportar, ou seja, tornar acessível a outros módulos. Assim, em C, um ficheiro `.h` define a API do respetivo ficheiro `.c`, declarando em particular os designados protótipos ou **declarações** das funções a exportar.

Para que outros módulos possam utilizar as funções tornadas públicas por um módulo, basta que incluam o ficheiro cabeçalho do módulo que querem utilizar, e.g., `#include "A.h"`.

A título de exemplo, o ficheiro `stack.h` abaixo representa a interface de um módulo que implementa a stack, e que expõe funcionalidades para inicializar uma stack, colocar um novo elemento na stack, remover um elemento da stack e indicar se a stack se encontra ou não vazia.

```
1  //stack.h
2    #ifndef STACK_H
3    #define STACK_H
4
5    void initStack(void);
6    void pushOntoStack(int number);
7    int popFromStack(void);
8    int stackEmpty(void);
9
10   #endif
```

Note-se o uso das diretivas `#ifndef STACK_H`, `#define STACK_H` e `#endif`. Estas devem estar presentes em todos os ficheiros cabeçalho, alterando o nome da definição de acordo com o nome do módulo (neste caso *STACK_H*), de forma a evitar que surjam problemas de compilação. Em programas grandes estas diretivas permitem evitar que um ficheiro de implementação `.c` inclua um ficheiro de cabeçalho `.h` várias vezes.

3.2 Ficheiros de implementação *.c*

Os ficheiros de implementação contêm a **definição**, ou implementação, das funções do módulo, contendo todo o código-fonte necessário para executar as funcionalidades fornecidas pelo mesmo. No ficheiro de implementação podemos encontrar não só as funções declaradas no cabeçalho do módulo, mas também outras funções, tipos, macros e variáveis que sejam exclusivamente internas ao módulo. Como iremos ver no guião de encapsulamento #3-2, é a capacidade de conseguirmos definir elementos que não são exportados através do ficheiro cabeçalho que nos permite garantir o encapsulamento em módulos implementados na linguagem C.

O ficheiro de implementação, e.g., `"stack.c"` deverá incluir o seu próprio ficheiro cabeçalho `#include "stack.h"`. Existem várias razões para esta prática: o compilador irá detetar discordâncias entre as declarações no cabeçalho e as definições no ficheiro de implementação (e.g., argumentos diferentes); tipos definidos no cabeçalho estarão disponíveis para uso no ficheiro de implementação; as funções podem ser definidas no ficheiro de implementação em qualquer ordem; entre outros. Abaixo encontra-se um exemplo do ficheiro de implementação da stack, `"stack.c"`.

```
1    #include <stdio.h>
2
3    #include "stack.h"
4
5    /* Inclui os seus próprios protótipos. Isto é uma boa prática.
6       Sendo os ficheiros .c compilados em separado, quanto mais
7       informação sintática tiver o compilador melhor. */
8
9    /* Variáveis que implementam a stack de inteiros.
10      São globais no ficheiro stack.c mas privadas fora deste.
11      stackPointer indica a próxima posição livre.
12      Se stackPointer == 0 a stack está vazia.
13   */
14
15   #define MAX_STACK_SIZE 500
16
17   static int stack[MAX_STACK_SIZE];
18   static int stackPointer = 0;
19
20   /*****************************************************/
21
```

```
22  void initStack(void) {
23    stackPointer = 0;
24  }
25
26  /*****************************************************/
27
28  void pushOntoStack(int number) {
29    stack[stackPointer] = number;
30    stackPointer++;
31  }
32
33  /*****************************************************/
34
35  int popFromStack(void) {
36    stackPointer--;
37    return stack[stackPointer];
38  }
39
40  /*****************************************************/
41
42  int stackEmpty(void) {
43    if (stackPointer > 0)
44      return 0;     /* false - stack não vazia */
45    else
46      return 1;     /* true - stack vazia.*/
47  }
```

A combinação do ficheiro cabeçalho `"stack.h"` e do ficheiro de implementação `"stack.c"` constitutem o módulo `stack`. Em baixo é apresentado um exemplo da utilização do módulo stack por um outro módulo.

```
1   //main.c
2     #include <stdio.h>
3     #include <stdlib.h>
4
5     #include "stack.h"
6
7     int main(int argc, char* argv[]) {
8       int numero = 10;
9
10      // Inicialização da stack
11      initStack();
12
13      //Adição de um número
14      pushOntoStack(numero);
15
16      //Remoção de um número, verificando que a stack não está vazia
17      if( !stackEmpty() ) { //Se a stack não estiver vazia..
18        {
19          /* Faz pop & escreve no ecrã */
20          numero = popFromStack();
21          printf("%d\n", numero);
22        }
23      }
24    }
```

4 Como separar um programa em diferentes tipos de módulos

Cada módulo desempenha uma função diferente no sistema, de tal forma que cada um destes se torna logicamente independente dos restantes. A primeira categoria de módulo a considerar num programa C é o módulo de **estrutura de dados**.

4.1 Módulos de estruturas de dados

Um módulo deste tipo implementa uma estrutura de dados e todas as operações sobre a mesma (abstraindo de quem o usa os detalhes de implementação). Exemplos de módulos de estruturas de dados incluem módulos que implementem: listas, tabelas de hash, árvores, arrays, entre outros. O exemplo dado anteriormente da stack é também um módulo de estrutura de dados.

Estas estruturas básicas de dados são usadas recorrentemente em programas em C, e as suas propriedades básicas de organização em memória, pesquisa, inserção e remoção de dados, entre outros, mantêm-se constantes. Assim, ao isolarmos o comportamento de uma estrutura de dados no seu próprio módulo, podemos reutilizar a mesma implementação em diferentes contextos, evitando ter que a programar mais do que uma vez. Veja-se o exemplo abaixo da utilização do módulo stack em dois contextos diferentes.

```c
//fila_LIFO_int.c
#include "stack.h"

void inicializar_fila(){
  // Inicializacão da stack
  initStack();
}

void adicionar_a_fila(int numero){
  //Adição de um número
  pushOntoStack(numero);
}

int remover_da_fila(){
  numero = popFromStack();
  return numero;
}
```

```c
//pilha_de_paletes_armazem.c
#include "stack.h"

void inicializar_pilha_armazem(){
  // Inicialização da stack
  initStack();
}

void colocar_palete(int idPalete){
  //Adição de um número
  pushOntoStack(idPalete);
}

int remover_palete(){
  idPalete = popFromStack();
  return idPalete;
}
```

Nos exemplos acima vemos uma implementação para uma fila Last-In-First-Out (LIFO) de inteiros, bem como de um módulo que descreve uma pilha de paletes num armazém. Apesar de se tratarem de contextos bastante diferentes,

ambos podem ser representados por uma stack. Ao isolarmos a implementação da stack no seu próprio módulo evitamos a necessidade de a implementarmos para cada um destes casos de uso, podendo ambos utilizar a mesma implementação.

Estruturas de dados genéricas Uma estrutura de dados genérica permite-nos guardar dados de qualquer tipo, o que, comparando com uma estrutura específica a um único tipo de dados, permite uma maior reutilização da mesma. Se repararmos na implementação anterior da stack, esta implementação apenas permite que guardemos na stack valores do tipo inteiro (`int`). No entanto, através do uso do tipo `void *` podemos guardar qualquer tipo de dados na stack. Vejamos as modificações a serem feitas às funções push, pop e à estrutura da stack para que tal seja possível:

```
1  //stack.h
2    #ifndef STACK_H
3    #define STACK_H
4
5    void initStack(void);
6    void pushOntoStack(void * number);
7    void * popFromStack(void);
8    int stackEmpty(void);
9
10   #endif
```

```
1  //stack.c
2    #include <stdio.h>
3    #include "stack.h"
4
5    #define MAX_STACK_SIZE 500
6
7    static void * stack[MAX_STACK_SIZE];
8    static int stackPointer = 0;
9
10   /*******************************************************/
11
12   void initStack(void) {
13     stackPointer = 0;
14   }
15
16   /*******************************************************/
17
18   void pushOntoStack(void *data) {
19     stack[stackPointer] = data;
20     stackPointer++;
21   }
22
23   /*******************************************************/
24
25   void *popFromStack(void) {
26     stackPointer--;
27     return stack[stackPointer];
28   }
29
30   /*******************************************************/
31
32   int stackEmpty(void) {
33     if (stackPointer > 0)
34       return 0;     /* false - stack não vazia */
35     else
36       return 1;     /* true - stack vazia.*/
```

```
37   }
```

Como consequência do uso de estruturas de dados com tipo genéricos, o módulo que implementa a estrutura deixa de conhecer o tipo dos dados que guarda. Como tal, certas operações que exigem conhecimento sobre o tipo dos dados passam a necessitar do auxílio de funções por apontador. Veja-se o exemplo abaixo para uma função que imprime para a consola o conteúdo da `stack`.

```
1  //On stack.c
2    void printStack(void (*printFunc) (void * data)){
3      printf("Stack:\n")
4      for (int i = 0; i < stackPointer; i++){
5        printFunc(stack[i]);
6        if(i + 1 < stackPointer){
7          printf("->");
8        }
9      }
10   }
```

```
1  //Module that is using stack
2
3  void printChar(void * data){
4    char * character = (char *) data;
5    printf("%c", *character);
6  }
7
8  void testStack(){
9    char string[] = "Hello world";
10   for(int i = 0; i < strlen(string); i++){
11     pushOntoStack(&string[i]);
12   }
13
14   printStack(printChar);
15
16 }
```

Uma vez que o módulo externo é que define e conhece o tipo de dados que está a ser guardado pela stack, é também ele que tem que implementar a função que é capaz de imprimir os elementos da mesma. O mesmo irá acontecer sempre que quisermos implementar na stack uma função que dependa do tipo dos dados a ser guardados.

Estruturas de dados instanciáveis De forma semelhante a uma estrutura de dados que é construída para tipos genéricos de dados, uma estrutura pode e deve também ser instanciável para maximizar a sua reusabilidade. Uma estrutura de dados instanciável permite que várias instâncias de uma mesma estrutura possam coexistir num só momento, de forma a que diferentes módulos possam fazer uso da mesma estrutura em simultâneo. No exemplo anterior, a stack recorre a duas variáveis globais, `stack` e `stackPointer`, para guardar os elementos da stack. Desta forma, apenas uma `stack` pode existir a cada momento, sendo os seus dados guardados nestas variáveis. Se quisermos, por exemplo, usar uma stack num módulo que gere contas, e uma outra stack numa fila de espera, não seria possível guardar os dados de cada um em estruturas separadas. Poderiam inclusive ocorrer acessos de um módulo aos dados do outro, de forma inadvertida.

Assim, para que cada módulo externo possa fazer uso da sua própria stack necessitamos, em primeiro lugar, de criar um tipo de dados `Stack` para que com ele se possam declarar múltiplas variáveis deste tipo de dados. Ao criar uma variável deste novo tipo estaremos a criar uma nova `Stack`. Para que as funções do módulo consigam aceder à `Stack` pretendida, a variável deve ser passada como argumento para cada uma das funções. Veja-se o exemplo abaixo.

```
1  // node.h
2  #ifndef NODE_H
```

```c
#define NODE_H

typedef struct node {
  void* data;
  struct node* next;
} Node;

// Cria um novo Node
Node* nodeCreate(void* data);

#endif
```

```c
// node.c
#include "node.h" #include <stdlib.h>
Node* nodeCreate(void* data) {
    Node* new = malloc(sizeof(Node));
    new->data = data;
    new->next = NULL;
    return new;
}
```

```c
// stack.h
#ifndef STACK_H
#define STACK_H

#include "node.h"
typedef struct stack {
  int size;
  Node* top;
  void (*elemPrint)(void *); // função para imprimir um elemento da stack
} Stack;

Stack* stackCreate(void (*elemPrint)(void *));
void stackPush(Stack* stack, void* data);
void* stackPop(Stack* stack);
void stackPrint(Stack* stack);

#endif
```

```c
// stack.c
#include "stack.h"
#include <stdio.h>
#include <stdlib.h>
Stack* stackCreate(void (*elemPrint)(void *)) {
  Stack* stack = malloc(sizeof(Stack));
  stack->size = 0;
  stack->top = 0; // mesmo que NULL
  stack->elemPrint = elemPrint;
  return stack;
}

void stackPush(Stack* stack, void* data) {
    Node* node = nodeCreate(data);
    node->next = stack->top;
```

```
16        stack->top = node;
17        stack->size++;
18    }
19
20    void* stackPop(Stack* stack) {
21      if (stack->size == 0) {
22        return 0; // mesmo que NULL
23      }
24      Node* top = stack->top;
25      stack->top = top->next;
26      void* data = top->data;
27      free(top);
28      stack->size--;
29      return data;
30    }
31
32    void stackPrint(Stack* stack) {
33      Node* top = stack->top;
34      while (top) {
35        stack->elemPrint(top->data);
36        printf(" -> ");
37        top = top->next;
38      }
39      printf("x\n");
40    }
```

Evitando o uso de variáveis globais e recorrendo antes ao uso de um novo tipo que representa a nossa estrutura de dados, neste caso a estrutura `Stack`, a nossa estrutura passa a ser instanciável. Vejamos agora como podemos invocar duas instâncias de stack:

```
1     #include "stack.h"
2     #include <string.h>
3     #include <stdio.h>
4
5     void printInt(void *data){
6       int *num = (int *) data;
7       printf("%d",num);
8     }
9
10    void printChar(void *data){
11      int *character = (int *) data;
12      printf("%c",character);
13    }
14
15    int main(int argc, char *argv[]){
16      Stack * stack1 = stackCreate(printInt);
17      Stack * stack2 = stackCreate(printChar);
18
19      int num = 10;
20      char line[] = "Hello World";
21
22      stackPush(stack1, (void *) &num);
23      for(int i = 0; i < strlen(line); i++){
24        stackPush(stack2, (void *) &line[i]);
25      }
26
27      stackPrint(stack1);
28      stackPrint(stack2);
```

```
29  }
```

É de salientar que o uso de variáveis globais, por norma, restringe a reutilizabilidade do código, sendo preferível evitar o seu uso sempre que possível.

4.2 Módulos de Entidades do sistema

As entidades do sistema representam os componentes base do nosso programa. Cada entidade é um ator no sistema que representa uma unidade lógica independente. Por exemplo, num programa que gere uma oficina de carros, podemos ter as seguintes entidades: **peça**, que mantém informações básicas sobre o id, nome, modelo, versão e carros compatíveis com uma determinada peça; **reparação**, que mantém informações sobre o modelo e matrícula do carro a ser arranjado, peças envolvidas no arranjo, mecânico responsável pelo arranjo, etc.; **mecânico**, que indica o seu nome, data de nascimento, especialidade, id de empregado, anos de serviço e quanto recebe por hora de trabalho. De forma semelhante, um outro sistema que fizesse a gestão de táxis, por exemplo, poderia ter os seguintes módulos de entidade: utilizador, condutor, viagem, recibo, etc.

Cada módulo de entidade tem associado, por norma, um tipo de dados que a representa (e.g., `struct`). Uma entidade expõe através da sua API (ficheiro cabeçalho `.h`) funções para aceder às informações da entidade, bem como informações derivadas dessas. E.g., um módulo de *mecânico* pode expor uma função que retorna a sua idade, sendo esta calculada a partir da sua data de nascimento. Além disso, da mesma forma que uma estrutura de dados, o módulo de *mecânico* deve ser instanciável para que se possam ter múltiplas instâncias da mesma entidade. Veja-se o exemplo abaixo para uma entidade *mecânico*:

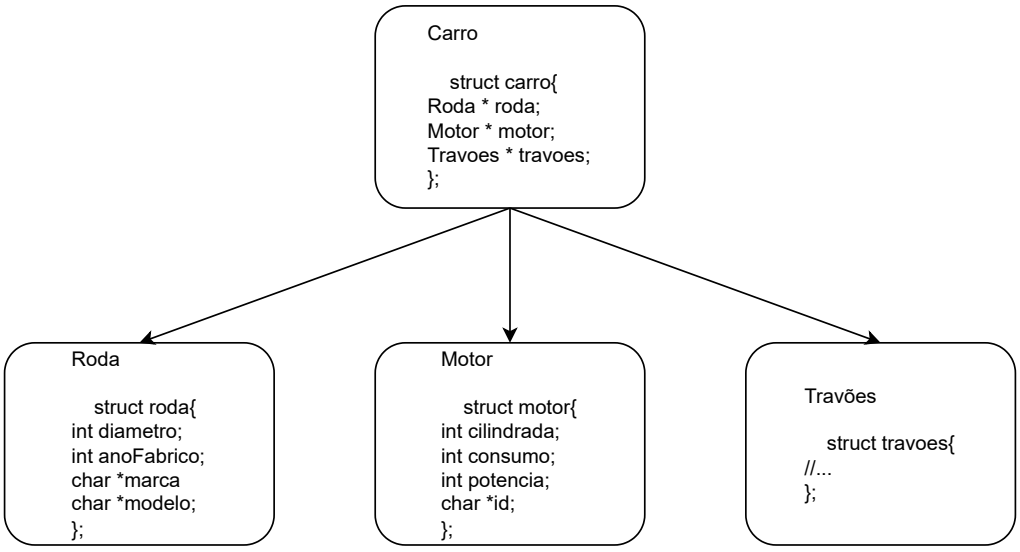
```c
//mechanic.h
typedef struct mechanic{
  long id;
  char *name;
  int hourlyRate;
  //Could perhaps be better handled using a Date data module!
  int birthYear;
  int birthMonth;
  int birthDay;
  //...

} Mechanic;

Mechanic * createMechanic(long id, char * name, int hourlyRate, char * birthDate);
void increase_hourly_rate(Mechanic * mechanic, int percentage);
int getAge(Mechanic *mechanic);
//...
```

```c
//mechanic.c
#include "mechanic.h"


Mechanic * createMechanic(long id, char * name, int hourlyRate, char * birthDate){
  Mechanic * mechanic = malloc(sizeof(Mechanic));
  mechanic->id = id;
  mechanic->name = name;
  mechanic->hourlyRate = hourlyRate;
  splitBirthDate(birthDate, &mechanic->birthYear, &mechanic->birthMonth, &mechanic->birthDay);

  return mechanic;
}

void increase_hourly_rate(Mechanic * mechanic, float percentage){
```

```
16    mechanic->hourlyRate = (1 + (percentage/100)) * mechanic->hourlyRate;
17  }
18
19  int getAge(Mechanic * mechanic){
20    int currentYear = get_current_year();
21    int age = currentYear - birthYear;
22
23    //Has mechanic already had his birthday?
24    if (mechanic->birthMonth < get_current_month()){
25      age -= 1;
26    }
27    else if(mechanic->birthMonth == get_current_month() && mechanic->birthDay < get_current_day){
28      age -= 1;
29    }
30    return age;
31  }
32
33  //...
```

Note que a maneira como foi codificado este módulo não garante o encapsulamento! Iremos ver porquê no guião de encapsulamento #3-2.

Os módulos de entidade podem ainda formar uma estrutura hierárquica, em que uma entidade se estrutura em sub-entidades. Assim, uma entidade, e.g., **carro**, pode conter referências para sub-entidades roda, motor, travões, etc. Veja-se o exemplo abaixo.



Inclusive é possível recorrer a estruturas de dados em entidades. E.g., a entidade mecânico poderia conter uma lista de números de telefone.

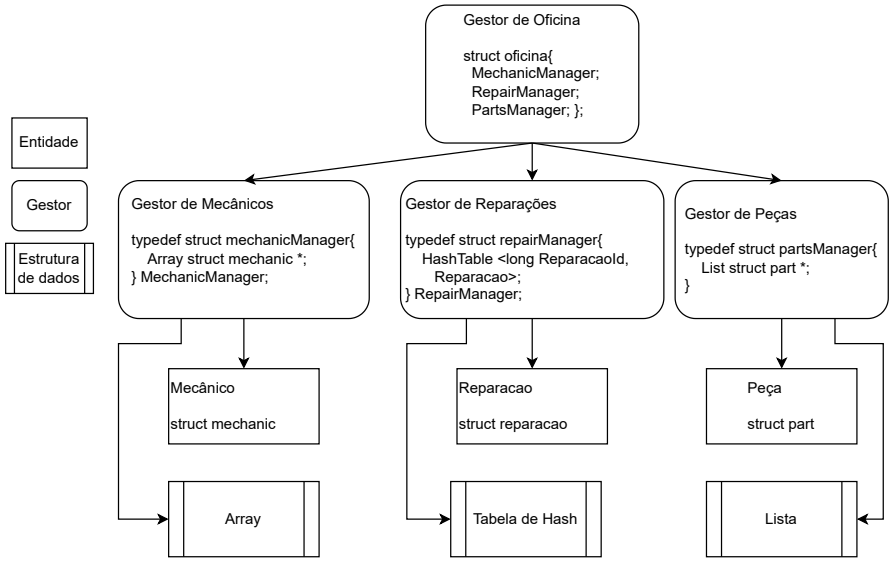
4.3 Módulos de Gestão do sistema

Os módulos de gestão fazem uso das estruturas de dados anteriormente referidas para guardar e gerir instâncias de entidades. Continuando o exemplo anterior de uma oficina de carros, podemos ter os seguintes gestores no nosso sistema: **gestor de peças**, que mantém uma lista das peças que existem, e que para cada peça contém informação sobre o número de peças em stock, e a sua localização na oficina; **gestor de reparações**, que mantém uma tabela de hash onde se encontram instâncias de carros que estão a ser ou já foram reparados, o tipo de reparação a ser feita, incluindo as peças necessárias, o mecânico atribuído à reparação e a data de entrada/saída do veículo na oficina; **gestor de mecânicos**, que contém um array dos mecânicos que trabalham na oficina.

Fazendo uso destas estruturas de dados bem como das funções expostas pelas entidades que gerem, os gestores implementam funções de mais alto nível que implicam conhecimento sobre múltiplas entidades do sistema. Por exemplo, o gestor de reparações pode responder a questões como: quantos carros estão na oficina neste momento, se a reparação do carro está concluída ou não, saber quantas reparações um cliente já fez na nossa oficina, entre outros.

Note-se que no exemplo anterior cada gestor faz uso de uma estrutura de dados diferente. A escolha da estrutura de dados de um gestor deve ser feita de forma apropriada ao tipo de funções que o gestor implementa. Da mesma forma, um gestor pode fazer uso de múltiplas estruturas de dados para organizar os mesmos dados, fazendo a cada momento uso da estrutura de dados que melhor se adequa à operação a efetuar. Por exemplo, para um gestor de reparações encontrar uma reparação em particular a partir de um id de reparação, o gestor pode organizar as reparações numa tabela de hash. Já para encontrar reparações que foram efetuadas entre duas datas, fazer uso de uma segunda estrutura em árvore poderá ser mais eficiente.

O sistema de oficina acima mencionado poderia ter a seguinte hierarquia de entidades, gestores e estruturas de dados:



Note-se que entidades, gestores e estruturas de dados podem formar estruturas hierárquicas entre si (na figura vemos isto a acontecer com os gestores).

4.4 Módulos de I/O

Os módulos de I/O implementam lógica de interface entre o programa e entidades externas ao mesmo. Assim, estes módulos devem implementar toda a lógica de abertura, leitura, e processamento do texto de ficheiros, bem como de manipulação de input e output feitos através da linha de comandos ou qualquer outra interface textual ou visual. Assim, alguns exemplos de possíveis módulos de I/O incluem: **parser de input** que efetua a leitura de dados de ficheiros bem como o processamento das linhas de texto que daí recebe, efetuando operações como divisão de linhas em sub-linhas, e conversão dessas linhas noutros formatos, como estruturas (`struct`) definidas pelo programa; **processador de output** que recebe dados, converte-os para um formato textual apropriado e escreve-os para um ficheiro; **processador de input interativo** que implementa uma interface textual onde mostra na linha de comandos as operações que um utilizador pode efetuar, recebe as operações indicadas pelo utilizador e invoca os métodos apropriados para efetuar essas operações.

De forma semelhante aos módulos de estruturas de dados, os módulos de I/O devem ser implementados de forma genérica para garantirem a sua máxima reutilização, fazendo uso de funções por apontador como argumento.

4.5 Módulos de utilidade

Dá-se o nome de módulo de utilidade a um módulo agrega um conjunto de funções que, em geral, possuem afinidades funcionais (cf. por exemplo um módulo matemático, ou um módulo de processamento de datas) mas que não armazenam

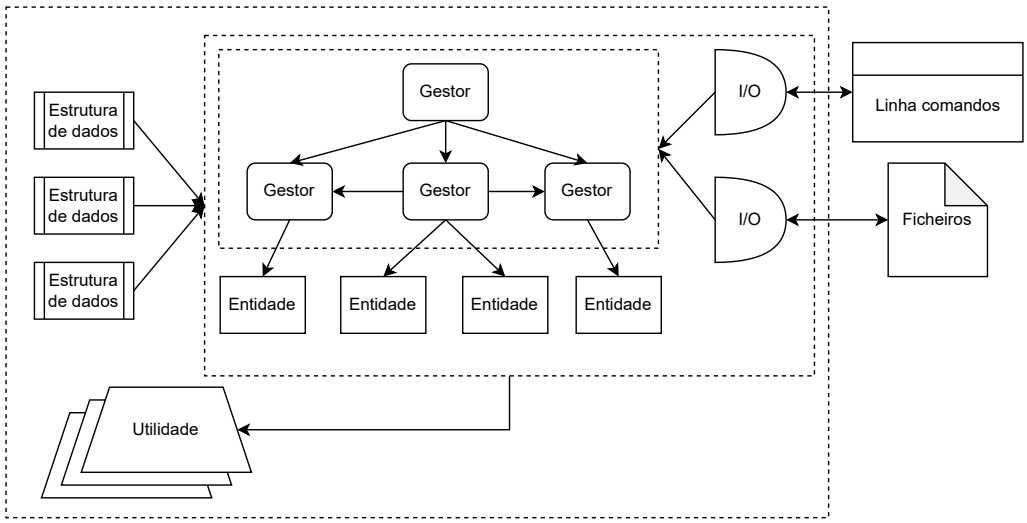
dados (à exeção da definição de eventuais valores constantes), fazendo apenas conversões ou operações sobre dados que lhe são passados. Veja-se o exemplo abaixo para um módulo de matemática:

```
1  //math.h
2
3  int sum(int a, int b);
4  int subtract(int a, int b);
5  int pow(int a, int b, int power);
6  double divide(double a, double b);
```

É ainda comum a existência de um módulo com o nome *utils* ou *commons* que agrega funções que são úteis a múltiplos módulos no sistema, mas que não possuem afinidade funcional, ou que tendo afinidade não existem em número suficiente para criar o seu próprio módulo. Por exemplo, veja-se o seguinte módulo:

```
1   //utils.h
2   struct Date{
3     int year;
4     int month;
5     int day;
6   }
7
8   Date stringToDate(char *str);
9   void removePunctuation(char *str);
10  char ** strToSubStrings(char *str, char *delim);
11  int stringToInt(char *str);
12  int stringToDouble(char *str);
```

Em suma, um programa bem modularizado deverá ter a seguinte arquitetura:



5 Exercícios

Considere um sistema de gestão dos balcões de registos portugueses. O sistema contém um conjunto de balcões, definidos pelo seu id, nome, localização, e horário de atendimento. A cada balcão estão associados vários clientes (um cliente pode estar associado a vários balcões), definidos pelo número de cartão de cidadão, nome, data de nascimento, e morada. Considere ainda que cada balcão pode ter múltiplas filas de espera, uma por cada serviço prestado (definido pelo nome, e.g., "Renovar cartão de cidadão"), sendo que a fila deverá dar prioridade de atendimento a utilizadores com mais de 80 anos. O sistema deverá suportar as seguintes operações:

- Consultar a informação de um cliente, fornecendo o seu número de cartão de cidadão;
- Consultar a lista de clientes de um balcão, fornecendo o seu id;
- Num dado balcão, adicionar um cliente a uma fila de espera, fornecendo o id do balcão, o número de cartão de cidadão do cliente, e o nome do serviço em questão;
- Num dado serviço de um balcão, chamar (i.e., remover) o primeiro cliente na fila, fornecendo o id do balcão e o nome do serviço.

1. Que entidades identifica neste sistema?
2. Que módulos é que definiria? Quais seriam as interações entre estes?
3. Que estruturas de dados é que usaria na implementação deste sistema?