Revisão de C

Laboratórios de Informática III
Guião #1

Departamento de Informática
Universidade do Minho

2023/24

Conteúdo

1 Conceitos

1.1 Função `main`

A função `main` é o ponto de entrada de qualquer programa em C, tendo a seguinte assinatura[1]:

```c
int main(int argc, char** argv);
```

É uma função que retorna um inteiro (normalmente 0, indicando que o programa executou com sucesso) e recebe dois parâmetros:

- `argc` – número de argumentos passados ao programa;
- `argv` – array de argumentos em formato textual.

O primeiro elemento de `argv` é reservado para o nome do executável e o último é sempre o endereço nulo (`NULL`), respetivamente, `argv[0]` e `argv[argc]`.

Considere um programa `repeat`, muito simples, que recebe como argumentos uma string e o número de vezes que deverá ser impressa na consola:

```c
#include <stdlib.h> // para usar o "atoi"
#include <stdio.h> // para usar o "printf"

int main(int argc, char** argv) {
    printf("%s recebeu %d argumentos.\n", argv[0], argc);
    int repeats = atoi(argv[2]); // atoi converte uma string para um inteiro
    while (repeats > 0) {
        printf("%s\n", argv[1]);
        repeats--;
    }
    return 0;
}
```

Usando o `gcc`, podemos compilar o programa da seguinte forma:

```
$ gcc repeat.c -o repeat
```

Executando `./repeat "Hello, world!"4` obtemos o seguinte output:

```
./repeat recebeu 3 argumentos.
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

1.2 Inclusão de ficheiros

Antes de usarmos funções ou tipos de dados externos ao nosso código-fonte devemos sempre – e em muitos casos *temos* mesmo – que incluir as respetivas declarações. Por exemplo, se queremos usar a função `printf`, devemos incluir a sua declaração, a qual, por convenção da biblioteca de C, encontra-se no ficheiro `stdio.h`. A diretiva `#include` permite-nos importar código de ficheiros externos para que o possamos usar no nosso ficheiro. Existem duas variantes para importação de ficheiros. A primeira variante vai procurar o ficheiro `file.h` num conjunto de caminhos pré-definidos:

```c
#include <file.h>
```

[1] Em alternativa mas de forma equivalente, o argumento `argv` também pode ser declarado como `char* argv[]`.

A segunda variante irá primeiramente procurar o ficheiro `file.h` na mesma pasta em que o nosso ficheiro se encontra. Caso não o encontre, irá procurar o ficheiro da mesma maneira que a variante anterior:

```
#include "file.h"
```

1.3 Arrays

Um array é uma estrutura de dados que nos permite agrupar dados do mesmo tipo em posições contíguas de memória. Acessos a posições de memória fora dos limites do array podem levar a comportamentos indesejados, como a alteração dos valores de outras variáveis, ou causar a terminação abrupta da execução do programa. Por isso, os limites de tamanho do array devem ser respeitados. O seguinte excerto apresenta exemplos de declarações de arrays:

```
int num_pointer_stack1[10]; //Array de inteiros com espaço para 10 elementos.
int num_pointer_stack2[] = {1,2,3,4,5,6,7,8,9,10}; //Igual ao exemplo anterior, mas as 10 posições são preenchidas no momento de declaração da variável. Declarar o tamanho do array é opcional, pois o compilador consegue inferi-lo.
int num_pointer_stack3[10] = {1,2,3}; //As restantes posições são inicializadas com o valor 0.
```

Arrays de caracteres podem ser instanciados também a partir de *string literals*. O conteúdo da string imutável é copiado para as posições do array, mas é importante notar que isto só é possível na inicialização da variável. Após a inicialização, a string contida no array poderá ser copiada – como qualquer string – usando a função `strcpy`. O compilador consegue inferir o tamanho de um array a partir de um string literal:

```
char char_pointer_stack1[10];
char char_pointer_stack2[]={'a','b','c','d','e','f','g','h','i','\0'};
char char_pointer_stack3[]= "abcdefghi";  //Forma alternativa. Copia o string literal para as posições do array
```

Valores de um array podem ser acedidos através de índices que começam no valor 0 e se estendem até [tamanho do array - 1]:

```
char stringPointer[10];
stringPointer[0] = 1; //Escrita no primeiro elemento do array
stringPointer[9] = 5; //Escrita no último elemento do array
```

É importante perceber que uma variável de array não é um apontador. Para mais detalhes sobre a distinção entre os dois, ver secção 1.6.

1.4 Structs

As structs em C permitem agrupar um conjunto de variáveis logicamente. Por exemplo, se quisermos representar os atributos de um produto de uma loja, podemos usar a seguinte struct:

```
struct item {
    char name[30];
    int stock;
    double basePrice;
    double discount; // [0, 1[
};
```

Para criarmos um produto, basta usar `struct item i;`. Para aceder às variáveis de uma struct, usamos o operador `.`:

```c
struct item i;
strcpy(i.name, "item 1"); // <string.h> precisamos de copiar a string para a struct
i.basePrice = 9.99;
i.stock = 14;
i.discount = 0;
```

1.5 Escopo e tempo de vida de variáveis

O escopo de uma variável define as **zonas do código em que esta pode ser manipulada**. O tempo de vida de uma variável define o **período durante o qual a variável se mantém alocada em memória**.

Quando uma variável é definida dentro de uma função, sem qualquer modificador, esta só pode ser manipulada dentro dessa função e o seu tempo de vida será o tempo de execução da função. A isto chamamos uma variável local:

```c
void func2() {
    //Não podemos manipular num1, porque a func2 está fora do seu scope, mas ainda está alocado em memória no momento de execução desta função (tempo de vida estende-se a func2)
    num1 = 1; // erro de compilação
}

void func1() {
    int num1;
    //Podemos manipular num1
    num1 = 10;
    func2();
}
```

Duas variáveis locais com o mesmo nome podem existir no mesmo programa, desde que não se sobreponham no seu escopo, ou seja, desde que sejam declaradas em funções diferentes:

```c
//Cada uma das variáveis num1 tem o scope local à sua função, e não é afetada pelas operações da outra função.

void func1() {
    int num1;
    num1 = 10;
}

void func2() {
    int num1;
    num1 = 10+3;
}
```

Uma variável pode também ser definida fora de uma função. Neste caso, o escopo da variável estende-se a todas as funções do programa e tem um tempo de vida que se estende até à conclusão da execução do programa, atribuindo-se-lhe o nome de variável global:

```c
int a;

void func1() {
    a = a + 2;
}

void func2() {
    a++;
}
```

```
10
11 int main(int argc, char** argv) {
12     a = 0;
13     func1(); //a passa a ter valor 2
14     func2(); //a passa a ter valor 3
15 }
```

Quando duas variáveis, com o mesmo nome, se sobrepõem no mesmo escopo de tal forma que uma variável é local, e a outra é global, a variável local irá sobrepor-se à global:

```
1  int a;
2
3  void func1() {
4      int a = 0;
5      a = a + 2; //a tem valor 2
6  }
7
8  void func2() {
9      a++; //a tem valor 4
10 }
11
12 int main(int argc, char** argv) {
13     a = 3;
14     func1(); // variável global 'a' não é alterada nesta função
15     func2(); // variável global 'a' passa a ter valor 4
16 }
```

1.6 Apontadores

As variáveis em C podem armazenar diferentes tipos de dados, como `int`, `float`, `char`, `long long`, entre outros. Contudo, também podem guardar um endereço de memória (e.g., o local onde uma variável está guardada), sendo designados por "apontadores". Os apontadores são representados por `tipo*`, e.g., `int*`. Podemos usar o símbolo `*` num apontador para aceder ao seu valor (i.e., desreferenciar o apontador) e o símbolo `&` para obter o endereço de uma variável:

```
1 int x = 10; // x é um inteiro de valor 10
2 int* y = &x; // y é um apontador para um inteiro
3 printf("Endereço de x = %p\n", y);  // Endereço de x = 0x7fffea9d58ac
4 printf("*y = %d\n", *y); // *y = 10
5 x *= 2;
6 printf("*y = %d\n", *y); // *y = 20
```

Os apontadores tornam-se especialmente úteis na leitura e modificação de estruturas de dados mais complexas, sobretudo quando queremos manipular as mesmas estruturas de dados a partir de funções diferentes, sem ter que efetuar várias cópias. Por exemplo:

```
1  struct house {
2      char street[150];
3      int area;
4      char owner[50];
5  }
6
7  void fill_house_details(struct house* my_house) {
8      strcpy((*my_house).street, "foo");
9      (*my_house).area = 50;
10     strcpy((*my_house).owner, "bar");
11 }
```

```
12
13 int main(int argc, char** argv) {
14     struct house my_house;
15     fill_house_details(&my_house);
16 }
```

A estrutura `struct house` é instanciada apenas uma vez. Passando o endereço da estrutura em vez da própria estrutura evita que sejam feitas múltiplas cópias. É necessário no entanto ter cuidado com a partilha de estruturas de dados através de apontadores, uma vez que as funções a quem são passados os apontadores podem modificar autonomamente o conteúdo das estuturas.

Aritmética de apontadores

Tal como com valores numéricos, também é possível aplicar operações aritméticas a apontadores. No entanto, estas operações não são idênticas a operações numéricas (e.g., somar 1 a um apontador não incrementa necessariamente o valor do apontador em 1 unidade). Por exemplo, considere o seguinte array:

```
1 char l[] = {'H', 'e', 'l', 'l', 'o', '\0'}; // \0 é o char NULL
2 char* l_ = l; // l_ contém o endereço para o primeiro elemento do array l
```

Se quisermos aceder ao primeiro elemento de `l`, basta fazer `*l`; para aceder ao segundo, fazemos `*(l+1)`; para aceder ao n-ésimo, fazemos `*(l+n-1)`. Sendo assim, podemos definir a função `strlen_`, que calcula o tamanho de uma string, da seguinte forma:

```
1 void strlen_(char* s) {
2     int length = 0;
3     for (; *s; s++, length++);
4     return length;
5 }
```

Neste caso, somar 1 ao apontador incrementa o endereço num byte, dado que o tamanho de `char` é 1. Contudo, nem todos os valores têm tamanho 1. Por exemplo, considere o seguinte array de `short`[2]:

```
1 short s[] = {123, 456, 789};
```

Neste caso, a operação `s += 1` irá incrementar o endereço guardado em `s` 2 unidades (dado que o tamanho de um `short` é 2 bytes), como podemos comprovar no seguinte exemplo e no esquema da Figura 1:

```
1 printf("l:%p; l+1:%p\n", l, l+1); // l:0x7ffff1b1b5f3; l+1:0x7ffff1b1b5f4
2 printf("s:%p; s+1:%p\n", s, s+1); // s:0x7ffff1b1b5ec; s+1:0x7ffff1b1b5ee
```

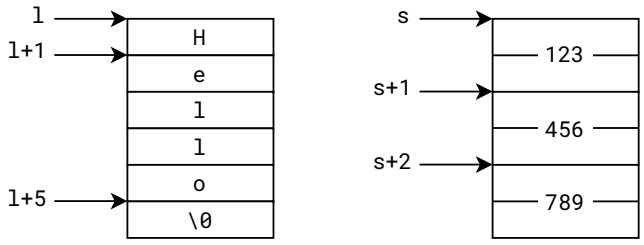

Figura 1: Exemplo da estrutura de dois arrays, `char* l` e `short* s`.

[2] `short` representa um valor número inteiro com mínimo -32768 e máximo 32767.

Por último, é relevante salientar o apontador especial `void*`, que pode apontar para qualquer tipo de dados. Este apontador é usado sobretudo na implementação de estruturas de dados genéricas, como iremos ver mais à frente na Secção 1.9. Dado que pode referir-se a qualquer valor, não devemos somar ou subtrair diretamente ao apontador sem antes o converter para o seu tipo concreto (cast).

Apontadores de structs

Se tivermos um apontador para uma struct, podemos aceder ao seu conteúdo da forma anteriormente abordada, Ou seja, desreferenciando o apontador através de `*` e depois usando o operador `.`. Contudo, podemos simplificar o acesso a elementos de uma struct através do seu apontador, usando `->`. No seguinte exemplo, as duas funções são equivalentes:

```c
void printItemDereference(struct item* i) {
    printf("{\n\tNome: %s,\n\tPreço base: %.02f,\n\tStock: %d,\n\tDesconto: %d%%\n}\n",
        (*i).name, (*i).basePrice, (*i).stock, (int)((*i).discount * 100));
}

void printItemArrow(struct item* i) {
    printf("{\n\tNome: %s,\n\tPreço base: %.02f,\n\tStock: %d,\n\tDesconto: %d%%\n}\n",
        i->name, i->basePrice, i->stock, (int)(i->discount * 100));
}
```

Arrays vs apontadores

É importante notar que uma variável do tipo array, ainda que semelhante, não é idêntica a um apontador para o início desse array. Uma variável do tipo array aloca espaço para todos os elementos de um array. Um apontador apenas aloca espaço para um endereço. Uma variável do tipo array, quando lhe é aplicada a função `sizeof`, irá retornar o tamanho do array inteiro, enquanto que um apontador retornaria o tamanho de um endereço. Para além disso, variáveis do tipo array são imutáveis. Os seus elementos podem ser modificados, mas não a própria variável:

```c
int arr1[10];
int arr2[12] = {1,2,3,4,5};
int* arr_pointer;

arr1 = arr2; //Operação não permitida! O valor de arr1 é imutável!
arr_pointer = arr1; //arr1 é implicitamente convertido para o endereço do primeiro elemento do
↪  array. Operação válida.

sizeof(arr1); //Retorna o 10*sizeof(int)
sizeof(arr_pointer) //Retorna o tamanho de um endereço de memória

//Um caso interessante
char* string = "Hello world"; //String literals são alocados em memória apenas de leitura. Uma vez
↪  que a variável string não se trata de um array, vai apenas apontar para o local onde o string
↪  literal está alocado, não sendo possível, por esse motivo, alterar o valor da string.
```

Quando um array é passado como argumento para outra função, é implicitamente convertido no endereço do seu primeiro elemento. Assim, o valor retornado por `sizeof` quando aplicado ao argumento será o de um endereço, e pode ser-lhe atribuído um valor diferente. Esta conversão implícita acontece também quando imprimimos o valor da variável do tipo array:

```c
void called_function(int arr1[], int* arr2) {
    // As duas variáveis são apontadores
    sizeof(arr1); //Devolve tamanho de um endereço (warning: ‘sizeof’ on array function parameter
↪  ‘arr1’ will return size of ‘int*’)
    sizeof(arr2); //Devolve tamanho de um endereço
    int arr3[10];
    arr1 = arr3; //Operação permitida mas não correta, dado que a referência a arr3 tornar-se-á
↪  inválida quando a função terminar
```

```c
}

int main(int argc, char** argv) {
    int arr1[10];
    int arr2[10];
    called_function(arr1,arr2);
    return 0;
}
```

1.7 Gestão de memória

A Figura 2 mostra uma visão geral da estrutura de memória de um programa C. Existem duas secções de memória de tamanho variável, a Stack e a Heap. O facto destas secções serem variáveis significa que podem crescer ou diminuir ao longo da execução do programa, tal como mostra a Figura 2:

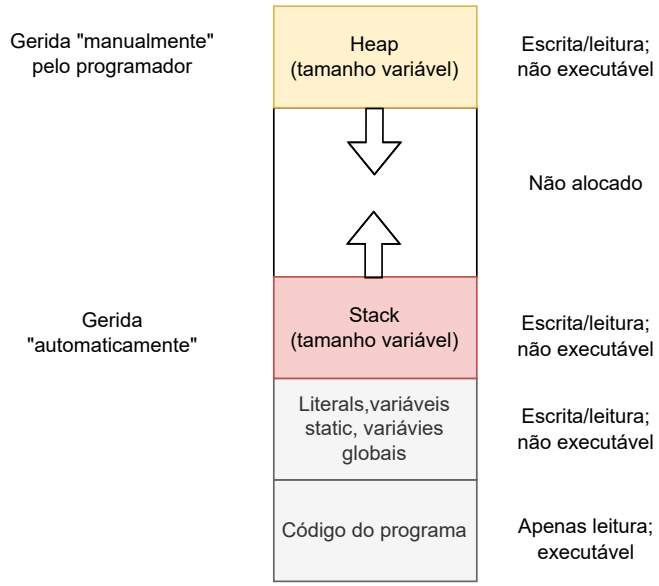

Figura 2: Estrutura de memória de um programa C

A Stack pode-se considerar como sendo *"gerida automaticamente"* pelo programa. Quando uma função é chamada, a Stack cresce. É alocado espaço para as variáveis locais da função, para os argumentos passados à função, e para outros elementos necessários ao normal funcionamento do programa. Quando uma função termina, a Stack encolhe. O espaço alocado para as variáveis locais e argumentos é libertado, mantendo-se apenas o valor retornado pela função. Tudo isto é feito sem qualquer intervenção explícita do programador.

Já a Heap é controlada pelos comandos `malloc`, `calloc`, `realloc` e `free`. A Heap só cresce ou encolhe quando é explicitamente manipulada pelo programador, através das funções anteriormente descritas. A terminação de funções não afeta a alocação de variáveis na Heap.

Vejamos os exemplos abaixo:

```c
//Na stack é alocado espaço para as variáveis 'result', 'constant', e para os argumentos 'a' e 'b'.
↪  Os argumentos 'a' e 'b' são na verdade cópias dos valores que lhe foram passadas pela main.
int const_sum_values(int a, int b) {
    int constant = 5;
    int result = a + b + constant;
    return result; //Apenas o valor retornado é mantido em memória.
}

int main(int argc, char** argv) {
    int a = 2;
```

```
    int b = 3;
    const_sum_values(a,b);
}
```

```
//Memória gerida "automaticamente" pelo programa.
int* build_stack_array() {
    int arr[10]; //Espaço para 10 inteiros é alocado em stack. Este espaço será desalocado no fim da execução da função
    int* arr_pointer = arr;
    return arr_pointer;
}

//Memória gerida "manualmente" pelo programador, na Heap
int* build_heap_array() {
    int* num_pointer_heap; //Um apontador não inicializado. Neste momento aponta para um local aleatório da memória.
    num_pointer_heap = malloc(10 * sizeof(int)); //A função malloc aloca um bloco de memória para 10 elementos do tipo int em memória heap e retorna o endereço para o início desse bloco. Esse endereço é guardado no apontador anteriormente definido.
    return num_pointer_heap;
}

int main () {
    int* num_stack;
    int* num_heap;

    num_stack = build_stack_array(); //Aponta para uma zona de memória desalocada, não pode ser usado
    num_heap = build_heap_array(); //Aponta para um array em zona de memória heap. O array pode ser usado enquanto não for desalocado.

    free(num_heap); //Desaloca o espaço anteriormente alocado pela função malloc.
}
```

1.8 #define e typedef

O operador `#define` cria um alias (ou macro) para variáveis, tipos ou funções de um programa. Antes do programa ser compilado, o pré-processador substitui no texto do programa esses alias, funcionando no fundo como um mecanismo de ”copy+paste”. Por convenção, os alias atribuídos são escritos em letras maiúsculas.

```
#define ONE 1

int main(int argc, char** argv) {
    //Antes do programa ser compilado, o termo ONE vai ser substituído por 1
    int a = ONE;
    return 0;
}
```

O operador `typedef` permite renomear tipos existentes, criando um novo tipo.

```
typedef long int lu;

int main(int argc, char** argv) {
    lu a = 1000;
    printf("Long int has value: \%ld\n",a);
```

```
    return 0;
}
```

Este operador é bastante útil quando estamos a lidar com structs. Vejamos a definição abaixo:

```
struct item {
 char name[50];
 float basePrice;
 float stock;
 float discount;
};
```

De momento, precisamos de indicar `struct item` para representar o tipo de um item. Contudo, isto pode ser simplificado através do uso do `typedef`:

```
typedef struct item Item;
```

Também pode ser feito diretamente na definição da struct:

```
typedef struct item {
    char name[20];
    ...
} Item;
```

Com isto, o produto já pode ser declarado da seguinte forma:

```
Item i;
```

Quando estamos a lidar com tipos, não devemos usar `#define`. Este operador não define um novo tipo, apenas substitui texto, subsituindo todas as instâncias de uma palavra, independentemente de se tratarem de tipos ou variáveis, podendo levar a erros complicados de detetar. O operador `typedef` garante que o novo tipo estará sujeito a regras de escopo e de sintaxe, e será interpretado pelo compilador como qualquer outro tipo. Vejamos os exemplos abaixo, que comparam os dois casos. A função `concatenate_bytes` concatena dois bytes guardados numa struct, e devolve um inteiro de valor correspondente a essa concatenação.

```
#define byte unsigned char


int main(int argc, char** argv) {
    byte a;
    int byte = 3;

    a = byte;
    return 0;
}
```

Após passar pelo pré-processador, a função main ficaria com o seguinte aspeto:

```
// Código obviamente não compilável
int main(int argc, char** argv) {
    byte a;
    int unsigned char = 0;
    a = unsigned char;

```

```
    return 0;
}
```

Já se usarmos antes `typedef unsigned char byte;` o programa funcionará da forma esperada.

1.9 Estruturas de dados abstratas

As estruturas de dados abstratas são estruturas definidas pelo seu comportamento, podendo ter diversas implementações. Uma das estruturas abstratas mais usadas é a `Lista`, que representa uma sequência de valores onde a ordem é relevante. Para representar estruturas de dados abstratas em C, usamos o `void*`. Por exemplo, podemos implementar uma simples `Lista` da seguinte forma:

```
typedef struct list {
    void* data;
    List* next;
} List;
```

Para adicionar qualquer valor à cabeça lista, podemos usar a seguinte função:

```
List* create(void* data) {
    List* new = malloc(sizeof(List));
    new->data = data;
    new->next = NULL;
    return new;
}

List* prepend(List* l, void* data) {
    List* new = create(data);
    new->next = l;
    return new;
}
```

Funções por referência

Para efetuar operações sobre estruturas de dados abstratas são muitas vezes usadas funções de ordem superior, ou seja, funções que recebem outras funções por argumento. Considere a função `apply`, que aplica uma função a todos os elementos da nossa lista:

```
void apply(List* l, void (*f)(void*)) {
    while (l) {
        f(l->data);
        l = l->next;
    }
}
```

Neste caso, `apply` recebe uma função `f` que por sua vez recebe `void*` por argumento e não retorna nada, aplicando-a a todos os elementos da lista. Um exemplo de `f` é a função `square`, que converte um número no seu quadrado:

```
void square(void* i) {
    int* i_ = i; // precisamos fazer cast para calcular o quadrado
    *i_ *= *i_;
}
```

Considerando a lista `*l` com valores `[10, 20, 30, 40]`, aplicar `apply(l, &square)` irá resultar nos novos valores `[100, 400, 900, 1600]`.

1.10 Ficheiros .h vs .c

Declaração *vs* definição de funções

Há uma distinção importante entre declaração e definição de uma função. A **declaração** de uma função indica o tipo de valor retornado pela função, e o número e tipo de argumentos que essa função recebe, mas não inclui as operações efetuadas pela função. A **definição** de uma função, para além de incluir a sua declaração, define a lógica da função, ou seja, define a sequência de operações a serem executadas pela função.

```
1 //Declaração de função
2 int sum(int a, int b);
3
4 //Definição de função
5 int sum(int a, int b) {
6     return a+b;
7 }
```

Ficheiros .h

Os ficheiros **.h** contêm **declarações de funções** e **definições de tipos (entre outros) que queremos partilhar** com outros ficheiros. Funcionam como a definição de uma interface, que explicita a quem quiser usar o nosso código as funções que estamos a disponibilizar, e eventualmente novos tipos de variáveis que são retornadas pelas nossas funções. Assim, evitamos ter que definir uma função em todos os ficheiros que precisam dela, promovendo assim a reutilização de código, já que só temos que definir a função uma vez. Para além disso, permite-nos também restringir as funções que outros ficheiros podem usar.

Um ficheiro **.h** deve ser importado tanto por quem quer usar as funções que ele declara, como pelos ficheiros que implementam (i.e. definem) essas funções.

Para além disso, qualquer ficheiro .h deve começar pelas diretivas `#ifndef` e `#define`, seguidas de um nome único, de modo a evitar reimportar o ficheiro múltiplas vezes, o que levaria a erros de compilação. Aqui adotamos a convenção de usar o nome do ficheiro, em maiúsculas e unido por `_`, como identificador único. O ficheiro deve ainda terminar em `#endif`. Veja-se o exemplo abaixo:

```
1 #ifndef FICHEIRO_A
2 #define FICHEIRO_A
3
4 void func1();
5 void func2();
6 //...
7
8 #endif
```

Ficheiros .c

Os ficheiros **.c** devem conter as definições (i.e. implementações) de funções do programa. Devem também incluir todos os tipos e variáveis que não queremos que sejam partilhadas com outros ficheiros. Uma das vantagens desta estruturação é que podemos ter diferentes implementações (usando diferentes **.c**) para diferentes interfaces, podendo optar por uma ou por outra dependendo das nossas necessidades, na altura da compilação.

Por exemplo, considere novamente a lista genérica definida acima. A sua interface conterá a declaração da estrutura e das operações a implementar:

```
1 // list.h
2 #ifndef LIST
3 #define LIST
4 typedef struct list {
5     void* data;
```

```
    List* next;
} List;

List* create_list(void* data);
List* prepend_list(List* l, void* data);
...
#endif
```

Já o ficheiro com a implementação irá primeiro importar a interface e de seguida implementar as operações descritas:

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

List* create_list(void* data) {
    ...
}

List* prepend_list(List* l, void* data) {
    ...
}

...
```

Esta lista poderá depois ser utilizada noutros ficheiros simplesmente importando o ficheiro header:

```
// main.c
#include "list.h"

int main(){
    int* a = malloc(sizeof(int));
    List* list = create_list(a);
    return 0;
}

...
```

1.11 Compilação e execução

Um programa passa por 4 ferramentas antes de se tornar num executável, como descrito na Figura 3, nomeadamente o Pré-processador, o Compilador, o Assembler e o Linker.

O pré-processador efetua múltiplas funções. Entre estas inclui-se a remoção de todos os comentários do ficheiro e a substituição de Macros definidas com `#define` pelo texto correspondente. Para além disso, é feita a substituição de diretivas `#include <file.h>` pelo conteúdo desses ficheiros, como indicação futura de que as funções declaradas nesse ficheiro deverão ser procuradas em ficheiros externos.

O compilador converte o output produzido pelo pré-processador para código Assembly.

O Assembler, por sua vez, converte o ficheiro de código Assembly num *object file*, que contém código máquina (ou binário).

Quando temos um projeto com múltiplos ficheiros, cada ficheiro .c irá originar um ficheiro de código máquina `.o` . E cada ficheiro `.c` poderá fazer uso de funções que não estão definidas no próprio ficheiro. Para combinar as definições das diferentes funções que se encontram nos diferentes ficheiros, de forma a formarmos um ficheiro executável com todo o código necessário, recorremos ao Linker. O Linker combina os diferentes ficheiros `.o` do nosso projeto, bem como ficheiros `.o` de bibliotecas que estejam a ser usadas no nosso programa, para formar o executável final.

Considere o exemplo onde temos o ficheiro `list.h`, que define a interface de uma lista, e o ficheiro `list.c`, que define uma possível implementação. Considere ainda a existência do ficheiro `main.c`, que usa a lista declarada em `list.h`. O primeiro passo a tomar consiste na geração dos ficheiros `.o`, através do comando `gcc -c`:

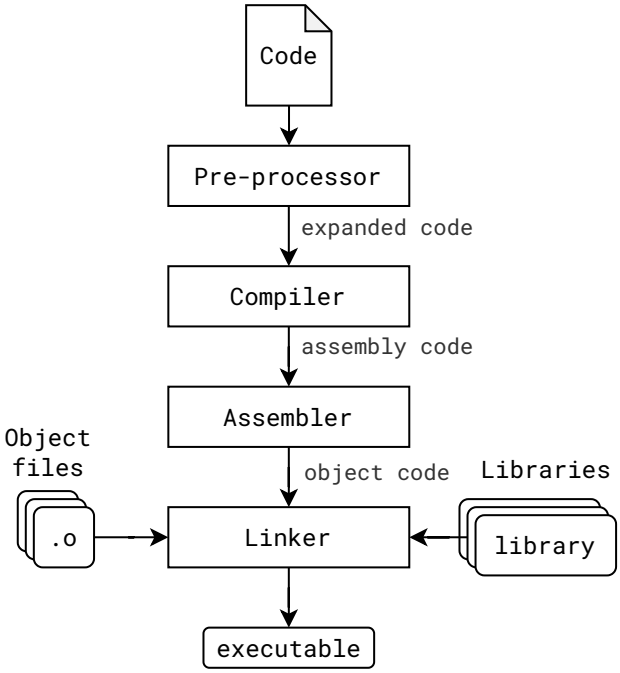

Figura 3: Passos do processo de compilação.

Adaptado de `https://www.javatpoint.com/compilation-process-in-c`

```
$ gcc -c list.c # gera list.o
```

De seguida, geramos o executável `main`, fornecendo o ficheiro `list.o` como argumento:

```
$ gcc main.c list.o -o main
```

Considere agora uma segunda implementação de `list.h`, definida em `list_fast.c`, que consume mais memória mas obtém tempos de execução mais rápidos. Para usar esta nova implementação, simplesmente trocamos a implementação de `list.h` no momento da compilação:

```
$ gcc -c list_fast.c # gera list_fast.o
$ gcc main.c list_fast.o -o main
```

Não foi necessário modificar o ficheiro que usa `list.h`, visto que a declaração da interface não foi alterada.

2 Exemplo prático

O exemplo de seguida consiste na criação e teste de uma `Stack` genérica. A `Stack` é uma estrutura de dados que suporta a operações de `push` – adicionar um elemento no top – e `pop` – retira o elemento do top, retornando-o.

Comecemos por definir as interfaces do nosso exemplo. A primeira refere-se a `Node`, que representa o elemento base da nossa `Stack`:

```c
// node.h
#ifndef NODE_H
#define NODE_H

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

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

#endif
```

A segunda é a `Stack`, que contém os elementos e um contador, e declara diversas operações de modificação e leitura:

```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
```

De seguida, passámos à implementação da duas interfaces:

```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.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;
    stack->top = node;
    stack->size++;
}

void* stackPop(Stack* stack) {
    if (stack->size == 0) {
        return 0; // mesmo que NULL
    }

    Node* top = stack->top;
    stack->top = top->next;
    void* data = top->data;
    free(top);
    stack->size--;
    return data;
}

void stackPrint(Stack* stack) {
    Node* top = stack->top;
    while (top) {
        stack->elemPrint(top->data);
        printf(" -> ");
        top = top->next;
    }
    printf("x\n");
}
```

Por último, criamos uma `main` para testar a estrutura. Neste caso, começamos por criar a `Stack`, alocando e inserindo 5 inteiros aleatórios. Depois, removemos e libertamos os elementos um a um. Finalmente, libertamos a `Stack`:

```c
// main.c
#include "stack.h"
#include <time.h>
#include <stdlib.h>
#include "stdio.h"
#include "stack.h"

void printInt(void* i) {
    int* i_ = i;
    printf("%d", *i_);
}

```

```c
int main(int argc, char** argv) {
    srand(time(NULL)); // seed do random, para ser diferente a cada run

    Stack* s = stackCreate(&printInt);
    for (int i = 0; i < 5; i++) {
        stackPrint(s);
        int* i = malloc(sizeof(int));
        *i = rand() % 100; // [0, 100[
        stackPush(s, i);
    }
    stackPrint(s);

    int* i;
    // termina quando retornar NULL, ou seja,
    // não existe mais elementos para remover
    while (i = stackPop(s)) {
        free(i); // é preciso libertar os dados pois stackPop liberta apenas o nó
        stackPrint(s);
    }

    free(s);

    return 0;
}
```

Para gerar um executável, precisamos primeiro de compilar os vários módulos usados pela nossa `main`. De seguida, compilamos o ficheiro com a `main`, indicando a implementação das dependências:

```
$ gcc -c node.c
$ gcc -c stack.c
$ gcc main.c node.o stack.o -o main
```

Executando `./main`, obtemos o seguinte output:

```
$ ./main
x
8 -> x
70 -> 8 -> x
41 -> 70 -> 8 -> x
35 -> 41 -> 70 -> 8 -> x
99 -> 35 -> 41 -> 70 -> 8 -> x
35 -> 41 -> 70 -> 8 -> x
41 -> 70 -> 8 -> x
70 -> 8 -> x
8 -> x
x
```

3 Exercícios

Considere a estrutura de dados abstrata `Deque` (*double-ended queue*), que modela uma fila onde valores podem ser inseridos no início ou no fim, podendo ainda ser removidos do início ou do fim (Figura 4).

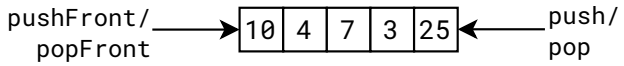

Figura 4: Exemplo de uma `Deque`.

A `Deque` deverá suportar as seguintes operações:

- `Deque* create()` – inicializa uma `Deque` sem elementos;
- `void push(Deque* deque, void* data)` – adiciona `data` no fim da fila;
- `void pushFront(Deque* deque, void* data)` – adiciona `data` no início da fila;
- `void* pop(Deque* deque)` – remove o elemento do fim da fila e retorna-o; retorna `NULL` se a fila não tiver elementos;
- `void* popFront(Deque* deque)` – remove o elemento do início da fila e retorna-o; retorna `NULL` se a fila não tiver elementos;
- `int size(Deque* deque)` – retorna o número de elementos da fila;
- `bool isEmpty(Deque* deque)` – indica se a fila está vazia; [3]
- `void reverse(Deque* deque)` – inverte a ordem dos elementos da fila;
- `void printDeque(Deque* deque, void(*printFunc)(void*))` – imprime os elementos da fila com o auxílio da função `printFunc`;
- `void destroy(Deque* deque)` – liberta a memória associada à `Deque`.

O design da `Deque` deverá permitir que as operações acima sejam executadas com o número mínimo de iterações possíveis.

1. Defina a interface da `Deque`;
2. Implemente a `Deque`;
3. Crie uma função para testar a estrutura, inserindo/removendo valores (e.g., `int*`) no início ou no fim da `Deque` de forma aleatória, imprimindo para a consola o seu estado a cada operação.

[3] Para usar o tipo de dados `bool` é necessário incluir `stdbool.h`

4 Conceitos adicionais

Enums

Os enums são enumerações de valores constantes inteiros. São usados para representar categorias, estados, tipos, ..., de forma simples. Por exemplo, o seguinte enum representa o género de um filme:

```
enum genre {
    Action,
    Comedy,
    Drama,
    ...
}
```

Os valores de um enum podem ser implícitos, como exemplificado em cima, ou explícitos, com mostra o exemplo seguinte:

```
enum genre {
    Action = 0,
    Comedy = 1,
    Drama = 2,
    ...
}
```

De seguida, apresenta-se o uso do `enum genre` numa struct de filmes:

```
struct movie {
    char name[50];
    int runtime;
    enum genre genre;
}

struct movie m;
m.genre = Comedy;
```

union

O construtor `union` permite definir variáveis que partilham o mesmo espaço de memória. Ao definir uma variável `union`, será alocado espaço suficiente para manter a maior das variáveis dessa `union`. Ao usar uma `union`, o único valor válido será o da última variável escrita, uma vez que escrever uma das variáveis irá corromper a representação das restantes. Na prática, uma variável do tipo `union` pode ser vista como uma única variável que pode adotar diferentes tipos.

Uma alternativa ao uso de variáveis `union` seria fazer uso de `void *`, o que inclusive evitaria ter uma variável que ocupa mais espaço do que o necessário. No entanto, com o uso de `union` garantimos que a variável apenas poderá adotar um dos tipos definidos na `union`. Torna também possível rapidamente alternar o tipo da variável a ser representada, sem ser necessário voltar a alocar memória:

```
//O tamanho de uma variável deste tipo será sempre 8 bytes
union classification {
    int i;
    float f;
    char s[8];
}

int main() {
    union classification clf;
    clf.i=10;
    clf.f=0.1;
```

```
12      strcpy(clf.s,"Bad");
13      //Apenas o valor de clf.s é válido neste ponto do código
14      return 0;
15  }
```

Modificadores de armazenamento

O *scope* e tempo de vida de variáveis pode ser controlado através do uso de modificadores de armazenamento (Tabela 1).

Modificador	Scope	Tempo de vida
auto	Bloco	Execução da função
static	Ficheiro	Fim do programa
register	Bloco	Execução da função
extern	Programa	Fim do programa

Tabela 1: Modificadores de armazenamento

De um modo geral os modificadores `extern` e `register` **não deverão ser usados**. O modeificador `extern`, em particular, introduz a possibilidade de declaração e uso incorreto de funções externas, situação de erro particularmente difícil de identificar e depurar. O modificador `auto` é o modificador aplicado por defeito a variáveis **locais**, e não deve ser aplicado a variáveis globais, logo para os propósitos desta UC pode ser ignorado.

O modificador `static` permite-nos estender o tempo de vida de uma variável local até ao fim do programa. Quando usado em variáveis locais, a variável é inicializada apenas na primeira chamada da função e o valor da variável será mantido entre chamadas.

```
1   void func1() {
2       static int a = 0;
3       a = a + 2;
4   }
5
6   int main(int argc, char** argv) {
7       func1();
8       func2();
9       //variável a da func1 tem agora o valor 4
10  }
```