INSTITUTO POLITÉCNICO DE VISEU

Escola Superior de Tecnologia e Gestão de Lamego

ASPETOS DE PROGRAMAÇÃO MODULAR EM *C*

Carlos Jorge Almeida Costa

Viseu, julho de 2017

INSTITUTO POLITÉCNICO DE VISEU

Escola Superior de Tecnologia e Gestão de Lamego

ASPETOS DE PROGRAMAÇÃO MODULAR EM *C*

Carlos Jorge Almeida Costa

Lição a apresentar no Instituto Politécnico de Viseu, para prestação de provas públicas de avaliação de competência pedagógica e técnico-científica, no âmbito do Art.º 6º do Decreto-Lei n.º 45/2016, de 17 de agosto, a que se referem os n.ºs 9, 10 e 11 do artigo 6.º do Decreto-Lei n.º 207/2009, de 31 de agosto, alterado pela Lei n.º 7/2010, de 13 de maio.

TABELA DE CONTEÚDOS

LISTA DE FIGURAS v

LISTA DE TABELAS vi

1 INTRODUÇÃO 1

2 OBJETIVOS 5

3 PRINCÍPIOS DE DESENVOLVIMENTO DE SOFTWARE 6

3.1 Fases do desenvolvimento de software 7

3.2 Princípios e práticas fundamentais 8

3.3 Metas da decomposição modular 10

3.4 Recursos disponíveis nos ambientes de desenvolvimento 10

4 ORGANIZAÇÃO MODULAR DE UM PROGRAMA EM C 12

4.1 Decomposição modular 13

4.2 Princípios e práticas de estruturação em funções 14

4.3 Princípio da separação da interface da implementação 17

5 PROGRAMAÇÃO MODULAR EM C 20

5.1 Ambiente de desenvolvimento 21

5.2 Tutorial de Programação Modular 21

5.3 Pré-processador 32

5.4 Regras de âmbito e classes de armazenamento 35

6 ESTRATÉGIAS PEDAGÓGICAS 42

6.1 Metodologias de Ensino/Aprendizagem 42

6.2 Recursos didáticos 43

6.3 Avaliação 43

7 EXERCÍCIOS DE CONSOLIDAÇÃO DE CONHECIMENTOS 44

8 CONCLUSÃO 47

ANEXO 49

BIBLIOGRAFIA 55

LISTA DE FIGURAS

Figura 1 – Uma visão simples do desenvolvimento de software (Vliet, 2007, p. 11)...... 6

Figura 2 – Estrutura Hierárquica de um programa 15

Figura 3 – Formato geral de uma função.. 16

Figura 4 – Exemplo de definição de funções 16

Figura 5 – Exemplo do header file para a definição da interface de um módulo. 18

Figura 6 – Exemplo do ficheiro fonte de implementação de um módulo. 19

Figura 7– Exemplo do header file de definição da interface do módulo myMat 24

Figura 8 – Exemplo do ficheiro fonte de implementação do módulo myMat................ 25

Figura 9 – Exemplo do ficheiro fonte que implementa um programa de testes 26

Figura 10 – Esquema de construção de um programa baseado em vários ficheiros fonte 27

Figura 11– Exemplo do ficheiro fonte de outra implementação do módulo myMat 31

Figura 12 – Exemplo de utilização da classe de armazenamento auto.......................... 37

Figura 13 – Exemplo de utilização da classe de armazenamento register...................... 37

Figura 14 – Exemplo de utilização do modificador static, na definição de uma função 38

Figura 15 – Exemplo de utilização da classe static no âmbito de uma função............... 38

Figura 16 – Exemplo de utilização da classe de armazenamento extern........................ 39

Figura 17 – Exemplo de aplicação das regras de âmbito e classes de armazenamento. 40

LISTA DE TABELAS

Tabela 1 – Principais header files da Biblioteca Standard de C 23

1 INTRODUÇÃO

Esta publicação pretende ser um documento de suporte à lição de 60 minutos, a que alude a alínea b) do n.º 1 do artigo 2.º do Regulamento n.º 116/2011, de 16 de fevereiro, publicado em Diário da República, 2ª série – N.º 33 – 16 de Fevereiro de 2011, anexo ao Despacho n.º 57/2016, de 22 de dezembro de 2016, emanado pelo Ex.º Senhor Presidente do Instituto Politécnico de Viseu (IPV) e que define as provas públicas de avaliação de competência pedagógica e técnico-científica, a que se referem os n.ºs 9, 10 e 11 do artigo 6.º do Decreto-Lei n.º 207/2009, de 31 de agosto, alterado pela Lei n.º 7/2010, de 13 de maio.

As provas em questão serão prestadas na área ou áreas disciplinares de Engenharia Informática, em geral e Ciências Informáticas, em particular, em que o autor desempenha funções como Equiparado a Professor Adjunto na Escola Superior de Tecnologia e Gestão de Lamego (ESTGL) do Instituto Politécnico de Viseu (IPV). Atualmente, o autor é regente e leciona aulas teóricas, teórico-práticas e práticas das seguintes unidades curriculares: Fundamentos de Programação, Algoritmia e Estruturas de Dados, Programação Orientada a Objetos, Tópicos Avançados de Programação, Sistemas Operativos e Sistemas Distribuídos. Este também desempenha as restantes funções dos docentes do ensino superior, em geral e da categorial funcional de professor adjunto, em particular.

A lição em causa situa-se no âmbito da unidade curricular (UC) de Fundamentos de Programação do curso de Licenciatura em Engenharia Informática, da ESTGL do IPV. Trata-se de uma UC de cariz teórico-prática e constitui a primeira UC do curso, na área técnico-científica de programação de sistema computacionais.

Indubitavelmente, a programação de sistemas computacionais é uma área fundamental no perfil profissional de um Engenheiro Informático e Telecomunicações, uma vez que este compreende as atividades inerentes ao desenvolvimento de *software*, concretamente da implementação de programas que permite a materialização de soluções informáticas.

Para o desenvolvimento de profissionais competentes, é fundamental uma formação inicial consistente em fundamentos de programação. É efetivamente um desafio, numa unidade curricular inicial de programação, estabelecer os princípios e práticas subjacentes às disciplinas de desenvolvimento de *software*, uma vez que é necessário que os

estudantes desenvolvam maioritariamente competências de programação. No entanto, consideramos que, é fundamental desde cedo aplicar os princípios e práticas fundamentais inerentes às disciplinas de desenvolvimento de *software*, nomeadamente no que concerne à abstração, modularização, ocultação de informação e independência de contextos. Só com a aplicação destes princípios se consegue favorecer a robustez, manutenção, reutilização e evolução do *software*, cada vez mais complexo.

Sendo esta uma UC teórico-prática de fundamentos de programação, foi necessário optar por uma linguagem de programação. A opção recaiu, naturalmente na linguagem *C*, por ser uma linguagem considerada apropriada para a aprendizagem de técnicas de programação, uma vez que é uma linguagem concisa e poderosa (P. J. Deitel & Deitel, 2016; Kernighan & Ritchie, 1988). Esta é utilizada para o desenvolvimento ao nível do sistema operativo, de microcontroladores, de *embeded systems* e sistemas de tempo-real, entre outras áreas de aplicação. Para além disso, a sua sintaxe é utilizada na maioria dos ambientes e linguagens de programação mais utilizados atualmente, para o desenvolvimento de *software* (*e.g., C++, Java, C#*).

São assim definidos como objetivos de ensino/aprendizagem para a UC, o estudo dos conhecimentos teórico-práticos e desenvolvimento de aptidões e atitudes que envolvem a programação de sistemas computacionais, nomeadamente, em termos de fundamentos de algoritmia, técnicas de programação, em geral e na linguagem de programação *C*, em particular. É assim pretendido, que as competências a desenvolver pelos estudantes sejam:

- Interpretar e descrever problemas de forma a poderem ser resolvidos por meios computacionais;
- Identificar, adaptar e/ou desenvolver algoritmos para a resolução de problemas e propor soluções constituídas por algoritmos e estruturas de dados de média complexidade;
- Conhecer e aplicar os princípios da programação imperativa, estruturada e modular, bem como os diferentes recursos da linguagem de programação *C*;
- Desenvolver programas de média complexidade na linguagem de programação *C*;
- Analisar, adaptar e, eventualmente otimizar soluções implementadas na linguagem de programação *C*.

Em anexo, é fornecido o programa da UC, para que possa ser melhor compreendido todos os aspetos que a caracterizam.

Sendo a linguagem *C* poderosa e versátil, poderá encerrar alguns perigos, quando utilizada para a aprendizagem dos fundamentos de programação. No entanto, consideramos que não é uma desvantagem, mas sim uma vantagem, que poderá ser obtida se formos rigorosos nos princípios e práticas de programação, incluindo *standards* e convenções fundamentais.

Assim, o tema desta lição incide sobre os aspetos de específicos da programação modular na linguagem *C*. Insere-se no tópico 6. Funções e estruturação de programas, do programa da UC, em anexo. Naturalmente que numa lição, não é possível cobrir todos os aspetos respeitantes ao tópico na sua totalidade. É condição, para a apresentação da lição, que os estudantes já tenham conhecimentos de fundamentos de funções em *C*. De qualquer, forma, no que concerne a esta publicação, para poder ser considerada compreensiva, aborda os aspetos necessários ao efetivo ensino/aprendizagem de programação modular na linguagem *C*.

Pensamos que a lição "Aspetos de Programação Modular em *C*", é recorrente no processo de ensino/aprendizagem de fundamentos de programação, uma vez que a tendência de quem aprende, eventualmente influenciado pelos manuais que estuda, é colocação de toda a solução num único ficheiro fonte, eventualmente numa única função. Ora, como é compreensível. Assim, será fundamental incutir princípios e práticas de abstração, modularização, ocultação de informação e independência de contextos ou separação de preocupações (*concerns*), desde o início no processo de ensino/aprendizagem de técnicas de programação.

O resto do documento é organizado da seguinte forma. No Capítulo 2, são definidos os objetivos específicos da lição. Naturalmente que estes se enquadram nos objetivos e competências a desenvolver na UC pelos estudantes.

Seguidamente, são desenvolvidos 3 capítulos com os conteúdos teórico-práticos, que efetuam o enquadramento teórico e descrevem com profundidade conveniente, as matérias a abordar na lição. Assim, no Capítulo 3, são descritos os princípios de desenvolvimento de *software*, independentemente da metodologia de engenharia de *software*. São abordadas as fases, princípios e práticas fundamentais do desenvolvimento de *software*. Seguidamente são definidas as metas a atingir com a decomposição modular

e, por fim, neste capítulo, é apresentada uma perspetiva geral, com a identificação de casos concretos nos vários ambientes de desenvolvimento. Aqui é definida uma perspetiva evolutiva e histórica dos ambientes de desenvolvimento.

No Capítulo 4, são descritos e caracterizados os aspetos da organização modular de um programa em *C*. Nomeadamente, são abordadas as vantagens e formas de decomposição modular, em geral e a problemática de estruturação modular na linguagem *C*. A seguir, são definidos os princípios e prática de programação estruturada e da estruturação do código em funções, fundamental no âmbito de um módulo. Aqui, é ainda caraterizado o princípio da separação da *interface* da implementação, como metodologia de desenvolvimento, bem como a sua aplicação no âmbito da linguagem *C*.

O Capítulo 5, corresponde, em termos técnico-científicos, ao cerne da questão subjacente ao tema desta lição. Aqui, são aplicados os princípios, recursos e boas práticas de programação modular em *C*. Começamos com a apresentação do ambiente de desenvolvimento, constituído por ferramentas da *GNU*, característica por ser *software livre,* utilizado para a demonstração e teste dos exemplos desenvolvidos. Seguimos com uma abordagem tutorial, com a demonstração de exemplos de aplicação completos. Nesta, é efetuado o desenvolvimento de um módulo reutilizável, bem como a possibilidade da sua substituição e/ou evolução, pela implementação de uma versão mais eficiente. Depois são abordados os recursos específicos, que permitem o desenvolvimento de programas modulares, concretamente são descritos e exemplificados: o pré-processador e a sua linguagem, para a definição das *interfaces* e respetiva implementação; bem como, as regras de âmbito e classes de armazenamento, fundamentais para a aplicação dos princípios de abstração, modularização, ocultação de informação e separação de contextos.

No Capítulo 6, são definidas as estratégias pedagógicas a adotar para a apresentação da lição, recursos didáticos a utilizar, bem como, as atividades de avaliação para a validação das competências a desenvolver.

No Capítulo 7, são apresentados exercícios de consolidação dos conhecimentos a disponibilizar aos estudantes.

Finalmente, no Capítulo 8, são apresentadas as principais conclusões e por fim um anexo, com o programa da UC, utilizado no curso de Eng. Informática e Telecomunicações.

2 OBJETIVOS

Tendo em atenção os objetivos e competências estabelecidas para a UC de Fundamentos de Programação, os objetivos a cumprir após esta lição, são os que a seguir se apresentam.

Assim, é pressuposto que no final da lição os estudantes sejam capazes de aplicar os princípios e boas práticas no âmbito da programação modular de sistemas computacionais, em geral e na linguagem de programação *C*, em particular.

Concretamente:

- Conhecer os princípios de modularidade e independência de contextos nos âmbitos do desenvolvimento de software, em geral e da programação de sistemas computacionais, em particular;
- Compreender a necessidade de organização modular de uma aplicação e/ou programa;
- Aplicar os princípios, recursos e boas práticas de programação modular no âmbito da linguagem de programação *C*.

Naturalmente que é compreensível que na programação, o desenvolvimento das matérias é cumulativo. Também é percetível que no âmbito de uma única lição, os objetivos traçados são muito ambiciosos, principalmente num processo de aprendizagem, no âmbito dos fundamentos de programação. Assim, os objetivos traçados, são os pretendidos nesta fase de aprendizagem, após o desenvolvimento de todos os tópicos até ao tópico 6. Funções e estruturação de programas. É preciso notar o tema desta lição, não engloba toda a temática deste tópico, pelo que os dois primeiros objetivos concretos, é pressuposto, já estarem adquiridos, antes desta lição.

Para esta lição é assim, definido especificamente o último objetivos concreto.

De qualquer forma, considerando a necessidade de efetuar um enquadramento teórico mais abrangente e descrever com profundidade os conceitos a abordar, para efeitos desta publicação, retomamos a totalidade dos objetivos traçados, como os objetivos a serem alcançados, com a exposição nos próximos 3 capítulos.

3 PRINCÍPIOS DE DESENVOLVIMENTO DE *SOFTWARE*

Independentemente das metodologias de engenharia de *software*, é usual que o desenvolvimento de *software* se estruture nas seguintes fases (ver Figura 1): **engenharia de requisitos**, **conceção**, **implementação**, **testes** e **manutenção** (Vliet, 2007, sec. 1.2).

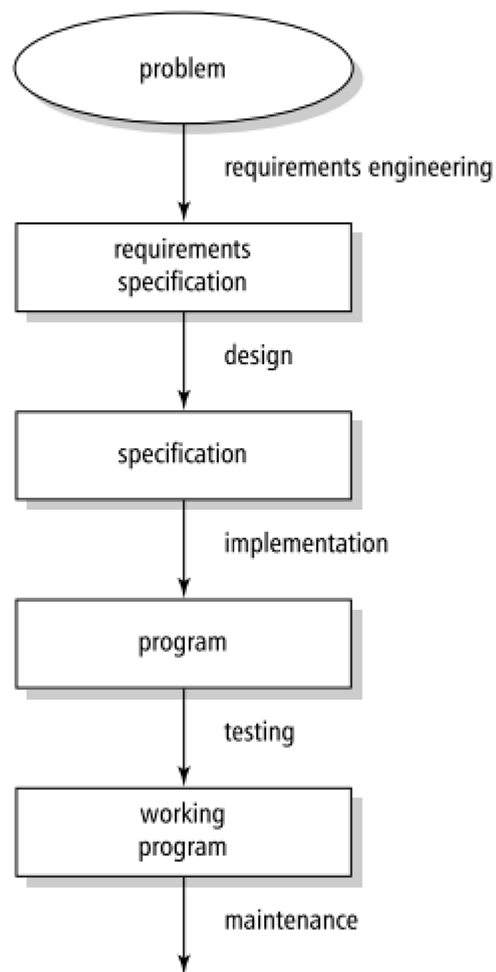

Figura 1 – Uma visão simples do desenvolvimento de *software* (Vliet, 2007, p. 11)

Naturalmente, que quando se trata de programas de pequena dimensão, dependendo do tipo de projeto em causa ou ambiente de desenvolvimento, estas fases não são estritamente sequenciais e separadas, havendo normalmente sobreposição. Segue-se uma breve caracterização de cada fase.

Tendo em atenção que o desenvolvimento de *software* é tão efetivo, quanto os princípios e práticas aplicados, estes serão apresentados a seguir. Estes são abstração, modularização, ocultação de informação e independência de contextos e, normalmente, estão principalmente ligados às fases de engenharia de requisitos, conceção e implementação. Isto porque é com estes princípios e práticas que é possível cumprir as fases de testes e manutenção.

De seguida são definidas as metas a alcançar com a decomposição modular, com vista a perceber, quais os recursos a serem disponibilizados pelos ambientes de desenvolvimento.

Por fim serão definidos os recursos necessários ao desenvolvimento modular de *software*, bem como os principais ambientes que os disponibilizam de forma explicita, nos vários paradigmas de desenvolvimento.

3.1 Fases do desenvolvimento de *software*

O principal objetivo da fase de **engenharia de requisitos** é efetuar a análise e a descrição completa do problema a resolver, bem como dos requisitos impostos pelo ambiente no qual o sistema vai funcionar. Parte desta fase deverá versar um estudo de viabilidade económica e técnica. O documento resultante desta fase é normalmente designado por **especificação de requisitos.**

Durante a fase de **conceção**, é desenvolvido um modelo de todo o sistema. Normalmente é seguida uma aproximação de refinamento progressivo (abordagem *top-down*), onde o problema é decomposto em peças manegáveis, a partir da **arquitetura** global, que designaremos por conveniência por **componentes**. A **funcionalidade destes componentes** e a **interface** entre eles, são especificadas de forma precisa. O resultado desta fase, a **especificação (técnica)** é o suporte e ponto de partida para a fase de implementação.

Na fase de **implementação**, concentramo-nos nos componentes individuais, na sua conceção (definição das opções de implementação de cada componente) e concretização num ambiente de desenvolvimento, utilizando uma linguagem de programação conveniente. É preciso notar que a primeira meta de um programador deverá ser o desenvolvimento de um programa correto, flexível, confiável, fácil de ler e bem documentado. O resultado desta fase são **programas executáveis.**

Atualmente, e principalmente no âmbito das metodologias ágeis, falar numa fase de **testes** autónoma e após a implementação é uma falácia. Estes deverão ser levados a cabo durantes todas as fases anteriores, como forma de correção dos erros antes da fase de implementação. Normalmente, os testes são assumidos nas fronteiras das fases como **verificação** da correção, das transições entre as fases subsequentes, bem como para a **validação** do cumprimento dos requisitos impostos.

Após a implantação do *software* em ambiente de execução, é necessário proceder à **manutenção** corretiva, bem como, numa perspetiva atual, fazer progredir o sistema com base em necessidades de alteração e evoluções verificadas. A manutenção compreende assim todas as atividades necessárias para manter o sistema operacional após a sua implantação em ambiente de produção.

3.2 Princípios e práticas fundamentais

A implementação, como fase de concretização das fases anteriores (engenharia de requisitos e conceção do sistema), normalmente inclui também a conceção dos componentes e constitui o suporte às fases posteriores (testes e manutenção). Esta deverá ter subjacente os seguintes princípios e práticas fundamentais: abstração, modularização, ocultação de informação e independência de contextos.

Abstração significa a concentração nos aspetos essenciais e ignorar os detalhes em cada nível e/ou fase no desenvolvimento. Normalmente pode falar-se em dois tipos de abstração: procedimental e de dados. A **abstração procedimental** corresponde à natural subdivisão das soluções em procedimentos e/ou funções, sendo que cada um destes, é uma abstração da operação a desempenhar. Esta permite a natural organização hierárquica dos componentes. A **abstração de dados**, corresponde também em termos de nível e/ou fase, a ignorar os tipos de dados envolvidos, relevando efetivamente as operações subjacentes sobre estes. Como nota final, pode ser identificado um terceiro tipo de abstração, abstração de controlo. Na **abstração de controlo** é ignorada a ordem precisa em que uma sequência de eventos deve ser manipulada (Vliet, 2007, sec. 12.1).

Modularização é a decomposição do sistema num conjunto de módulos e na definição das relações entre esses módulos. Até ao momento, utilizamos a noção de componente de forma intuitiva (unidade identificável da conceção do *software*). Por questões de clareza de linguagem, esta poderá corresponder, na implementação, à noção intuitiva de

"módulo". Generalização, um componente poderá ser constituído por pelo menos um módulo.

Um **módulo** é assim um elemento com uma responsabilidade (*concern*), mas que não pode ser confundido com um subprograma, procedimento ou função. Compreende naturalmente, procedimentos e funções fortemente relacionados, para além de definições de tipos de dados e os dados propriamente ditos. Um módulo deve ser especificado, em termos formais, pela separação entre ***interface*** e **implementação**. A interface define a informação necessária e suficiente para a utilização efetiva do módulo, que deverá ser exportada. A implementação é constituída pela definição da funcionalidade, sem especificar os recursos a utilizar e respetivos detalhes de implementação (Parnas, 1972a; Parnas, Clements, & Weiss, 1984).

Podemos assim falar em programação modular, que corresponde ao desenvolvimento das técnicas de escrita e tecnologia de construção de programas, que permitem que um módulo seja escrito com pouco conhecimento do código de outro módulo e que permita que módulos sejam reconstruídos e substituídos sem reconstruir todo o sistema. Assim, ao nível da linguagem de programação, usualmente, um módulo corresponde a uma unidade de compilação.

A efetividade da "modularização" depende dos critérios utilizados para a decomposição do sistema em módulos. Em (Parnas, 1972b) são descritos e comparados dois tipos de decomposição, uma "convencional", seguindo o fluxo sequencial de execução dos procedimentos e funções (fluxograma) e uma apelidada de "não convencional", baseada num critério de ocultação de informação. Naturalmente foi provado a efetividade da segunda abordagem, por privilegiar a ocultação de informação entre módulos e por consequência promover a facilidade de manutenção do sistema.

Pelas provas dadas no desenvolvimento de *software* inerentemente complexo, a técnica de **ocultação de informação** tem sido um princípio e prática recorrente e fundamental (Parnas et al., 1984). Pressupõe o encapsulamento dos dados no âmbito do módulo, não permitindo o seu acesso direto, nem o conhecimento dos recursos empregues para a sua salvaguarda e eventual estruturação. O acesso é controlado e validado a partir de operações que constituem a interface pública, exportada pelo módulo.

Este condicionamento de acesso às informações (operações disponibilizadas pela *interface* do módulo formalmente definida), impõe a necessária **independência de contextos**, no que diz respeito aos dados que pertencem a cada um dos módulos**.**

Todos estes princípios e práticas estão intrinsecamente relacionados. O processo de refinamento progressivo e ocultação de informação concretizam a abstração, modularidade e independência de contextos nos âmbitos do desenvolvimento de *software*.

3.3 Metas da decomposição modular

O principal objetivo da decomposição em módulos, é a redução do custo total do *software* (económico e/ou temporal), permitindo que os módulos possam ser concebidos, implementados e mantidos independentemente uns dos outros. Concretamente, a decomposição em módulos pressupõe os seguintes objetivos:

- A estrutura do módulo deverá ser o suficientemente simples para que seja completamente compreendida;
- Deverá ser possível modificar a implementação de um módulo sem ter conhecimento da implementação de outros módulos e sem afetar o comportamento de outros módulos;
- Deverá ser possível efetuar modificações previsíveis sem modificar a *interface* do módulo e caso seja fundamental, por não ser previsível, tanto quanto possível, deverá afetar os módulos menos utilizados;
- Deverá ser possível efetuar alterações significativas, como um conjunto de alterações independentes a módulos individuais, i.e., exceto para alterações de *interface*, as equipas de programadores para alterar os módulos individuais não necessitam de comunicar.

Como consequência das metas definidas, o *software* desenvolvido é composto por muitos módulos pequenos, sendo este desenvolvimento efetuado por composição.

3.4 Recursos disponíveis nos ambientes de desenvolvimento

Independentemente do paradigma de programação (imperativo, orientado a objetos, funcional) e plataforma de desenvolvimento de alto nível, têm sido introduzidos recursos específicos nas linguagens de programação, para dar suporte à definição de módulos e potenciar as vantagens, que serão apresentadas no capítulo seguinte.

A primeira linguagem a dar suporte efetivo à estrutura modular, conforme foi definida anteriormente, foi a linguagem MODULA (Wirth, 1976) e a sua sucessora imediata MODULA-2 (Wirth, 1978). Trata-se de uma evolução da linguagem PASCAL, também proposta por Niklaus Wirth (Wirth, 1971), que implementou o recurso designado por *module*. Este permite agrupar procedimentos, funções, tipos de dados e variáveis, tendo o programador controlo preciso sobre os elementos (nomes) importados e exportados no âmbito do módulo.

Todos os ambientes de desenvolvimento e respetivas linguagens de programação, com a generalização do paradigma orientado a objetos e, posteriormente, baseado em componentes ou arquiteturas orientadas a serviços, tem incluindo recursos de suporte à modularização. Esses recursos podem ser designados de forma distinta, mas englobam as facilidades associadas aos módulos, já descritas. A título de exemplo, apresentamos os exemplos dos ambientes de desenvolvimento de *software* mais utilizados atualmente. A linguagem *Java*, disponibiliza *interfaces*, que permitem separar a definição das *interfaces* da implementação (em *classes*), bem como os *packages* que permitem agregar, num mesmo espaço de nomes, *interfaces*, *classes* e *enumerations*. Estas são organizadas de forma hierárquica, permitindo a sua importação noutros módulos (*packages*). É possível várias restrições de acesso aos elementos constituintes (Schildt, 2014, Chapter 9 e 12). A linguagem *C#*, que disponibiliza *interfaces*, que como no *Java*, permitem separar a definição das *interfaces* da implementação (aqui designados por tipos, concretamente *reference types* ou *classes* e *value types* ou *structs*). Também disponibilizam os *namespaces*, que permitem agregar num mesmo espaço de nomes, as *interfaces* e tipos. O mesmo que já foi dito relativamente à hierarquia, utilização e restrição de acesso nas *packages,* se aplicam aos *namespaces* (Albahari & Albahari, 2016, Chapter 2). A linguagem *C++*, embora não inclua o recurso *interface*, permite a definição de *classe* puramente abstratas, par o efeito. Atualmente, esta permite a separação de âmbitos de identificadores, pela disponibilização de *namespaces*, que também permitem agregar identificadores (constantes, variáveis, tipos e funções), num mesmo espaço de nomes, que permite ser utilizado noutros âmbitos (P. Deitel & Deitel, 2014, sec. 23.4).

4 ORGANIZAÇÃO MODULAR DE UM PROGRAMA EM *C*

Muitas vezes são utilizados indiferentemente os termos programa ou aplicação. Assim, começamos por definir cada um dos termos. Um **programa** é um conjunto de instruções que implementam as operações num computador ("dizem ao computador o que fazer"). Isto é, um programa é qualquer unidade de *software* que pode executar num computador. Uma **aplicação**, é um programa ou, usualmente, um grupo de programas para a resolução de problemas do utilizador final, num sistema computacional. Também são designadas *Software* de Aplicação e permitem ao utilizador executar um grupo coordenado de operações, tarefas ou atividades para um fim específico. Podemos dizer que aplicações são programas, mas programas podem não ser necessariamente aplicações (*e.g.,* os serviços do sistema operativo, por exemplo, que executam em *background,* não são aplicações).

A programação modular, agrupa um conjunto relacionado de funções num módulo. Um módulo é dividido na sua *interface* e na sua implementação. O módulo exporta a *interface* e os clientes do módulo (programa), importam a *interface* para poderem aceder à funcionalidade do módulo. A implementação dos módulos deverá ser privada e naturalmente ocultada dos clientes. A divisão de programas em módulos, é um princípio poderoso de organização, para a conceção e implementação de programas. Os módulos fornecem **abstração**, **encapsulamento** e **ocultação de informação**, tornando a estrutura de um programa de grande dimensão inteligível. Uma conceção e implementação cuidada dos módulos, também promove a manutenção e reutilização de *software*.

Muitas dos ambientes de desenvolvimento muito utilizados atualmente, para o desenvolvimento de aplicações, disponibilizam recursos específicos para o desenvolvimento de programação modular, como já foi descrito na secção 3.4.

A linguagem *C*, apesar de ser utilizada para o desenvolvimento de programas complexos (*e.g.,* sistemas operativos), não fornece explicitamente recursos para suportar programação modular. No entanto existem facilidades na linguagem, incluindo o pré-processador e algumas convenções, que podem ser combinadas para, convenientemente, permitir desenvolver uma técnica e prática efetivas de programação modular em C.

Assim, no resto do capítulo, começamos por abordar as vantagens que envolve a decomposição modular, bem como os recursos disponibilizados nos ambientes de desenvolvimento e a problemática de estruturação modular na linguagem *C*.

A seguir são abordados os princípios e prática de estruturação de um programa em *C*, como forma de estruturação no âmbito, de um módulo, da funcionalidade deste.

Por fim será apresentada e caracterizada uma metodologia, baseada na separação da *interface* da implementação, aplicada à linguagem *C*, que permite potenciar a reutilização de *software*.

4.1 Decomposição modular

Normalmente as aplicações e os programas são decompostos em módulos. Assim, as vantagens da decomposição modular são as seguintes:

- Criação de módulos, em que cada um implementa uma responsabilidade (*concern*) do sistema, pela separação entre *interface* e implementação;
- Encapsulamento numa unidade compilada dos dados e das operações que alteram os seus estados e ocultação da informação e detalhes de implementação;
- Favorecimento e facilidade do desenvolvimento em equipa, pela natural atribuição de módulos às equipas e à necessidade reduzida de comunicação a não ser para a definição das *interfaces* e funcionalidade de cada módulo;
- Com a utilização de recursos que permitem a abstração procedimental e de dados, favorecimento e facilidade da reutilização de *software*;
- Favorecimento e facilidade da manutenção, quer dos módulos criados (pela substituição de módulos após a correção) quer da evolução natural do sistema, pela natural concentração de esforço em cada um dos módulos.

Como já foi descrito, os ambientes de desenvolvimento e respetivas linguagens de programação, utilizadas atualmente para o desenvolvimento de aplicações, suportam e disponibilizam recursos específicos para programação modular (ver secção 3.4). Independentemente da designação desses recursos, todos têm, pelo menos, os seguintes aspetos em comum:

- Permitem a separação entre *interface* e implementação;
- Definem módulos ou espaços de nomes (*e.g., namespaces* ou *packages*), constituídos por tipos (*e.g., classes, structs, enumerations*), dados (variáveis ou

constantes) e operações (*e.g.,* funções ou métodos), organizados de forma hierárquica;

- Restringem acessos;
- Exportam (pela definição de elementos públicos ou protegidos) e importam módulos (pela indicação dos módulos que pretendem utilizar).

Como já foi introduzido, a linguagem de programação *C*, não suporta explicitamente recursos, especificamente para programação modular. As principais razões prender-se-ão, eventualmente, com a sua génese e natureza (versatilidade e utilização ao nível dos sistemas operativos, em geral e do sistema *UNIX*/Linux, em particular; muitas vezes para o seu próprio desenvolvimento).

No âmbito do desenvolvimento de *software* complexo, utilizando a linguagem *C* (aplicações ou sistemas operativos), como organizar programas de média ou grande dimensão? Pouca bibliografia fornece especificamente conhecimento sobre estes aspetos, principalmente ao nível do ensino/aprendizagem da linguagem. Normalmente os exemplos são colocados num único ficheiro fonte. Assim, sem nenhum princípio de orientação sobre como organizar o código, os programas de grande dimensão, para além de serem de difícil compreensão, são praticamente impossíveis de manter (Hayes, 2001).

4.2 Princípios e práticas de estruturação em funções

Seguindo os princípios e práticas de programação estruturada, comummente aceites, na organização de um programa, devem ser utilizados estruturas de bloco (com definição de início e fim, condicionais e de repetição) sub-rotinas e não utilizar saltos incondicionais (P. J. Deitel & Deitel, 2016, Chapter 3; Dijkstra, 1968; "Structured programming," n.d.).

Na linguagem *C*, um ficheiro com código fonte (utilizaremos também a designação de ficheiro fonte) é constituído por definições (*e.g., structs, unions, enums*), declarações e funções. Todas as instruções são exclusivamente colocadas em funções (sub-rotinas) (Kernighan & Ritchie, 1988, Chapter 1).

Para a constituição de um programa executável é necessário a presença de uma função, dita principal ou *main*, a partir da qual começa a execução. Esta pode ter vários protótipos predefinidos, isto é, declaração da função, com valor de retorno, identificador (nome da função) e tipos dos parâmetros formais. Seguindo o *Standard C11* da ISO (*International*

Standards Organization) (P. J. Deitel & Deitel, 2016, Chapter 1), o protótipo mínimo é o seguinte:

int main(void); (1)

Tal como acontece com a maioria dos ambientes de desenvolvimento, é fornecido uma biblioteca de módulos que permitem desempenhar tarefas específicas e fundamentais para os programas e cujas operações, podem ser invocadas e integradas nos programas do utilizador via instruções da linguagem de programação.

No caso do ambiente de desenvolvimento em *C*, é fornecida uma biblioteca normalizada (*Standard Library*), constituída por um conjunto extensivo de funções para desempenhar tarefas fundamentais, tais como: efetuar entradas e saídas, gestão de memória, manipulação de *strings* (sequências de caracteres) e outras. Estas são normalmente designadas pelas funções predefinidas e são agrupadas em módulos de funções relacionadas (*e.g.*, funções matemáticas, de *standard input*/*output*).

O programador também pode definir funções, que deverão efetuar uma única tarefa específica. Dentro do possível, é conveniente evitar dependências externas de dados (aplicação do princípio e prática de modularização e independência de contextos). As funções são utilizadas, através da sua invocação pela especificação do seu nome e dados a passar como argumentos ou parâmetros.

O programa deverá funcionar, pela invocação, a partir da função *main()*, das várias funções descritas, que por sua vez poderão invocar outras funções, aplicando os princípios e práticas subjacentes à programação estruturada. Assim, um programa exibe naturalmente uma estrutura hierárquica em árvore (ver Figura 2), cuja raiz implementa a função *main()* e os nós intermédios e folhas, implementam funções do utilizador ou constantes da Biblioteca *Standard* (princípio e prática da abstração procedimental).

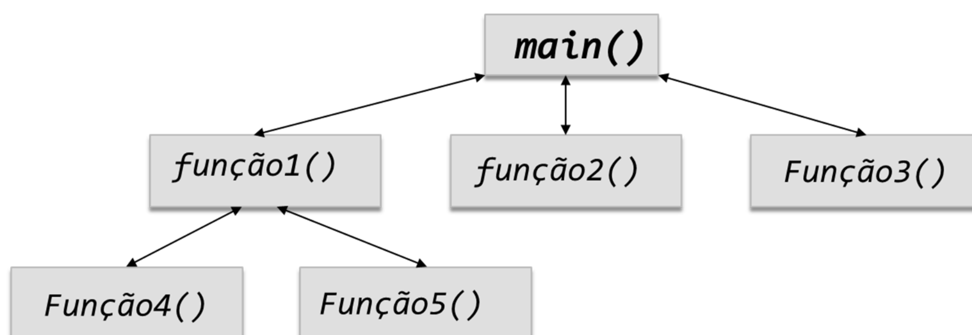

Figura 2 – Estrutura Hierárquica de um programa

Uma função poderá ser definida, conforme o formato apresentado na Figura 3. Todas as variáveis declaradas na função (*declarações*) ou declaradas na *lista-de-parâmetros* (lista de parâmetros formais), são locais e só podem ser acedidas no âmbito da função em causa. O *tipo_valor_de_retorno* é um tipo nativo, definido pelo programador ou *void*, caso a função não retorne um valor (se não for especificado um tipo, é assumido pelo compilador, o tipo *int*, sedo uma situação a evitar).

```
tipo_valor_de_retorno nome_da_função (lista-de-parâmetros)
{
  declarações;

  instruções;
}
```

Figura 3 – Formato geral de uma função

Segue um exemplo da definição de funções (ver Figura 4).

```
1  /*
2   * Função quadrado
3   */
4  #include <stdio.h>
5
6  /* protótipo da função */
7  int square( int );
8
9  int main(void) {
10     int x;
11     for ( x = 1; x <= 10; x++ )
12       printf( "%d  ", square( x ));
13     printf("%s", "\n" );
14     return 0;
15 }
16
17 /* Definição da função */
18 int square( int y )
19 {
20    return y * y;
21 }
```

Figura 4 – Exemplo de definição de funções

Aqui, na linha 5, são incluídos as definições e protótipos da biblioteca de funções de entradas e saídas, para a utilização da função *printf()*. Na linha 7 é definido o protótipo da função *square()* , para que esta possa ser utilizada na função *main()* e o compilador

validar os tipos do valor de retorno, número e tipos de dados dos parâmetros, bem como a ordem destes, antes da função ser definida nas linhas 18 a 21.

Por fim, e neste momento, é suficiente dizer que na linguagem *C*, todos os parâmetros são passados às funções por valor (é passada na pilha do programa, na *frame* associada à invocação da função, uma cópia do valor atual do parâmetro). Se houver alteração ao valor do parâmetro dentro da função, não se reflete no exterior da mesma. Caso seja pretendida a passagem por referência, é necessário utilizar apontadores, no entanto esta matéria está fora do âmbito desta lição.

4.3 Princípio da separação da *interface* da implementação

Os programadores lidam naturalmente com *Application Programming Interfaces* (*API*s). A maioria utiliza *API*s e as bibliotecas que as implementam (normalmente são designadas só por *API*s, mas efetivamente incluem as bibliotecas, normalmente dinâmicas, que podem ser reutilizadas) na grande maioria dos programas que desenvolvem.

Relativamente poucos criam e distribuem *API*s. Normalmente, é mais fácil escrever código específico para o programa. *API*s bem definidas e genéricas são bastante mais complicadas de conceber e implementar.

No entanto, o ponto de partida, passa pela separação entre *interface* e a sua implementação. Esta também poderá ser designada por uma metodologia de conceção baseada em *Interface* (*Interface-based design*). Normalmente, esta é independente de uma linguagem de programação e ambiente de desenvolvimento, em particular.

A conceção baseada em *interfaces* torna o desenvolvimento mais rápido pela construção e reutilização de uma fundação de *API*s genéricas que são utilizadas em vários programas. Normalmente, é necessário a aplicação dos princípios e práticas de abstração de dados, no âmbito de uma linguagem de programação (*e.g.*, apontadores para *void*, na linguagem *C, Templates* na linguagem *C++*, ou *Generics* nas linguagens *Java* e *C#*).

Assim, um programa de grande dimensão, é construído a partir de pequenos módulos. Estes módulos fornecem funções para manipular as estruturas de dados que encapsulam. Idealmente, a maioria destes módulos deveriam estar prontos e incorporados a partir de bibliotecas. Infelizmente, na prática, não é realidade, sendo estes, desenvolvidos de raiz. A conceção e implementação de bibliotecas é um processo complicado. Normalmente é

necessário ponderar cuidadosamente entre generalização, simplicidade e eficiência (Hanson, 1997).

Uma biblioteca desenvolvida em linguagem *C*, deverá exportar um conjunto de módulos, que fornecem funções e encapsulam tipos de dados abstratas (*ADT – Abstract Data Types*) ou, também designadas, estruturas de dados genéricas.

Um módulo compreende duas partes: a ***interface*** e a sua **implementação**. A *interface* específica o que o módulo faz. A implementação especifica como o módulo cumpre o propósito disponibilizado pela *interface*. Um módulo, normalmente, só tem uma *interface*, mas pode ter várias implementações.

Como já foi referido, um cliente é uma peça de *software* que utiliza um módulo. Um cliente importa *interfaces*, tendo necessidade de conhecer unicamente estas *interfaces*. A implementação concretiza o módulo, ocultando estruturas de dados e algoritmos. A *interface* especifica os identificadores (de tipos e de funções) que o cliente pode utilizar.

A linguagem *C* tem suporte mínimo para a separação da *interface* da implementação, mas com algumas convenções podem ser alcançados a maioria dos benefícios desta metodologia de separação de *interface*/implementação.

Vejamos como isto se concretiza. Uma *interface* é especificada num ficheiro com a terminação *.h* (muitas vezes a terminação é designada por extensão). Este ficheiro fonte é chamado *header file* ou ficheiro de cabeçalho.

Este *header file* permite efetuar exclusivamente declarações, nomeadamente tipos e protótipos de funções (ver Figura 5).

```
1  /* arith.h */
2
3  extern int intMax( int, int );
4  extern int intMin( int, int );
```

Figura 5 – Exemplo do *header file* para a definição da *interface* de um módulo.

A implementação é efetuada num ficheiro fonte característico da linguagem *C*, isto é, um ficheiro com a terminação *.c*. Este implementa o(s) algoritmo(s) específicos, em funções da linguagem *C* (ver Figura 6). Naturalmente que, é necessário que no ficheiro fonte de implementação conheça a *interface* em implementação, Isso é efetuado pela diretiva do pré-processador *#include* que consta da linha 2 da mesma figura.

```c
/* arith.c */
#include "arit.h"

int intMax( int x, int y) {
    return (( x > y ) ? ( x ) : ( y ));
}

int intMin( int x, int y) {
    return (( x > y ) ? ( y ) : ( x ));
}
```

Figura 6 – Exemplo do ficheiro fonte de implementação de um módulo.

Também o cliente importa a *interface*, com a diretiva do pré-processador *#include*, não necessitando de conhecer a implementação.

Em termos de distribuição da biblioteca, esta poderá ser construída, como tal, utilizando uma ferramenta de construção de programas (*e.g., gcc*). Contudo, esta mesma biblioteca pode ser distribuída em ficheiro objeto, isto é, um ficheiro com a terminação *.o*, produzido pela ferramenta de construção de programas, parametrizada para só efetuar a compilação (*e.g.,* parâmetro *-c* no *gcc*). Em qualquer um dos casos, o *header file* também tem que ser distribuído com a biblioteca.

Os exemplos apresentados pretendem demonstrar os recursos fundamentais e convenções desta metodologia de separação de interface e implementação, aplicada à linguagem *C*. Estes não demonstram, por questões de simplificação e melhor compreensão da exposição, os aspetos que envolvem a abstração de dados.

A exploração mais aprofundada dos vários recursos, que permitem efetivar esta metodologia, utilizando a linguagem *C*, é matéria do Capítulo 5.

5 PROGRAMAÇÃO MODULAR EM C

Após a descrição dos aspetos técnico-científicos, relativos aos princípios de desenvolvimento de *software*, em geral e da programação de sistemas computacionais, em particular; bem como, da necessidade, princípios e prática da organização modular de *software* de grande dimensão e complexo, em geral e na linguagem *C*, em particular; é hora de aplicar os princípios, recursos e boas práticas de programação modular no âmbito da linguagem de programação *C* e do seu ambiente de desenvolvimento.

Aqui será seguida a mesma abordagem introduzida por *Brian Kernighan* e *Dennis Ritchie*, aquando da publicação de *The C Programming Language* em 1978, referida em *Preface to the First Edition* do seu livro *The C Programming Language, Second Edition* (Kernighan & Ritchie, 1988). Foi assumido por estes, tratar-se um livro para auxiliar o leitor a aprender como programar em *C*. Aliás, esta abordagem tem sido seguida, desde então, para a estruturação dos vários livros, que descrevem a forma de programar numa linguagem de programação. Aliás, é normal que praticamente todos os livros de programação, independentemente da linguagem e ambiente de desenvolvimento, comecem com o emblemático "Olá, mundo" (*"hello, world"*, em inglês).

Assim, começamos por apresentar o ambiente de desenvolvimento, constituídos por um conjunto de ferramentas da *GNU*. Estas ferramentas são multiplataforma (computador e sistema operativo), permitindo ter testado os exemplos nos sistemas *Linux* e *Windows*.

Seguidamente apresentamos um tutorial de introdução à programação modular, que permitirá a aplicação dos princípios, recursos e práticas fundamentais de programação modular na linguagem *C*.

Por fim serão descritos, em secções separadas, os recursos e convenções disponibilizados pela linguagem *C* ambiente de desenvolvimento, de suporte à programação modular. Concretamente, são descritas o pré-processador e as diretivas que permitem operacionalizar a definição das *interfaces* e implementação de módulos, bem como, as regras de âmbito e classes de armazenamento, que permitem operacionalizar a ocultação e exportação de recursos, no âmbito dos módulos. Estes são demonstrados com exemplos elucidativos, permitindo complementar a exposição tutorial da secção 5.2.

5.1 Ambiente de desenvolvimento

Todos os exemplos foram testados nos sistemas operativos *Windows* e *Linux*. Para a criação dos programas foi utilizada a ferramenta de *software livre* (gratuito) de criação de programas executáveis em linha de comando *gcc* (*the GNU Compiler Collection*) e que é, vulgarmente, designada por compilador. Esta é uma ferramenta multilinguagem de programação e ambiente de desenvolvimento. Inclui *front ends* para *C*, *C++*, *Objective-C*, *Fortran*, *Ada* e *Go*, bem como bibliotecas para essas linguagens de programação ("GCC, the GNU Compiler Collection," n.d.). O *gcc* normalmente é livremente disponibilizado em todas as distribuições de *Linux*.

No entanto é uma ferramenta multiplataforma, cujos binários são disponibilizados para outros sistemas operativos, nomeadamente: *Windows, Solaris*, *macOS*, etc. No sistema operativo *Windows*, são disponibilizados principalmente os seguintes projetos: *Cygwin project* ("Cygwin," n.d.); *MinGW* ("MinGW," n.d.) e *mingw-w64* ("Mingw-w64," n.d.). O último é um avanço do projeto original *MinGW*, especificamente para suportar a arquitetura de *64-bits*.

5.2 Tutorial de Programação Modular

Resumindo e estendendo os princípios e práticas de estruturação em funções (ver secção 4.2), para aplicar no âmbito da programação modular, um programa ou biblioteca em linguagem *C* é constituído por funções, eventualmente, separadas por vários ficheiros fonte. Se se tratar de um programa executável, um, dos ficheiros fonte que o constitui, deverá possuir uma função principal onde a execução inicia e, cujo protótipo pode ser *int main(void);*. Podem ser adotados outros, em que o *void* é substituído por parâmetros formais, que permitem ao sistema operativo passar argumentos ao programa na invocação através da linha de comando, bem como permitir o acesso às variáveis de ambiente do sistema operativo.

Normalmente é mais simples conceber um programa seguindo a aproximação de refinamento sucessivo (aproximação *top-down*). Assim, na subdivisão do problema deverão ser identificadas as funções, seguindo o princípio de abstração procedimental e concebida a função principal, também designado de programa principal, noutros ambientes.

Se se pretender, desenvolver uma biblioteca, é normal concentramo-nos na funcionalidade a implementar e reutilizar noutros programas, identificando as funções a implementar, bem como o seu agrupamento, por afinidade, em módulos.

As funções definidas pelo programador, deverão devolver um valor de um tipo específico, ou caso não devolvam qualquer valor, o identificador deverá ser precedido de *void* (ver secção 4.2). Convém relembrar, que pelo princípio da separação e independência de contextos, cada função deverá conter os parâmetros necessários, para possibilitar a passagem dos dados, no momento da invocação. Preferencialmente, as funções devem trabalhar sobre variáveis locais, sendo de evitar a definição e/ou acesso a variáveis globais.

Na organização lógica dos ficheiros fonte, deverá ser privilegiada a criação de módulos, aplicando a metodologia de separação de *interface* e implementação. Estes módulos poderão permitir a criação de bibliotecas de funções, reutilizáveis noutros contextos, ou eventualmente uma melhor organização lógica, do programa em questão. Cada módulo deverá ser constituído por funções relacionadas.

A implementação dos módulos, pressupõe a separação em dois ficheiros fonte. Um para a definição da *interface* ou *header file* (ficheiro com terminação *.h*) e, pelo menos outro, para a implementação do módulo (ficheiro com terminação *.c*). No *header file* são colocadas declarações globais (incluído a importação de *interfaces* de outros módulos, através da diretiva de pré-processador *#include*); definições de tipos e/ou protótipos de funções. Nos ficheiros de implementação são colocadas as diretivas de importação de *header files* (dos módulos de funções da Biblioteca *Standard,* no formato *#include <ficheiro.h>* e do(s) *header files* dos módulos definidos pelo programador, no formato *#include "caminho-absoluto-ou-relativo-a-partir-do-diretorio-de-trabalho-para-o-ficheiro.h"*) e o código em linguagem *C,* que implementa os algoritmos das funções (definição das funções).

A Biblioteca *Standard* (*C Standard Library*) da linguagem *C* ou *API* do *C*, é organizada em módulos, que integram as funções prontas a serem utilizadas nos programas do utilizador. Cada um dos módulos agrega as macros, definições de tipos e funções relacionadas para determinado fim (por conveniência de linguagem, muitas vezes poderá ser designado por biblioteca). Por exemplo são disponibilizados módulos para: manipulação de *strings*, funções matemáticas, processamento de *input/output*, gestão de memória e vários outros serviços do sistema operativo. Cada módulo da Biblioteca

Standard, possui um *header file* que define a *interface* desse módulo. A seguir apresenta-se uma tabela com os *header files* mais utilizados nos programa, bem como uma breve descrição da funcionalidade do módulo respetivo (ver Tabela 1).

Tabela 1 – Principais *header files* da Biblioteca *Standard* de *C*

Header file	**Funcionalidade**
<*errno.h*>	Teste de códigos de erros reportados pelas funções da biblioteca
<*stdlib.h*>	Conversão entre tipos numéricos, geração de números pseudoaleatórios, reserva e libertação de memória, controlo de processos
<*math.h*>	Operações matemáticas
<*stdio.h*>	*Standard input/output* fundamental
<*string.h*>	Manipulação de *strings* (sequências de caracteres)
<*time.h*>	Manipulação de datas e horas

Tal e qual como a linguagem, também a Biblioteca *Standard,* está normalizada pela *ISO* ("C Standard Library," n.d.). Naturalmente que existem várias implementações desta Biblioteca *Standard*, sendo que, no caso do ambiente de desenvolvimento *gcc*, corresponde a um projeto da *GNU*, designado por *The GNU C Library*, que fornece as bibliotecas nucleares, para o *GNU/Linux system* ("The GNU C Library," n.d.).

A organização modular de bibliotecas ou programas de média e grande dimensão, não é fácil. Assim, pela análise da Biblioteca *Standard* é possível recolher algumas orientações a respeito da estruturação do nosso *software*.

Para exemplificar todo o processo de programação modular, procederemos à conceção de uma biblioteca simples de funções matemáticas, bem como ao respetivo programa de testes. Esta designar-se-á *myMat* e será constituída por 3 funções: uma para determinar se um determinado número, positivo, é primo, outra para determinar se um número inteiro é par e a última para determinar se um número inteiro é impar. Embora sejam implementadas 3 funções, o programa de testes a apresentar posteriormente, efetuará só o teste da função de verificação se os números são primos. Naturalmente, que essas funções foram testadas, noutro programa de testes, que não apresentaremos, uma vez que a lógica é semelhante, variando apenas as funções a invocar.

Comecemos por definir a *interface* do módulo (ver **Erro! A origem da referência não foi encontrada.**). Sem entrar para já muito em pormenor, uma vez que será abordado posteriormente, a *interface* começa com algumas declarações globais, com as diretivas do pré-processador para, condicionalmente definir um símbolo (*MYMAT_H*), para testar se a o *header file* já foi incluído no programa presente (linhas 3 a 13). Caso não fosse utilizada esta prática, no momento da construção da biblioteca, resultaria naturalmente num erro de *sintaxe*, por declarações repetidas.

```
1   /* myMat.h */
2
3   #ifndef MYMAT_H
4   #define MYMAT_H
5
6   #define VERDADEIRO 1
7   #define FALSO 0
8
9   int primo( unsigned n ); // Protótipo da função primo()
10  int par( int n );        // Protótipo da função par()
11  int impar( int n );      // Protótipo da função impar()
12
13  #endif
```

Figura 7– Exemplo do *header file* de definição da *interface* do módulo *myMat*

Ainda com diretivas do pré-processador, são declaradas duas constantes simbólicas para definir o tipo booleano, que não existe na linguagem *C* (linhas 6 e 7), este será o tipo de retorno das funções. Finalmente, são definidos os protótipos das funções a implementar (linhas 9, 10 e 11). Note-se que, não era necessário colocar os identificadores para os parâmetros formais (*n*), bastando, como vimos anteriormente definir os tipos (ver Figura 5). No entanto, parece-nos uma convenção preferível, uma vez que torna a *interface* mais inteligível. Note-se ainda que não foi utilizado o modificador *extern*, uma vez que como veremos posteriormente é redundante (ver secção 5.4).

Passemos de seguida à implementação do módulo. Em termos didáticos optamos por exemplificar a implementação da função *primo()*, utilizando o algoritmo exaustivo, que corresponde à definição clássica, no âmbito da álgebra, que a seguir se específica.

> Dado um número natural *n*, verificar se existe algum número primo natural *m*, de *2* até *n-1*, que divida inteiramente *n* (o resto da divisão inteira entre *n* e *m* é *zero*). Se *n* não é divisível por qualquer *m*, este é primo. De outra forma não é primo.

Como veremos posteriormente, é um algoritmo ineficiente, mas permite, a quem está a aprender, compreende-lo melhor. A implementação do módulo é apresentada na Figura 8.

```
1   /* myMat.c */
2
3   #include "myMat.h"
4   /*
5    * primo()       : Função para determinar se um número é primo
6    * retorno       : VERDADEIRO, se o número for primo ou FALSO, se não
7    * parâmetros    : n - número inteiro positivo a testar
8    */
9   int primo( unsigned n ) {
10      unsigned m;
11
12      for( m = 2; i < n; m++ )
13          if ( !( n % m ) ) return (FALSO); // if( ( n % m ) == 0 )
14      return (VERDADEIRO);
15  }
16
17  /*
22  int par( int n ) {
23      if( !(n % 2) {
24          return (VERDADEIRO);
25      } else {
26          return (FALSO);
27      }
28  }
29
30  /*
35  int impar( int n ) {
36      return ( !par( n ) );
37  }
```

Figura 8 – Exemplo do ficheiro fonte de implementação do módulo *myMat*

Inicialmente tem que ser incluído o *header file* que define a *interface* (linha 3). Note-se ainda que é exemplificado para a implementação da função *primo()*, uma possível documentação da mesma função (linhas 4 a 8). Embora também tenha sido feito para as outras duas funções, estas foram ocultadas para que a figura não fosse tão extensa. Em relação aos algoritmos, já foi mencionado a opção pela versão exaustiva do algoritmo para implementar a função *primo()*. No entanto é de notar, que pelo facto do ciclo ser desenvolvido de *2* a *n – 1* (linha 12), o algoritmo é mais eficiente se fosse ao contrário, uma vez que os números mais pequenos, são divisores de mais números. Isto é, se o número a testar for par, o ciclo só efetua uma iteração, terminando a função imediatamente. Muitas vezes, pela nossa experiência, os estudantes propõem esta última solução, sedo apresentado este argumento. Em relação aos outros dois algoritmos, o que implementa a função *par()*, devolve, o resultado do teste da divisão do número por 2. Ao testa o resto da divisão inteira de *n* por *2* (linha 23), devolve a negação dessa expressão. Alternativamente e mais compreensível, está a expressão em comentário na mesma linha.

O algoritmo de implementação da função *impar()*, baseia-se no princípio que se um número não é par é impar.

Passemos finalmente ao programa de testes (ver Figura 9). Como foi mencionado anteriormente, só apresentamos o teste à função *primo()*. Assim é possível ainda, demonstrar que num programa não é necessário a utilização da totalidade das funções presentes numa biblioteca, tal e qual acontece com os módulos que constituem a Biblioteca *Standard* do *C*.

```
1   /* testaPrimos.c */
2
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <limits.h>
6   #include <myMat.h>
7
8   int main ( void ) {
9       unsigned num;
10
11      puts("Teste da funcao primo() que verifica se um numero positivo e' primo");
12      do {
13          printf("%s", "Introduza um numero positivo: ");
14          scanf("%u", &num);
15          if((num > 0) && (num < UINT_MAX)) {
16              printf("O numero %u %s\n", num,
17                  primo(num) ? "e' um numero primo" :
18                               "nao e' um numero primo");
19          }
20      } while( num > 0 );
21      puts("Terminou o programa");
22      return (EXIT_SUCCESS);
23  }
```

Figura 9 – Exemplo do ficheiro fonte que implementa um programa de testes

Em relação ao teste apresentado, neste ficheiro fonte, começa-se por incluir os *header files,* dos módulos das bibliotecas a utilizar (importação de *interfaces*). Para o efeito, nas linhas 3 a 5, são incluídos *header files* da Biblioteca *Standard* do *C* e na linha 6 o *header file* do módulo que desenvolvemos. A lógica subjacente à função *main()*, passa por declarar uma variável do tipo de dados nativo *unsigned* (linha 9) para introduzir um número natural (linhas 13 e 14), enquanto este for positivo (expressão definida na linha 20). O número introduzido é testado, quanto ao limite superior do *unsigned*, através da utilização da constante simbólica *UINT_MAX*, definida em *limits.h* (linha 15). Para a mensagem a apresentar ao utilizador, é utilizado o operador ternário, que consoante o resultado *booleano* (*VERDADEIRO* ou *FALSO*) da invocação da função *primo()*, apresenta a mensagem correspondente (instrução que se desdobra nas linhas 16 a 18). Como já foi mencionado, a linguagem *C*, não possui um tipo *booleano*. Assim, nas expressões lógicas (*e.g.*, envolvendo o operador ternário nas linhas 17 e 18, de teste do *if*, ou ciclos *while* ou *for*), uma avaliação para *zero* é considerada falsa e *um* ou outro valor,

verdadeira). Finalmente, na instrução de retorno da função *main()*, que embora possa não existir, do ponto de vista da programação obedecendo aos *standards*, deverá existir (linha 22), é utilizada a constante simbólica *EXIT_SUCCESS,* definida em *stdlib.h*, para comunicar ao sistema operativo o estado de finalização do processo, que neste caso é com sucesso. Na realidade o valor desta constante simbólica é *zero*, que o sistema operativo entende, como o estado de finalização normal de um programa. Para indicar um estado de finalização com erro, por exemplo, deverá ser utilizada a constante simbólica *EXIT_FAILURE.*

O processo de criação do programa executável, utilizando a ferramenta *gcc*, é esquematizado na Figura 10.

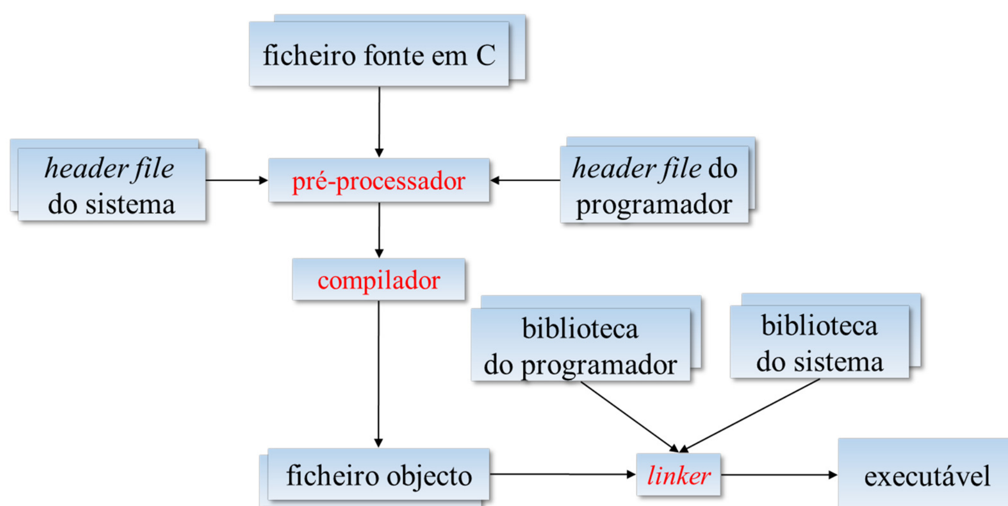

Figura 10 – Esquema de construção de um programa baseado em vários ficheiros fonte

Nas ferramentas de construção de executáveis, em geral e no *gcc*, em particular, o processo passa por três entidades distintas: pré-processador, compilador e *linker* (ou de ligação com bibliotecas compiladas). No pré-processador, são interpretadas as diretivas de pré-processador ou de compilação, que veremos posteriormente (ver secção 5.3) e que, entre outras ações, efetua a inclusão de *header files* no código fonte ou efetua as substituições da diretiva *#define*, nas ocorrências no(s) ficheiro(s) fonte em *C*. O código fonte, resultante desta entidade é submetido ao compilador para ser traduzido para código máquina, gerando, por cada, ficheiro fonte, um ficheiro objeto (*e.g.*, ficheiro com terminação *.o*, no caso do *gcc*). Atualmente, no caso do *gcc*, quando o processo de criação é completo, resultando num executável, estes ficheiros (objeto) são limpos pela ferramenta de forma automática, quando o processo é finalizado com êxito. Finalmente,

os ficheiros objeto, resultantes da compilação, são ligados com as bibliotecas do sistema e do programador, criando o ficheiro executável do programa.

Para a criação do executável ou biblioteca a partir do módulo desenvolvido, utilizar-se-á a ferramenta *gcc*, em linha de comando. No caso da criação da biblioteca estática, é utilizada a ferramenta *ar* (colocação de módulos no formato de ficheiros objeto, num único ficheiro, que constituirá a biblioteca). Para uma concentração específica nas opções destas ferramentas para os fins pretendidos, não serão utilizadas quaisquer opções de otimização ou outras disponíveis. Em relação à ferramenta *gcc*, com maior funcionalidade, estas opções poderão ser consultadas em ("Using the GNU Compiler Collection (GCC)," n.d.).

Assim, a forma mais simples de criação do executável, com base em todos os ficheiros fonte desenvolvidos é:

$ gcc testaPrimos.c myMat.c -o testaPrimos (2)

Fundamentalmente, são indicados como argumentos da ferramenta *gcc,* na linha de comando (2), os ficheiros fonte envolvidos e o nome do ficheiro executável a ser criado no seguimento da opção *-o*. Caso esta opção não seja utilizada, é sempre criado um ficheiro executável com o nome *a.out*, no caso do *gcc*.

Para executar o ficheiro, na linha de comados do *Windows*, no diretório de trabalho onde está o programa executável, basta fazer:

$ testaPrimos (3)

No caso da linha de comandos do *Linux*, é necessário indicar o caminho para a execução e caso, o programa executável, se encontre no diretório de trabalho atual, o comando é:

$ ./testaPrimos (4)

Caso seja pretendida a disponibilização da biblioteca criada, a forma mais simples é a criação de um ficheiro objeto, que sendo distribuído com o respetivo *header file* (neste caso *myMat.h*), permitirá ser integrado com outros programas, disponibilizando as funções desenvolvidas. Assim, deverá começar-se pela compilação do módulo desenvolvido (opção *-c*):

$ gcc -c myMat.c -o myMat.o (5)

Da invocação do comando (5), resulta o ficheiro *myMat.o*, no entanto os argumentos *-o MyMat.o* são redundantes, uma vez que por defeito, quando é utilizada a opção de

compilação *-c*, resultam os mesmos nomes dos ficheiros fonte, retirada a terminação *.c* e acrescentada a terminação *.o*.

Para a criação do executável é agora necessário possuir os ficheiros *testaPrimos.c*, *myMat.o* e *myMat.h*. Para facilitar o processo, estes poderão estar todos no mesmo diretório de trabalho, sendo que o comando a executar seria:

$ gcc testaPrimos.c myMat.o -o testaPrimos (6)

Caso o ficheiro objeto *myMat.o*, não estivesse no mesmo diretório, seria necessário na linha de comando, indicar o seu, caminho relativo ao diretório de trabalho ou caminho absoluto. Chama-se a atenção que, caso o *header file* não esteja no diretório de trabalho, é necessário, no ficheiro fonte constante da Figura 9 – Exemplo do ficheiro fonte que implementa um programa de testes, na linha 6, indicar o caminho relativo ao diretório de trabalho ou o caminho absoluto de *myMat.h*. Naturalmente, o comando (6) poderá ser desdobrado em compilação do ficheiro *testaPrimos.c*, nos mesmo moldes do comando (5) e ligação nos mesmos moldes do comando (6), substituindo *testaPrimos.c* por *testaPrimos.o*.

Se a opção, fosse pela distribuição como uma biblioteca estática, isto é, um conjunto de ficheiros objeto empacotados num único ficheiro, após a compilação da biblioteca com o comando (5), seria necessário a utilização do comando *ar* da seguinte forma:

$ ar rc libmyMat.a myMat.o (7)

Neste comando, o nome da biblioteca a criar (ficheiro terminado em *.a*), tem que ser iniciado com *lib*, conforme o comando (7). As opções *rc*, têm o seguinte significado: *r*, na adição à biblioteca, substitui o ficheiro objeto caso exista; *c*, cria a biblioteca caso não exista.

Para a criação do executável, mantem-se a necessidade de possuir os ficheiros *testaPrimos.c* e *myMat.h*. Parte-se do princípio que a biblioteca estática está no diretório de trabalho. Assim, para criar o executável, ligando-o à biblioteca estática o comando seria:

$ gcc -static testaPrimos.c -L. -lmyMat -o testaPrimos (8)

No comando (8), a opção *-Lcaminho* permite indicar o caminho onde reside a biblioteca, que neste caso é indicado o diretório de trabalho ou atual (.). A opção *-lBiblioteca*, é a

indicação da biblioteca, que como se pode constatar é o nome do ficheiro terminado em *.a*, criado pelo comando (7), retirando o *lib* inicial e a terminação *.a*.

Atualmente, as bibliotecas são disponibilizadas como bibliotecas partilhadas, cuja principal vantagem natural, é tornar os ficheiros executáveis mais pequenos, uma vez que os programas executáveis partilham as bibliotecas. Optamos por não colocar aqui o processo de criação de uma biblioteca partilhada, uma vez que envolve praticamente os mesmos passos da criação da biblioteca estática, utilizando a ferramenta *gcc* em vez da *ar* com outras outras opções. Consideramos que este processo, não acrescentaria muito valor à exposição, face à necessidade de texto necessário à explicação, que seria nos mesmos moldes do anterior, com outras opções.

Para exemplificar agora a possibilidade de fazer evoluir a implementação do módulo em causa, permitindo a sua substituição sem afetar as restantes partes do programa, efetuaremos uma nova implementação da função *primo()*, agora com um algoritmo mais eficiente que o anterior. O princípio, passa pela adoção da definição seguinte.

> Dado um número natural *n*, verificar se existe algum número primo natural *m*, de *2* até $\sqrt{n}$, que divida inteiramente *n* (o resto da divisão inteira entre *n* e *m* é *zero*). Se *n* não é divisível por qualquer *m*, este é primo. De outra forma não é primo.

Relativamente, ao algoritmo anterior, a alteração é no limite superior do ciclo, uma vez que só é necessário testar os números primos *m* até $\sqrt{n}$. Acima deste limite já não existem divisores de *n*.

A implementação do módulo em causa é a que se apresenta na (Figura 11).

Note-se que nesta implementação como é utilizada a função *sqrt()*, implementada no módulo de funções matemáticas da Biblioteca *Standard*, é necessário importar o respetivo *head file* (*#include <math.h>* na linha 3). Na instrução que contém a expressão do ciclo *for* (linha 13), na condição que testa a finalização do ciclo é fundamental efetuar a conversão explicita para *unsigned*, uma vez que o resultado de *sqrt()* é *double* e o *m* é *unsigned*.

Considerando a distribuição como uma biblioteca estática a sequência de comandos de compilação da biblioteca seria pela utilização em sequência dos comandos (5), (7) e (8), como as respetivas adaptações, nomeadamente:

$ gcc -c myMat2.c -o myMat.o (9)

$ ar rc libmyMat.a myMat.o (10)

$ gcc -static testaPrimos.c -L. -lmyMat -lm -o testaPrimos (11)

```
1   /* myMat2.c */
2
3   #include <math.h>
4   #include "myMat.h"
5   /*
6    * primo()       : Função para determinar se um número é primo
7    * retorno       : VERDADEIRO, se o número for primo ou FALSO, se não
8    * parâmetros    : n - número inteiro positivo a testar
9    */
10  int primo( unsigned n ) {
11      unsigned m;
12
13      for( m = 2; m <= ((unsigned) sqrt(n)); i++ )
14          if ( !( n % m ) ) return (FALSO); // if( ( n % m ) == 0 )
15      return (VERDADEIRO);
16  }
17
18  /*
23  int par( int n ) {
24      if( !(n % 2) ) {
25          return (VERDADEIRO);
26      } else {
27          return (FALSO);
28      }
29  }
30
31  /*
36  int impar( int n ) {
37      return ( !par( n ) );
38  }
```

Figura 11– Exemplo do ficheiro fonte de outra implementação do módulo *myMat*

O que efetuamos, permitiu a efetiva substituição do módulo desenvolvido por uma nova versão, a distribuir como biblioteca estática. Salienta-se, ainda, que na *GNU C Library*, normalmente há módulos da Biblioteca *Standard* que necessitam ser indicados no momento da construção da biblioteca ou programa, como é o caso do módulo de matemática, em que é necessário indicar o parâmetro *-lm*, no comando (11).

Para finalizar, este Tutorial de Programação Modular em *C*, resta-nos dizer que todo o processo de implementação da biblioteca e/ou programa, seria mais fácil, se utilizássemos um *IDE* (*Integrated Development Environment*), como por exemplo o *Eclipse* ("Eclipse C/C++ Development User Guide," n.d.). Isto porque, neste *software,* as tarefas de edição e construção, seriam mais intuitivas, através da disponibilização e seleção das opções em *menus*. Este ambiente, em concreto (aliás como a maioria de outras opções), integra um editor de código inteligente (facilita as tarefas de edição de código e obtenção de ajuda à escrita do código) e um *front end* de acesso à ferramenta *gcc.* Para além disso, também são disponibilizados *front ends* de acesso a ferramentas de gestão de projetos e *debugging*

(que não abordamos), entre outras, de forma mais facilitada, podendo ser utilizadas em linha de comando. Estamos a falar, por exemplo, das ferramentas *Make* ("GNU Make," n.d.) e *GDB* ("GDB: The GNU Project Debugger," n.d.), para a gestão de projetos e controlo de versões, bem como de *debugging*, respetivamente.

5.3 Pré-processador

O pré-processador é a entidade da ferramenta de construção de programas, que é executada antes do processo de compilação e que examina o código dos ficheiros fonte e excuta determinadas ações, com base nas instruções que lhe são dirigidas, nomeadamente:

- A inclusão de outros ficheiros fonte, no código a ser compilado;
- Definição de constantes simbólicas e macros;
- Compilação condicional do programa e código;
- Execução condicional de diretivas do pré-processador.

As instruções dirigidas ao pré-processador, chamam-se diretiva do pré-processador ou diretivas de compilação e são todas as instruções do código fonte iniciadas por #. Estas não são compiláveis.

Apesar de algumas diretivas serem desaconselhadas, para a implementação de programas, em geral e módulos em particular, a maioria dos atuais programadores profissionais de *C*, continuam a conhecer e a utilizar estas diretivas "problemáticas". É preciso salientar que o *C* é a linguagem de programação, que mantém uma grande quantidade de "*legacy code"* (P. J. Deitel & Deitel, 2016, Chapter 13). As diretivas em questão, são maioritariamente, as que definem constantes simbólicas e macros, uma vez que não existe qualquer validação de tipos, quando utilizadas para a definição de "constantes" ou "funções *inline*" (pequenas funções implementadas como expressões a substituir no código, para melhor desempenho do *software*).

Contudo as diretivas do pré-processador, no nosso caso particular, para programação modular em *C*, são fundamentais para organizar, definir e implementar as facilidades dos módulos, nos ficheiros fonte de *interfaces* e implementação. Como iremos constatar a seguir, estas permitirão a importação de *interfaces* de módulos, exportação de uma *interface* constituída por definição de tipos e funções "leves" (*e.g.,* definição de

constantes simbólicas e macros); bem como a, eventual, compilação de código e/ou execução de diretivas do pré-processador de forma condicional.

Há medida da exposição, serão dadas algumas recomendações de técnicas mais seguras, de utilização, essencialmente, da diretiva *#define*.

Relativamente à diretiva *#include*, não há mais nada a acrescentar, a não ser relembrar a sua utilidade e sintaxe. Esta permite incluir o conteúdo do ficheiro especificado a seguir à mesma, no ponto do código em que aparece. Poderá ser considerada uma diretiva de "expansão" / "condensação" de código. Tal como já foi descrito em secções anteriores, assume duas sintaxes: uma para a importação de *header files* da Biblioteca *Standard* do *C* – *#include <header-file>* (*e.g., #include <stdio.h>*) e outra para a importação de *header files* do programador (outras bibliotecas) – *#include "caminho-para-o-header-file"* (*e.g., #include "myMat.h"*).

A diretiva *#define*, é provavelmente a diretiva mais problemática, uma vez que sem a perceção do programador, poderão ser introduzidos erros lógicos, difíceis de detetar e corrigir. Isto acontece, porque a maior parte das situações que levam ao aparecimento dos erros, são para casos particulares, que normalmente não testamos. Vejamos esta diretiva em particular, chamando à atenção para pormenores que tornarão mais segura a sua utilização.

Genericamente, esta é uma diretiva de substituição de texto no código fonte no local em que aparece. Assim a sintaxe geral é:

#define IDENTIFICADOR texto-de-substituição (12)

Esta pode ser utilizada para a definição de símbolos, passíveis de serem testados posteriormente ao seu aparecimento. Convencionalmente o identificador é expresso todo em maiúsculas:

#define IDENTIFICADOR (13)

Por exemplo, *e.g., #define MYMAT_H.*

Poderá, ainda, ser utilizada para a definição de constantes simbólicas (a forma inicial de definição de constantes na linguagem *C* original, antes da inclusão clausula *const*). Também o identificador deverá ser todo expresso em maiúsculas:

#define IDENTIFICADOR texto-de-substituição (14)

Como exemplo de utilização, podemos ter: *#define VERDADEIRO 1* ou *#define PI 3.141592*. Neste âmbito é necessário ter especial atenção à sintaxe, uma vez, não existe qualquer verificação da substituição e, por exemplo, se por qualquer motivo, definirmos a seguinte constante simbólica "*#define PI = 3.141592;*" as substituições serão de *PI* por *" = 3.141592;"*. Dependendo do caso, poderá resultar num erro de sintaxe (situação ideal) ou numa situação complicada, de alterar a avaliação de uma determinada expressão de forma subtil, levando ao aparecimento de um erro de lógica, muitas vezes difícil de detetar. Pela razão exposta, para além da não verificação de tipos, que pode levar ao aparecimento de outros erros de lógica subtis (eventualmente derivados de conversões implícitas), recomenda-se a utilização da clausula *const* a preceder a definição da variável (*e.g., const double PI = 3.141592;*), quando a intensão for definir uma constante.

Por fim esta diretiva poderá ser utilizada para definir macros com parâmetros, para definir uma funcionalidade. A razão da existência desta facilidade, prende-se com a possibilidade de implementar funcionalidade "leve", de forma mais eficiente, que as tradicionais funções. Naturalmente, deverá ser utilizada esta funcionalidade baseada, eventualmente, numa só expressão. A sintaxe geral de uma macro é:

#define macro(PARAMETROS) expressão-com-os-PARAMETROS (15)

Por exemplo, considere o seguinte exemplo:

#define max(A, B) (((A) > (B)) ? (A) : (B))

Destacámo-lo, para chamar à atenção para a importância do envolvimento dos *PARAMETROS* na expressão, com parêntesis, de forma a não serem possíveis alterações, subtis, na avaliação da expressão, que poderiam introduzir erros de lógica. Se não vejamos um exemplo de um erro lógico.

Considere a seguinte macro:

#define ABS(X) ((X < 0) ? (-X): (X))

Esta situação poderá ser perigosa, dependendo do parâmetro *X*.

Se a macro for invocada como *ABS(2-3)*, o resultado da expressão *(2-3 < 0) ?*, após a substituição e antes da avaliação, resultaria em *-2-3,* que, algebricamente, seria avaliado em *-5* e não em *1*, que seria o resultado pretendido.

A forma de conseguir corrigir esta alteração subtil, seria utilizando parêntesis na *expressão-com-os-PARAMETROS* de (15), a envolver cada *PARAMETRO:*

#define ABS(X) ((X < 0) ? (- (X)) : (X))

Neste caso, utilizando o mesmo caso, teríamos *(2-3 < 0) ?,* que resultaria na substituição *(-(2-3))*, que após a avaliação resultaria em *1*, que é o resultado pretendido.

Por fim, poderá ser notado pelo leitor a eventual inconsistência na convenção seguida nos dois exemplos apresentados. No entanto esta foi propositada, uma vez que, inicialmente, as macros eram confundíveis com as funções (identificador todo em minúsculas) para os programadores utilizador destas e neste momento, é aconselhado, seguir a convenção geral da diretiva *#define,* para as definições de símbolos e constantes simbólicas, permitindo distinguir, imediatamente, pela leitura do código do programador utilizador, que se trata de uma macro ou de uma função, que mantém a convenção inicial.

Se pretendermos indefinir uma determinada definição já efetuada, deverá ser utilizada a diretiva:

#udefine IDENTIFICADOR (16)

Uma vez que esta secção não pretende ser um manual de referência de diretivas de compilação, a não ser as que normalmente são utilizadas no âmbito da programação modular, as restantes diretivas, poderão ser referenciadas, quando apropriado, especificando, concretamente, as que são mais utilizadas neste âmbito.

A compilação condicional, permite controlar a execução de diretivas do pré-processador e da compilação condicional de código. As diretivas em causa são do tipo *#if … #endif.* Estas aproximam-se e seguem regras semelhantes à instrução *if()*, tendo sintaxes e utilidades específicas. Destas, as que têm normalmente aplicabilidade no âmbito da programação modular, como já foi descrito, têm haver com a importação condicional de *header files* (ver Figura 7– Exemplo do *header file* de definição da *interface* do módulo *myMat*, bem como a explicação a respeito na secção 5.2) para prevenir a múltipla importação dos *header files* no mesmo programa e os respetivos erros na ligação do programa/biblioteca.

5.4 Regras de âmbito e classes de armazenamento

Antes de iniciar esta abordagem, convém definir os seguintes termos. Falamos em declaração, quando escrevemos uma expressão que indica a forma de um identificador para poder ser acedido (*e.g., declaração de um protótipo de função*). Falamos em definição, quando escrevemos uma expressão ou conjunto de instruções para definir um

identificador (*e.g.,* definição de uma função ou variável). Falamos em instrução, para designar operações de um algoritmo na linguagem *C* (*e.g., i+=5;*).

Regras de âmbito (*scope*), de um identificador, é a porção do programa, no qual o identificador é conhecido e pode ser acedido pelo seu nome. O âmbito de um identificador, pode ser: módulo, função ou bloco. Os módulos, normalmente, coincidem com os ficheiros fonte que os implementam, sendo que o âmbito dos identificadores declarados ou definidos num módulo, correspondem aos identificadores globais, declarados fora de qualquer função. Os identificadores declarados ou definidos no âmbito de uma função, são locais a essa função e não podem ser acedidos fora dessa função. Os identificadores declarados ou definidos no âmbito de um bloco, são os que são declarados dentro de um bloco delimitado por *{* e *}*, não podendo ser acedidos fora desse bloco.

As funções são globais por natureza, não sendo possível declarar ou definir funções dentro de funções. Naturalmente, não é possível existirem instruções fora de uma função. Por seu turno, os blocos podem ser definidos dentro de blocos, mas só dentro de funções. Apesar de ser possível definir blocos no âmbito das instruções condicionais e de repetição (*e.g.,if, switch, do…while, while* e *for*), não é possível declarar ou definir identificadores dentro destes blocos. Não é possível definir o mesmo identificador mais do que uma vez no mesmo âmbito. Isto é, não é possível definir o mesmo identificador no mesmo módulo, função ou bloco, conforme o âmbito. O mesmo identificador pode ser definido simultaneamente, no módulo, função dentro do módulo e/ou bloco dentro da função (são considerados âmbitos diferentes). Neste caso o identificador declarado no âmbito menor, oculta o identificador declarado no âmbito maior (*e.g.,* se for declarada uma variável chamada *nome*, num módulo e numa função, dentro deste módulo; na função o acesso é no âmbito da função, não sendo possível, na função aceder ao conteúdo da variável no âmbito do módulo).

Naturalmente que as regras de âmbito descritas, permitem cumprir os princípios e práticas de ocultação de informação e separação de contextos, na organização modular de um programa em *C*.

Relacionados com o âmbito dos identificadores, existem várias classes de armazenamento. Estas últimas definem a visibilidade e tempo de vida de identificadores e/ou espaço de armazenamento associados, conforme os casos (variáveis ou funções) especificados a seguir.

As classes de armazenamento, num programa em *C*, são definidas, através dos seguintes modificadores, que normalmente precede a declaração do identificador a que se aplica: *auto*, *register, static* e *extern*.

A classe de armazenamento *auto*, é a classe de armazenamento por defeito para a definição de todas as variáveis locais. Estas variáveis são criadas e mantidas enquanto o âmbito estiver ativo. Isto é, são definidas à entrada do âmbito e destruídas à saída desse âmbito. É preciso ter em atenção que na definição, não é feita a inicialização do espaço de armazenamento, pelo que se aceder à variável em leitura, antes de lhe atribuir um valor, o valor obtido é lixo.

A título de exemplo, considere a Figura 12.

```
1  {
2      int mount;
3      auto int month;
4  }
```

Figura 12 – Exemplo de utilização da classe de armazenamento *auto*

Neste caso, no âmbito do bloco definido, ambas as variáveis têm a classe de armazenamento *auto*. Na declaração da linha 2 esta classe está implícita, porque é a classe de armazenamento por defeito no âmbito de uma função, único sítio em que pode ser definido um bloco. Na declaração da linha 3, com a explicitação redundante do modificador.

A classe de armazenamento *register* é utilizada para definir variáveis locais das funções, que devem ser guardadas e mantidas em registo do processador. O tipo destas variáveis, devem ocupar no máximo o tamanho em *bits* do registo do processador. Normalmente devem ser do tipo *int*.

Devem ser usadas só para variáveis que requerem acesso mais rápido, como por exemplo, contadores. Este modificador não obriga à manutenção da variável no registo, tratando-se, só, de uma indicação ao compilador para efetuar uma otimização do código traduzido, nesse sentido.

A título de exemplo, considere a Figura 13.

```
1  {
2      register int count;
3  }
```

Figura 13 – Exemplo de utilização da classe de armazenamento *register*.

A classe de armazenamento *static,* pode ser utilizado na definição de variáveis ou funções. No caso de utilização na definição de uma função, torna-a acessível exclusivamente, no ficheiro, em que foi definida. É uma solução para a implementação de funções utilitárias, no âmbito de um módulo que não pretende disponibilizar essas funções aos programadores utilizadores do módulo. A título de exemplo ver Figura 14.

```
1   /* fonte1.c */
2   static void fun1(void)
3   {
4     puts("fun1 chamado");
5   }
6
7   /* fonte2.c  */
8   int main(void)
9   {
10    fun1();
11    return 0;
12  }
13
14  gcc fonte2.c fonte1.c
```

Figura 14 – Exemplo de utilização do modificador *static,* na definição de uma função

Neste exemplo, a criação do programa, com o comando da linha 14, gera o seguinte erro de sintaxe: *"undefined reference to 'fun1'"*.

Ao utilizar o modificador *static,* na declaração de uma função local, instrui o compilador, para manter uma variável local durante o tempo de vida do programa. Se esta for inicializada no momento da declaração, só é inicializada a primeira vez que a função executa (ver a Figura 15). Aqui, a primeira vez que a função é invocada, a variável *conta* é inicializada com o valor *5*, sendo incrementada para *6*. Nas restantes invocações, *conta* na entrada da função, tem o valor da saída da invocação anterior, sendo incrementada antes da função terminar. Isto é, na segunda invocação, na entrada da função *conta* tem o valor *6* e na saída, o valor *7*, na terceira invocação, na entrada da função, *conta* tem o valor *7* e na saída, o valor *8*, etc.

```
1  void func( void ) {
2      static int conta = 5;
3
4      conta++;
5      return;
6
7  }
```

Figura 15 – Exemplo de utilização da classe *static* no âmbito de uma função.

Este modificador pode ser aplicado a variáveis globais, sendo que o efeito é idêntico à aplicação na definição de funções. Isto é, torna o acesso restrito ao ficheiro (módulo) em que é declarada a variável.

A classe de armazenamento *extern* é utilizada para declarar o acesso a uma variável global ou função, que está definida noutro âmbito. No caso de ser uma variável global, permite o acesso ao conteúdo dessa variável. Na declaração da variável global com o modificador *extern*, não pode haver inicialização na própria declaração.

Esta classe de armazenamento é normalmente utilizada, quando existem dois ou mais ficheiros que partilham as mesmas variáveis globais e/ou funções (ver Figura 16).

```
1   /* main.c */
2   #include <stdio.h>
3
4   int count ;
5   extern void write_extern();
6
7   int main(void) {
8       count = 5;
9       write_extern();
10  }
11
12  /* support.c */
13  #include <stdio.h>
14
15  extern int count;
16
17  void write_extern(void) {
18      printf("count is %d\n", count);
19  }
```

Figura 16 – Exemplo de utilização da classe de armazenamento *extern*.

Neste caso, no ficheiro *main.c* é declarada uma função que será definida noutro ficheiro (*suporte.c*), sendo possível na função *main()* invocar essa função sem conhecer a sua implementação. No ficheiro *suporte.c* é declarada uma variável global com o modificador *extern*. Esta variável é definida em *main.c* como global. A função *write_extern()*, definida em *support.c*, consegue aceder ao valor de *count* que contém o valor atribuído na função *main()*. Desta forma é possível partilhar o conteúdo de variáveis globais entre módulos.

Para compreender as regras de âmbito conjugadas com as classes de armazenamento mais comuns, apresentamos e comentamos o exemplo constante da Figura 17.

Na linha 5, é definida e inicializada a variável global *x* com o valor *1*. Na linha 7 é impresso o conteúdo da variável global *x*, ou seja *1*, sendo apresentada a seguinte mensagem:

local x in outer scope of main is 1

```
1  #include <stdio.h>
2  void a( void ); // functions prototypes
3  void b( void );
4  void c( void );
5  int x = 1;          /* global variable */
6  int main(void) {
7      printf("local x in outer scope of main is %d\n", x );
8      int x = 5;      // local variable
9      {               /* start new scope */
10         int x = 7;
11         printf("local x in inner scope of main is %d\n", x );
12     }               /* end new scope  */
13     printf("local x in outer scope of main is %d\n", x );
14     a(); b(); c();
15     a(); b(); c();
16     printf("local x in main is %d\n", x );
17     return 0;
18 }
19 void a( void ) {
20     int x = 25;  /* initialized each time a is called */
21     printf( "\nlocal x in a is %d after entering a\n", x++ );
22 }
23 void b( void ) {
24     static int x = 50; // static initialization only first time b is called
25     printf("\nlocal static x is %d on entering b\n", x++ );
26 }
27 void c( void ) {
28     printf("\nglobal x is %d on entering c\n", x ); x *= 10;
29     printf( "global x is %d on exiting c\n", x );
30 }
```

Figura 17 – Exemplo de aplicação das regras de âmbito e classes de armazenamento.

Na linha 8, é definida e inicializada a variável local *x*, com o valor *5*. Neste caso, estamos na presença de um novo âmbito – função *main()*, ocorrendo uma ocultação da variável global *x*, por ter o mesmo nome. É definido um novo âmbito (bloco) entre as linhas 9 e 12. Neste, na linha 10 é definida e inicializada a variável local *x,* com o valor *7*, que oculta a variável local à função *main()*. Na linha 11, é impresso o conteúdo da variável local ao bloco *x*, ou seja *7*, sendo apresentada a seguinte mensagem:

local x in inner scope of main is 7

Na linha 13, é impresso o conteúdo da variável *x* local à função *main()*, ou seja *5*, sendo apresentada a seguinte mensagem:

local x in outer scope of main is 5

Na linha 14, são invocadas as funções *a()*, *b()* e *c()*. A função *a(),* que está definida nas linhas 19 a 22, define e inicializa uma variável local *x*, que também oculta o conteúdo da variável global *x.* A classe de armazenamento é *auto*, pelo que cada vez que a função executa, inicializa a variável com o valor *25.* Assim, esta função imprime o conteúdo desta variável, ou seja *25*, incrementando uma unidade o conteúdo da variável, no âmbito da função, que não se reflete fora deste âmbito. Ao finalizar a função a variável é destruída, mas ainda no âmbito da função é apresentada a seguinte mensagem:

local x in a is 25 after entering a

A função *b(),* que está definida nas linhas 23 a 26, definida e inicializada uma variável local *x* com a classe de armazenamento *static*, que também oculta o conteúdo da variável global *x.* Assim a variável só é inicializada na primeira execução da função, concretamente como valor *50.* A função imprime o conteúdo desta variável, ou seja *50*, incrementando uma unidade o conteúdo da variável, no âmbito da função. Ao finalizar a função o conteúdo da variável é mantido, sendo, no âmbito da função apresentada a seguinte mensagem:

local static x is 50 on entering b

A função *c()*, que está definida nas linhas 27 a 30, imprime o conteúdo da variável global *x*, concretamente *1.* A função incrementa o conteúdo desta variável, pelo produto por 10, passando a conter o valor *10.* Antes de terminar, volta a imprimir o valor da variável global, que atualmente é de *10*, sendo apresentadas as seguintes mensagens:

global x is 1 on entering c

global x is 10 on exiting c

Na linha 15, são invocadas novamente as funções *a()*, *b()* e *c()*, em sequência, apresentando as seguintes mensagens:

local x in a is 25 after entering a

local static x is 51 on entering b

global x is 10 on entering c

global x is 100 on exiting c

Finalmente, na linha 16 é impresso o conteúdo da variável local à função *main()*, cujo valor continua a ser *5*, sendo apresentado a seguinte mensagem:

local x in main is 5

6 ESTRATÉGIAS PEDAGÓGICAS

Neste capítulo serão definidas as estratégias pedagógicas a adotar, no âmbito da lição.

Como é compreensível, devido à extensão dos conteúdos teórico-práticos desenvolvidos, tendo a lição a apresentar, uma duração de 60 minutos, seria uma tarefa complicada, se não impossível, a apresentação da totalidade da matéria, com a profundidade necessária. Mais, foi definido, nos objetivos (Capítulo 2), que como resultado final e específico da lição em causa, se pretende que os estudantes, sejam capazes de aplicar os princípios, recursos, e boas práticas de programação modular no âmbito da linguagem de programação *C*. Assim, todas as estratégias aqui definidas se aplicam ao cumprimento desse objetivo em particular.

Concretamente, estamos a falar no processo de ensino/aprendizagem da matéria constante do Capítulo 5, desta publicação.

As estratégias pedagógicas irão ser subdivididas em três secções. Primeiro, começaremos por definir as metodologias de ensino/aprendizagem, em concordância com as metodologias, da própria unidade curricular, especificadas para a lição em questão. Depois definiremos, os recursos didáticos a disponibilizar aos estudantes. E por fim, definiremos, o regime de avaliação, concreto, para esta matéria.

6.1 Metodologias de Ensino/Aprendizagem

No âmbito da unidade curricular, as metodologias de ensino/aprendizagem definidas foram a utilização dos métodos expositivo, demonstrativo e experimental.

Assim, para cada lição de cariz teórico-prática, é pressuposto o desenvolvimento dos conteúdos, suscitando a participação ativa, reflexiva e critica, consolidada com debates e exercícios de aplicação.

Assim, serão expostos os conteúdos programáticos com exemplos elucidativos, com questões, orientadoras da exposição e do processo de ensino/aprendizagem, lançadas durante a mesma. Muitas destas questões prender-se-ão, com a proposta de solução aos enunciados dos exemplos e antes da apresentação das soluções.

Dado que a duração da lição é de 60 minutos, o método experimental, é sugerido. Nomeadamente, dar-se indicação para que os estudantes comecem pela implementação

dos exemplos apresentados, permitido a verificação dos resultados salientados na exposição. Também será dada indicações, para a resolução de exercícios de aplicação de consolidação, disponibilizados pelo docente.

Será incentivado o recurso à *WWW*, para auxiliar a resolução dos exercícios, disponibilizando-se o docente, para a realização de tutorias individuais. Assim, será possível a deteção e suprimento das necessidades individuais, centrando todo o processo no estudante.

6.2 Recursos didáticos

Para a exposição dos conteúdos será utilizado um computador com o ambiente de desenvolvimento para a linguagem de programação *C,* uma aplicação de apresentações, um vídeo projetor e *slides* de suporte à exposição a efetuar.

Para os estudantes será disponibilizado um documento coerente, constituído por uma versão mais reduzida do Capítulo 3 e pelos Capítulos 4 e 5, bem como do Capítulo 7, desta publicação, para servir de material de estudo.

Cada estudante deverá ter um computador, com o ambiente de desenvolvimento para a linguagem *C* para as realizações práticas.

Disponibilização de uma ficha de exercícios de resolução obrigatória, criada a partir dos exercícios constantes do capítulo 7 ou outros entretanto elaborados.

6.3 Avaliação

O estudante, deverá ser capaz de demonstrar que é capaz de aplicar os princípios e boas práticas no âmbito da programação modular de sistemas computacionais, em geral e na linguagem de programação *C*, em particular.

Assim, durante a exposição da lição as intervenções dos estudantes são consideradas. Será observado e acompanhados a resolução dos exercícios, no âmbito das tutorias individuais.

Será efetuada uma avaliação formativa da ficha de exercícios, com relato do desempenho alcançado.

7 EXERCÍCIOS DE CONSOLIDAÇÃO DE CONHECIMENTOS

- No âmbito dos princípios e práticas fundamentais de desenvolvimento de *software*, relacione abstração, modularização, ocultação de informação e independência de contextos.

- No âmbito da decomposição modular, apresente as vantagens da seguinte meta: deverá ser possível modificar a implementação de um módulo sem ter conhecimento da implementação de outros módulos e sem afetar o comportamento de outros módulos.

- Um programa na Linguagem de programação *C* (escolha pelo menos uma opção):
  - ☐ é constituído por várias funções *main()*, desde que cada uma esteja no seu ficheiro fonte.
  - ☐ dos ficheiros fonte a implementação (código executável) é colocada nos ficheiros com extensão **.h* (*header files*).
  - ☐ pode possuir várias funções definidas pelo programador.
  - ☐ é constituído por funções, eventualmente separado por vários ficheiros fonte.
  - ☐ pode conter a instrução *#include "my_funcs.h"* num ficheiro do tipo **.h*.

- A função cujo protótipo é *int dobro(int x)*; devolve o dobro de um inteiro. Escreva a instrução que que permita utilizar esta função, para atribuir á variável *oDobro* o dobro do valor guardado em *num.*

- Defina uma função que recebe como parâmetro o raio de uma esfera e calcula o seu volume ($v = 4/3.PI.R^3$).

- Elabore um programa que leia dois números reais. O programa deverá escrever no ecrã qual o maior dos valores. Deverá ser definida uma função *double maior(double, double)* que devolve o maior entre os dois números passados como parâmetros.

- Escreva o código necessário para que a função *int maximum(int x, int y, int z)* retorne o máximo entre 3 números.

- Considere o programa seguinte e identifique o output do mesmo:

```
include <stdio.h>
void func();
int i = 10;
int main(void) {
    int i = 20;
    func(); printf("i= %d ", i);
    if(i >= 20) { i = 30; func(); printf("i= %d ", i); }
}
void func() { printf("i = %d ", i); }
```

  - ☐ *i= 10 i= 20 i=10 i= 30*
  - ☐ *i= 20 i=10 i=10 i= 30*
  - ☐ *i= 10 i=10 i=10 i=10*
  - ☐ *i= 20 i= 20 i= 30 i= 30*
  - ☐ nenhuma das outras opções.

- Crie uma biblioteca que implemente a seguinte funcionalidade:

  Macro *MAIOR(A, B)* que devolve o maior entre os dois números *A* e *B;*

  Função *double arredonda(double val, int dec)*, para arredondar um número para um determinado número de casas decimais;

  Obs.:

  A função *floor* pode ser usada para arredondar um número em vírgula flutuante para um certo número de casas decimais.
  Por exemplo, a instrução: *y = floor(x * 100 + 0.5) / 100*; arredonda o número *x* para duas casas decimais.

  Crie um programa para testar as facilidades solicitadas.

- Elabore um programa para apresentar os *n* primeiros números primos, sendo que *n* deverá ser positivo e introduzido pelo utilizador. Deverá ser pesquisada e implementada uma solução eficiente.

- Elabore um programa para determinar o fatorial de ordem *n*, onde *n* é introduzido pelo utilizador. Deverá ser definida a função *int fact(int)* para calcular o fatorial de um valor inteiro passado como parâmetro. Por exemplo: $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$.

- Elabore um programa que determine o máximo divisor comum, dados dois números inteiros positivos. Deverá ser desenvolvida uma função *int mdc(int, int)* que implemente o algoritmo de Euclides.

  Exemplo do cálculo *mdc(24, 15) = 3*:

Dividendo	24	15	9	6
Divisor	15	9	6	3
Resto	9	6	3	0

mdc

8 CONCLUSÃO

A presente publicação pretende ser um documento de suporte à lição a apresentar, como requisito parcial, no âmbito da prestação de provas públicas de avaliação de competência pedagógica e técnico-científica.

Foi definido pelo docente, como tema da lição "Aspetos de Programação Modular em *C*", na área ou áreas disciplinares em que desempenha funções, nomeadamente de Engenharia Informática, em geral e Ciências Informáticas, em particular.

Como é percetível é um desafio, no âmbito duma primeira unidade curricular de programação, desenvolver competências de aplicação dos princípios e prática de desenvolvimento de *software*, em geral e da programação modular, em particular.

Este desafio, ainda é maior, numa linguagem de programação versátil e poderosa, como é a linguagem *C*. Esta, muitas vezes é a base das linguagens de programação de microcontroladores e *embeded systems*. Também é a linguagem de eleição para a programação de sistemas de tempo-real e sistemas operativos, nomeadamente os de base *UNIX/LINUX.* Esta linguagem é adotada em muitas instituições para lecionar fundamentos de programação ou introdução à programação, uma vez que para além de ser concisa, é a base de muitas linguagens de programação muito utilizadas atualmente, como é o caso, do *Java, C++* ou *C#.*

Assim, foi traçado como meta fundamental, que os estudantes sejam capazes de aplicar os princípios e boas práticas no âmbito da programação modular de sistemas computacionais, em geral e na linguagem de programação *C*, em particular.

Para o efeito, foram enquadrados e abordados os princípios e práticas fundamentais de desenvolvimento de *software*, incluindo as metas de decomposição modular e os recursos disponibilizados nos ambiente de desenvolvimento, independentemente do paradigma de programação envolvido.

Foi abordada a organização modular de um programa em C. Para isso, começou-se por definir as vantagens, recursos em geral para suportar a decomposição modular e a problemática da mesma, no caso da linguagem C. Para começar a contornar os problemas identificados, foram apresentados e demonstrados os princípios e práticas de estruturação

em funções na linguagem C e a seguir a criação de módulos, baseados no princípio da separação da *interface* e implementação em C.

Em termos técnico-científicos, complementamos a matéria abordada, com a programação modular em C e a aplicação dos princípios e práticas abordados anteriormente. Aqui, foi caracterizado o ambiente de desenvolvimento utilizado. Seguidamente, foi desenvolvido um tutorial de programação modular, com o desenvolvimento de exemplos completos de módulos reutilizáveis e com possibilidade de evolução. Por fim, foram abordados os recursos específicos, que permitem o desenvolvimento de programas modulares, nomeadamente as diretivas de pré-processador e as regras de âmbito e classes de armazenamento. Fundamentalmente, foi possível mostrar que é possível, com os recursos disponibilizados pela linguagem C e o seu ambiente de desenvolvimento, aplicar os princípios subjacentes à programação modular, isto é, abstração, modularização, ocultação de informação e separação de contextos. Com esta prática é possível desenvolver *software* complexo modular, robusto, reutilizável e que favorece a manutenção, utilizando a linguagem *C*.

Como foi referido o conteúdo técnico-científico explorado nesta publicação é compreensivo, sendo complicado o seu ensino/aprendizagem numa lição de 60 minutos. Assim, em termos pedagógicos, foi selecionado o último objetivo concreto definido, sendo assumido que os restantes, já se encontram adquiridos antes da lição a desenvolver.

Assim a lição incidirá, naturalmente, no Capítulo 5, deste documento, tendo as estratégias pedagógicas sido definidas, tendo em atenção os pressupostos definidos no parágrafo anterior. Nestas foram definidas as metodologias de ensino/aprendizagem, os recursos didáticos e o processo de avaliação.

Foram ainda elaborados exercícios de aplicação, para a consolidação dos conhecimentos e aptidões adquiridos.

É uma convicção profunda do autor, poder demonstrar que possui a competência pedagógica e técnico-científica necessária, para continuar a desempenhar funções como professor adjunto.

Futuramente, é intensão do autor, o desenvolvimento de objetos de *eLearning* que permitam disponibilizar, de forma interativa, na plataforma *moodle*, os conteúdos técnico-científicos das unidades curriculares que leciona, com exercícios de consolidação de conhecimentos.

ANEXO

INSTITUTO POLITÉCNICO DE VISEU

ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO DE LAMEGO

CURSO

CÓDIGO DO CURSO – (9122) – Regime Diurno

Área CNAEF – 523 – Eletrónica e Automação

PROGRAMA

UNIDADE CURRICULAR **CÓD. (IPV.ESTGL.9122.301105)**

FUNDAMENTOS DE PROGRAMAÇÃO

DEPARTAMENTO	INFORMÁTICA, COMUNICAÇÕES E CIÊNCIAS FUNDAMENTAIS
ÁREA CIENTÍFICA	INFORMÁTICA

ANO LETIVO		
1º	2º	3º
☒	☐	☐

SEMESTRE	
1º	2º
☒	☐

Horas Totais	162	Horas de contacto	60	Semanas Letivas	15	Horas Letivas Semanais	4
Carga horária (Teórica)		Carga horária (Teórico-Prática)	30	Carga horária (Prática)	30	ECTS	6

Categoria	Semestral	Regime de frequência	Presença Obrigatória

Docente Responsável	Carlos Jorge Almeida Costa	Cód. (028)

Aprovado em Conselho Técnico Científico a _____/_____/_____

Válido a partir do Ano Letivo de _____/_____/_____

ESTGL – Avenida Visconde Guedes Teixeira, 5100-070 Lamego - Telf. 254615477 - Fax. 254613029 – mail: estgl@estgl.ipv.pt

Introdução
O perfil profissional de um Engenheiro Informático e Telecomunicações, compreende as atividades inerentes ao desenvolvimento de *software*. De entre estas, a implementação de programas permite a materialização de soluções informáticas. Para o desenvolvimento de profissionais tecnicamente competentes, é essencial uma formação consistente em fundamentos de programação. Assim, o ensino/aprendizagem de fundamentos de algoritmia e respetiva implementação em linguagens de alto nível, em geral e na linguagem C, em particular, constituem fatores decisivos no desenvolvimento destes profissionais. Esta, constitui a primeira unidade curricular no âmbito da área técnico-científica de programação de sistemas computacionais.

Pré-requisitos
Nível B1 de Leitura (Compreender) de Língua inglesa, segundo o Quadro Europeu Comum de Referência (CECR).

Objetivos de aprendizagem (conhecimentos, aptidões e competências a promover pela UC):
Estudo dos conhecimentos teórico-práticos e desenvolvimento de aptidões e atitudes que envolvem a programação de sistemas computacionais, nomeadamente, em termos de fundamentos de algoritmia, programação e da linguagem de programação C. Concretamente: • Compreensão das técnicas fundamentais de interpretação de problemas a serem resolvidos por meios computacionais; • Estudo dos conceitos, técnicas e práticas no âmbito do desenvolvimento de algoritmos fundamentais e de média complexidade; • Estudo dos conceitos, técnicas e práticas no âmbito da implementação de programas na linguagem de programação *C*.

Competências a desenvolver pelos estudantes:
• Interpretar e descrever problemas de forma a poderem ser resolvidos por meios computacionais; • Identificar, adaptar e/ou desenvolver algoritmos para a resolução de problemas e propor soluções constituídas por algoritmos fundamentais e de média complexidade; • Conhecer e aplicar os princípios da programação imperativa e modular, bem como os diferentes recursos da linguagem de programação C; • Desenvolver programas de média complexidade na linguagem de programação *C*; • Analisar, adaptar e, eventualmente otimizar soluções implementadas na linguagem de programação *C*.

Conteúdos Programáticos:

1. **Conceitos iniciais e algoritmia fundamental:** conceitos fundamentais; interpretação e resolução de problemas; fundamentos de algoritmia.
2. **Ambientes de Desenvolvimento:** caracterização, instalação, configuração e exploração.
3. **Introdução tutorial à programação:** primeiros passos; tipos fundamentais; variáveis, constantes e expressões; input/output fundamental; principais instruções condicionais e de repetição; *arrays*; funções; sequência de caracteres.
4. **Tipos, operadores e expressões:** tipos de dados; fundamentos de expressões e avaliação; operadores aritméticos, relacionais, lógicos e outros; expressões relacionais e lógicas; conversão entre tipos.
5. **Fluxo de controlo de programas:** instruções condicionais (*if, if...else e switch*); instruções de repetição (*while, do...while e for*); quebras de instruções e ciclos (*break, continue*).
6. **Funções e estruturação de programas:** fundamentos; passagem de parâmetros; regras de âmbito e classes de armazenamento; passagem de *arrays* para funções; pré-processador; estruturação de programas em módulos.
7. **Tipos de dados compostos/derivados:** fundamentos de estruturas de dados; definição e utilização de estruturas de dados (*structs*); estruturas e *typedef*; *union*; tipos de dados enumerados.
8. **Apontadores/*Pointers***: apontadores e endereços; apontadores e argumentos de funções; apontadores e *arrays*; aritmética de apontadores; *arrays* de apontadores; apontadores para apontadores; estruturas e apontadores.
9. ***Standard Input/Output:*** entradas/saídas por defeito; entradas/saídas formatadas; fundamentos de manipulação de ficheiros.

Demonstração da coerência dos conteúdos programáticos com os objetivos de aprendizagem da unidade curricular:

Os conteúdos são organizados de forma a permitir, paulatinamente, a aquisição dos conhecimentos e o desenvolvimento das aptidões e competências da unidade curricular.

Começa-se por explorar os aspetos preparatórios para o desenvolvimento da unidade curricular, com o enquadramento do âmbito desta, no ciclo de estudos e no perfil profissional, com conceitos fundamentais, nomeadamente: interpretação, resolução de problemas e fundamentos de algoritmia. Antes de iniciar o processo de aquisição de conhecimentos e o desenvolvimento de aptidões e competências de programação, é introduzido o ambiente de desenvolvimento de aplicações para a linguagem de programação a adotar.

A partir daqui, tem lugar a exploração, com monitorização conteúdos vs. objetivos, dos aspetos teórico-práticos, bem como a prática e teste na linguagem de programação adotada. Todos os tópicos que constituem o programa são ilustrados com vários exemplos elucidativos e são também disponibilizados exercícios de aplicação com complexidade crescente.

ESTGL – Avenida Visconde Guedes Teixeira, 5100-070 Lamego - Telf. 254615477 - Fax. 254613029 – mail: estgl@estgl.ipv.pt

Metodologias de ensino:

É aplicado o método expositivo, demonstrativo e experimental. Assim, inicialmente é efetuado um diagnóstico de cada estudante. Para suscitar a motivação é feita a sensibilização para a importância da unidade curricular no domínio profissional. São apresentados os conteúdos, definidos objetivos, competências a adquirir, integração no currículo e interligação com o perfil profissional.

Durante o semestre são desenvolvidos os conteúdos, suscitando-se a participação ativa, reflexiva e crítica, consolidados com exercícios de aplicação, debates, trabalhos individuais e de grupo. Como forma de monitorização, as atividades são alvo de avaliação formativa e sumativa, permitindo o acompanhamento e o direcionamento para apoio.

A avaliação é individual e sumativa, permitindo a aferição do nível de conhecimentos pela demonstração aplicada das aptidões e competências desenvolvidas. Ao longo do percurso é possível detetar necessidades de apoio e aplicar medidas adequadas de promoção do sucesso.

Metodologias de avaliação (contínua, periódica e final; recurso e melhoria):

Avaliação teórico-prática individual, constituída por testes, exercícios/práticas laboratoriais em contexto de aula e/ou submetidas via plataforma de e-learning. Avaliação de trabalhos desenvolvidos em grupo, com acompanhamento obrigatório.

Esta UC pressupõe as seguintes modalidades de avaliação: periódica; final; recurso e melhoria.

Na modalidade de avaliação periódica os estudantes são avaliados com as componentes teórico-prática individual e de trabalhos desenvolvidos em grupo. Na modalidade de avaliação final poderá existir a avaliação de trabalhos desenvolvidos em grupo, caso seja possível o acompanhamento dos mesmos. Alternativamente, os estudantes poderão desenvolver trabalhos individuais. Na modalidade de avaliação por recurso e melhoria, a avaliação teórico-prática individual é realizada por exame e/ou trabalhos individuais. Poderá ser considerada a avaliação de trabalhos desenvolvidos em grupo, caso tenha tido lugar no decorrer do semestre.
A classificação final é obtida pela média ponderada dos vários elementos de avaliação. Para aprovação o aluno tem que obter 9,5 valores nas avaliações teórico-prática individual e de trabalhos desenvolvidos em grupo.

Demonstração da coerência das metodologias de ensino com os objetivos de aprendizagem da unidade curricular:

As metodologias de ensino são orientadas de forma a: centrar o ensino no estudante; dar importância ao processo de aprendizagem; facilitar o acesso aos meios e recursos de aprendizagem; orientar a aprendizagem; diversificar os métodos e os contextos de aprendizagem; privilegiar a aplicação e integração dos saberes; validar as competências adquiridas e demonstradas e premiar o mérito.

ESTGL – Avenida Visconde Guedes Teixeira, 5100-070 Lamego - Telf. 254615477 - Fax. 254613029 – mail: estgl@estgl.ipv.pt

As metodologias descritas enquadram-se nestes princípios, uma vez que todo o processo é monitorado e orientado para identificar necessidades individuais de apoio e assente na demonstração de competências e aptidões adquiridas individualmente e em grupo.

Uma vez que a unidade curricular é eminentemente prática, os exemplos e exercícios são estudos de caso, inicialmente mais simples e, à mediada que a unidade curricular progride, mais complexos, permitindo uma forte ligação a outras temáticas do ciclo de estudos e ao futuro profissional dos estudantes.

Para o trabalhador-estudante é suscitado o processo de autoaprendizagem orientada, podendo a monitorização ser efetuada num misto de presencial e/ou via plataforma de *e-learning* da instituição.

Bibliografia de consulta obrigatória:

Deitel, P. J., & Deitel, H. M. (2016). *C How to Program, Global Edition, 8th Edition*. Pearson Education.

Kernighan, B. W., & Ritchie, D. M. (1988). *The C Programming Language, 2nd Edition*. Prentice Hall.

Rocha, A. (2006). *Introdução à Programação usando C – 2ª Edição*. Editora FCA.

ESTGL – Avenida Visconde Guedes Teixeira, 5100-070 Lamego - Telf. 254615477 - Fax. 254613029 – mail: estgl@estgl.ipv.pt

BIBLIOGRAFIA

Albahari, J., & Albahari, B. (2016). *C# 6.0 in a Nutshell.*

C Standard Library. (n.d.). Retrieved July 28, 2017, from https://en.wikipedia.org/wiki/C_standard_library

Cygwin. (n.d.). Retrieved July 28, 2017, from https://www.cygwin.com/

Deitel, P., & Deitel, H. (2014). *C++ How to Program, 9th Edition* (Vol. 1). Pearson Education. Retrieved from https://www.pearsonhighered.com/product/Deitel-C-How-to-Program-Early-Objects-Version-9th-Edition/9780133378719.html

Deitel, P. J., & Deitel, H. M. (2016). *C How to Program, Global Edition, 8th Edition.* Pearson Education. Retrieved from http://www.pearsoned.co.uk/bookshop/detail.asp?item=100000000608187

Dijkstra, E. W. (1968). Letters to the Editor: Go to Statement Considered Harmful. *Commun. ACM*, *11*(3), 147–148. https://doi.org/10.1145/362929.362947

Eclipse C/C++ Development User Guide. (n.d.). Retrieved July 29, 2017, from http://help.eclipse.org/oxygen/index.jsp?topic=%2Forg.eclipse.cdt.doc.user%2Fconcepts%2Fcdt_o_home.htm&cp=13

GCC, the GNU Compiler Collection. (n.d.). Retrieved July 28, 2017, from https://gcc.gnu.org/

GDB: The GNU Project Debugger. (n.d.). Retrieved July 29, 2017, from https://www.gnu.org/software/gdb/

GNU Make. (n.d.). Retrieved July 29, 2017, from https://www.gnu.org/software/make/

Hanson, D. R. (1997). *C Interfaces and Implementations : Techniques for Creating Reusable Software*. Addison-Wesley. Retrieved from https://www.pearson.com/us/higher-education/program/Hanson-C-Interfaces-and-Implementations-Techniques-for-Creating-Reusable-Software/PGM118741.html

Hayes, J. R. (2001). Modular Programming in C. Retrieved July 17, 2017, from http://www.embedded.com/design/prototyping-and-development/4023876/Modular-Programming-in-C

Kernighan, B. W., & Ritchie, D. M. (1988). *The C Programming Language, 2nd Edition. Book* (Second Edi). Prentice Hall. Retrieved from https://www.pearson.com/us/higher-education/program/Kernighan-C-

Programming-Language-2nd-Edition/PGM54487.html

MinGW. (n.d.). Retrieved July 28, 2017, from http://www.mingw.org/

Mingw-w64. (n.d.). Retrieved July 28, 2017, from https://mingw-w64.org/doku.php

Parnas, D. L. (1972a). A technique for software module specification with examples. *Commun. ACM*, *15*(5), 330–336. https://doi.org/10.1145/355602.361309

Parnas, D. L. (1972b). On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM*, *15*(12), 1053–1058. https://doi.org/10.1145/361598.361623

Parnas, D. L., Clements, P. C., & Weiss, D. M. (1984). The Modular Structure of Complex Systems. In *Proceedings of the 7th International Conference on Software Engineering* (pp. 408–417). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=800054.801999

Schildt, H. (2014). *Java The Complete Reference, 9th Edition* (9th Ed). McGraw-Hill Education. Retrieved from https://www.mhprofessional.com/9780071808552-usa-java-the-complete-reference-ninth-edition-group

Structured programming. (n.d.). Retrieved July 27, 2017, from https://en.wikipedia.org/wiki/Structured_programming

The GNU C Library. (n.d.). Retrieved July 28, 2017, from https://www.gnu.org/software/libc/manual/

Using the GNU Compiler Collection (GCC). (n.d.). Retrieved July 29, 2017, from https://gcc.gnu.org/onlinedocs/gcc/

Vliet, H. Van. (2007). *Software Engineering: Principles and Practice*. Wiley. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.128.2614

Wirth, N. (1971). The Programming Language Pascal. *Acta Inf.*, *1*(1), 35–63. https://doi.org/10.1007/BF00264291

Wirth, N. (1976). MODULA: a language for modular multiprogramming. *Eidgenössische Technische Hochschule Zürich, Institut Für Informatik*, *18*. https://doi.org/10.3929/ETHZ-A-000199440

Wirth, N. (1978). MODULA-2. *Eidgenössische Technische Hochschule Zürich, Institut Für Informatik*, *27*. https://doi.org/10.3929/ETHZ-A-000153014