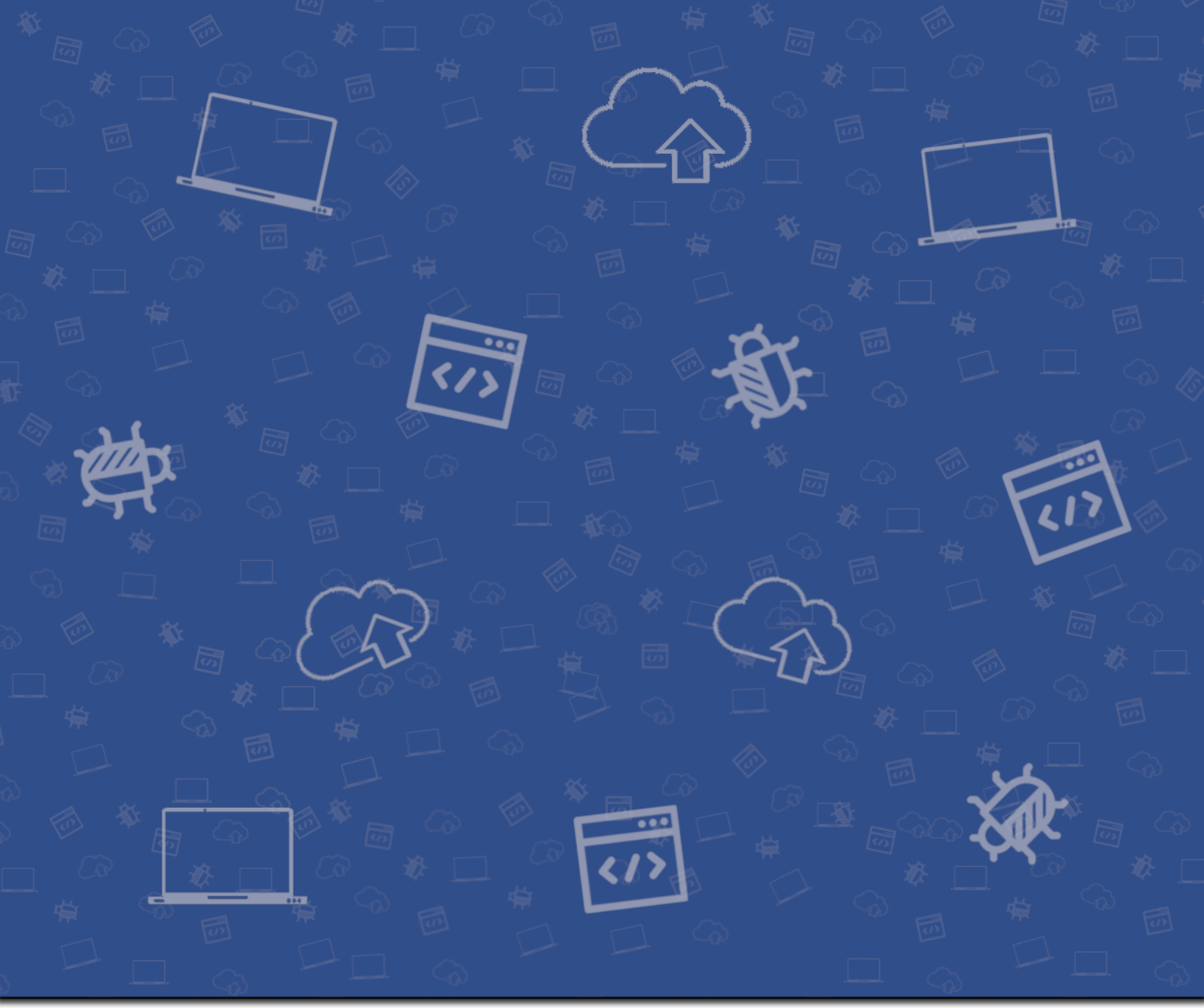

Introdução a Conceitos Base de Engenharia Informática

1ª Edição

Apontamentos de Apoio e Guias Laboratoriais de Laboratórios de Programação

Pedro R. M. Inácio, Tiago Roxo e Tiago M. C. Simões

Publicado como eBook
www.di.ubi.pt/~inacio

Primeira edição, datada de setembro de 2021

Sobre os Autores

Pedro R. M. Inácio

Pedro Ricardo Morais Inácio é professor associado do Departamento de Informática da Universidade da Beira Interior (UBI). Leciona unidades curriculares relacionadas com segurança e garantia da informação, programação e simulação assistida por computador, a cursos de licenciatura, mestrado e doutoramento, nomeadamente aos de Engenharia Informática. É coordenador da equipa de resposta a incidentes CSIRT.UBI e instrutor da Academia CISCO @ UBI.

Tem uma licenciatura pré-Bolonha em Matemática Informática e um Doutoramento em Engenharia Informática, obtidos na Universidade da Beira Interior (UBI) em 2005 e 2009, respetivamente. Os trabalhos de Doutoramento foram desenvolvidos no ambiente empresarial da Nokia Siemens Networks Portugal S.A., ao abrigo de uma bolsa de investigação da Fundação para a Ciência e a Tecnologia.

É membro sénior IEEE, associado ACM e investigador do Instituto de Telecomunicações (IT). Os seus interesses de investigação incluem segurança e garantia da informação, simulação assistida por computador, e monitorização, análise e classificação de tráfego de rede. Tem cerca de 50 publicações em livros, revistas e conferências científicas internacionais com revisão por pares. Revê com frequência artigos para revistas IEEE, Springer, Wiley e Elsevier. Foi membro do Comité Técnico de diversas conferências e seminários nacionais e internacionais, como a ACM Symposium on Applied Computing - Track on Networking. É atualmente Editor Associado da revista IEEE Access.

inacio@di.ubi.pt | www.di.ubi.pt/∼inacio | @in4cio - Twitter | LinkedIn

Tiago Roxo

Nascido em Portugal a 20 de setembro de 1990, concluiu a Licenciatura em Engenharia Informática na Universidade da Beira Interior (UBI) em 2019. Desde 2020 que ingressa no programa doutoral da UBI, associado ao Instituto de Telecomunicações e financiado por uma bolsa de doutoramento da Fundação para a Ciência e a Tecnologia (FCT). É membro IEEE e investigador do Instituto de Telecomunicações (IT). Os seus principais interesses de investigação incluem visão computacional e inteligência artificial.

tiago.roxo@ubi.pt | LinkedIn

Tiago M. C. Simões

Tiago M. C. Simões é atualmente investigador auxiliar no Instituto de Telecomunicações (IT) e professor auxiliar convidado do departamento de Informática da Universidade da

Beira Interior (UBI). Leciona nesta instituição unidades curriculares relacionadas com as temáticas de programação, segurança e tecnologias baseadas na nuvem. Estas unidades pertencem essencial a cursos de licenciatura e mestrado em Engenharia Informática e Informática Web.

Na Universidade da Beira Interior obteve em 2010 uma licenciatura em Engenharia Informática, em 2012 o mestrado em Engenharia Informática e por fim, em 2019, e através de uma bolsa financiada pela Fundação para a Ciência e a Tecnologia (FCT), um doutoramento em Engenharia Informática. Ele é autor e coautor de diversos artigos em conferências e revistas internacionais. Atualmente, os seus principais interesses de investigação são focados nas aplicações e serviços para a Web, segurança, tecnologias baseadas na nuvem e Internet das Coisas (IoT).

tsimoes@di.ubi.pt | @tiagomiguelcs - Twitter | LinkedIn

Prefácio

A maior parte do conteúdo deste livro eletrónico foi elaborado durante a preparação da unidade curricular designada por *Laboratórios de Programção*, incluída no 1º ano do curso de Engenharia Informática da Universidade da Beira Interior. Por esse motivo, o tom da exposição é coloquial em algumas partes, ou até mesmo jocoso, se tal resultar, no entendimento dos autores, em prol da pedagogia. É incluído um sumário no início de cada capítulo, o que também prova a adaptação quase direta dos referidos conteúdos para este formato. Os sumários estão em Português e em Inglês, apenas porque são também assim incluídos nas aulas, para ajudar eventuais estudantes de outras nacionalidades, de visita pelo programa Erasmus, a encontrar conteúdo sobre os temas abordados.

O texto está largamente estilizado, e é recorrente a utilização de **negrito** para **realçar várias partes** que os autores pensam ser importantes. Este conteúdo é fornecido como material de apoio ao estudo de uma unidade curricular, concretizando este livro um esforço para colar todo esse conteúdo, deixando alguns dos artefactos que fazem deles apontamentos.

Este trabalho encontra-se em constante desenvolvimento, e apesar de ter sido revisto várias vezes e por várias pessoas antes de ser disponibilizado, pode conter erros ou falhas, que serão resolvidas em iterações futuras. Caso queira reportar algum erro, ou apenas fazer chegar algum comentário aos autores, queira fazê-lo para inacio@di.ubi.pt.

Este livro está dividido em duas partes principais: a primeira parte concatena os conteúdos de apoio às aulas teóricas; enquanto que a segunda compila uma série de guias laboratoriais. A primeira parte é, por isso, de índole mais estático, de leitura e análise mais dura; enquanto que a segunda tem um tom mais ligeiro e interativo. A maior parte dos tópicos abordados na primeira parte encontram um paralelo prático na segunda. Cada guia laboratorial contém diversas tarefas que guiam o executante com a resolução de exercícios de programação simples, intercaladas com perguntas de escolha múltipla ou direta, cujo objetivo é o de ajudar a estruturar o caminho. Irão perdoar o tom mais brincalhão desta parte em relação à primeira. Sugere-se, claro, a execução prática dos vários guias laboratoriais.

Finalmente, aqui fica expresso o agradecimento aos revisores deste documento. Sem nenhuma ordem em particular, para além da alfabética, um *Muito Bem Haja* à Joana Costa e à Sara Martins.

Conteúdo

I Apontamentos Teóricos

Lista de Figuras

Parte I

Apontamentos Teóricos

1 Definição de Engenharia Informática e Conceitos Associados

Sumário

Âmbito para estudo dos temas que a unidade curricular aborda. Definição de Engenharia Informática.

Summary

Scope for studying the topics covered by this course. Definition of Computer Science and Engineering.

1 Introdução

Em jeito de introdução, este capítulo apresenta os principais objetivos de aprendizagem que se pretendem atingir com estes apontamentos e exercícios. O capítulo termina com uma definição de Engenharia Informática, bem como conceitos associados a este tema.

2 Objetivos de Aprendizagem

O objetivo desta unidade curricular é **introduzir conceitos base** de Engenharia Informática, procurando uniformizar o conhecimento dos estudantes, e expôr as metodologias para **configurar** e explorar ambientes e **ferramentas de programação e desenvolvimento**. Pretende-se fornecer uma **estrutura e formato base para relatórios técnicos**, desenvolvendo aptidões para **gestão de projetos** de programação, nomeadamente **gestão de versões**. Finalmente, pretende-se introduzir a **automação** de tarefas de desenvolvimento, e **boas práticas de programação e documentação**.

No final desta unidade curricular o estudante deve ser capaz de:

- **Configurar o ambiente de trabalho** e manusear ferramentas de desenvolvimento e perfilagem (*profiling*);
- **Gerir versões** de um projeto de *software*;
- **Lidar** com o ambiente de **linha de comandos** e com a **compilação** de projetos de programação;
- **Fazer depuração** de programas;
- **Elaborar relatórios técnicos com qualidade**.

3 Definição de Engenharia Informática

Incluem-se nesta secção três definições (de Engenharia, Engenheiro(a) e Informática) que sustentam, por fim, a discussão do que poderá ser a Engenharia Informática.

3.1 Definição de Engenharia

Uma pesquisa pelos dicionários *Oxford Languages* e *Merriam-Webster* devolve as seguintes definições de Engenharia:

> *"1. the branch of science and technology concerned with the design, building, and use of engines, machines, and structures.*
> *a field of study or activity concerned with modification or development in a particular area (e.g., "software engineering")*
> *2. the action of working artfully to bring something about."*

in Google's English dictionary, *"Engineering."* Oxford Languages, 2020 [Online.] Last Access: October 10, 2020. Available: https://languages.oup.com/google-dictionary-en/.

> *"the work of designing and creating large structures (such as roads and bridges) or new products or systems by using scientific methods (...) the control or direction of something (such as behavior)"*

in Merriam-Webster, *"Engineering."* Merriam-Webster.com Dictionary, 2021 [Online.] Last Access: August 13, 2021. Available: https://www.merriam-webster.com/dictionary/engineering.

Uma definição de Engenharia
Aproveitando as duas definições de dicionários comummente utilizados e com algum prestígio, pode-se dizer que a **Engenharia** é simultaneamente um **ramo da ciência e da tecnologia** que endereça as fases da **conceção**, **construção** ou desenvolvimento **e utilização** de motores, máquinas e estruturas. A **Engenharia** é referida como **a ação de trabalhar com arte** para criar algo **utilizando métodos científicos**. Pode ainda ser considerada como uma forma de controlar ou guiar algo.

3.2 Definição de Engenheiro

Definição de Engenheiro(a)
Um(a) engenheiro(a) é, portanto, **uma pessoa hábil, que cultiva o rigor e que sabe utilizar o conhecimento científico** para conceber, implementar, manter e utilizar motores, máquinas e estruturas.

Note-se que saber conceber implica saber imaginar e **planear com detalhe, não só o que se vai criar, como também o processo** (incluindo ferramentas) usado para essa implementação, sustentado em conhecimento científico. Realça-se ainda o aspeto da multidisciplinaridade associada à definição de Engenharia, bem como o vasto conjunto de aptidões e conhecimento inerentemente associados a esta definição.

3.3 Definição de Informática

Informática e Ciência da Computação (*Computer Science*) são conceitos diferentes mas, em alguns contextos, os termos são usados de forma quase indiferenciada.

Uma definição de Informática
A palavra Informática resulta da aglutinação de duas *substrings* das palavras Informação e Automática: Informática = Infor + mática. Assim, Informática refere-se ao **conjunto de ciências que permitem o processamento racional da informação por meios automáticos** (tipicamente computadores ou, de forma mais lata, dispositivos de processamentos de dados). Em alguns casos, admite-se que esta compreende outras ciências ou áreas específicas, como a ciência da computação, a teoria da informação, sistemas de informação, cálculo, análise numérica e o estudo da modelação e representação do conhecimento e de problemas para fins de processamento automático.

Note-se que a definição evoca os termos ***racional*** **e** ***automático***, que especificam que aquele processamento tem por base uma lógica concretizada por um conjunto de passos a executar (uma algoritmia), e que esses passos podem ser traduzidos em instruções de uma máquina que os execute, eventualmente de forma repetitiva.

3.4 Definição de Engenharia Informática

Engenharia Informática é, portanto, o **ramo da ciência e da tecnologia** que endereça as fases da **conceção**, **construção** ou desenvolvimento, **e utilização** (o que inclui a manutenção) de **sistemas de software e hardware**, que permitem o processamento racional da informação por meios automáticos.
Tem raízes na engenharia eletrotécnica, matemática e linguística; e assume hoje em dia vastos conhecimentos em programação, algoritmia e desenho de programas, administração de sistemas e hardware computacional.

Apesar da discussão na secção anterior, a Engenharia Informática é mais frequentemente referida, em inglês, como *Computer Science and Engineering* do que como *Informatics Engineering*.

2 Estruturação de um Relatório

Sumário

Estrutura e conteúdo de um relatório técnico. Apresentação de metodologias para redação de relatórios com qualidade.

Summary

Structure and content of technical reports. Presentation of methodology for writing of good quality reports.

1 Introdução

Quando se escreve um relatório (aplica-se a mais tipos de documentos), deve-se considerar começar sempre do geral para o detalhe. *i.e.*, deve-se sempre começar por esboçar o índice do documento, começando pelas partes (pré-corpo, corpo, anexos), depois pelos capítulos, depois pelas secções e, finalmente, foca-se no conteúdo.

Note-se que um relatório técnico ou científico é feito por alguém altamente especializado; neste caso, por um(a) engenheiro(a). Assim, **o relatório é**, ele mesmo, **um resultado de um trabalho de engenharia**, seguindo o mesmo rigor científico e planeamento que rege todos os trabalhos de engenharia. Mais, a estrutura do próprio relatório segue, grosso modo, o método de engenharia, passando pela análise do problema (Introdução, Estado-da-Arte), planeamento (Introdução e Planeamento), Implementação/Execução, Testes e Manutenção, e Conclusão.

A maior parte dos comentários ou indicações deste capítulo devem ser entendidos como sugestões, podendo ser aplicados por omissão caso falte indicação em contrário, mas nunca sobrepondo-se a especificações do formato fornecidas ou de um orientador do trabalho.

2 Comentários de Aplicação Geral

Nesta secção aprensentam-se comentários a aplicar na generalidade de um documento técnico ou científico. Dividimos estes comentários em duas vertentes: **qualidade da Língua escrita**; e **qualidade do relatório**.

2.1 Comentários de Qualidade da Língua Escrita

Em termos da forma e da escrita (qualidade da Língua escrita), pode-se apresentar os seguintes comentários:

- O texto deve estar impecável, tanto sintática como semanticamente. Um parágrafo deve descrever uma ideia. Para além disso, como regra heurística, pode considerar que não existem parágrafos com uma só frase;
- Tudo num documento técnico é uma frase, à exceção dos títulos;
- Todas as listas devem começar com uma introdução, ter os itens separados com ponto e vírgula, exceto o último, que termina com ponto final;
- Os títulos devem estar consistentemente capitalizados. Um exemplo de forma de capitalização é capitalizar todas as palavras, exceto os elementos de ligação;
- As legendas (de figuras, tabelas ou trechos de código) devem ser frases, terminar com ponto final e ser auto-contidas, no sentido de descreverem fielmente, mas não exaustivamente, o seu conteúdo.

Alguns dos piores problemas em termos de escrita são os erros ortográficos, os problemas de semântica e **o ritmo**. A colocação de pontuação, nomeadamente vírgulas e ponto-e-vírgulas, é especialmente gravoso. Considere-se dar especial atenção a esse aspeto.

Existem ferramentas gratuitas baseadas na *Web* para identificação de problemas sintáticos, de concordância e ortografia, nomeadamente:

- https://www.languagetool.org/; e
- http://www.expresso-app.org/.

Considerem a utilização destes recursos aquando da escrita de relatórios, especialmente o *Language Tool*. A segunda ferramenta (http://www.expresso-app.org/) chega a fazer sugestões, em termos de sinónimos, para melhorar o texto. Contudo, esta funcionalidade deve ser usada com cautela, porque nem sempre a sugestão serve no contexto. Para

significados de palavras em Língua Portuguesa, o ideal é o dicionário da Porto Editora [1]. **Note que os estrangeirismos** (palavras em língua estrangeira à que está a ser escrita no documento) **devem estar em itálico**. Por vezes, a visita a dicionários ajuda a perceber se a palavra já é considerada nativa ou não. Finalmente, a plataforma *Overleaf* deve ser considerada para escrita em colaboração de documentos LaTeX online.

2.2 Comentários de Qualidade do Relatório

Em termos dos detalhes de qualidade do relatório, apresentam-se os seguintes comentários:

- Todas as tabelas, figuras ou trechos de código colocados dentro de um relatório são referenciados pelo menos uma vez dentro desse relatório e são sempre referidos de forma consistente;
- Os recursos para além do texto (como figuras, tabelas ou trechos de código) têm de ter qualidade adequada (*e.g.*, o tamanho de letra em figuras tem de ser minimamente consistente com o documento em termos de tipo e tamanho da fonte);
- Cada capítulo, exceto a *Introdução* e a *Conclusão e Trabalho Futuro*, tem as secções introdução e conclusão;
- Todos os estrangeirismos devem estar em *itálico*;
- Todos os termos ou referências a código, comandos, nomes de ficheiros ou instruções devem estar em `monospace`;
- Trechos de código só devem estar no corpo do documento se forem fulcrais para a explicação (*e.g.*, são a instanciação de um pequeno algoritmo importante) e ocuparem até 1/4 de página. Excertos de código maiores devem ser colocados em anexo (mas sempre referidos e explicados no texto);
- Os trechos de código devem estar formatados para o próprio relatório (se for necessário, deve ser refeita a indentação) e conter comentários que melhorem a percetibilidade;
- Não devem existir secções vazias nem desperdício de espaço (isto é válido para figuras);
- Os anexos devem ser tratados como capítulos e devem sempre conter uma breve descrição do recurso que contêm e de onde esse recurso é referido no corpo do relatório;

[1] http://www.priberam.pt/DLPO/

- O documento deve ser equilibrado em termos de número de capítulos, secções e páginas. Deve ser também equilibrado em relação à profundidade. Deve-se fazer um esforço para escrever um documento simétrico (*e.g.*, cinco capítulos, cinco secções em cada);
- Não se devem fazer citações após ponto final ou fora de frases;
- As expressões matemáticas devem fazer parte de frases, quer sejam escritas em linha ou de forma isolada;
- As margens das páginas devem ser consistentes ao longo do documento e as margens não devem ser ultrapassadas. A revisão final do documento deve procurar identificar e resolver casos destes.

Note que o **plágio**, tome a forma que tomar (auto-plágio também é plágio), **é uma atividade considerada desonesta e indigna** numa sociedade evoluída. Caso se use um recurso ou pequeno trecho de texto de outro autor, deve-se sempre indicar a fonte e assinalar de forma precisa o que foi usado.

3 Estrutura de um Relatório Técnico

Um relatório técnico é estruturado por diferentes partes, cada uma com a sua finalidade. Em cada uma das secções seguintes apresentam-se as diferentes componentes de uma estrutura geral de relatório. Cada título corresponde a uma parte do relatório, acompanhado de recomendações de redação e uma breve descrição do propósito de cada parte para o documento técnico.

3.1 *Frontmatter*

A *Frontmatter* é composta por secções, texto e recursos que **antecedem** o corpo principal do documento. Seguidamente, apresentam-se as as componentes de um relatório associadas a esta divisória.

Capa

A capa contém um título, apresenta o(s)/a(s) executante(s), bem como a instituição e a data em que o documento foi produzido.

Agradecimentos

É uma boa ideia começar os agradecimentos pelo orientador e terminar nos amigos e família, ou vice-versa.

Resumo

Quando se redige um resumo, dever-se-á ter em consideração se será um resumo do trabalho ou do relatório. Caso considere fazer um resumo do trabalho, sugere-se a seguinte organização:

- Duas a três frases a introduzir o tema e a motivá-lo;
- Duas a três frases a descrever como é que se perseguiu a resolução daquele problema;
- Uma a duas frases a descrever o que é que foi conseguido (*i.e.*, os melhores resultados alcançados).

Palavras-Chave e *Keywords*

Os principais detalhes a ter em conta nesta secção:

- Colocar sempre as palavras-chave ou *keywords* por ordem alfabética;
- Identificar as palavras-chave como sendo aquelas que usaria para procurar o trabalho num motor de busca;
- Devem estar em texto corrido, separadas por vírgula ou ponto-e-vírgula.

Acrónimos

Alguns detalhes a ter em conta nesta secção:

- Colocar sempre os acrónimos por ordem alfabética;
- Os acrónimos devem estar alinhados;
- Os acrónimos têm de ser usados no texto de uma forma consistente;
- Têm de ser definidos por extenso na primeira utilização;
- Em LaTeX devem considerar utilizar o pacote `acronym`;
- Os acrónimos devem estar alinhados. Para isso, deve usar-se: `\begin{acronym}[MAIOR ACRONIMO USADO NO DOCUMENTO]`
- Quando colocarem acrónimos em legendas de figuras ou títulos de capítulos ou secções, certifiquem-se que usam comandos como `\acs*{}` (notar o `s` e o `*`). Por exemplo: `\section{Protocolo \acs*{TCP}}`. O uso do `s*` força a versão curta (short) e que o acrónimo não conte para a primeira definição do mesmo (caso contrário, o acrónimo fica definido no índice de figuras, tabelas ou índice geral).

3.2 Introdução

O capítulo *Introdução* é geralmente divido em quatro partes, apresentadas nas subsecções seguintes.

Âmbito, Enquadramento e Motivação

Esta parte da *Introdução*, tem como intuito responder a questões como:

- O que é este documento?
- De uma forma sintética, qual o contexto deste projeto (convergindo depois para a resposta às duas questões seguintes)?
- Qual a área em que se insere este projeto?
- Quais as duas sub-áreas em que se insere este projeto?
- Por que é importante abordar o problema abordado neste projeto? A resposta deve ser impessoal. Não se trata da motivação pessoal, mas sim da motivação técnica a alimentar o projeto.

A resposta a cada uma das questões apresentadas deve ser dada em um parágrafo. Note que o *Enquadramento* e a *Motivação* podem, por vezes, ser secções diferentes.

Problema e Objetivos do Trabalho

O propósito desta parte da *Introdução* é responder a questões como:

- Qual o problema específico endereçado com este projeto?
- Qual o objetivo principal a alcançar/o que é que se pretende conseguir no final deste projeto?
- Quais os objetivos secundários a alcançar (se existirem)?

A resposta a cada uma das questões apresentadas deve ser dada em um (mínimo) ou dois (máximo) parágrafos.

Abordagem

A *Abordagem* pretende responder a questões como:

- Qual a abordagem seguida para conseguir resolver o problema identificado antes ou atingir o objetivo do trabalho? A resposta deve ser dada em um (mínimo) ou dois (máximo) parágrafos;

- Qual o conjunto de passos ordenados para resolver o problema?

A resposta a estas questões deve ser uma lista ordenada com os passos e métodos, e pode ser complementada por um diagrama de Gantt com a planificação do trabalho.

Organização do Documento

A última parte da *Introdução* pretende descrever a organização do documento. Um exemplo de apresentação desta organização pode ser a seguinte forma:

> *De modo a descrever fielmente o trabalho que foi feito, este documento encontra-se estruturado da seguinte forma:*
>
> - *No capítulo 2 – **Estado-da-Arte, Trabalhos Relacionados** – descrevem-se o estado das tecnologias subjacentes e alguns trabalhos relacionados;*
> - *No capítulo 3 – ...*

Note que não é preciso descrever o próprio capítulo 1 neste ponto, mas devem-se descrever os anexos, se existirem.

3.3 Estado-da-Arte e Trabalhos Relacionados

A estrutura seguinte representa um exemplo de organização deste capítulo num relatório técnico. Tal como referido anteriormente, este capítulo deverá conter as secções *Introdução* e *Conclusão*.

Introdução

Esta secção deverá apresentar uma descrição do que é que o capítulo trata. Um exemplo de exposição desta informação seria:

> *Este capítulo descreve algumas das ferramentas mais parecidas com a que foi desenvolvida neste trabalho. Este capítulo encontra-se estruturado da seguinte forma: a secção 2.2 descreve o estado-da-arte em termos de tecnologias importantes para o projeto; a secção 2.3 descreve outros trabalhos semelhantes ao que foi desenvolvido neste trabalho.*

Estado-da-Arte

Nesta secção pretende-se aferir o estado da tecnologia que permite a resolução dos problemas apontados. Geralmente, a resposta deve ser dada em um (mínimo) ou dois (máximo) parágrafos **por cada tecnologia ou trabalho descrito**. Os trabalhos ou tecnologias

aqui descritas devem ser acompanhados de citações a referências bibliográficas de qualidade reconhecida (ver secção 3.8).

Trabalhos Relacionados

Esta secção tem o intuito de apresentar os trabalhos mais parecidos ao descrito no relatório. A resposta deve ser dada em um (mínimo) ou dois (máximo) parágrafos **para cada trabalho relacionado**, que devem resumir o trabalho de forma simples, e indicar as semelhanças e diferenças para o trabalho atual. Por vezes, é boa ideia terminar esta secção com uma tabela a comparar os trabalhos existentes com o que foi desenvolvido no projeto descrito no relatório. Similarmente ao *Estado-da-Arte*, os trabalhos ou tecnologias aqui descritas devem ser acompanhados de citações a referências bibliográficas de qualidade reconhecida (ver secção 3.8).

Conclusão

A *Conclusão* deverá descrever o que de mais importante se abordou neste capítulo e que permitirá evoluir para as próximas etapas de trabalho descritas. Algumas das conclusões que se podem incluir neste capítulo indicam os trabalhos que mais influenciaram o desenvolvimento do projeto relatado, colocando também as razões por detrás dessa influência.

3.4 Planeamento

Os nomes dos capítulos apresentados nesta estrutura de relatório são **meramente sugestivos**. Muitas vezes, o capítulo do *Planeamento* é simplesmente chamado de *Engenharia de Software*. À semelhança do capítulo 3.3, **este também deverá incluir as secções *Introdução* e *Conclusão***. Para efeitos de simplicidade, neste capítulo serão analisadas as secções que lhe são exclusivas.

Engenharia de Software

Esta secção contém e discute requisitos funcionais e não funcionais, diagramas de casos de uso, diagramas de atividade e sequência, diagramas de arquitetura do sistema e diagramas dos modelos de dados.

Se, devido à quantidade de informação e diagramas a colocar nesta secção, o capítulo começar a ficar demasiado grande, considere-se transformar esta secção num capítulo isolado, colocar as tecnologias importantes no final do capítulo anterior ou no início do próximo, e colocar alguns diagramas em anexo. Ressalva-se a importância de ter sempre **muito cuidado com o equilíbrio do documento**.

Todos os diagramas usados no contexto de um relatório técnico devem advir de linguagens de representação oficiais ou devem ser formas de representação de conceitos muito reconhecidas.

Tecnologias Importantes

O propósito desta secção é identificar **qual ou quais as tecnologias escolhidas para implementar este trabalho** (qual é a justificação para as usar), e mostrar **de que forma é que essas tecnologias foram encadeadas/combinadas para conseguir atingir os objetivos do trabalho**. A resposta a ambas as perguntas deve ser dada em um (mínimo) ou dois (máximo) parágrafos. No caso da identificação da escolha de tecnologias, cada tecnologia deverá ter um parágrafo dedicado à sua descrição.

3.5 Execução / Desenvolvimento

A organização e descrição do trabalho desenolvido varia de trabalho para trabalho. Por essa razão, apenas se apresenta **nomes sugestivos** de secções, aplicáveis a um relatório técnico, sem detalhar o intuito de cada uma delas. Uma sugestão de estrutura deste capítulo seria:

1. Introdução;
2. Dependências;
3. Detalhes de Implementação;
4. Procedimento de Instalação;
5. Conclusão.

3.6 Testes e Validação

Um trabalho desenvolvido com qualidade deverá ser devidamente testado e validado, sendo por isso comum dedicar um capítulo do relatório técnico a esta parte. De forma análoga ao capítulo 3.5, apenas se apresenta **nomes sugestivos** de secções. Uma possível de estrutura deste capítulo seria:

1. Introdução;
2. Especificação dos Testes;
3. Execução dos Testes;
4. Análise dos Resultados e Aprimoramento;
5. Conclusão.

3.7 Conclusões e Trabalho Futuro

O capítulo de *Conclusões e Trabalho Futuro* é, geralmente, divido em duas secções: **Conclusões Principais** e **Trabalho Futuro**. Desta forma, torna-se mais claro quais as conclusões que advêm do trabalho realizado e qual o trabalho que se projeta realizar, no futuro.

Conclusões Principais

Esta secção pretende expôr **se foram atingidos todos os objetivos do trabalho**, evidenciando, em caso afirmativo, **provas de que tal foi feito**. Adicionalmente, é esperado que se apresentem as **conclusões principais a que o(a) estudante chegou no fim do trabalho**. A resposta a ambos os pontos apresentados deve usar tantos parágrafos quanto necessário.

Trabalho Futuro

A secção de *Trabalho Futuro* deve **apresentar o que ficou por fazer (e porquê)**, mostrar o **que seria interessante fazer**, mas não foi feito por não ser exatamente o objetivo deste trabalho, e referir **que outros casos ou situações ou cenários** – que não foram estudados no contexto deste projeto por não ser o seu objetivo – **é que o trabalho aqui escrito pode ter aplicações interessantes** (e porquê). Novamente, não existe uma restrição do número de parágrafos a usar para expôr a informação necessária.

3.8 Bibliografia

A *Bibliografia* deve seguir o formato sugerido no *template* ou, caso não exista, deve ser considerado o formato `IEEEtrans`. Não se devem poupar esforços em relação aos recursos bibliográficos a usar. A bibliografia deve ser completa, cristalizando de forma fiel as referências que foram usadas durante o trabalho.

Para a pesquisa e inclusão de referências bibliográficas referentes ao estado da arte ou trabalho relacionado, devem favorecer, **por esta ordem**:

1. Artigos científicos em revista;
2. Artigos científicos em conferência;
3. Livros ou capítulos em livro;
4. Teses e dissertações;
5. Relatórios técnicos reconhecidos;
6. Referências eletrónicas (artigos, posts, *etc.*).

Alguns apontamentos importantes relativamente a referências bibliográficas:

- **Todas** as referências bibliográficas devem conter, pelo menos, **o nome do autor, o título e o ano**;
- As referências eletrónicas devem conter adicionalmente a data do último acesso e o *Uniform Resource Locator* (URL);
- Os nomes das revistas científicas ou dos encontros científicos devem ser escritos de forma consistente (*e.g.*, sempre por extenso);
- Os títulos das referências, o nome das revistas ou conferências ou o título das obras (*e.g.*, livros, teses e dissertações) devem estar devidamente capitalizados;
- Todas as referências na lista bibliográfica têm de estar obrigatoriamente citadas no texto pelo menos uma vez.

Nunca usar a *wikipedia* como referência bibliográfica (embora possam usar sem citar, claro). Se encontrarem algo na *wikipedia* e precisarem da citação correta, procurem a melhor referência na secção de referências da própria página *wikipedia*.

Uma ferramenta excelente para fazerem pesquisas bibliográficas (para além do Google) é o IEEExplore.

3.9 *Backmatter*

A *backmatter* contém os anexos e apêndices ao documento. Estes recursos são aqui colocados porque não couberam no corpo do documento ou porque são recursos de caráter opcional mas importante para a discussão. Todos os anexos devem ser referidos no texto do corpo principal, e todos os anexos devem ter uma pequena introdução onde também se refira onde são mencionados no corpo principal.

4 Processo de Revisão

É sempre boa ideia **definir editores e um processo de revisão para documentos técnicos**. Quando um relatório é feito no seio de um grupo de trabalho, podem alguns dos elementos ser os editores e outros serem os revisores (costuma ser boa ideia nomear editores e revisores diferentes, já que é por vezes difícil a um editor notar os seus próprios erros). Adicionalmente, deve ser definido o próprio processo de revisão em termos de datas. Por exemplo, dizer que:

- O capítulo 1 deve ser escrito pelo Pedro até dia 30 de outubro;

- O capítulo 1 deve ser revisto pela Joana até dia 2 de novembro.

Após a compilação que se considera como sendo a final de um documento LaTeX, deve fazer-se uma pesquisa no *Portable Document Format* (PDF) por `!` (ponto de exclamação) e `?` (ponto de interrogação) para encontrar eventuais problemas de referências falhadas. Estas devem ser corrigidas, se necessário.

3 Introdução ao Ambiente de Linha de Comandos

Sumário
Introdução ao ambiente de linha de comandos: criação de ficheiros e diretorias, navegação pelo sistema de ficheiros e execução de programas.

Summary
Introduction to the command line interface: creating files and directories, browsing the file system and running programs.

1 Introdução ao Ambiente de Linha de Comandos

O ambiente de linha de comandos constitui **uma forma de interagir com o Sistema Operativo (SO)** e, portanto, gerir o computador (leia-se "recursos computacionais") através de **interfaces baseadas em texto**, tanto para entrada como para saída de dados. A maior parte dos SOs contém um fácil acesso a um ambiente destes, sendo que alguns apenas oferecem este ambiente (*e.g.*, o *Cisco Internetwork Operating System*).

Será de simples compreensão que a concretização de um ambiente deste tipo só é possível se existirem programas que operem daquela forma (*i.e.*, recebendo *inputs* e produzindo *outputs* em texto) e um interpretador que medeie a relação entre esses programas e o utilizador.

A **interface que permite interagir usando apenas texto em linha** (tanto para entrada como saída) com um sistema, com partes do sistema ou com um programa é designada por Interface de Linha de Comandos (da designação inglesa ***Command Line Interface*** **(CLI)**).

Em vários SOs com interface gráfica é possível aceder ao ambiente de linha de co-

mandos através de um programa específico com aspeto semelhante para todos: o programa `consola` (Windows) ou `terminal` (Unix ou Unix-*like*). Por exemplo, o programa `gnome-terminal`, exibido na figura 3.1, é um terminal para o ambiente de *desktop* Gnome.

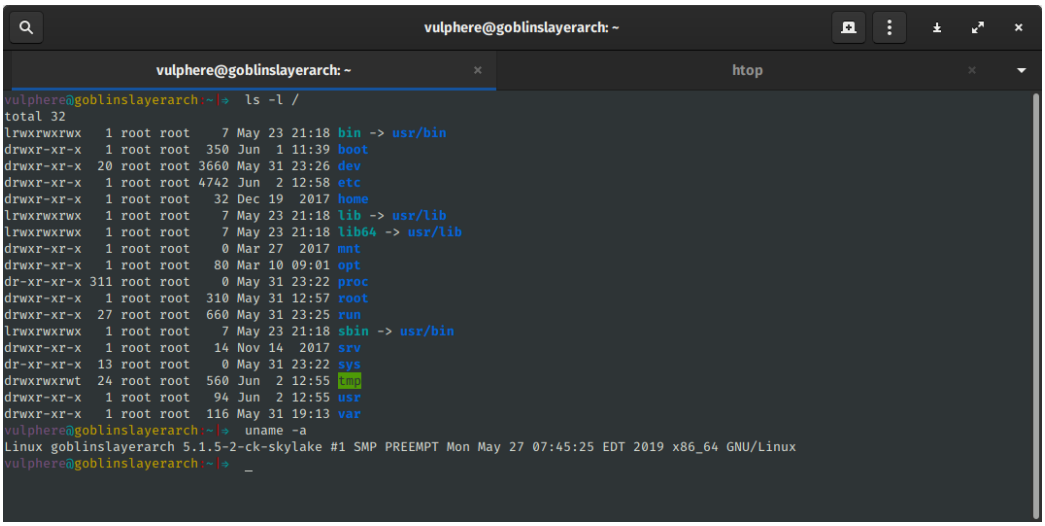

Figura 3.1: Captura de ecrã de um `gnome-terminal`.

Note que é normalmente possível evoluir diretamente para um ambiente de linha de comandos através da reinicialização de um sistema operativo.

Ao aceder ao ambiente de linha de comandos, o utilizador é normalmente confrontado com um pouco de texto e um cursor a piscar, indicativo de que o interpretador está pronto a receber *input*. **Esse pedaço de texto e cursor é conhecido por *prompt*.**

É importante saber interpretar a *prompt*. No caso dos sistemas Unix ou Unix-*like*, a *prompt* é tipicamente construída da seguinte forma: `[user@hostname diretoria] $`, em que:

- O campo `user` tem o nome do utilizador autenticado no sistema;
- O símbolo `@` significa *at* no sentido de "*estar autenticado no domínio*";
- O campo `hostname` tem o nome do sistema em que o utilizador está autenticado;
- O campo `diretoria` tem o nome da diretoria onde o ambiente se encontra.

Quando o utilizador autenticado tem privilégios de administração, o símbolo `$` muda para `#`. Assim, a seguinte *prompt* significa que o utilizador `root` tem a permissão do sistema para fazer tudo: `[user@hostname diretoria] #`.

Note que até a palavra *prompt* dá uma ideia de "pronto (estar à espera) para fazer algo".

Deve ser entendido que **a *prompt* é dada por um programa que serve de interpretador de comandos**. É inclusive possível instalar diferentes programas com essa funcionalidade (*e.g.*, a `z shell` e a `tee-see-shell`) e até configurar, ao gosto, essa *prompt*. Estes programas, por servirem de ponto de interface com o SO, são designados por `shell` (casca, no sentido de rodearem o núcleo do SO).

A *shell* mais comum dos sistemas Unix e Unix-*like* atuais é a *Bourne Again Shell* (bash). Atualmente, há `shells` bastante poderosas que oferecem adicionalmente serviços e comodidades (como o autocompletar ou a pesquisa avançada no histórico de comandos) aos utilizadores, bem como a definição de linguagens de programas.

2 Introdução ao Sistema de Ficheiros

A **navegação e manipulação do sistema de ficheiros** é uma das principais habilidades a adquirir no contexto da interação por linha de comandos.

O **sistema de ficheiros** é o conjunto de métodos e estruturas de dados que o SO usa para gerir, controlar e indexar os ficheiros num disco ou numa partição e que determina como é que esses ficheiros são organizados e entendidos. Sem o sistema de ficheiros, os dados gerados ou guardados num determinado meio de armazenamento ficariam guardados sem organização nem forma de saber onde começavam ou terminavam.

Uma das ideias mais interessantes a reter em termos de sistemas Unix e Unix-*like* é que **quase tudo nestes sistemas é considerado ou tratado como sendo um ficheiro**. Assim, em sistemas Unix e Unix-*like*:

- **Uma diretoria** é considerada um ficheiro (é um ficheiro que tem outros ficheiros lá dentro);
- **O monitor** é considerado um ficheiro (para onde se escrevem dados);
- **O teclado** é considerado um ficheiro (de onde se leem dados);
- **A memória** é considerada um ficheiro (de onde se leem e escrevem dados);
- **A impressora** é considerada um ficheiro (para onde se escrevem dados);
- **Um disco inteiro** pode ser montado como um único ficheiro (para onde se escrevem e leem dados);
- Finalmente, **um ficheiro** é considerado um ficheiro.

No ambiente de linha de comandos, há três fluxos de dados (*streams*) que interessa conhecer: `stdin` (*standard input*), `stdout` (*standard output*) e `stderr` (*standard error*). **Estes três fluxos de dados são tratados como ficheiros** (o primeiro é um ficheiro

de onde se podem ler dados de entrada, e os outros são ficheiros onde se podem escrever dados).

Por omissão, **qualquer comando que funcione em CLI usa automaticamente aqueles três fluxos de dados:** o `stdin` está normalmente ligado ao teclado; e o `stdout` e `stderr` estão ligados ao monitor.

3 Ajuda, Comentários e Ecos em *Bash*

Alguns dos comandos mais interessantes e com uso mais perene são os que permitem obter ajuda acerca de outros comandos. Em sistemas Unix e Unix-*like*, os programas para linha de comandos incluem, frequentemente, uma documentação rica que pode ser acedida com o comando `man`. Por exemplo, o comando seguinte permite obter o manual para o comando `echo`:

```
man echo
```

Note que, no comando anterior, a palavra `echo` é o parâmetro de entrada do comando `man`.

Depois de emitir o comando `man`, o contexto do ambiente muda para o da documentação. É possível navegar por este ambiente com as setas direcionais (linha a linha) ou com a tecla de espaço (página a página). Para sair do ambiente, pode-se usar a tecla `q` (proveniente da palavra inglesa ***q**uit*).

Muitos comandos mostram também uma ideia muito concisa da sua forma de operar e serem invocados se precedidos com a opção `--help`. Por exemplo, a instrução seguinte mostra apenas a lista de opções disponíveis para o próprio comando `man`.

```
man --help
```

É também tipicamente possível obter a versão do *software* de um comando com a opção `--version`, *e.g.*,

```
man --version
```

que produz um *output* semelhante a

```
man 2.9.3
```

É possível colocar comentários no ambiente de linha de comandos usando o caractere `#`:

```
# isto é um comentário; não é interpretado de forma alguma
```

Um comando que, inicialmente, pode parecer não ter muita utilidade é o comando `echo`. A sua utilidade ficará mais clara à medida que for usando a CLI, sobretudo *scripts*. O comando `echo` pode ser usado para fazer eco de texto para a `stdout`:

```
# O próximo comando pode ser usado para fazer eco de
# texto para uma stream especial chamada stdout:
echo "Pipalacaquigrafos."

# output:
Pipalacaquigrafos.
```

Note que foi incluído um comentário junto com o comando para precisamente ilustrar a integração de um comentário antes de um comando. Foi também mostrado o *output* do comando no mesmo excerto.

4 Autocompletar

A maior parte das *shells* disponibiliza a funcionalidade de autocompletar um comando nativamente. Isto significa que é tipicamente possível começar a escrever o comando e carregar na tecla `tab` para acabar de escrever o comando ou obter sugestões de possíveis comandos para determinada combinação de letras. Se houver apenas um comando que satisfaça determinada combinação de letras, carregar no `tab` acaba de preencher o comando; se houver mais do que um comando que satisfaça essa combinação, são dadas várias possibilidades. No exemplo seguinte, ao escrever `where` + `tab` obtêm-se duas sugestões de comandos (comandos `where` e `whereis`):

```
where (tab)
where whereis
```

5 Criação e Navegação em Diretorias

Nesta secção listam-se comandos importantes no contexto da criação e navegação no sistema de ficheiros. Quando acede ao ambiente de linha de comandos, encontra-se localizado numa determinada diretoria do sistema (tipicamente a diretoria `home` do utilizador atual). O comando `pwd` *(**p**rint **w**orking **d**irectory)* imprime o caminho completo da diretoria onde a *prompt* se encontra:

```
pwd
```

O comando `ls` lista todos os ficheiros e sub-diretorias da diretoria atual:

```
ls
```

O próximo comando lista (`ls`) todos os ficheiros e sub-diretorias da diretoria atual, mas **mostrando mais detalhes**, nomeadamente as permissões, o dono, o grupo do ficheiro e a data de modificação:

```
ls -l

# Exemplo de output:
drwxr-xr-x  2 in4cio in4cio  4096 ago 26 15:48 Pictures
drwxr-xr-x  2 in4cio in4cio  4096 set  5  2019 Public
```

O comando anterior está guarnecido pela opção `-l`.

Em sistemas Unix e Unix-*like*, os comandos podem levar opções e parâmetros. **As opções mudam o comportamento dos comandos e seus *outputs* e os parâmetros fornecem dados adicionais ao comando**. As opções são introduzidas por `-` ou por `--`, enquanto que os parâmetros não. Os parâmetros aparecem tipicamente mais para o final de um comando. As opções começadas com `-` têm apenas uma letra a seguir, enquanto que a versão verbosa de uma opção é introduzida por `--`. Nesse sentido, o comando

```
rm -r
```

é o mesmo que

```
rm --recursive
```

Em sistemas Unix e Unix-*like*, os ficheiros e diretorias começadas por `.` estão escondidos. Para se verem todos os ficheiros e diretorias, incluindo os que estão escondidos, pode ser usada a opção `-a` (de ***a**ll*) do comando `ls`, como se exemplifica a seguir:

```
ls -la

# Exemplo de output:
drwx------ 35 in4cio in4cio   4096 out 28 15:54 .
drwxr-xr-x  4 root   root     4096 set  5  2019 ..
drwx------  3 in4cio in4cio   4096 mar 13  2020 .adobe
...
drwxr-xr-x  2 in4cio in4cio  4096 ago 26 15:48 Pictures
drwxr-xr-x  2 in4cio in4cio  4096 set  5  2019 Public
```

Note as duas primeiras entradas no exemplo de *output* em cima. Em sistemas Unix e Unix-*like*, é comum representar a diretoria atual com um `.` e a diretoria anterior com `..`

Note que o comando mostrado antes combina duas opções: `-l` e `-a`. Na verdade, o comando é equivalente a

```
ls -l -a
```

Em muitos casos é possível unir opções para uma escrita mais compacta. Isto só é possível para opções curtas (introduzidas por `-`) e desde que não tenham parâmetros.

O comando `mkdir`, seguido do nome da diretoria (parâmetro obrigatório), pode ser usado para criar uma diretoria, conforme se exemplifica a seguir:

```
mkdir Teste
```

Para navegar para uma determinada diretoria, pode usar o comando `cd`, seguido do nome da diretoria (ou do caminho dessa diretoria). O comando seguinte evolui o contexto do ambiente de linha de comandos para a diretoria *Teste*:

```
cd Teste
```

Para voltar para a diretoria anterior pode-se usar `cd ..`, cujo significado é *change directory* ***up***:

```
cd ..
```

Note-se que o caractere ∼ é tipicamente usado para simbolizar a diretoria padrão de determinado utilizador. Assim, é possível voltar rapidamente para essa diretoria com o atalho seguinte:

```
# o comando seguinte permite voltar para a diretoria home/user
cd ~
```

A remoção de diretorias pode ser conseguida através do comando `rm` (da designação inglesa ***rem****ove*). Contudo, por se tratar de uma diretoria, é necessário colocar a opção `-r` (da palavra ***r****ecursive*), de modo a indicar explicitamente que se quer eliminar uma diretoria, todos os ficheiros que contenha, bem como outras sub-diretorias (daí a recursividade). O exemplo seguinte mostra como apagar uma diretoria chamada *Teste*:

```
rm -r Teste
```

6 Criação e Edição de Ficheiros

Esta secção apresenta alguns comandos destinados à criação e edição de ficheiros. A criação de um ficheiro vazio, chamado `vazio.md`, é conseguida usando o comando:

```
touch vazio.md
```

Na verdade, o comando `touch` pode também ser usado para mudar a data do último acesso a um ficheiro, sem alterar o conteúdo desse ficheiro.

A eliminação de um ficheiro pode ser conseguida através do comando `rm`, seguido do nome do ficheiro. Por exemplo, o próximo comando pode ser usado para eliminar o ficheiro `vazio.md`:

```
rm vazio.md
```

Também é possível criar um ficheiro e imediatamente inserir texto neste. Para tal, podemos usar o comando:

```
echo "Esta frase tem sujeito e predicado." > comtexto.md
```

O comando anterior faz uso do recurso de redirecionamento de fluxos (`>`) para conseguir o efeito desejado. Este recurso é discutido, em detalhe, na secção 8. Contudo, aproveita-se a oportunidade para enfatizar a utilidade do comando echo na criação de um ficheiro (`comtexto.md`, no exemplo anterior).

O comando `cat` pode ser usado para rapidamente ver o conteúdo de um ficheiro, chamado `comtexto.md`, como se ilustra a seguir:

```
cat comtexto.md

# Exemplo de output:
Esta frase tem sujeito e predicado.
```

Note que este comando abre o ficheiro, mostra (lê) o seu conteúdo, e fecha logo o ficheiro.

7 Programas e Miscelânea

Neste secção apresentam-se comandos de aplicação variada, aquando do uso de linha de comandos.

O comando `whereis`, seguido do nome do comando procurado, mostra o caminho absoluto do programa no sistema de ficheiros. Por exemplo, ao procurar `ls` obtém-se o output seguinte

```
whereis ls

# Exemplo de output:
ls: /usr/bin/ls /usr/share/man/man1/ls.1.gz /usr/share/man/man1/ls.1p.gz
```

O *output* do exemplo anterior indica que o comando `ls` está instalado em `/usr/bin/ls`, e a sua documentação está em `/usr/share/man/man1/`.

O comando `wc` conta o número de linhas, palavras e bytes num ficheiro cujo nome é dado como argumento ao comando. Um exemplo de aplicação deste comando:

```
echo "Esta frase tem sujeito e predicado." > comtexto.md
wc comtexto.md

# Exemplo de output:
1  6 36 comtexto.md
```

O comando `grep` pode ser usado para encontrar uma palavra dentro de um ficheiro, mostrando o contexto (uma ou mais linhas) diretamente no ambiente de linha de comandos em caso afirmativo. No trecho seguinte, o comando `grep` mostra apenas a linha do programa que tem um `printf`:

```
echo '#include<stdio.h>\n\nvoid main(){\n printf("Olá!");\n}' > program.c
grep print program.c

# Exemplo de output:
printf("Olá!");
```

Os dois comandos mencionados a seguir permitem encontrar ficheiros no sistema de ficheiros. O comando `locate` (que não está disponível em todas as distribuições) aceita o nome do ficheiro como parâmetro e procura a sua localização numa base de dados que vai atualizando frequentemente:

```
locate comtexto.md

# Exemplo de output:
/home/in4cio/comtexto.md
```

O comando `find` é muito mais poderoso que o `locate`, mas pode demorar mais, uma vez que a procura é feita para o sistema de ficheiros, e não para uma base de dados previamente construída. A seguir incluem-se dois exemplos de utilização do comando `find`. O primeiro mostra como se pode encontrar o ficheiro `comtexto.md` numa diretoria em particular (neste caso `/home`) e o segundo mostra como encontrar ficheiros vazios na diretoria `/home/in4cio`:

```
find /home -name comtexto.md

# Exemplo de output:
/home/in4cio/comtexto.md

find /home/in4cio -empty

# Exemplo de output:
/home/in4cio/vazio.txt
```

8 Redirecionamento e *Pipes*

Como referido na secção 2, cada comando usa, por omissão, três fluxos de dados, que são entendidos como ficheiros no sistema: o `stdin`, para entrada de dados e normalmente ligados ao teclado; o `stdout`, para saída de dados normal, tipicamente ligados ao terminal e monitor; e o `stderr`, para saída de dados relacionados com erros, tipicamente ligados ao terminal e monitor. Duas das funcionalidades mais interessantes oferecidas pelos ambientes de linha de comandos Unix e Unix-*like* são os *pipes* (túneis) e os redireci-

onamentos, **que permitem ligar ou redirecionar esses fluxos**. Nas secções seguintes apresentam-se comandos referentes a estas duas funcionalidades.

8.1 Redirecionamento

Em ambiente de linha de comandos é possível redirecionar o `stdout` para um ficheiro (símbolo `>`), ao invés de este ser exibido no ecrã. O exemplo seguinte demonstra esta funcionalidade:

```
# O comando seguinte mostra o texto no próprio terminal (e ecrã):
echo "Esta frase tem sujeito e predicado."

# O comando seguinte NÃO mostra o texto, mas coloca-o dentro do ficheiro:
echo "Esta frase tem sujeito e predicado." > comtexto.md
```

De forma análoga, é possível redirecionar o `stderr` para um ficheiro (usando os símbolos `2>`), ao invés de ser exibido no ecrã:

```
# O comando seguinte mostra o erro no próprio terminal (e ecrã),
# porque o ficheiro não existe:
rm ficheiro-que-nao-existe.md

# O comando seguinte NÃO mostra o erro no próprio terminal (e ecrã),
# mas envia-o para o ficheiro erro.txt:
rm ficheiro-que-nao-existe.md 2> erro.txt
```

Note que é possível redirecionar **ambos os fluxos** para um ficheiro com o símbolo `&>`.

Os exemplos anteriores demonstram como funciona o redirecionamento para o `stdout` e `stderr`, mas deve adicionalmente ser referido que, tanto o `>` como o `2>`, criam ou reescrevem sempre o ficheiro apontado como destino. Caso se queira **concatenar** texto no final de um ficheiro de saída, devem ser usados os símbolos `>>`, `2>>` ou `&>>`.

Finalmente, é importante referir que também é possível redirecionar o `stdin` recorrendo ao símbolo `<`. Contudo, essa possibilidade não é explorada no contexto desta unidade curricular.

8.2 *Pipes*

Os *pipes* constituem **formas de ligar o fluxo de saída (`stdout`) de um comando ao fluxo de entrada (`stdin`) de outro comando**. Para isso, usa-se o símbolo `|`, que representa, neste caso, um tubo virado para cima. O exemplo seguinte mostra como combinar o comando `ls` com o comando `wc` para se saber quantos ficheiros e sub-diretorias contém a diretoria atual:

```
ls

# Exemplo de output:
BitSync  Desktop  Documents  Downloads  music  Music
Pictures  Public  Templates  trash  Videos  Virtual-Machines

# Uso do pipe para contagens:
ls | wc

# Exemplo de output (12 palavras, 12 linhas e 104 bytes):
12   12   104
```

Repare que, apesar de ainda não ser óbvio, o redirecionamento e, sobretudo, os *pipes* **adicionam um potencial enorme**, já que permitem combinar vários comandos. **Podem-se combinar tantos pipes quanto necessário**. Isto é possível porque todos os comandos usam entradas em saídas padrão: texto.

9 Edição, Compilação e Execução de Programas

Uma das funcionalidades mais importantes da linha de comandos é a edição, compilação e execução de programas. É possível, com apenas **três** instruções em *bash* **criar**, **compilar** e **executar** um programa. O exemplo seguinte demonstra, exatamente, esta situação:

```
echo '#include<stdio.h>\n\nvoid main(){\n printf("Olá!");\n}' > program.c
cc -o program.exe program.c
./program.exe

# Exemplo de output:
Olá!
```

No exemplo anterior, a primeira linha correspondeu à criação de um ficheiro (e imediatamente colocar um programa lá dentro), a segunda demonstra um comando de compilação desse programa e a terceira linha refere-se à execução do mesmo.

Embora seja possível realizar os três processos (criar, compilar e executar) por linha de comandos, o ideal é usar um editor de texto para editar um programa. Uma boa escolha para editar ficheiros em ambiente de linha de comandos é o editor `nano`. Este editor de texto é leve e simplista, mas incorpora diversas funcionalidades, como a pesquisa de texto, navegação por palavra ou linha, e pesquisa e substituição. Na figura 3.2, é exibida a interface deste editor, onde se mostra as várias funcionalidades na parte de baixo, bem como o atalho para lhes aceder. Por exemplo, para gravar (*Write Out*), basta emitir a combinação de teclas `ctrl+O` (o símbolo ^ significa `ctrl`).

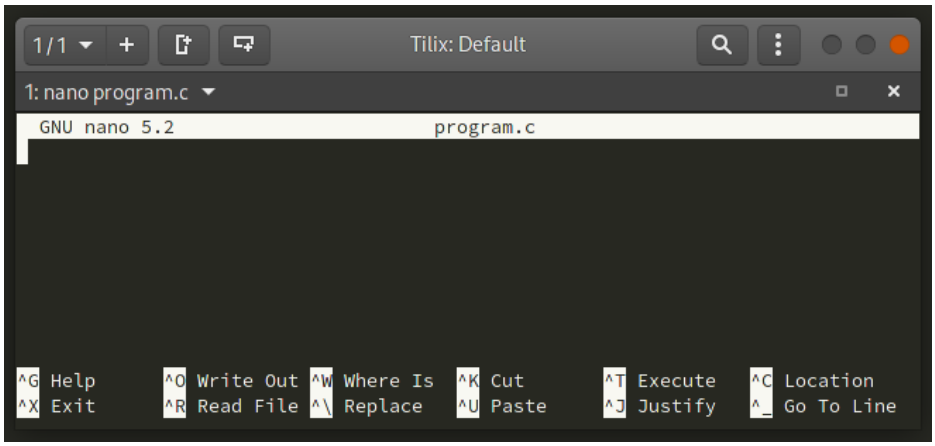

Figura 3.2: Interface do editor `nano`.

Para editar um ficheiro chamado `program.c`, basta emitir uma instrução semelhante à próxima:

```
nano program.c
```

Aquando da execução da instrução, se o ficheiro `program.c` não existir, este é criado depois de se guardarem as alterações efetuadas.

Considere que abre o ficheiro `program.c` e implementa nele o código seguinte:

```
#include<stdio.h>

int main(){
  printf("Hello Planet Earth!\n");
  return 0;
}
```

Após implementar o código, emite-se a sequência `ctrl+x`, escolhe-se o nome do ficheiro de saída (ou carrega-se em `enter` para aceitar o nome atual) e pressiona-se `y` (de *yes*) para confirmar.

Para compilar o programa, invoca-se o compilador de programas escritos em C, que aceita o nome do programa em C e, se não houverem erros a reportar, produz um ficheiro executável chamado `a.out`:

```
cc program.c
```

Caso se queira guardar o ficheiro executável com determinado nome (*e.g.*, `program.exe`), pode sempre usar-se a opção `-o`, seguida do nome do ficheiro pretendido, como se mostra a seguir:

```
cc -o program.exe program.c
```

Para executar o programa, deve invocar-se o nome do ficheiro, **embora precedido de** `./`, como se ilustra a seguir:

```
./a.out
# ou
./program.exe

# Exemplo de output:
Hello Planet Earth!
```

Note que, normalmente, não é preciso colocar `./` para executar um programa no ambiente de linha de comandos. **Isto acontece porque a maior parte dos comandos estão em diretorias conhecidas, e referidas numa variável chamada `PATH`.** Como a diretoria onde os projetos são normalmente implementados não está referida na `PATH`, é preciso indicar **explicitamente** que o programa a executar está na diretoria atual com `./` (recorde que `.` é usado para simbolizar a diretoria atual). É possível ver quais são as diretorias que estão na `PATH` emitindo o comando:

```
echo $PATH
```

Repare que, para implementar programas em outras linguagens, o processo é em todo semelhante, com ajustes menores. Por exemplo, para um programa em Java, começa-se por editar o ficheiro da classe principal (*e.g.*, `HelloWorld.java`):

```
nano HelloWorld.java
```

e implementa-se no ficheiro o código seguinte:

```
class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello Planet Earth!");
  }
}
```

O **c**ompilador Java (`javac`) tem um nome semelhante ao **c**ompilador C (`cc`),

```
javac HelloWorld.java
```

Finalmente, para se executar um programa em Java, instancia-se uma máquina virtual Java que execute o programa através do comando `java`:

```
java HelloWorld

# Exemplo de output:
Hello Planet Earth!
```

Em Python, o processo é mais simples, já que a linguagem é normalmente interpretada (em vez de compilada). Assim, a elaboração e execução de um programa em Python pode ser feita em dois passos: edição e interpretação. O comando seguinte abre o ficheiro `program.py` para edição:

```
nano program.py
```

Considere que é implementado nesse ficheiro o seguinte trecho de código Python:

```
if __name__ == "__main__":
  print("Hello Planet Earth!\n")
```

O programa pode ser *executado* (leia-se *interpretado*) com o seguinte comando:

```
python program.py

# Exemplo de output:
Hello Planet Earth!
```

Em *Hypertext PreProcessor* (PHP), o processo é semelhante ao que foi usado para Python. O comando seguinte abre o ficheiro `program.php` para edição:

```
nano program.php
```

Considere que é ali implementado o seguinte trecho de código PHP:

```
<?php
  echo("Hello Planet Earth!");
?>
```

O programa pode ser *executado* (leia-se *interpretado*) com o seguinte comando:

```
php program.php

# Exemplo de output:
Hello Planet Earth!
```

Note que só é possível compilar ou interpretar código se tiver os respetivos programas de compilação ou de interpretação instalados. Os exemplos anteriores assumem que os programas `cc`, `javac`, `java`, `python` e `php` estão instalados corretamente na máquina.

10 *Scripts*

Os *scripts* são programas ou sequências de instruções que são interpretadas por outro programa (um interpretador), ao invés de serem compilados e executados diretamente no processador.

É muito comum os ambientes de linhas de comandos atuais definirem uma linguagem de *scripting* para potenciar ainda mais aquele ambiente. A *bash* não é exceção.

No caso específico da linha de comandos, um ***script* é sobretudo constituído por conjuntos de comandos que são executados sequencialmente**, eventualmente adorna-

dos por instruções de controlo de fluxo (como `if...else` e `while`). Para fazer e executar um *script* em *bash* são necessários quatro passos principais:

1. Editar o ficheiro que irá conter o *script* (*e.g.*, ficheiro `script.sh`):

```
nano script.sh
```

2. Escrever o conjunto de comandos a executar dentro do ficheiro:

```
#!/bin/bash

echo "A data de hoje e a hora são as seguintes"
date
```

3. Tornar o ficheiro com o *script* executável com o comando `chmod u+x nome_ficheiro`:

```
chmod u+x script.sh
```

4. Executar o *script* como se de outro programa se tratasse:

```
./script.sh

# Exemplo de output:
A data de hoje e a hora são as seguintes
sex 30 out 2020 11:08:03 WET
```

Há alguns detalhes da explicação anterior que importa realçar:

1. Que o conteúdo do ficheiro contendo o *script* começa com `#!/bin/bash` (a combinação `#!` lê-se *shebang!*), o que basicamente significa que este *script* deve ser sempre interpretado pelo *bash* (e não por outro interpretador);

2. O *script* contém duas instruções (`echo` e `date`) que são executadas sequencialmente. Neste caso, é equivalente a corrermos os dois comandos, um a seguir ao outro, num terminal;

3. O comando `chmod`, que ainda não havia sido referido antes, pode ser usado para mudar as permissões de um ficheiro, nomeadamente para o marcar como executável.

Não é objetivo estar a aprofundar em demasia a elaboração de *scripts*, mas podem-se dar vários exemplos rápidos que introduzem conceitos e recursos interessantes neste contexto, e é assim que evolui esta discussão em baixo.

10.1 Parâmetros de Entrada

É possível executar um *script* com parâmetros (dentro do *script* os parâmetros são denotados por `$1`, `$2`, `$3`, *etc.*). O trecho seguinte mostra um *script bash* que usa dois parâmetros de entrada:

```
#!/bin/bash

echo "Bom dia, $1."
echo "Hoje é $2."
```

O *script* pode ser invocado da seguinte forma:

```
chmod u+x script.sh
./script.sh Pedro Quarta-Feira

# Exemplo de output:
Bom dia, Pedro.
Hoje é Quarta-Feira.
```

10.2 Variáveis

É possível usar variáveis em *scripts* e na própria linha de comandos. A definição e utilização de variáveis usa a seguinte notação:

```
NOMEVARIAVEL=26
echo "A idade do Pedro é $NOMEVARIAVEL."
```

Note que a definição da variável não usa o sinal de `$`, mas a sua invocação (para obter o valor) sim.

10.3 Controlo do Fluxo

É possível controlar o fluxo de um *script* com instruções de `if... then... else... fi`, ciclos `while...done` e `for...done`. O trecho seguinte ilustra precisamente essa possibilidade para o ciclo `for...done`:

```
#!/bin/bash
for VAR in 1 2 3 4 5
do
  echo $VAR
done
```

Note que a execução do *script* acima referido daria origem a um *output* semelhante ao seguinte:

```
chmod u+x script.sh
./script.sh

# Exemplo de output:
1
2
3
4
5
```

Note ainda que, em *bash*, os ciclos terminam com a *keyword* `done`.

10.4 Ler Valores de Entrada

Finalmente, talvez seja útil referir que se podem pedir e guardar valores ao utilizador, guardando-os em variáveis através do comando `read NOMEVAR`. O trecho seguinte ilustra essa possibilidade:

```
#!/bin/bash

echo -n "Introduza um número entre 1 e 200: "
read VAR

if [[ $VAR -gt 100 ]]
then
  echo "O número é superior a 100."
else
  echo "O número é inferior a 100."
fi
```

Note que a execução do *script* acima referido daria origem a um *output* semelhante ao seguinte:

```
chmod u+x script.sh
./script.sh

# Exemplo de output:
Introduza um número entre 1 e 200: 120
O numero é superior a 100.
```

Deve ser enfatizado que, em *bash*, uma construção `if` termina com a palavra `fi`.

4 Boas Práticas de Programação, Ferramenta Make e Makefile

Sumário

Boas práticas de programação: automatização do processo de construção de programas. Ferramenta `Make` e `Makefile`.

Summary

Good programming practices: automation of the program construction process. `Make` *tool and* `Makefile`*.*

1 Introdução e Motivação

Embora a maior parte dos projetos de programação desenvolvidos numa fase introdutória às linguagens de programação é relativamente simples, visando poucas funcionalidades e resolvíveis em relativamente poucas linhas de código, implementadas frequentemente em um só ficheiro, **os projetos de software dos tempos modernos incorporam tipicamente milhares ou milhões de linhas de código, espalhadas por muitos ficheiros**. Nesses casos, o processo de compilação do código, gestão dos ficheiros suplementares de auxílio à compilação e teste pode tornar-se um processo oneroso que interessa tornar célere através de automatismos. É neste contexto que um sistema de automação de compilação como o `Make` se torna útil.

2 Processo Geral de Desenvolvimento de um Programa

Antes de prosseguir, interessa recordar o procedimento generalizado de implementação de uma peça de *software*. O processo típico em **linguagens compiladas** incorpora os seguintes passos:

1. Implementação do código;

2. Compilação e interligação do código (**este passo incorpora muitos outros sub-passos**). Caso se detetem problemas neste passo, voltar ao passo 1;

3. Execução e teste do código. Caso se detetem problemas neste passo, voltar ao passo 1.

Note-se que o processo típico em **linguagens interpretadas** é mais simples que o anterior:

1. Implementação do código;

2. Interpretação do código (execução e teste). Caso se detetem problemas neste passo, voltar ao passo 1.

Utilizando os comandos introduzidos no capítulo 3, podemos concretizar rapidamente os dois procedimentos de compilação referidos. Assim, para linguagens compiladas:

1. Implementação do código;

```
nano main.c
```

2. Compilação e interligação do código;

```
cc -o main.exe main.c
```

3. Execução e teste do código.

```
./main.exe
```

Para linguagens interpretadas:

1. Implementação do código;

```
nano main.py
```

2. Interpretação do código (execução e teste).

```
python main.py
```

3 Exemplo Explicativo de Desenvolvimento de um Programa

De forma a prover a explicação com um exemplo de um projeto que usa vários ficheiros, considere que se estava a implementar um (ainda assim) simples programa `Hello World!`, em linguagem C, que usava funções e um ficheiro de cabeçalho, resultando num total de três ficheiros:

- O ficheiro principal `main.c`:

```c
#include "helloworld.h"

int main(){
  helloworld();
  return 0;
}
```

- O ficheiro `helloworld.h`:

```c
void helloworld();
```

- O ficheiro `helloworld.c`:

```c
#include<stdio.h>

void helloworld(){
  printf("Hello World!\n");
}
```

Foque novamente os trechos de código e repare nos seguintes detalhes:

1. O ficheiro `main.c` contém um `include` para um ficheiro de cabeçalho local. Por isso é que usa aspas em vez de < e >;
2. O ficheiro `main.c` faz uso de uma função que não está definida no próprio ficheiro (mas sim em `helloworld.c`);
3. O ficheiro `helloworld.h` apenas contém a declaração da função `helloworld()`;
4. Finalmente, é no ficheiro `helloworld.c` que está a instrução de `printf()` que imprime a mensagem.

A compilação e *linkagem* do projeto, bem como a sua execução, pode ser feita com os dois comandos em baixo:

```
# O seguinte comando compila e linka todos os ficheiros do projeto
cc -o main.exe main.c helloworld.c

# O seguinte comando executa o programa resultante
./main.exe
```

Note-se que se podem compilar primeiro os ficheiros individualmente e só depois fazer a sua ligação (*linkagem*), conforme se ilustra na sequência de comandos seguinte (atenção à utilização da opção `-c` do `cc`).

```
# Os dois comandos seguintes compilam cada
# um dos ficheiros fonte para ficheiros .o
cc -c main.c
cc -c helloworld.c

# O comando seguinte faz a linkagem e
# produz o ficheiro executável main.exe
cc -o main.exe main.o helloworld.o

# O seguinte comando executa o programa resultante
./main.exe

# O comando seguinte limpa todos
# os ficheiros .o criados entretanto
rm *.o
```

Chama-se a atenção para o **último comando, que limpa os ficheiros temporários criados entretanto**. Note que o processo apresentado acarreta várias tarefas que podem rapidamente tornar-se onerosas:

1. Compilar;
2. *Linkar*;
3. Limpar ficheiros temporários; e
4. Executar.

Neste sentido, torna-se vantajoso (principalmente para projetos de maior dimensão), fazer uso de uma ferramenta que agilize o processo de desenvolvimento de um programa. A secção seguinte aborda, exatamente, esse tema.

4 O Sistema de Automação Make

O `Make` é uma ferramenta de engenharia de *software* que controla a geração de executáveis e outros ficheiros associados a partir dos ficheiros fonte de um programa. É tão mais útil

quanto maior for o número de componentes que contribuem para um programa, e permite diminuir a probabilidade de erro humano no processo de compilação, *e.g.*, gralhas nos nomes de ficheiros ou esquecimento de especificação de componentes a compilar, executar ou limpar. A ferramenta vem muitas vezes pré-instalada em sistemas Unix ou Unix-*like* ou pode ser facilmente instalada.

As instruções específicas de como construir um programa são descritas num ficheiro tipicamente chamado `Makefile`, que lista os ficheiros e como estes devem ser processados. Este `Makefile` deve fazer parte do projeto de *software*.

Note que o ficheiro **pode ter qualquer nome**. Contudo, se tiver um nome diferente de `Makefile` ou `makefile`, então a ferramenta `Make` terá de ser invocada com um comando semelhante a

```
make -f nome-do-ficheiro
```

Caso o nome do ficheiro descritor seja `Makefile` ou `makefile`, então a ferramenta pode ser simplesmente invocada com

```
make
```

A ferramenta procura automaticamente o ficheiro dentro da diretoria atual.

4.1 Exemplo Simples de um Ficheiro Makefile

Como não podia deixar de ser, o `Makefile` é um ficheiro de texto, podendo assim ser facilmente editado, mesmo a partir de um ambiente de linha de comandos. A seguir inclui-se uma possível concretização deste **ficheiro descritor** que está em linha com o exemplo já incluído anteriormente:

```
main.exe : main.o helloworld.o
    cc -o main.exe main.o helloworld.o

helloworld.o : helloworld.c
    cc -c helloworld.c

main.o : main.c helloworld.h
    cc -c main.c

clean :
  rm *.o
```

Aparte as explicações que serão dadas em baixo, pode ser imediatamente dito que, se o ficheiro `Makefile` com o conteúdo anterior estiver presente na diretoria com o código fonte do projeto, então este pode ser compilado e construído com a emissão do comando

```
make main.exe
```

ou, simplesmente,

```
make
```

Por outro lado, a limpeza dos ficheiros auxiliares com extensão `.o` pode ser conseguida com um comando semelhante a

```
make clean
```

Se, por ventura, fosse necessário compilar apenas o ficheiro `helloworld.c` para o ficheiro `helloworld.o`, poder-se-ia executar o `Make` com esse objetivo, *i.e.*,

```
make helloworld.o
```

4.2 Notação e Sintaxe

Um ficheiro descritor é sobretudo composto por *regras* cuja sintaxe é aqui descrita. Também é possível definir variáveis e utilizar comentários nestes ficheiros, bem como outros recursos interessantes no contexto da compilação de programas. Note-se contudo, que optou-se por se endereçar o tema da ferramenta `Make` na sequência da introdução à linha de comandos por esta ter um modo de operação semelhante a um *script* em alguns aspetos, nomeadamente por incorporar comandos. Em baixo, à medida que se forem enunciando vários conceitos, vai-se adornando a explicação com uma melhoria do exemplo anterior sempre que tal se demonstrar útil.

Comentários

É possível colocar texto num `Makefile` que não é de forma alguma interpretado, bastando para isso preceder essa linha com `#`. O trecho de código seguinte exemplifica o uso de comentários:

```
# o phony target seguinte limpa todos os ficheiros auxiliares (*.o)
clean :
  rm *.o
```

Sintaxe das Regras

O ficheiro descritor é sobretudo composto por regras.

Cada regra tem **um objetivo** (também designado por alvo ou *target*), define **as dependências** para a execução desse objetivo e **os comandos** para o atingir. A sua sintaxe é a seguinte:

```
target : dependencies
    (tab) command1
    (tab) command2
    ...
    (tab) commandn
```

Note-se que, no trecho de código anterior, o nome do objetivo e as dependências estão **obrigatoriamente separadas por** `:` e que **os comandos começam na segunda linha da regra**, precedidos **obrigatoriamente por um** `tab`.

O exemplo incluído no início da secção 4.1 contém várias regras definidas usando a sintaxe referida e que podem agora ser analisadas com mais detalhe:

1. A primeira regra tem como **objetivo** produzir o ficheiro `main.exe`, mas **depende** dos ficheiros `main.o` e `helloworld.o`. O **comando** para construir o ficheiro é `cc -o main.exe main.o helloworld.o`. Note que o nome do objetivo coincide com o nome do ficheiro a produzir;
2. A segunda e a terceira regras têm como objetivos produzir, precisamente, as duas dependências em falta para a primeira regra. Assim, a segunda regra tem como **objetivo** produzir o ficheiro `helloworld.o`, tendo como **dependência** `helloworld.c` e usando o **comando** `cc -c helloworld.c`, enquanto a terceira regra tem como **objetivo** construir `main.o`, a partir das **dependências** `main.c` e `helloworld.h`, com o **comando** `cc -c main.c`.

Funcionamento

O funcionamento do `Make` é bastante intuitivo. Quando se executa o comando `make` com um objetivo definido, este lê a respetiva regra e verifica as dependências (que são ficheiros ou outros objetivos). **Se** as dependências já estiverem satisfeitas, avança para o comando; **caso contário**, procura nas outras regras a forma de satisfazer as dependências que ainda faltam cumprir.

Note-se que a forma de funcionamento do `Make` permite, em algumas situações, poupar a compilação de partes do projeto, porque ele reconhece automaticamente casos em que ficheiros já foram antes compilados e que não sofreram alterações entretanto.

No exemplo incluído nestes apontamentos, torna-se assim claro que, ao invocarmos o `Make` pela primeira vez, este tenta executar a regra `main.exe` mas, como as dependências não estão garantidas, **vai primeiro procurar e executar os comandos que geram essas dependências**. O `Make` pode fazer isso recursivamente.

Tipos de Regras

Conforme já implicitamente referido em cima, existem dois tipos de *objetivos* em ficheiros descritores:

- Aqueles em que um ficheiro é produzido, simplesmente conhecidos como **objetivos**. Tipicamente, o nome do objetivo é o nome do ficheiro gerado. No exemplo anterior, o objetivo `main.exe` concretiza um destes casos;

- E os objetivos em que não é produzido nenhum ficheiro, conhecidos como ***objetivos falsos***, ***objetivos nulos*** ou, em inglês, ***phony targets***. No exemplo anterior, `clean` concretiza um desses objetivos.

Constitui de resto uma boa prática identificar exatamente quais são os *phony targets* num `Makefile` através de uma entrada começada por `.PHONY` no ficheiro descritor, como se exemplifica a seguir:

```
.PHONY: clean

main.exe : main.o helloworld.o
    cc -o main.exe main.o helloworld.o

helloworld.o : helloworld.c
    cc -c helloworld.c

main.o : main.c helloworld.h
    cc -c main.c

# o phony target seguinte limpa todos os ficheiros auxiliares (*.o)
clean :
    rm *.o
```

Deve ainda ser mencionado que, caso o comando `Make` seja emitido sem nenhum argumento, então este **executa sempre a primeira regra no ficheiro descritor**. Contudo, é boa prática definir um *phony target* chamado `all`, que passa a ser o predefinido pelo `Make`, o que permite controlar o que acontece se alguém executar o comando sem argumentos. Muitas vezes define-se esse objetivo como um ponteiro para o que queremos correr primeiro:

```
.PHONY: all clean

all : main.exe

main.exe : main.o helloworld.o
    cc -o main.exe main.o helloworld.o

helloworld.o : helloworld.c
    cc -c helloworld.c

main.o : main.c helloworld.h
    cc -c main.c

# o phony target seguinte limpa todos os ficheiros auxiliares (*.o)
clean :
    rm *.o
```

Variáveis

É possível definir variáveis nos ficheiros descritores seguindo a sintaxe seguinte:

```
nomevariavel = string
```

O valor da variável pode depois ser obtido usando a seguinte sintaxe:

```
$(nomevariavel)
```

Note-se que a definição da variável é semelhante à operação análoga em *bash*, aparte os espaços (que não devem existir em *bash*). A referência à variável também é semelhante, embora em *bash* não se usem os parêntesis.

Uma utilidade bastante simples e direta das variáveis está relacionada com a especificação de comandos, nomeadamente do compilador. Imagine a situação em que pretendia mudar de compilador (*e.g.*, trocar o `cc` por outro). No exemplo apresentado, teria de mudar todas as regras que usam aquele comando. Contudo, se o nome do comando for uma variável, a alteração seria simples. A redação do `Makefile` em baixo ilustra como usar uma variável como nome do comando:

```
# a seguinte linha define a variável compiler como tendo a string cc
compiler = gcc

.PHONY: all clean

all : main.exe

main.exe : main.o helloworld.o
    $(compiler) -o main.exe main.o helloworld.o

helloworld.o : helloworld.c
    $(compiler) -c helloworld.c

main.o : main.c helloworld.h
    $(compiler) -c main.c

# o phony target seguinte limpa todos os ficheiros auxiliares (*.o)
clean :
    rm *.o
```

Variáveis Automáticas e Vetores

Embora existam ainda aspetos avançado do `Make` não discutidos aqui, importa referir duas variáveis automáticas (por vezes designadas por macros) e um *wildcard* que se revelam úteis em muitos cenários:

- a variável `@` é igual ao nome do objetivo em cada regra;
- a variável `<` é igual ao nome do primeiro elemento na lista de dependências;
- o *wildcard* `%`, que significa qualquer sequência de caracteres.

Assim, o ficheiro descritor usado como exemplo nesta secção pode ser simplificado para a seguinte forma:

```
# a seguinte linha define a variável compiler como tendo a string cc
compiler = cc

.PHONY: all clean

all : main.exe

main.exe : main.o helloworld.o
    $(compiler) -o $@ main.o helloworld.o

%.o : %.c
    $(compiler) -c $<

# o phony target seguinte limpa todos os ficheiros auxiliares (*.o)
clean :
    rm *.o
```

É ainda possível definir vetores (de *strings*) usando uma sintaxe semelhante à seguinte

```
SOURCEFILES = main.c helloworld.c
```

ou ainda, dando mais responsabilidades ao `Make`, da seguinte maneira

```
SOURCEFILES = $(wildcard *.c)
```

No caso referido em último, a ferramenta `Make` procura todos os ficheiros terminados em `.c` na diretoria do ficheiro descritor, construindo um vetor com o nome dos ficheiros que encontrar.

Para finalizar, deve ainda ser referido que é comum colocar alguma forma de retorno em objetivos que não criem ficheiros ou output (*e.g.*, o objetivo `clean`). Pode-se usar o comando `echo` para o efeito.

```
...
# o phony target seguinte limpa todos os ficheiros auxiliares (*.o)
clean :
    rm *.o
    @echo "Ficheiros removidos."
```

Note-se que, no excerto anterior, o comando `echo` é precedido de `@`. Tal é feito para **evitar que o próprio comando seja impresso no ecrã**. Desta forma, **apenas o resultado do comando é que aparece**.

5 Ambientes de Desenvolvimento Integrados e Depuração

Sumário
Ambientes e ferramentas de desenvolvimento integrado. Depuração e perfilagem de programas.

Summary
Integrated development environments and tools. Program debugging and profiling.

1 Introdução

Um Ambiente de Desenvolvimento Integrado (da sua designação inglesa, pela qual é mais conhecido, **IDE**) é uma **aplicação informática** que reúne, numa ou em várias perspetivas integradas e de (mais) fácil acesso e utilização, várias ferramentas de apoio à implementação, preparação e teste de *software*. Estes ambientes de desenvolvimento integram frequentemente, e no mínimo, **um editor de código** (muito semelhante a editor de texto), **ferramentas de automação** de preparação do *software* e um **depurador**.

2 Funcionalidades de Ambientes de Desenvolvimento

Entre outras, os IDEs podem integrar o seguinte conjunto de funcionalidades e ferramentas:

1. Um **editor de código** para o código fonte das linguagens de programação suportadas. Normalmente integra também um conjunto de sub-funcionalidades relacionadas com o editor, nomeadamente a funcionalidade de auto-completar (quando

começamos a escrever uma *keyword* da linguagem, este termina-a ou escreve-a toda) e a funcionalidade descrita no ponto seguinte;

2. **Geração automática de ficheiros e de código** dentro desses ficheiros nas linguagens suportadas. Por exemplo, quando se começa um novo projeto na linguagem C, o IDE gera automaticamente um ficheiro chamado `main.c` e coloca lá um protótipo da função `main`. Esta funcionalidade pode ainda ser tornada mais robusta em combinação com o que é descrito no ponto seguinte;

3. Suporte à **modelação de dados e do *software***, na qual é possível especificar o *software* ou o modelo dos dados usando uma linguagem normalizada (*e.g.*, *Unified Modelling Language* (UML)) e depois pedir ao IDE para transformar as especificações em protótipos de funções e código;

4. Um **compilador** (que compila os vários recursos de código fonte) e um ***linker*** (que junta os vários recursos compilados) ou um **interpretador** (que interpreta o código em linguagens interpretadas) são ferramentas que **devem estar integradas ou ser acedidadas de forma integrada pelo IDE**. É, de resto, típica a situação em que estas ferramentas são acedidas através de um botão *Play* no IDE;

5. **Um conjunto de recursos para depuração e perfilagem**, que inclui **integração com ferramentas** de depuração (*e.g.*, o `gdb`) e **perspetivas próprias** para visualização do *output* produzido por essas ferramentas;

6. **Uma perspetiva sobre um ambiente de linha de comandos (uma consola ou terminal),** na qual é possível ver os comandos que o IDE executa automaticamente ou inclusive interagir com os ficheiros e fornecer entradas a um programa em execução;

7. **Automatismos ligados à distribuição do *software* produzido,** nomeadamente assistentes e *scripts* de configuração que permitem ligar o IDE a lojas de *software online* para publicação de aplicações;

8. **Automação de testes,** permitindo a especificação de passos de testes (repetitivos ou não) ao *software*, que podem ser aplicados automaticamente, produzindo relatórios com resultados desses testes;

9. Suporte à **refatoração**, que consiste num conjunto de recursos que permite a melhoria consistente do código fonte, nomeadamente através da mudança uniforme do nome de variáveis, funções ou métodos, otimização (e limpeza) de código;

10. **Organização do trabalho de desenvolvimento em projetos**, sendo que o código fonte continua em ficheiros com extensão típica, mas permitindo adicionar meta-informação relativa a configurações, entradas, testes, entre outros;

11. Integração de **recursos de controlo de versões** que permitem, por exemplo, enviar ou obter o código de repositórios centrais, manter um registo de alterações do código, manter e navegar por várias versões e bifurcações do código;

12. Integração de **perspetivas de visualização de registos (*logs*) de execução** dos programas.

Muitos IDEs atuais recorrem a um **sistema de *plugins*** por uma questão de versatilidade, **permitindo que o(a) programador(a) os adapte às suas necessidades**, partindo de um conjunto de funcionalidades base. Por exemplo, muitos dos IDEs da atualidade são especialmente vocacionados para uma determinada linguagem de programação, mas podem suportar outras através da instalação de *plugins*.

Nas subsecções não numeradas seguintes apresentam-se os IDEs atualmente mais usados, capacitados das funcionalidades anteriormente descritas.

Netbeans

O Netbeans é um IDE bastante conhecido atualmente, provavelmente por ser de código aberto e suportar algumas das linguagens de programação mais populares dos últimos anos, nomeadamente Java, JavaScript, *Hypertext Markup Language* (HTML) 5, PHP, C/C++ e Python, embora a sua linguagem base nativa inicial fosse Java (através de *plugins*). Integra a grande maioria (senão todas, através de *plugins*) das funcionalidades descritas em cima. É um IDE gratuito, disponível para várias plataformas (Linux, Windows e macOS), cuja interface é mostrada na figura 5.1.

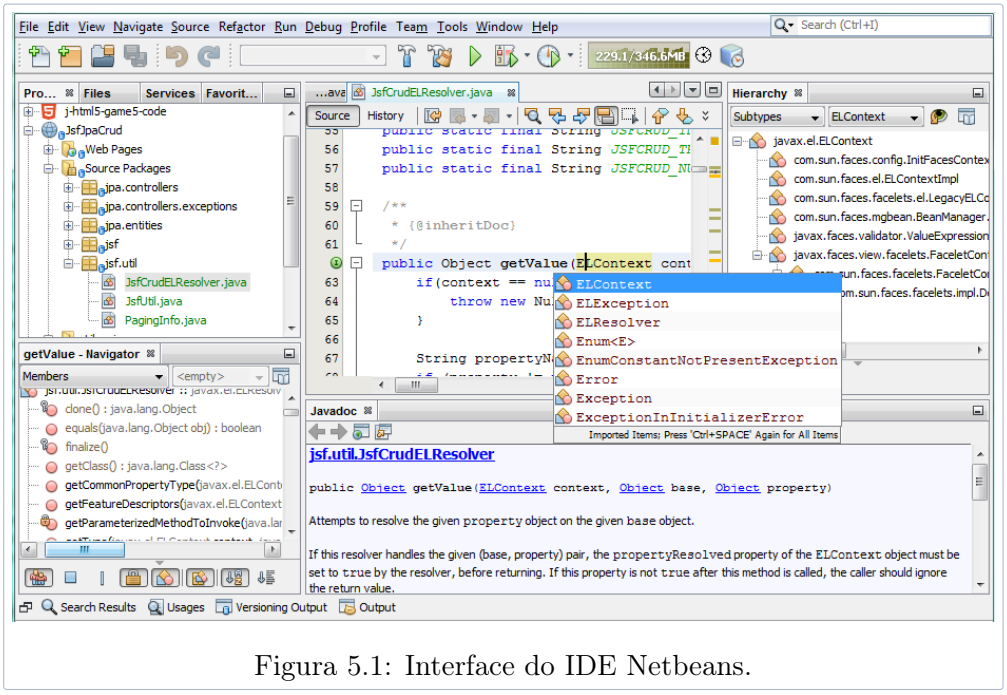

Figura 5.1: Interface do IDE Netbeans.

Visual Studio Code

O Visual Studio Code é definido como sendo um editor de código gratuito desenvolvido pela Microsoft. É também multi-plataforma, estando disponível para Windows, Linux e macOS. Apesar de não ser descrito como um verdadeiro IDE, integra muitas das funcionalidades acima descritas, nomeadamente **enfatização de sintaxe, suporte para depuração, auto-completar contextual de código, refatoração de código e gestão de versões** (com Git). A interface desta ferramenta é exibida na figura 5.2.

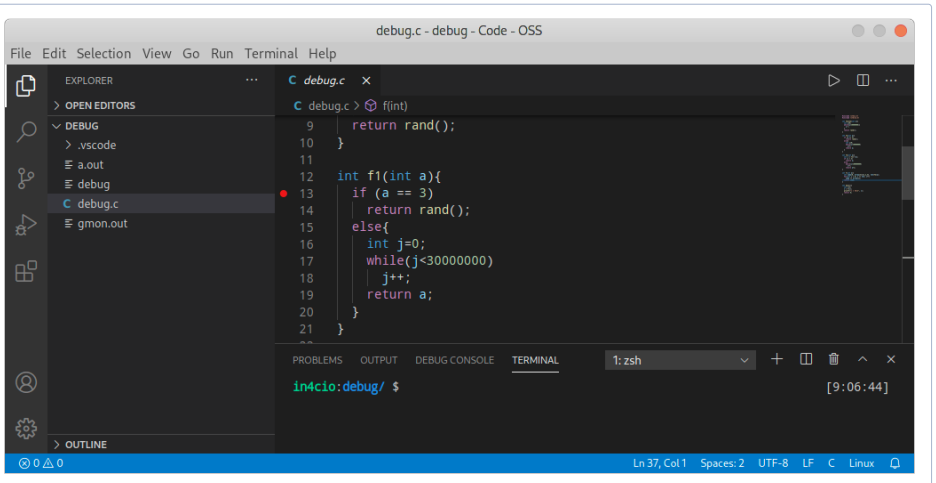

Figura 5.2: Interface do IDE Visual Studio Code.

Code::Blocks

O Code::Blocks é um IDE gratuito e de código aberto para várias plataformas (Linux, Windows e macOS), especialmente vocacionado para as linguagens de programação C, C++ e Fortran, nomeadamente para depuração. Como em muitos outros IDEs, é possível estender o seu conjunto de funcionalidades. É aqui referido por se integrar especialmente bem com ferramentas do ecossistema Linux.

Android Studio

O Android Studio constitui um bom exemplo de IDE por mostrar também que as **ferramentas de preparação e distribuição podem incluir ambientes virtualizados de teste** do *software* desenvolvido, nomeadamente de dispositivos virtuais Android em que se simulam *smartphones* ou *tablets*.

3 Depuração e Perfilagem de Programas

Quando se desenvolvem programas, podem ser **introduzidos ou existir erros nas várias fases de desenvolvimento e execução**; e os erros podem ser de **tipos diferentes**,

nomeadamente **erros de sintaxe** ou **erros de execução**. De forma análoga, alguns erros podem ser detetados pelo próprio compilador ou interpretador (erros de sintaxe), ou apenas detetados durante a execução, quer por análise dos resultados do programa, quer por erros que fazem o programa parar ou entrar em estados de inconsistência. É neste contexto que a depuração e perfilagem de programas se revela útil.

3.1 Erros de Sintaxe

Os erros de sintaxe são **erros bastante comuns** por se deverem ou à inexperiência (e desconhecimento) do programador, ou a gralhas típicas de humanos. O excerto de código seguinte contém um exemplo de um erro de sintaxe na linha 4. Neste caso, falta um ponto-e-vírgula a terminar a instrução `printf()`, que **será detetado pelo próprio compilador** (*e.g.*, pelo `gcc`), porque se **deve a um desvio a uma regra normalizada da linguagem**.

```
#include<stdio.h>

int main(){
  printf("Hello world!\n")
}
```

De facto, ao compilar, obtém-se o *output* seguinte:

```
debug0.c: In function ‘main’:
debug0.c:4:27: error: expected ‘;’ before ‘}’ token
    4 |   printf("Hello world!\n")
      |                           ^
      |                           ;
    5 | }
      | ~
```

Note que, neste caso, o *output* do compilador faz um esforço para dizer inclusive a linha onde o erro ocorre (linha 4) e indicar o local onde o `;` deve ser colocado.

3.2 Erros de Execução - Regras de Negócio

Alguns erros devem-se à **má concretização de regras de negócio na programação**. Dado estas regras não serem normalizadas nem passíveis de serem colocadas no compilador ou no interpretador, não podem ser por eles verificadas. O excerto de código seguinte procura exemplificar um destes erros, embora apenas de forma pedagógica, por não concretizar nenhuma funcionalidade ou cristalizar nenhum requisito específico.

Considere que implementava uma funcionalidade que verificava se determinado contador havia conseguido chegar ao fim de uma contagem, percorrendo todas iterações possíveis

até 10, imprimindo `Sucesso!` no caso disso acontecer. Como sabe que o código no ciclo `for` apenas chega a nove **dentro do ciclo**, o(a) programador(a) coloca o `if` a verificar isso mesmo e a imprimir `Sucesso!` nesse caso. Contudo, ao executar o código, verifica que este imprime sempre `Falhanco!`.

```
#include<stdio.h>

int main(){
  int i;
  for( i = 0; i < 10; i++ )
    ;
  if( i == 9)
    printf("Sucesso!\n");
  else
    printf("Falhanco!\n");
}
```

Depuração Baseada na Impressão dos Valores das Variáveis

O tipo de erro referido na secção anterior fica óbvio quando se executa o programa e, neste caso, a **técnica de depuração que consiste em imprimir o valor das variáveis** ao longo da execução é suficiente para detetar e resolver o problema em questão. O excerto de código seguinte ilustra as providências que o(a) programador(a) tomou para tentar perceber o problema.

```
#include<stdio.h>

int main(){
  int i;
  for( i = 0; i < 10; i++ )
    ;

  // A instrução seguinte foi adicionada só para efeitos
  // de depuração e deve ser retirada do programa adiante.
  printf("%d\n",i);

  if( i == 9)
    printf("Sucesso!\n");
  else
    printf("Falhanco!\n");
}
```

Note que a introdução e utilização de instruções para imprimir o valor de variáveis ao longo do código para efeitos de depuração **deve apenas acontecer numa fase de testes** (na verdade, passe a redundância, apenas na fase de depuração). Isto significa que o(a) programador(a) deve **manualmente retirar estas instruções** (no máximo, imediatamente) **antes da construção (*build*) da versão final** do *software*.

Esta técnica de depuração **não requer o uso de ferramentas adicionais** para além do próprio compilador e interpretador já usados no desenvolvimento e apresenta-se como sendo **uma das mais simples**.

3.3 Erros de Execução - Erros de Implementação

Os erros de implementação **acontecem quando o(a) programador(a) desconhece algumas regras de sintaxe ou funcionamento da linguagem** (*e.g.*, que os índices em *arrays* começam em zero e terminam em $n-1$, em que n é o tamanho do *array*) ou quando, mesmo conhecendo essas regras, **falham em cumpri-las ou fazê-las cumprir** (*e.g.*, via validação de dados de entrada). A execução (depois de devidamente compilado) do código no excerto de código seguinte daria um erro ou incorreria numa situação inusitada.

```
#include<stdio.h>

int main(){
  int array[3]={1,2,3};
  int i;
  for(i = 0; i <= 3; i++ ){
    printf("%d\n",array[i]);
  }
}
```

O *output* resultante da execução do código apresentado antes seria semelhante a:

```
1
2
3
-1002724352
```

Repare que o número que aparece em último parece não fazer sentido no contexto dado pela implementação e deve-se ao facto de estarmos a mostrar o valor do `array[3]`, sendo que os índices do *array* só vão até $3-1=2$. Note também que **o compilador não deteta o problema, porque todas as regras sintáticas estão a ser cumpridas**.

Depuração Baseada em Pontos de Rutura

A depuração baseada em pontos de rutura (da expressão inglesa *break points*) é especialmente útil em casos semelhantes ao representado na secção anterior ou em casos em que o programa implementado tem uma grande pegada. Esta técnica **consiste em indicar ao processo de preparação da aplicação que injete pontos de paragem no programa e que mostre o valor das variáveis ou outros artefactos de execução nesses pontos.** Esta injeção é **muito frequentemente feita com a ajuda de IDEs** porque fazê-lo manualmente é moroso e desproporcionalmente difícil para o(a) programador(a) de determinada linguagem.

Pela razão referida em último, a depuração de um programa com base em pontos de rutura é **normalmente conseguida com a ajuda de um IDE, que permite a introdução desses pontos de uma forma gráfica**, como de resto se exemplifica na figura 5.3.

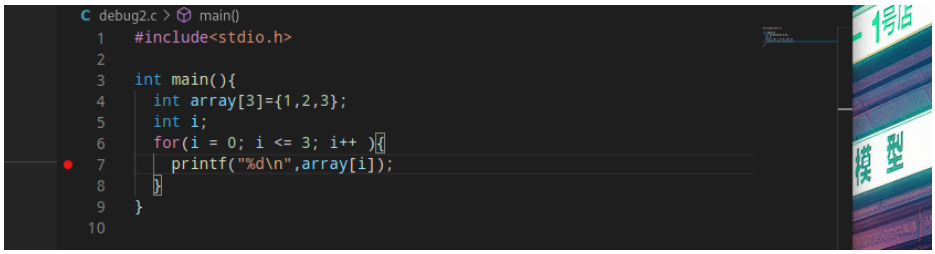

Figura 5.3: Colocação de *break points* em Visual Studio Code.

O ponto de rutura é colocado clicando à esquerda da linha em que se pretende parar o programa para análise durante a execução (linha 7, neste caso), ficando assinalado com um ponto vermelho.

A figura 5.4 mostra a perspetiva oferecida pelo Visual Studio Code aquando da depuração do programa com pontos de rutura. Entre outros mostra-se que é possível **navegar nas instruções através de controlos colocados no topo da janela** e que são apresentadas perspetivas (do lado esquerdo) que mostram os valores das variáveis (neste caso, a variável `i:3` e `array[3]`), bem como as funções executadas (que neste caso é a `main()`). É possível saltar entre instruções, avançar para outro ponto de rutura, ou terminar a depuração usando os controlos no topo da janela.

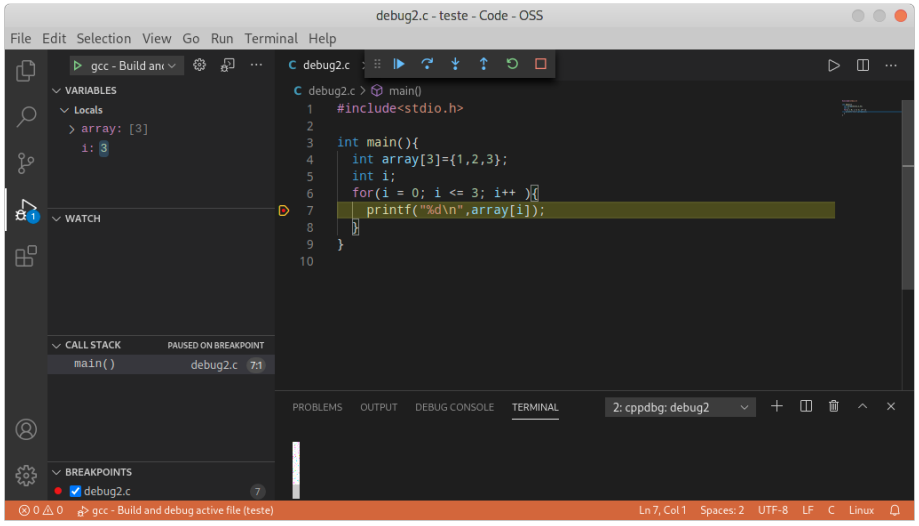

Figura 5.4: Captura de ecrã do IDE Visual Studio Code na depuração de um programa.

No caso apresentado, pode ser concluído que o problema do trecho de código apresentado tem a ver com o facto da variável `i` tomar o valor `3`.

Note que o uso **deste tipo de depuração não costuma introduzir artefactos adi-**

cionais ao código gerado para produção, pelo que **não requer** (contrariamente à *Depuração Baseada na Impressão dos Valores das Variáveis*) **a remoção manual desses artefactos** aquando da preparação (*build*) da versão final do *software*. No entanto, esta técnica de depuração **requer o uso de ferramentas adicionais** ao compilador ou interpretador (ou requer que estes já suportem depuração), e apresenta-se um pouco mais complexa que a referida anteriormente. Um exemplo de uma dessas ferramentas é o `gdb` (**G**NU ***Deb**ugger*).

3.4 Perfilagem

A perfilagem (da expressão inglesa *profiling*) de um programa é **uma análise dinâmica do programa que permite avaliar** a percentagem de tempo total utilizado por cada função, que funções foram chamadas, qual a árvore de chamada das funções, qual a memória ocupada por funções ou variáveis, quais os recursos de rede utilizados e por que partes do programa, aplicação ou sistema, entre outros.

Para obter uma ideia do que um *profiler* pode fazer, atente no trecho de código seguinte:

```
#include<stdio.h>

void f2(){
  int j;
  int k;
  for( j = 0; j < 100000; j++ )
    for( k = 0; k < j ; k++ )
      ;
}

void f1(){
  int i;
  for( i = 0; i < 1000000000; i++ )
    ;
  f2();
}

int main(){
  f1();
  printf("Fim!\n");
  return 0;
}
```

Como se pode ver no código anterior, a função `main()` invoca a função `f1()`. Por sua vez, a função `f1()` tem um ciclo `for` que faz primeiro uma contagem até 1.000.000.000 e depois invoca `f2()`. A função `f2()` faz várias contagens (tem dois ciclos `for` encadeados), mas o ciclo principal parece ir apenas até 100.000.

Se quisermos ter uma ideia do tempo e recursos utilizados por cada uma das funções, podemos usar uma ferramenta de perfilagem, como o `gprof`. Para isso, **compilamos**

o programa com a *flag* `-pg` e executamo-lo pelo menos uma vez. Esta opção força a criação de dados de perfilagem em programas (associando dados de perfil a cada rotina/função invocada num programa), que depois pode ser lida pela ferramenta `gprof` para obter um relatório. As três instruções seguintes serviriam para compilar, executar e gerar esse relatório, guardando-o em `profile_output.txt`:

```
gcc -pg debug3.c
./a.out
gprof a.out gmon.out > profile_output.txt
```

A consulta do ficheiro com o relatório (usando, por exemplo, `cat profile_output.txt`) revela que, para o exemplo estudado, a função `f2` ocupa cerca de 89% do tempo de execução:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 89.49      9.09     9.09        1     9.09     9.09  f2
 12.33     10.34     1.25        1     1.25    10.34  f1
```

É importante referir que o relatório contém outros detalhes não discutidos aqui. Note também que o uso **deste tipo de perfilagem não costuma introduzir artefactos adicionais ao código gerado para produção**, mas deve ter-se em atenção que a preparação do *software* para produção não deve ser compilado da mesma forma que para perfilagem! No exemplo dado nesta secção, o código era compilado com `-pg` para permitir a perfilagem. **Essa opção não deve ser usada para gerar o código para produção.**

6 Sistemas de Gestão e Controlo de Versões

Sumário
Controlo de versões de *software* com caso de estudo do sistema Git.

Summary
Software version control with Git study case.

1 Introdução

Os **sistemas de gestão de versões** são outra das **ferramentas essenciais ao desenvolvimento de *software* na era da informação**. Salvo exceções muito pontuais (*e.g.*, exemplos de código para efeitos pedagógicos), todos os projetos de *software* beneficiam de uma ou mais funcionalidades ou características que estes sistemas oferecem.

Definição

Pode definir-se um **sistema de gestão de versões** (por vezes também conhecido como **sistema de gestão ou controlo de revisões** ou **sistema de controlo do código fonte**) como **uma classe de sistemas de *software* que ajuda a gerir alterações a programas de computador**, documentos, sites de Internet ou, de uma maneira geral, a qualquer coleção com sentido de informação.

É muito comum identificarem-se as alterações através de um número (número de revisão) ou letra (nível da revisão) associada à versão. Por exemplo, o primeiro conjunto de ficheiros de implementação de um sistema de *software* é tipicamente nomeado como a *revisão 1*, sendo que, sempre que são feitas um conjunto de alterações até um ponto de consolidação, se incrementa esse número de versão, dando origem à *revisão 2*, *revisão 3*, e assim sucessivamente. Idealmente, **cada revisão deve estar associada com um selo**

temporal e à informação da pessoa que fez a alteração. Podem ser feitas operações sobre as revisões, nomeadamente comparações entre revisões, restauro de determinada revisão, fusão de revisões, entre outros.

Exemplos

Existem vários exemplos de sistemas de gestão e controlo de versões, muitos deles partilhando inclusive do conjunto de comandos ou sintaxe de utilização. Alguns desses exemplos são:

- O Git, que é gratuito e de código aberto, e que pode ser usado para lidar de forma rápida e eficiente com projetos pequenos ou grandes. É muito utilizado no ecossistema Linux, tendo sido desenvolvido pelo criador desse sistema operativo. A sua pegada de código é pequena e otimizada, e suporta ramificação local e múltiplos fluxos de trabalho. O Git será alvo de maior discussão ao longo deste capítulo;
- O Mercurial, é outro sistema de gestão de código descrito como sendo muito eficiente para projetos de diferentes envergaduras e que disponibiliza uma interface simples e intuitiva. Na verdade, os comandos deste *software* são muito semelhantes aos do Git. Está também disponível gratuitamente e na forma de código aberto;
- O Subversion, também conhecido por SVN, está também disponível em código aberto e é desenvolvido atualmente pela Apache Software Foundation, usufruindo da colaboração de uma grande comunidade. É muito utilizado na gestão de projetos de código aberto e corporativos.

2 Motivação

Uma das melhores formas de entender o funcionamento dos sistemas de gestão ou controlo de versões consiste em incorporar um exemplo de desenvolvimento de *software* e eventualmente refletir sobre alguns detalhes relativos à forma como se lidou com algumas necessidades durante o desenvolvimento de projetos no passado.

Para começar, considere que a Alice é uma programadora que está a trabalhar num projeto em linguagem de programação C para a unidade curricular de Laboratórios de Programação. O projeto tinha apenas um ficheiro chamado `program.c`. Constroem-se, de seguida, diversos cenários que demonstram a necessidade das funcionalidades oferecidas pelos sistemas de gestão ou controlo de versões.

Cenário 1 – Restaurar

No primeiro cenário, considere que a Alice trabalhou vários dias no seu projeto, tendo chegado a um estado bastante bom. Contudo, mesmo antes de entregar, notou que fez uma ou mais alterações que inviabilizaram o projeto. Como só tinha um ficheiro e não

se recordando bem do que havia mudado, teria algum trabalho para voltar a um estado consistente do seu projeto. Neste caso, **a necessidade específica era a de voltar a um estado consistente** (passado) do projeto. Uma das funcionalidades destes sistemas será, portanto, a de facilmente permitir voltar a estados anteriores do projeto.

Cenário 2 – Navegação entre Versões

Consideremos agora que a Alice já havia experimentado o problema referido no cenário 1. Desta feita, e para o evitar, a Alice começou a guardar várias versões do mesmo programa, nomeadamente com a nomenclatura: `program.1.c`, `program.2.c`, `program.3.c`, e assim sucessivamente.

Repare-se que a nomenclatura dá uma ideia do número de revisão. Por uma questão de consistência, e como já começa a ganhar alguma maturidade neste assunto, **a Alice mantém sempre um ponteiro para a versão mais recente do programa no ficheiro** `program.c`**.** A este ficheiro (ao que contém) o ponteiro para a versão mais recente, dá-se o nome de `CABEÇA`, mas mais conhecida pela sua designação em Língua Inglesa: `HEAD`. No contexto dos sistemas de gestão ou controlo de versões, `HEAD` **é o ponteiro que aponta para a versão em que atualmente se está a trabalhar** (pode não ser necessariamente a mais recente).

Neste cenário, se a Alice chegar a um ponto em que o programa que está a desenvolver tem um problema que o inviabiliza, e assumindo que ela foi gravando o progresso em momentos chave do desenvolvimento, então tem uma forma simples de voltar a um estado consistente. Para isso, basta mudar o ponteiro para a versão anterior pretendida. Note que a Alice não tem uma real necessidade de ver todos os ficheiros de todas as versões que guardou. A cada momento, apenas está interessada em ver o ficheiro em que está a trabalhar. Neste caso, **as necessidades específicas eram as de ter uma forma consistente de guardar várias versões, navegar entre elas e de só ver a versão em que está a trabalhar**. Algumas das funcionalidades destes sistemas serão, portanto, a de permitir navegar entre versões, numerar as versões e esconder as diferentes iterações do projeto por comodidade.

Cenário 3 – Comparação de Versões

O cenário 3 é semelhante ao anterior, mas adicionando o facto de que a Alice queria apenas reverter algumas das alterações feitas de uma versão funcional para uma versão não funcional. Neste caso, quando ela se dá conta de que estragou o programa, sente a necessidade de verificar quais as alterações específicas que foram feitas, para eventualmente poder incorporar algumas das que melhoram o projeto e evitar aquelas que o estragaram. Neste caso, **a necessidade específica era a de comparar versões** do projeto.

Naturalmente, existem ferramentas (como a `diff`) que podem ajudar nesta tarefa, mas será tanto melhor se essa funcionalidade for disponibilizada de uma forma integrada. Uma das funcionalidades destes sistemas será, portanto, a de facilmente permitir comparar

versões do projeto para diferentes ficheiros.

Cenário 4 – Cópia de Segurança

Considere agora que a Alice gostaria de guardar o seu trabalho, incluindo todo o histórico num ponto remoto (*i.e.*, fazer uma cópia de segurança), para o caso do seu computador se estragar. Neste caso, **a necessidade específica é a de permitir fazer cópias de todo o grafo de objetos** do projeto, incluindo todas as versões. Uma das funcionalidades destes sistemas é, frequentemente, a de facilmente permitir enviar o projeto para um outro sistema computacional, eventualmente remoto àquele em que se está a trabalhar (*e.g.*, na *cloud*). Este cenário podia agravar-se, se pensarmos que a Alice havia estado a trabalhar no seu projeto em casa, esquecendo-se de o guardar num repositório *online*, e depois precisava de lhe aceder quando estava na aula, a menos de 10 minutos da entrega do trabalho.

Cenário 5 – Trabalho em Vários Locais

Considere agora que a Alice já estava habituada a todos os percalços referidos antes, e que já guardava cópias no seu computador pessoal, na *cloud*, e até num computador do laboratório na universidade. Certo dia, a Alice trabalha no seu projeto em casa e carrega a nova versão para o repositório remoto antes de ir para a universidade. Quando chega ao laboratório, continua a trabalhar no projeto, mas **esquece-se de retirar a versão que está no repositório remoto antes de começar a trabalhar, acabando por ficar com duas versões diferentes que resultaram da mesma versão anterior**. A Alice fica a desejar que um qualquer sistema a ajudasse a juntar (mais conhecida pela sua designação em Língua Inglesa *merge*) as duas versões.

Neste caso, **a necessidade específica é a de existir uma funcionalidade que permita ajudar a juntar duas versões** do projeto. Uma das funcionalidades destes sistemas é, precisamente, a de facilmente permitir fazer *merge* de diferentes versões de ficheiros ou do projeto. É claro que constitui uma **boa prática** garantir sempre que se está a trabalhar na versão mais recente de um projeto. Assim, se estamos a usar várias cópias locais (em diferentes computadores) e um repositório central, **devemos sempre fazer o *push* da versão mais recente do *software* quando chegamos a um ponto de consolidação** (ou ao fim do trabalho), **e fazer o *pull* antes de começarmos a trabalhar** em novas alterações.

Cenário 6 – Ramificação

Satisfeita com o seu projeto, a Alice repara que gostava de experimentar algumas funcionalidades que não faziam parte dos objetivos iniciais. Para isso, faz várias novas cópias de todos os ficheiros do projeto (em diferentes diretorias) e desenvolve nelas essas novas funcionalidades em separado. Tendo corrido bem para duas delas, tenta integrá-las num só projeto, mas o número de alterações entretanto produzidas levam-na a perder demasiado tempo e não vão a tempo para a entrega do projeto na unidade curricular.

Neste caso, **as necessidades específicas são as de existir uma forma que permita criar ramos (*branches*) do projeto** para testar funcionalidades e outra que **permita facilmente fundir (*merge*) esses ramos no projeto principal**. Uma das funcionalidades destes sistemas é, precisamente, a de facilmente permitir criar e fundir *branches* do projeto, no sentido de ser um processo o mais transparente possível para o utilizador.

Cenário 7 – Trabalho de Grupo

Neste último cenário, explora-se o facto do trabalho que a Alice precisa fazer ser um trabalho de grupo, em que participam mais do que dois programadores. Como alguns dos programadores gostam de trabalhar nos mesmos horários, a Alice combina que todos trabalham na sua cópia do projeto e se juntam todos os dias da semana às 9h30 para juntar o trabalho que fizerem e voltar a redistribuir uma cópia. Combinaram também que cada um só iria alterar uma pequena parte do código, independente da dos outros, para evitar problemas na **fusão**. Infelizmente, numa das vezes, um dos elementos do grupo alterou bem mais do que a parte dele(a), dando imenso trabalho à junção do trabalho feito de forma distribuída e à resolução do problema.

Neste caso, **a necessidade específica é a de providenciar uma forma de trabalhar em cópias locais e de assistir na fusão (*merge*) desse trabalho no projeto**. Algumas das funcionalidades destes sistemas são a de facilmente permitir obter o projeto e grafo de objetos para gestão de versões de um repositório remoto e a de assistir na fusão do trabalho feito localmente no projeto global.

Conclusão

A **motivação principal** para estes sistemas é, portanto, ter uma **forma consistente e simples** de lidar com várias potenciais iterações de um ou mais ficheiros, com garantias de que **o trabalho feito não prejudica iterações passadas e que a história do trabalho feito não é perdida, mesmo que esse trabalho seja feito de forma colaborativa e distribuída**.

3 Terminologia de um Sistema de Gestão ou Controlo de Versões

Os próximos termos fazem tipicamente parte do jargão associado ao Sistema de Gestão ou Controlo de Versões:

- **Alteração** ou `diff` – a diferença entre duas versões;
- **Consolidação** (`commit`) – nome dado ao momento em que se guardam terminantemente os efeitos de alterações numa versão, adicionando informação contextual acerca do responsável, hora e data da consolidação e informação para navegação entre versões;

- **Cópia de Trabalho** — uma cópia local do projeto cujos ficheiros podem ser editados;
- **Ficheiro** — um projeto é frequentemente composto por vários ficheiros. O ficheiro é assim uma das unidades básicas com que estes sistemas lidam;
- `Head` – o ponteiro para a versão em que se está a trabalhar;
- **Repositório** — um ponto de armazenamento local ou remoto que é usado para guardar o projeto e todos os metadados que permitem a gestão das versões;
- **Versão** ou **Revisão** — um registo do conteúdo do projeto num determinado momento no tempo;
- ***Clone*** e ***Pull*** — frequentemente usados para referir a ação de obter um projeto e todos os metadados de controlo de versões, ou a versão mais recente de um (ramo do) projeto de um outro repositório;
- ***Push*** — frequentemente usado para referir a ação de colocação de um (ramo de um) projeto ou a sua versão mais recente num determinado repositório.

4 Propriedades de um Sistema de Gestão ou Controlo de Versões

Na sequência da discussão anterior, é agora possível identificar, protelando algumas explicações detalhadas, as principais características, propriedades ou funcionalidades que determinam um Sistema de Gestão ou Controlo de Versões:

- **Confiabilidade** – no sentido de assegurar a perenidade do trabalho feito, mantendo diferentes versões dos ficheiros de um projeto de forma (semi-)automática e consistente, e permitir armazenamento de cópias de segurança;
- **Monitorização** de vários ficheiros – no sentido de gerir e monitorizar os constituintes de um projeto de forma holística e não necessariamente ficheiro a ficheiro;
- Identificação de **versões com adorno contextual** – no sentido de fornecer uma forma normalizada para guardar informação pertinente acerca de uma consolidação;
- **Reversão** e navegabilidade – no sentido de permitir navegar entre versões e reverter os efeitos de alterações;
- **Comparação** de versões – no sentido de integrar mecanismos que facilitam distinguir alterações efetuadas entre versões;

- **Registo** histórico (*log*) – no sentido de integrar, automaticamente, funcionalidades relacionadas com o registo da evolução do projeto e da visualização dessa informação;
- Suporte à **ramificação** de projetos – no sentido de incluir ferramentas ou métodos que permitem criar ramos paralelos de um mesmo projeto, garantindo a gestão das versões em todos eles;
- **Assistência à fusão** de versões – no sentido de estruturar e assistir o procedimento para fundir duas versões ou ramos do projeto;
- Suporte à **colaboração** no projeto – descrita em baixo;
- **Versatilidade** – assegurando as propriedades e funcionalidades identificadas para projetos não necessariamente de implementação de código. Adicionalmente, deve assegurá-las também para diferentes tipos de ficheiros dentro do mesmo projeto, já que mesmo os projetos de implementação de código incluem atual e frequentemente vários tipos de ficheiros.

Note-se que, em cima, é referido o suporte à **colaboração** como uma das características principais desta classe de sistemas. Na verdade, a **assistência à fusão** faz já parte do conjunto de características que viabilizam esse suporte. Se um programador (ou equipa de programação) estiver a trabalhar numa das versões ou ramo de um projeto e pretender fundir esse trabalho com o de outro programador, é a **assistência à fusão** que o permitirá fazer. Contudo, há ainda outras características que o sistema deve apresentar para que essa colaboração seja eficiente:

- Atribuição e **rastreabilidade de responsabilidade** – que se refere à necessidade do sistema guardar informação contextual sobre quem fez determinada alteração e consolidou e quando, num projeto partilhado;
- **Repositórios partilhados e locais** (para suporte a trabalho em paralelo) – que se refere à integração com protocolos de comunicação que permitam rapidamente partilhar o projeto entre sistemas e com um repositório central;
- **Trabalho por subconjuntos** (trabalho em progresso) – que se refere à possibilidade de vários programadores poderem partilhar o seu trabalho em nichos, sem que isso tenha impacto no trabalho de outros programadores ou nichos e sem abrir mão do controlo de versões.

O ponto referido em último está intimamente ligado com o facto dos **sistemas serem distribuídos ou centralizados**. Nos sistemas distribuídos, como o Git ou o Mercurial, o **trabalho pode ser feito e trocado localmente ou por grupos de pessoas**; enquanto nos sistemas centralizados, como o Subversion, o **trabalho tem de ser partilhado através do repositório central** antes de poder ser editado por outros. Ambos os tipos de sistemas têm as suas vantagens e desvantagens que não são discutidas aqui.

5 Git

O Git, inicialmente desenvolvido por Linus Torvalds, é um **sistema de controlo de versões distribuído** utilizado para o desenvolvimento de *software*. No entanto, o mesmo pode ser utilizado para manter um registo histórico de qualquer tipo de ficheiros (*e.g.*, livros, artigos, *Curriculum Vitae*). O Git apresenta todas as características e funcionalidades descritas anteriormente.

Este sistema é muito frequentemente utilizado em combinação com repositórios de código baseados na *cloud* ou *online*, nomeadamente com o GitHub e o BitBucket. Estas plataformas *online* permitem que vários programadores possam contribuir para o desenvolvimento do mesmo projeto e manter também uma cópia de segurança *sempre* disponível.

Comandos Git

A utilização do Git enquanto ferramenta (e sua integração com o GitHub) será alvo de algumas aulas práticas. Assim, enumeram-se apenas alguns dos seus comandos mais úteis:

- `git init` que permite inicializar um repositório local na diretoria onde o comando é emitido;
- `git status` que permite verificar o estado atual da monitorização que o Git está a fazer, identificando ficheiros alterados e não monitorizados ainda (e que podem faltar consolidar);
- `git add nome-ficheiro(s)` que permite adicionar ficheiros à monitorização ou assinalar que devem ser consolidados no próximo `commit`;
- `git commit -m "mensagem"` que permite tentar uma consolidação (*commit*), adicionando uma mensagem que ficará no *log*;
- `git log` que permite visualizar o registo (*log*) até ao momento;
- `git reset -hard <id do commit>` que permite navegar para um *commit* anterior, **desfazendo/apagando** todas as versões que estão entre essa consolidação e a atual;
- `git revert <id(s) do(s) commit(s)>` que permite desfazer (*undo*) uma ou mais alterações identificadas nas consolidações (*commits*). Note que, neste caso, deve ser indicada o valor do *commit* que quer ver desfeito, e não o valor do *commit* para onde quer voltar (ao contrário do *reset*);
- `git branch` que permite ver em que ramo do desenvolvimento está atualmente;
- `git branch nome-branch` que permite criar um novo ramo chamado `nome-branch`;

- `git checkout nome-branch` que permite a navegação para o ramo denominado `nome-branch`;
- `git merge nome-branch` que permite fundir o ramo atual com o ramo indicado por `nome-branch`;
- `git remote add <name-of-repo> <site remoto>` que permite ajustar um repositório remoto com o nome `name-of-repo` e *Uniform Resource Identifier* (URI) `<site remoto>`;
- `git push origin master` que permite atualizar o ramo `master` no repositório em `origin` com a cópia local;
- `git pull origin master` que permite atualizar o ramo `master` no repositório local com o que estiver em `origin`;
- `git clone uri` que permite clonar na diretoria de trabalho todo o grafo de objetos de um repositório remoto, nomeadamente todos os ficheiros com código fonte e de gestão e controlo de versões.

7 Documentação de Código e Geração Automática

Sumário

A importância da qualidade da documentação e geração automática de documentação a partir de comentários no código, com caso de estudo do sistema `Doxygen`.

Summary

The relevance of documentation quality and automatic generation of documentation using code comments, with a study case of `Doxygen`.

1 Introdução

A documentação adequada de código é uma componente essencial no desenvolvimento de *software*. Nesse sentido, este capítulo focar-se-á em realçar a importância da documentação, refletindo sobre a estrutura usada para colocar comentários no código. Será também apresentada a geração de documentação estruturada, de forma automática, a partir de comentários inseridos no código. As metodologias apresentadas neste capítulo visam agilizar o processo de documentação, com qualidade, promovendo o desenvolvimento sustentado de *software*.

2 Documentação de um Projeto de *Software*

Qualquer projeto de *software* deve ser acompanhado de uma documentação rica e rigorosamente preparada. É possível identificar diversas categorias para essa documentação, nomeadamente:

1. A **documentação que cristaliza todo o processo de engenharia de *software***, sendo que a preparação de grande parte dessa documentação pode ser escrita antes da implementação de qualquer linha de código. Se bem feita, alguma desta documentação pode inclusive ser usada para automaticamente gerar algum código de implementação. Esta documentação **utiliza uma linguagem técnica** e, em algumas partes, uma linguagem **própria e normalizada**, como por exemplo a **UML**;

2. O **manual de instalação do *software***, que descreve os requisitos de instalação, eventuais dependências de outro *software* e como é que o ambiente deve ser configurado para o correto funcionamento do *software*. Esta documentação **utiliza uma linguagem técnica**;

3. O **manual de utilização do *software***, que descreve as funcionalidades e como estas podem ser corretamente usadas. Esta documentação é a que pode eventualmente **utilizar uma linguagem menos técnica** e orientada para o utilizador, devendo ser alvo de testes para atingir o objetivo de clareza e simplicidade de entendimento;

4. O **relatório de execução de um projeto de *software***, que pode incluir a documentação descrita em pontos anteriores (sobretudo no ponto 1) e que descreve a forma como o projeto foi preparado, conduzido (incluindo implementação de código), testado e entregue ao cliente final. Esta documentação **utiliza uma linguagem técnica**;

5. Os comentários colocados junto com a implementação do código, que devem usar **uma linguagem técnica**, bem estruturada;

6. A **documentação da interface de programação**, muito comummente fornecida em HTML para permitir uma navegação simples e rápida. Esta documentação **utiliza uma linguagem técnica** e, caso os comentários inseridos no código tenham uma estrutura e rigor que o permitam, pode ser automaticamente gerada a partir dos mesmos. Esta documentação reflete de forma muito fiel todo o código implementado;

7. **Um ficheiro** `README` (opcional, mas recomendado), que contém informação acerca de outros ficheiros numa diretoria de um projeto de *software* e informações rápidas acerca da preparação e instalação desse projeto, ou ponteiros para onde essa informação possa estar. Esta documentação **utiliza uma linguagem técnica**, sendo esse ficheiro um simples ficheiro em formato txt (`README.txt`, `READ.ME`) ou Markdown (`README.md`);

8. A **licença do *software***, tipicamente contida num ficheiro separado com o nome `LICENSE`. Esta documentação **deve utilizar uma linguagem próxima do utilizador e do legislador, sem descurar elementos técnicos**. Dado o potencial de uniformização deste tipo de documentos para vários projetos, existem atualmente

muitos textos pré-preparados e que podem ser facilmente adaptáveis, facilitando assim o processo de atribuição de uma licença a uma nova peça de *software*. Para código aberto, por exemplo, o site *Choose Your Licence* está inclusivamente desenhado para conduzir o(a) programador(a) para a licença mais adequada. **Todo o *software* deve ter uma licença.**

Há quem advogue que código correto, implementado seguindo normas bem definidas e claras, e optimizado não necessita de documentação. Contudo, está provado que, de uma forma geral, **a elaboração da documentação resulta em múltiplas vantagens**, mesmo para bons(as) programadores(as):

1. A inclusão de comentários **ajuda a estruturar e organizar o raciocínio**;
2. A **legibilidade do código é beneficiada** na grande maioria dos casos, especialmente em casos em que código não esteja implementado seguindo normas claras;
3. A documentação gerada é um **excelente complemento à disponibilização de um *software online*** para que outros o possam usar. Se um código estiver bem comentado, o programador que o utiliza não precisa, na maior parte dos casos, de olhar sequer para a implementação das funções/recursos que precisa de utilizar, bastando entender esses trechos de código como caixas negras que aceitam *inputs* e devolvem *outputs* com as devidas especificações;
4. Os comentários incluídos no código são **lidos por alguns IDEs quando se sobrevoa uma função com o rato**, facilitando a implementação de código. A exibição desta funcionalidade é apresentada na figura 7.1;

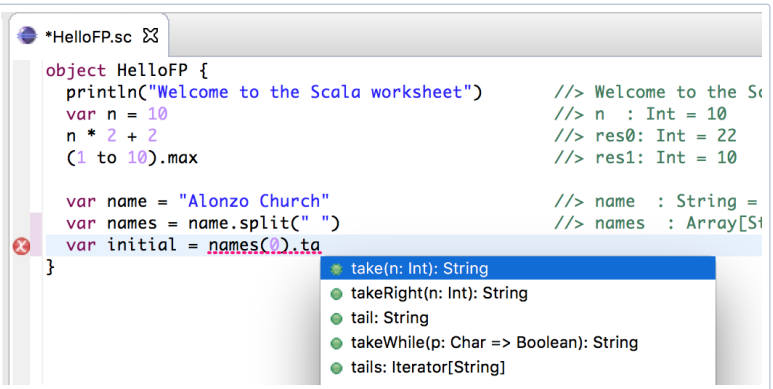

Figura 7.1: Exibição de documentação, no IDE Eclipse, usando *hover* de rato.

5. Finalmente, deve ter-se em conta que, **apesar de determinada implementação poder ser muito clara no momento em que se elabora**, por estar fresco e enquadrado o raciocínio nesse momento, **tal pode não ser verdade no futuro**.

Esta situação tem sido, de resto, alvo de vários piadas ao longo dos últimos anos, como o representado na figura 7.2.

```
// Dear programmer:
// When I wrote this code, only god and
// I knew how it worked.
// Now, only god knows it!
//
// Therefore, if you are trying to optimize
// this routine and it fails (most surely),
// please increase this counter as a
// warning for the next person:
//
// total_hours_wasted_here = 254
//

```

Figura 7.2: Exemplo de documentação incorreta do código fonte.

Apesar da importância da elaboração e disponibilização da documentação junto com o código ser inegável, é muito comum encontrarem-se peças de *software*, algumas até profissionais, com a documentação incompleta, mal feita ou em falta.

Um dos melhores exemplos de uma documentação do *software* bem feita é a disponibilizada com o *Software Development Kit* (SDK) *Android*. O reconhecimento de que uma documentação está bem escrita e é útil, bem como a capacidade de produzir documentação de qualidade para *Application Programming Interface* (API)s são competências importantes a adquirir no contexto da Engenharia Informática.

Alguns dos tipos de documentação referidos em cima são abordados noutras unidades curriculares dos cursos de Engenharia Informática. Por exemplo, a documentação relativa a Engenharia de *Software* é abordada na unidade curricular com o mesmo nome, incluindo, em alguns casos, UML. A elaboração de relatórios técnicos já foi também abordada na presente unidade curricular. As secções seguintes focam-se apenas na parte de documentação do código, sem que isso signifique um desmérito para os restantes tipos de documentação.

3 Documentação do Código

A maior parte das sugestões para documentação do código assentam na ideia de que **os comentários devem ser curtos, mas muito claros**. Normalmente, admitem-se **dois graus de verbosidade para descrições, uma breve e outra mais elaborada**, que podem aparecer em diferentes contextos. A forma como os comentários são construídos deve também seguir princípios de:

- **Consistência**, sempre escritos de forma semelhante e seguindo as mesmas regras ao longo de todo o projeto;
- **Escrita**, sempre que o comentário contenha frases, estas devem estar bem escritas na Língua nativa ou alvo e seguindo todas as suas regras sintáticas e de ortografia;
- **Completude**, apresentando de forma igual todos os elementos importantes.

Interpretando parte das indicações em cima, fica claro que **não se devem colocar comentários ao longo de todo o código, de forma dispersa ou arbitrária**, embora possam ser sempre colocados comentários para esclarecer alguns pontos nevrálgicos se necessário. Contudo, é possível identificar pontos na implementação onde a inclusão de documentação parece óbvia:

- No **início de um ficheiro de implementação** ou protótipos;
- Imediatamente **antes da implementação ou declaração de funções, rotinas, métodos ou classes** (esta última aplicável a linguagens orientadas a objetos);
- **Quando se declaram variáveis globais**.

Para complementar a explicação aqui contida, inclui-se um exemplo de um programa em C em que será integrada documentação seguindo os princípios e indicações descritas:

```
#include<stdio.h>

float a = 10;

int divide(float x, float y){
  if(y == 0)
    return 0.0;
  else
    return x/y;
}

int main(){
  float b = 20;
  printf("The division of a by b is %d\n", divide(a,b));
  return 0;
}
```

Uma boa prática de documentação é a **definição do nome do ficheiro no início do mesmo, bem como a inclusão do autor e da versão do *software*** (a informação referida em último é a mais dispensável, por mudar com mais frequência):

```
/*
 * File main.c.
 * Author Pedro R. M. Inácio
```

```
 * Version: 0.8
 */
#include<stdio.h>

float a = 10;

int divide(float x, float y){
  if(y == 0)
    return 0.0;
  else
    return x/y;
}

int main(){
  float b = 20;
  printf("The division of a by b is %d\n", divide(a,b));
  return 0;
}
```

A apresentação das funções, rotinas ou métodos é talvez um dos aspetos em que se deve ter mais cuidados. Esta apresentação **deve ter os seguintes elementos**:

1. Uma **apresentação breve desse trecho de código** que responde à questão: *O que é que este trecho de código faz exatamente?*;
2. A **apresentação de todos os parâmetros de entrada, o seu tipo e, eventualmente, como são usados**;
3. (Se aplicável) As **pré-condições a serem preenchidas** pelos parâmetros de entrada ou de variáveis globais para que o excerto de código execute corretamente;
4. (Se aplicável) **Uma explicação acerca daquilo que é devolvido**;
5. (Se aplicável) As **pós-condições** impostas pela execução desse código (*e.g.*, se forem mudadas variáveis globais ou sanitizados *inputs*).

Para reforçar o referido acima, inclui-se a seguir uma sugestão de formato para os comentários a colocar antes da declaração ou implementação de uma função:

```
/*
 * This funcion does this and that.
 * param1 is a <type of parameter> and it must fullfill condition1;
 * ...
 * paramn is a <type of parameter> and it must fullfill conditionn;
 *
 * This function returns ...
 *
 * Pre-conditions:
 * The global variable ... must be set to ...
 *
```

```
 * Post-conditions:
 * The global variable ... will be set to ... by the end of the function.
 */
```

Note-se, contudo, que a proposta anterior é (demasiado) verbosa e serve apenas para ajudar a construir aquela documentação em caso de dúvidas. Em muitos casos, o texto não precisa ser tão extenso, desde que bem organizado e enderece os pontos necessários. De seguida, melhora-se o exemplo de código incluído nesta secção com a integração da documentação para as funções:

```
/*
 * File main.c.
 * Author Pedro R. M. Inácio
 * Version: 0.9
 */
#include<stdio.h>

float a = 10;

/*
 * This function returns the division of two floats passed as parameters.
 * param x and y are floats, the dividend and the divisor, respectively.
 *
 * Returns the value of x divided by y if y is different from 0.
 * Returns 0 if y is equal to 0.
 *
 * Notice that the function checks if y is different from 0 before
 * performing the division.
 */
int divide(float x, float y){
  if(y == 0)
    return 0.0;
  else
    return x/y;
}

/*
 * This is the main function of the program.
 * Returns 0 on exit.
 */
int main(){
  float b = 20;
  printf("The division of a by b is %d\n", divide(a,b));
  return 0;
}
```

Finalmente, importa mencionar que **é comum colocar a documentação de variáveis (globais) na mesma linha em que são definidas** usando blocos de comentário em linha, conforme se mostra no excerto de código seguinte (propositadamente condensado para melhorar a legibilidade):

```
/*
```

```
 * File main.c.
 * Author Pedro R. M. Inácio
 * Version: 1.0
 */
#include<stdio.h>

float a = 10; /* The dividend is a global variable. */
...
```

Note que, no exemplo anterior, o comentário aparece **depois** do elemento a documentar e que, por isso, **vai ser preciso indicar explicitamente este facto às ferramentas de geração de documentação** (conforme será mencionado na secção 4). Note também que não é comum documentar variáveis locais, embora possa ser feito em ocasiões particulares.

4 Geração Automática de Documentação em Formato Estruturado

A geração automática de documentação a partir de comentários no código é normalmente **conseguida através da inclusão de anotações e pequenos indícios** nesses comentários. Essas anotações são **introduzidas por palavras-chave (*keywords*) reservadas com um significado para o gerador**, mas que são **também legíveis para o programador** (*i.e.*, são instruções não intrusivas) e até costumam contribuir para a estruturação desse comentários. Por exemplo, no caso de se estar a usar a ferramenta `Doxygen`, a descrição de uma variável pode ser introduzida pela *keyword* `@param`, enquanto a descrição do retorno de uma rotina poderia ser sinalizado pelo comando `@return`.

É interessante notar que outras ferramentas, como o `javadoc`, usam formas semelhantes às referidas antes. Na verdade, no caso dos parâmetros e do valor de retorno de uma função, as *keywords* utilizadas pelo `javadoc` são as mesmas que as utilizadas para o `Doxygen`. Uma das **maiores vantagens** de se gerar a documentação a partir do código é que esta abordagem garante, por construção, que essa documentação se mantém fiel a possíveis atualizações (já que podemos atualizar os comentários *in-situ*).

Note-se que nem todos os comentários introduzidos no código são lidos pelas ferramentas de geração de documentação. Aliás, **os comentários só são incluídos se os indícios ou anotações assim sinalizarem**. Por exemplo, no `Doxygen` (como também noutras ferramentas), os comentários só são processados se introduzidos com **dois asteriscos na primeira linha**, como se exemplifica no trecho seguinte (mas atente à discussão que sucede o exemplo):

```
/**
 * File main.c.
 * Author Pedro R. M. Inácio
 * Version: 1.0
 */
```

```c
#include<stdio.h>

float a = 10; /**< The divident is a global variable. */

/**
 This function returns the division of two floats passed as parameters.
 param x and y are floats, the dividend and the divisor, respectively.

 Returns the value of x divided by y if y is different from 0.
 Returns 0 if y is equal to 0.

 Notice that the function checks if y is different from 0 before
 performing the division.
 */
float divide(float x, float y){
  if(y == 0)
    return 0.0;
  else
    return x/y;
}

/*
 * This is the main function of the program.
 * Returns 0 on exit.
 */
int main(){
  float b = 20; /* Second variable (divisor). */
  printf("The division of a by b is %d\n", divide(a,b));
  return 0;
}
```

No exemplo anterior, o primeiro, o segundo e o terceiro blocos de comentários seriam processados pelo `Doxygen`, mas o quarto bloco e a descrição da variável `b` na `main()` não seriam. Note-se também que os asteriscos intermédios (entre `/**` e `*/`) são opcionais. Contudo, para assegurar a consistência, estes deveriam ter sido usados no segundo bloco.

A título de curiosidade, dizer que **existem outras formas equivalentes de assinalar que um comentário deve ser processado** por um gerador de documentação. Por exemplo, as seguintes representações também funcionavam:

```c
/*!
 * ... text ...
 */
```

ou

```c
///
/// ... text ...
///
```

No caso da linguagem Python, a introdução de documentação é feita usando especificamente a combinação de três aspas logo após o cabeçalho da função (também seriam necessárias configurações adicionais no `Doxygen`):

```
def func():
  """Documentation for the function.

  More details.
  """
  return 0
```

De modo a concretizar o que se entende com introdução de *keywords* para anotações, volta a melhorar-se o programa usado como exemplo nesta secção, de modo a que o `Doxygen` pudesse reconhecer e corretamente formatar os vários elementos a serem documentados:

```
/**
 * @file    main.c.
 * @author  Pedro R. M. Inácio
 * @version 1.1
 */
#include<stdio.h>

float a = 10; /**< The divident is a global variable. */

/**
 * This function returns the division of two floats passed as parameters.
 * @param x and @param y are floats, the dividend and the divisor,
     respectively.
 *
 * @return The value of x divided by y if y is different from 0.
 * @return 0 if y is equal to 0.
 *
 * Notice that the function checks if y is different from 0 before
 * performing the division.
 */
int divide(float x, float y){
  if(y == 0)
    return 0.0;
  else
    return x/y;
}

/**
 * This is the main function of the program.
 * @return 0 on exit.
 */
int main(){
  /** Second variable (divisor). */
  float b = 20;
  printf("The division of a by b is %d\n", divide(a,b));
  return 0;
}
```

Note que o exemplo incluído antes já está completo em termos de documentação e é totalmente compatível com o `Doxygen`.

Aproveita-se para se chamar a atenção para a descrição da **variável global** `a`. Neste caso, a variável é introduzida por `/**<`, sendo que **o sinal menor (`<`) é usado para explicitamente indicar ao `Doxygen` que este bloco de comentário está a aparecer depois do elemento que deve ser documentado**. O bloco que documenta a variável `b` foi colocado antes da declaração da variável para aparecer no local certo, já que o `Doxygen` não documenta da mesma forma variáveis locais.

Doxygen

O `Doxygen` é definido pelo seu autor como **a ferramenta *standard* para gerar documentação (estruturada) a partir de código fonte** C++ devidamente anotado, mas que também suporta outras linguagens de programação populares, como o C, Objective-C, C#, PHP, Java e Python, *Interactive Data Language* (IDL), entre outras.

O `Doxygen` pode ser usado para **gerar uma documentação navegável para *browsers* em HTML** ou um **manual de referência em PDF**, preparando esse documento em LaTeX. Também é possível exportar para *Rich Text Format* (RTF) e **páginas de manual Unix** `man`. Para além disso, pode ainda ser usado para gerar *websites* em HTML a partir de ficheiros de texto bem formatados e que sigam a sintaxe usada pela ferramenta.

É ainda interessante notar que **é possível gerar documentação a partir de código não comentado**. É claro que:

- A documentação gerada dessa forma **não terá a mesma riqueza** (no sentido do detalhe) que aquela gerada com comentários;
- Contudo, pode ser muito **útil para entender grandes projetos de *software* existentes**, já que produz um documento navegável que estrutura todos os ficheiros, classes, rotinas e variáveis desse projeto automaticamente.

Instalação

O `Doxygen` é uma ferramenta de uso gratuito com uma pegada bastante leve (menos de 20 *MegaByte* (MB)), especialmente orientada para a linha de comandos, embora possa ser integrado com alguns IDEs. Por ter sido desenvolvido para sistemas Unix-*like* e Linux, a sua instalação é simples nestes sistemas operativos (estando disponível nos repositórios de *software* comuns das distribuições) mas, por ser desenhado para ser altamente portátil, podem ser também encontrados binários para Windows. A título de exemplo, dizer que em várias distribuições Linux, a sua instalação resume-se a instruções semelhantes a:

```
sudo apt-get install doxygen # Em Ubuntu
```

```
sudo pacman -S doxygen # Em Arch Linux

sudo dnf install doxygen # Em Fedora
```

Utilização e Fluxo de Trabalho

A utilização da ferramenta `Doxygen` é simples e requer apenas **três passos** principais (assumindo que o código já está comentado de acordo com sintaxe necessária):

- **Geração de um ficheiro de configuração** da ferramenta na raiz do projeto com:

  ```
  doxygen -g [nome-ficheiro-de-configuracao]
  ```

  O comando anterior gera, por omissão, um ficheiro chamado `Doxyfile` (embora o nome do ficheiro possa ser definido pelo utilizador);

- **Edição do ficheiro de configuração** com um editor de texto (`nano Doxyfile`) para ajuste de detalhes do projeto (*e.g.*, ajustar o nome do projeto, o caminho para as diretorias onde o código fonte está incluído, ou a profundidade dos índices);

- **Geração da documentação**, emitindo o seguinte comando na raiz do projeto:

  ```
  doxygen [nome-ficheiro-de-configuracao]
  ```

É interessante notar que, por omissão, o `Doxygen` **gera tanto a documentação navegável *online* como o manual em LaTeX**. Este manual técnico pode ser rapidamente obtido da compilação do ficheiro `refman.tex`, produzido para a diretoria `latex` da documentação.

Comandos para Utilização do Doxygen

Na secção anterior, foi referido que o motor de geração de documentação precisa de encontrar determinadas *keywords* ou indícios nos comentários para corretamente estruturar e formatar os documentos produzidos. No `Doxygen`, estas *keywords* são conhecidas como comandos especiais. Ao longo desta aula já foram mostrados alguns desses comandos nos vários exemplos, nomeadamente:

- `@file <nome-do-ficheiro>` – usado para indicar o nome do ficheiro atual. Note-se que, na verdade, este comando deve estar obrigatoriamente no ficheiro ou o `Doxygen` não o irá incluir na documentação gerada;
- `@autor <name>` – usado para enfatizar o nome do autor;
- `@version <code>` – usado para indicar a versão do ficheiro ou *software*;
- `@param <desc>` – usado para introduzir a descrição de parâmetros;

- `@return <desc>` – usado para descrever os retornos do trecho de código documentado.

Existem **muitos** outros comandos para além dos enunciados e que podem ser usados para adicionar riqueza à documentação produzida. Sugere-se, por isso, a consulta da documentação em https://www.doxygen.nl/manual/, em particular a listagem e descrição de todos os comandos, em https://www.doxygen.nl/manual/commands.html. Também deve ser referido que os comandos podem ser introduzidos por `@` ou por \, *i.e.*, o comando `@return` é equivalente a `\return`, exibindo semelhanças à forma como o LaTeX define estes recursos.

Dois dos comandos adicionais que ainda não foram referidos, mas que podem ser interessantes e com utilidade mais imediata, são o `@brief <desc>` e o `@details <desc>`, que servem para distinguir duas descrições de um determinado elemento (uma mais breve e outra mais longa), que aparecem em diferentes contextos da documentação:

- A descrição introduzida com `@brief <desc>` aparece na descrição sumária dos vários elementos;
- Enquanto a descrição introduzida por `@details <desc>` acede nas páginas em que são fornecido detalhes.

Adaptando, novamente, o exemplo anterior (omitindo parte do trecho, por uma questão de legibilidade), obteríamos:

```
/**
 * @file    main.c.
 * @brief   This program outputs a division.
 * @details This program calculates and outputs the value of
 * the division of 10 by 20.
 * @author  Pedro R. M. Inácio
 * @version 1.2
 */
#include<stdio.h>
...
```

Geração de Documentação para Além dos Comentários

O `Doxygen` **permite ainda gerar documentação navegável ou manuais técnicos a partir de ficheiros de texto com sintaxe compatível**. Aliás, é interessante notar que, como o próprio autor da ferramenta refere, toda a documentação do `Doxygen` na *Web* foi produzida com a própria ferramenta. Com a configuração certa, a ferramenta suporta *tags* HTML e Markdown.

A título de exemplo, pode-se dizer que um ficheiro `main.md`, com o conteúdo incluído a seguir, daria origem a uma página principal (`@mainpage`) com o título `The main page` se fosse processado pelo `Doxygen`:

```
@mainpage The main page
This is the main page of the documentation of the project.
Its contents can be summarized as follows:
1. It lists the files implementing the software;
2. It lists each function of the program;
3. It contains reference links.
```

Este tipo de funcionalidade adicional **permite, facilmente, adornar a documentação gerada a partir do código**, colmatando lacunas decorrentes das limitações de gerar texto apenas a partir de comentários.

Parte II

Guias Práticos Laboratoriais

8 Introdução ao LaTeX

Sumário

Introdução ao LaTeX: instalação, estrutura base de um documento, títulos, índices, comandos e ambientes.

Summary

Introduction to LaTeX: installation, basic structure of a document, titles, indexes, commands and environments.

Pré-requisitos:

Algumas das tarefas propostas a seguir requerem um sistema com acesso à Internet e uma conta na plataforma ***Overleaf*** - *The Online LaTeX Editor*. Se optar por instalar o LaTeX no seu sistema sugere-se, o uso de uma distribuição comum de Linux, onde todas estas condições estarão provavelmente preenchidas.

1 Introdução ao LaTeX

TeX (que se pronuncia como *te**k***) é um sistema de escrita criado por Donald Knuth para compor documentos mais atraentes e consistentes. A ideia principal deste sistema é fornecer meios para que os autores se preocupem mais com o conteúdo e não com as regras tipográficas que produzem um documento esteticamente atraente.

O LaTeX (que se pronuncia como *la-te**k***) é um sistema de escrita de elevada qualidade, que se concretiza sobretudo por um pacote de comandos simplificados, construído sobre o TeX original. Estes comandos ou conjuntos de comandos são agregados em programas e disponibilizados numa distribuição. Assim, uma distribuição não é nada mais do que uma coleção de programas que podem ou não incluir editores, estilos, classes, *etc.* Atualmente, é possível encontrar as seguintes distribuições LaTeX:

- **TeX Live**[1] **e MiKTeX**, adequado para qualquer SO; e o
- **MacTeX**, variante do TeX Live para os SOs macOS.

A escrita de documentos LaTeX pode ser feita utilizando um editor especializado, que congrega diversas facilidades específicas para este sistema. Exemplos desses editores incluem o **TeXtudio**, **TeXmaker**, ou o **TeXShop** (este último apenas para os SOs macOS). Adicionalmente, existem serviços *online* que permitem a utilização e criação de documentos LaTeX sem a necessidade de realizar uma instalação local de uma distribuição.

2 Instalação do LaTeX

Para editar ficheiros `.tex` irá necessitar do sistema TeX e de um editor. A instalação no Windows poderá ser feita recorrendo à distribuição **MiKTeX**. Pode obter esta distribuição através do seguinte endereço *Web*: https://miktex.org/download.
Alternativamente, se utilizar um SO Linux, pode instalar a distribuição TeX Live com um conjunto de comandos semelhantes aos seguintes:

```
$ sudo apt-get install texlive-lang-portuguese
$ sudo apt-get install texlive
$ sudo apt-get install texlive-latex-extra
$ sudo apt-get install texlive-math-extra
$ sudo apt-get install texmaker
```

Instalação no Ubuntu.

```
$ su
$ yum install texlive
$ yum install texlive-latex
$ yum install texmaker
```

Instalação no Fedora.

```
$ sudo pacman -S texlive-core
$ sudo pacman -S texlive-latexextra
$ sudo pacman -S install texlive-publishers
$ sudo pacman -S texmaker
```

Instalação no Arch Linux.

Apesar da sugestão em cima, nas aulas práticas desta unidade curricular é **recomendada a utilização da plataforma *Overleaf*** (disponível em https://www.overleaf.com/project). Também pode usar esta plataforma para a elaboração do relatório da unidade curricular.

Q1.: O que é que está errado nas seguintes expressões/palavras?

`Tex Live`, `LaTeX` e `MikTex`

[1] https://www.tug.org/texlive/

☐ Não há nada de errado. Relaxa!
☐ Simples: as palavras estão em `monospace`, mas toda a gente sabe que deviam estar a *negrito*...
☐ Simples: as palavras estão em `monospace`, mas toda a gente sabe que deviam estar em **itálico**...
☐ Estas palavras requerem uma estilização muito própria, com letras acima e abaixo da linha de texto, que só se consegue com muito jeitinho (e com comandos LaTeX).

3 Estrutura Geral de um Documento LaTeX

A estrutura geral de um documento LaTeX é tipicamente divida em duas partes: uma primeira onde é definido o tipo de documento que estamos a redigir, os pacotes que queremos utilizar, entre outros elementos; e uma segunda parte onde está o conteúdo do documento.

Q2.: Como se denominam as duas partes que constituem a estrutura geral de um documento LaTeX (escolha duas respostas)?
☐ E como é que vou saber?
☐ Preâmbulo ☐ Introdução ☐ Cabeça ☐ Tronco ☐ Conclusão ☐ Membros
☐ Conteúdo ☐ Meio ☐ Principio ☐ Fim

Tarefa 1

Crie um novo projeto na plataforma *Overleaf* e insira, no ficheiro `.tex`, o seguinte trecho LaTeX:

```
% Preâmbulo
\documentclass[a4paper, 12pt]{report}
\begin{document}
  % Conteúdo
  This is my first document in \LaTeX.
\end{document}
```

Depois de criado o ficheiro, `compile` e verifique o resultado no visualizador do lado direito.

Q3.: As palavras `% Preâmbulo` e `% Conteúdo` aparecem no documento após a compilação?
☐ Claro que aparecem.
☐ Claro que não aparecem.
☐ Aparecem e desaparecem a piscar! Impecável!
☐ Claro que não aparecem, porque são comentários.
☐ Claro que não aparecem, porque só aparece texto em Inglês.

Q4.: O que acontece se colocar texto (*e.g.*, a palavra `teste`) após a instrução `\end{document}`?
☐ Sem olhar, digo logo que vai dar erro!
☐ Não dá erro, e a palavra `teste` aparece agora no documento.
☐ Não dá erro, e a palavra `teste` não aparece no documento (é ignorada).

Tarefa 2

Insira a frase

```
Eu devia ter experimentado logo!
```

no final do ficheiro que criou anteriormente (após a instrução `\end{document}`), `compile` e verifique o resultado.

O comando `\documentclass` deve aparecer no início de todos os documentos LaTeX. Este comando define as diferentes classes de documentos. Por exemplo, um livro (*book*), um artigo (*article*), um relatório (*report*) ou uma carta (*letter*).

A classe *report* é normalmente utilizada para estruturar documentos mais longos com capítulos e secções (*e.g.*, Ph.D. *thesis*). O texto dentro dos parênteses retos define as possíveis formatações a serem aplicadas ao documento, neste caso o tamanho do papel e o tamanho da fonte (*a4paper, 12pt*). Este campo é opcional.

No preâmbulo são incluídos todos os comandos que poderá usar ao longo do documento. Por fim, o conteúdo do documento é introduzido no ambiente `\begin{document}` e `\end{document}`, onde qualquer texto introduzido após `\end{document}` será ignorado.

4 Gerar um Título Automaticamente

Tarefa 3

Crie um novo ficheiro `.tex` e insira-lhe o seguinte trecho LaTeX:

```
\documentclass[a4paper]{report}
\usepackage[T1]{fontenc}
\title{Planet Description at R-16}
\author{Owen Lars}
\begin{document}
  \maketitle % Gerar o título do documento.
  This is my second document in \LaTeX.
\end{document}
```

`Compile` e analise o resultado.

Q5.: É verdade que foi gerada uma página rosto só com o titulo, o autor e a data?
☐ Mesmo! Ficou lindo!
☐ Não. Apareceu o título e o nome do autor imediatamente antes do texto...

O comando `\maketitle` permite gerar informações relacionadas com o título do documento. Para este fim deve definir o título do documento (comando `\title`) e, opcionalmente, informação referente ao seu autor (comando `\author`) e a data do documento (comando `\date`). Caso não especifique uma data, será utilizada a data atual.

Tarefa 4

Altere o título e o autor e insira uma data no exemplo anterior. Depois, `compile` e teste o resultado.

5 Capítulos, Secções e Referências Cruzadas

Deverá sempre organizar o seu documento em capítulos, secções e sub-secções. Isto é possível recorrendo aos seguintes comandos:

- `\chapter{...}`
- `\section{...}`
- `\subsection{...}`
- `\subsubsection{...}`
- `\paragraph{...}`
- `\subparagraph{...}`

Tarefa 5

A próxima tarefa consiste, por isso, na criação de um novo documento com as seguintes características:

- o título deve ser `Star Wars`;
- o autor deve ser `George Lucas`;

- deve ter cinco capítulos com os títulos `Introdução`, `A New Hope`, `The Empire Strikes Back`, `Return of the Jedi` e `Conclusão`;
- o primeiro capítulo deve ter duas secções com os títulos `Motivação` e `Organização do Documento`.

No final, `compile` e analise o resultado.

Tarefa 6

Analisando o exemplo seguinte, crie uma etiqueta (*label*) para cada um dos capítulos e/ou secções de forma a que sejam referenciados noutras partes do documento. Não se esqueça de ir `compilando` e testando o resultado.

```
...
\chapter{Name of the Chapter}
\label{ch-1}
You can find more info on chapter \ref{ch-1}.
...
```

Q6.: Qual é a diferença entre o comando `\label` e `\ref` ?

- ☐ O comando `\label` permite atribuir uma etiqueta a um elemento anterior, enquanto que o comando `\ref` permite fazer referência ao elemento.
- ☐ O comando `\ref` permite atribuir uma etiqueta a um elemento anterior, enquanto que o comando `\label` permite fazer referência ao elemento.
- ☐ A palavra `\label` lê-se quase da mesma maneira de trás para a frente e da frente para trás... Impecável!
- ☐ Nenhuma das opções anteriores.

Q7.: Qual é o comando que permite mostrar as páginas de uma referência?

---

6 Índice

Se conduziu a aula corretamente até aqui, deverá ter um documento com vários capítulos e secções (e pode adicionar sub-secções). Poderá gerar o índice do documento utilizando o comando `\tableofcontents` e o comando `\pagenumbering{...}`. Este último pode ser usado para alterar a numeração das páginas – árabe (`arabic`) ou romano (`roman`).

Tarefa 7

Altere o documento LaTeX que elaborou anteriormente de maneira a gerar um índice que utilize a numeração romana.

7 Comandos e Ambientes

Nos exercícios anteriores foram utilizados diversos comandos (`\usepackage`, `\section`, `\title`, `\author`, *etc.*) e um ambiente (`\begin{document}` ... `\end{document}`).

A diferença principal entre um comando e um ambiente é: o comando deve fazer algo em X, enquanto o ambiente define algo que deve funcionar de uma determinada forma de X até Y.

Por vezes é útil definir os nossos próprios comandos e/ou ambientes. Nestas situações pode-se utilizar a seguinte sintaxe para definir, respetivamente, um comando e um ambiente:

```
\newcommand{Nome_comando}[Núm.Arg]{...}
```

```
\newenvironment{Nome_Ambiente}[Núm.Arg] {X}{Y}
```

O nome do comando começa **sempre** por \. Relativamente ao ambiente, X refere-se a mudanças **antes** do texto dentro do ambiente e Y refere-se mudanças **após** o texto dentro do ambiente.

Tarefa 8

Considere incluir os seguintes trechos num ficheiro LaTeX, nos locais devidos. Depois, `compile` e analise o efeito do comando `eg` e do ambiente `centro` no documento compilado:

```
\newcommand{\eg}[1]{por exemplo, #1}
...
Distribuições Linux (\eg{Fedora}) são comuns.
```

```
\newenvironment{centro}[1]
{\begin{center} #1} % Antes
{\end{center}} % Após
...
Este texto está
\begin{centro}{centrado. Mas não sei se continua sempre}
\end{centro}
centrado...
```

Atente, primeiramente, ao trecho relativo ao comando. **Q8.: Qual é o nome do comando?**
☐ `\eg` ☐ `eg` ☐ `por exemplo` ☐ `#1`

Q9.: Nos trechos de códigos apresentados, quantos argumentos tem, respetivamente, o comando e o ambiente?
☐ 1 e 1 ☐ 1 e 2 ☐ 2 e 1 ☐ 2 e 2

Atente ao trecho relativo ao ambiente. Considere que o ambiente `center` tem como funcionalidade centrar palavras. **Q10.: Tendo em conta o número de argumentos passados ao ambiente `centro`, acha que toda a frase será apresentada centrada?**
☐ Não, apenas a palavra "centrado". O resto da frase aparecerá alinhada à esquerda.
☐ Sim, pois toda a frase é um argumento. Toda não, que o "centrado..." já deverá aparecer alinhado à esquerda. Acho eu...

Tarefa 9

Com base na sua análise aos trechos de código, ao documento compilado e nas respostas às questões anteriores, registe a finalidade do comando e do ambiente criados.

Tarefa 10

Modifique o ambiente da Tarefa 8 para que, **dentro do ambiente e antes de apresentar o texto centrado**, seja escrito "teoricamente". Para tal deverá criar um comando com o nome `\teorica` e incorporá-lo dentro do ambiente.

8 Codificação

A utilização de uma Língua que possui acentos ou caracteres diferentes daqueles existentes na codificação definida, por omissão, no editor e/ou na distribuição poderá requerer a definição da codificação no preâmbulo do documento. Caso isto não seja feito, letras, palavras com acentos ou outros símbolos especiais não serão corretamente interpretados (e.g., *água* daria origem a *gua* após `compilar`). Deve definir a codificação da seguinte forma:

```
...
\usepackage[utf8]{inputenc}
...
\begin{document}
A palavra água irá ser escrita corretamente.
\end{document}
```

O uso do comando `\usepackage[utf8]{inputenc}` irá permitir criar documentos LaTeX em *utf-8*, uma codificação *multi-byte*, na qual cada caractere poderá ser codificado no mínimo com um *byte* e, no máximo, com quatro *byte*.

Tarefa 11

Crie um novo ficheiro `.tex` e insira-lhe o seguinte trecho LaTeX:

```
\documentclass[a4paper, 12pt]{report}
% Preâmbulo
\usepackage[ascii]{inputenc}
\begin{document}
  % Conteúdo
  A Língua Portuguesa não usa acentos!
\end{document}
```

No final, `compile` e analise o resultado.

Q11.: Apareceu tudo direitinho?

☐ Eishh.... que engraçado: não! Neste caso, faltam lá letras! ☐ Sim, tudo bem!

Tarefa 12

Altere o ficheiro referido na tarefa anterior, mudando
`\usepackage[ascii]{inputenc}`
para
`\usepackage[utf8]{inputenc}`.
No final, `compile` e analise o resultado.

Q12.: Apareceu tudo direitinho?

☐ Ainda não... ☐ Já está impecável.

9 Introdução ao LaTeX II

Sumário

Introdução ao LaTeX: manipulação da letra, listas, tabelas, figuras, bibliografia e equações.

Summary

Introduction to LaTeX: font manipulation, lists, tables, figures, references, and equations.

Pré-requisitos:

Algumas das tarefas propostas a seguir requerem um sistema com acesso à Internet e uma conta na plataforma ***Overleaf*** - *The Online LaTeX Editor*. Se optar por instalar o LaTeX no seu sistema sugere-se, o uso de uma distribuição comum de Linux, onde todas estas condições estarão provavelmente preenchidas.

1 Tipos de Letra

A redação de um documento técnico necessita, muitas vezes, de usar diferentes formatações de texto. Alguns exemplos onde diferentes formatações podem ser úteis são: dar ênfase a uma ideia, isolar um conceito importante ou introduzir um estrangeirismo no texto. LaTeX dispõe de um conjunto de comandos para auxiliar nesta tarefa, nomeadamente:

- `\textit{itálico}`
  Muda a fonte para a versão *em itálico*;
- `\textbf{negrito}`
  Muda a fonte para a versão **a negrito**;
- `\underline{Sublinha}`
  Sublinha as palavras;

- `\textsc{smallcaps}`
  Coloca palavras em SMALLCAPS.

Tarefa 1

Crie um novo projeto na plataforma *Overleaf* com um ficheiro `.tex` e insira-lhe a palavra *Jabba* em itálico e a palavra **Naboo** a negrito. Não se esqueça de `compilar` e testar.

Q1.: Tendo em conta os comandos enunciados no início desta secção, acha que é possível escrever a palavra `teste` em itálico e, simultaneamente, a negrito?
☐ Nem pensar!
☐ Deve ser possível, mas vou esperar que o Prof. faça...
☐ Sim, encadeando os comandos da seguinte forma: `\textit{\textbf{teste}}`.
☐ Sim, encadeando os os comandos da seguinte forma: `\textbf{\textit{teste}}`.

Questão bónus → não é para todos(as)!
Q2.: O que faz o comando `\emph{palavra}`?
☐ Que giro, não faz nada! Só existe mesmo para encher t'choriços.
☐ Tem exatamente o mesmo efeito que `\textit{itálico}`.
☐ Tem efeito semelhante a `\textit{itálico}` em alguns casos. Contudo, a seguinte combinação `\textit{\textit{itálico}}` comporta-se de maneira diferente de `\emph{\emph{itálico}}`.
☐ Já estou cafuso(a), e prefiro não fazer esta questão...

2 Pacotes e Cores

O uso de cores em documentos pode também ter a sua utilidade. Similarmente à formatação de texto, o LaTeX contém um conjunto de comandos que permite alterar a cor do texto de um documento. No entanto, para fazer uso desta funcionalidade, deverá incluir um pacote, usando a seguinte sintaxe: `\usepackage[opções]{Nome_pacote}`.

Q3.: Qual é o nome do pacote para adicionar cor?
☐ O Professor mentiu e não é necessário adicionar nenhum pacote.
☐ O pacote para adicionar cor é o *color*.
☐ Pesquisei no *Google* mas não encontrei!
☐ O pacote para adicionar cor é o *colors*.

As cores básicas disponíveis no pacote são: black, red, green, blue, cyan, magenta, yellow e white.

Tarefa 2

Adicione ao ficheiro `.tex` um texto à sua escolha com a cor magenta. Para esse efeito, considere usar o comando `\color` e `compile` .

Relativamente à cor de texto, existe um outro comando chamado `\textcolor`. **Q4.: Qual(is) é(são) a(s) diferença(s) deste comando para o comando `\color`?**

- ☐ Este comando é exatamente igual ao comando `\color`!
- ☐ Ao contrário do comando `\color`, o `\textcolor` recebe como argumento um excerto de texto.
- ☐ O comando `\textcolor` altera apenas cor da primeira palavra de um texto.
- ☐ O comando `\color` permite alterar texto ao longo de múltiplas linhas e dentro de outros ambientes, enquanto `\textcolor` apenas modifica o texto num parágrafo e não pode conter outros ambientes.

Q5.: Qual o comando para introduzir uma cor de fundo numa porção de texto?

---

3 Tamanho de Letra

A definição de diferentes tamanhos de letra é outra funcionalidade no LaTeX. Para tal, existe um conjunto de comandos que permitem definir uma variedade de tamanhos, nomeadamente:

- \tiny palavra: palavra;
- \scriptsize palavra: palavra;
- \footnotesize palavra: palavra;
- \small palavra: palavra;
- \normalsize palavra: palavra;
- \large palavra: palavra;
- \Large palavra: palavra;
- \LARGE palavra: palavra;
- \huge palavra: palavra.

Tarefa 3

Usando o mesmo ficheiro `.tex`, introduza a frase seguinte:

```
This tiny little sentence has a large number of small words.
```

Coloque cada palavra da frase com um tamanho diferente, começando do mais pequeno para o maior (use **todos** os comandos acima descritos). `Compile` e certifique-se que o resultado se coaduna com o esperado.

Tarefa 4

Considere o seguinte cenário:

```
\tiny Alice Bob
```

Q6.: Tendo em conta o comando utilizado acima, assinale a opção correta:

- ☐ Ambas as palavras ficam com tamanho minúsculo.
- ☐ Apenas a palavra `Alice` fica com tamanho minúsculo, enquanto a palavra `Bob` fica com tamanho normal.
- ☐ Apenas a palavra `Bob` fica com tamanho minúsculo, enquanto a palavra `Alice` fica com tamanho normal.
- ☐ O comando é ignorado e ficam todos com tamanho gigante, só para contrariar!
- ☐ Dado esta pergunta estar dentro de uma tarefa isolada, deve ser mesmo para eu experimentar...

4 Listas

A apresentação de informação de uma forma estruturada permite uma melhor compreensão do conteúdo exposto. Uma forma de estruturar informação passa pelo uso de listas. Com esta finalidade, o LaTeX permite criar diferentes tipos de listas: não ordenadas, ordenadas e lista de termos. Nos exemplos abaixo são ilustrados excertos de código LaTeX, do lado esquerdo, e a sua representação, após compilação, do lado direito:

```
\begin{itemize}
 \item 5 ovos
 \item 1l de leite
 \item ...
\end{itemize}
```

→

- 5 ovos
- 1l de leite
- ...

Exemplo – Listas não ordenadas.

```
\begin{enumerate}
 \item Separar gemas
 \item Bater claras
 \item ...
\end{enumerate}
```

→

1. Separar gemas
2. Bater claras
3. ...

Exemplo – Listas ordenadas.

```
\begin{description}
 \item[Energia:] 657 kJ
 \item[Sódio:] 110 mg
 \item ...
\end{description}
```

→

Energia: 657 kJ

Sódio: 110 mg

...

Exemplo – Listas de termos.

Tarefa 5

Adicione ao ficheiro `.tex` uma receita culinária de crepes (*e.g.*, copie o conteúdo de https://www.vaqueiro.pt/recipes/crepes-198748). Use uma lista não ordenada para indicar os ingredientes e uma lista numerada para as instruções.

Foque-se na definição de ambientes. **Q7.: O termo utilizado, entre chavetas, em `\begin` e `\end` tem que ser o mesmo?**
☐ Não, só fazemos por conveniência. O `\end` até pode ficar vazio.
☐ Sim, claro. Senão até dá erro de compilação.

5 Comentários e Espaçamento

Em LaTeX é possível adicionar comentários utilizando o caractere %. Os comentários podem ser utilizados para escrever notas importantes como descrições de comandos, anotações das aulas ou até assinalar frases que não queremos que apareçam no documento final compilado. Quando o LaTeX encontra o caractere %, ignora todo o conteúdo desde este caractere até ao final da linha. O conteúdo ignorado não será compilado e, portanto, não aparecerá na versão final do documento.

O excerto de código

```
I guess you guys aren't ready for that
yet. % But your kids are gonna love it.
```

irá ter a seguinte aparência

```
I guess you guys aren't ready for that yet.
```

no documento final.

Relativamente ao **espaçamento**, vários espaços consecutivos vão ser tratados como um único espaço. Por outro lado, várias linhas vazias são tratadas apenas como uma linha vazia que, por sua vez, é entendida em LaTeX como sinalização de um novo parágrafo. Alternativamente, podemos também usar o comando \par para criar um novo parágrafo.

Q8.: Só para que fique bem sólido na memória: como é que se se pode começar um parágrafo novo em LaTeX?
☐ Começando o novo parágrafo numa linha nova.
☐ Deixando uma linha em branco antes desse novo parágrafo.
☐ **Deixando dois espaços no início do texto desse novo parágrafo.**

Em geral, o LaTeX ignora linhas em branco e outros espaços vazios. Por fim, duas barras invertidas (\\) podem ser utilizadas para iniciar uma nova linha (quebra de linha) **sem gerar um novo parágrafo**.

Tarefa 6

Insira as frases seguintes num ficheiro `.tex`:

```
Primeira frase.
Segunda frase.
```

Coloque cada frase em sua linha e adicione o comando \par a seguir à primeira linha. No final, `compile` e teste.

Q9.: Na tarefa anterior a segunda frase começou num novo parágrafo?
☐ Não sei bem... Como se vê isso?
☐ Sim, porque li ali atrás que é isso que o comando faz!
☐ Sim, pois a frase começou com um espaço em relação à margem esquerda.
☐ Sim, porque o colega do lado respondeu assim.

Tarefa 7

Altere o exemplo anterior, mas substituindo o comando \par por \\, e `compile` .
Q10.: Notou alguma diferença, relativamente ao resultado final, entre o uso de \\ e \par?
☐ Não notei nada. O resultado é exatamente o mesmo.
☐ Sim, notei. Usando \\ parece que ficou com mais espaço em relação à margem esquerda.
☐ Sim notei. Usando \\ parece que ficou mais próximo da margem esquerda.

6 Caracteres Especiais

O LaTeX dispõe de um conjunto de caracteres reservados com significados especiais. Um exemplo destes caracteres é o símbolo de comentário, %, referido na secção anterior. Os caracteres seguintes são outros exemplos de caracteres reservados em LaTeX:

```
# $ % ^ & _ { } ~ \
```

Todos estes caracteres, **com exceção da barra invertida**, podem ser adicionados ao texto (e serem visíveis na versão final do documento) se adicionarmos uma barra invertida (\) imediatamente antes do respetivo caractere, conforme se ilustra a seguir:

```
\# \$ \% \^{} \& \_ \{ \} \~
```

Resta a pergunta: **Q11.: como proceder se quisermos adicionar uma barra invertida e quisermos que esta apareça na versão final do documento, após compilação?**

☐ Não dá. É um daqueles mistérios que a Humanidade ainda não conseguiu resolver...
☐ Seguindo a lógica anterior, basta colocar \\.
☐ Colocando \, seguindo de um espaço, e depois outra \.
☐ Usando o comando \textbackslash.

Tarefa 8

Teste a sua resposta à questão anterior, só para ter a certeza...

Relativamente a exceções, existem ainda outros dois casos especiais. No caso dos acentos circunflexo e til (^, ~) é necessário adicionar, também, {} após o acento; caso não o faça, o acento será colocado sobre o caractere seguinte.

7 Tabelas

As tabelas são dos recursos mais interessantes para estruturação de informação em documentos técnicos. Em LaTeX podemos construir tabelas utilizando o ambiente `tabular`. O excerto de código seguinte exibe um exemplo de uma tabela:

```
\begin{tabular}{l | r | c}
\textbf{Nome} & \textbf{Idade} & \textbf{Naturalidade}\\
    % \hline
    Ana        & 40             & Lisboa\\
\end{tabular}
```

Q12.: Visto que já está no nível de *padawan* de LaTeX, e só de vista, quantas colunas é que acha que a tabela representada no código anterior tem?
☐ zero ☐ uma ☐ duas ☐ três ☐ quatro ☐ meia dúzia

Q13.: E quantas linhas?
☐ zero ☐ uma ☐ duas ☐ três ☐ quatro ☐ meia dúzia

Tarefa 9

Implemente o código exibido anteriormente no ficheiro `.tex` do seu projeto. `Compile` e verifique novamente as respostas dadas anteriormente, só para se certificar que respondeu corretamente. Adicionalmente, considere a questão seguinte:

Q14.: O que pode comentar, relativamente ao alinhamento das células da tabela?
☐ Alinhamento das células?! O que é isso?
☐ As células da primeira coluna estão alinhadas à esquerda, as da segunda à direita e as da terceira ao centro.
☐ As células da primeira coluna estão alinhadas à direita, as da segunda à esquerda e as da terceira ao centro.
☐ Todas as células de todas as colunas estão centradas.

Analisando o resultado da tarefa anterior podem-se extrair as seguintes conclusões:

- O número das colunas e o alinhamento horizontal são definidos pelos caracteres **l**, **r**, e **c**. Neste caso existem **três caracteres** a definir o alinhamento, o que originará uma tabela com **três colunas**.
- A tabela contém três colunas com alinhamentos à **esquerda** (***l****eft*), **direita** (***r****ight*) e ao **centro** (***c****enter*).
- As linhas da tabela são separadas utilizando `\\`.
- Em cada linha, o conteúdo das colunas é separado por um &.

Retire o % que precede o comando `\hline` no excerto de código anterior e `compile` .

Q15.: Qual é a finalidade do comando `\hline`?
☐ Não sei, mas o meu sentido aranha indica que irá gerar uma linha.
☐ Gera uma linha horizontal entre o conteúdo da tabela e o cabeçalho.
☐ Gera uma linha vertical entre o conteúdo da tabela e o cabeçalho.
☐ Nem o *Google* sabe!

Q16.: À semelhança do índice, demonstrado na aula anterior, qual é o comando para gerar uma lista de tabelas?
☐ Não é possível definir uma lista de tabelas no LaTeX.

☐ Esta é óbvia, é o comando `\tableofcontents`.
☐ Cheguei à conclusão, após cuidada deliberação, que é o comando `\listoftables`.
☐ Nenhuma das opções anteriores.

Tarefa 10

Utilize o comando para gerar a lista de tabelas e `compile` .

Observe a lista de tabelas gerada. **Q17.: É verdade que a lista está vazia?**
☐ Não! Está lá com a tabela que acabei de criar.
☐ Está vazia... Será que usei o comando certo?

Tarefa 11

Modifique o excerto de código da tabela de modo a que o ambiente `tabular` fique envolvido num ambiente `table` e insira o comando `\caption{tabela1}` entre `\end{tabular}` e `\end{table}`. No final, não se esqueça de `compilar` .

Observe, novamente, a lista de tabelas gerada. **Q18.: A lista continua vazia?**
☐ Parece que sim...
☐ Não! Agora já está lá com o nome `tabela1` e indica em que página está e tudo! Até as galhinhas dizem: imp, imp, impecável!

Na tarefa anterior fez uso da funcionalidade de inserção de legendas que o LaTeX disponibiliza. Para tal, fez uso do comando `\caption{nome_tabela}`, o que permitiu o aparecimento de uma linha na lista de tabelas (referente à tabela criada) e, adicionalmente, a exibição de uma legenda abaixo da tabela. Repare que não foi necessário a adição de um número de tabela, sendo esta informação preenchida automaticamente ao adicionar o comando acima referido.

8 Figuras

As figuras são outro dos recursos mais interessantes no contexto de uma explicação. Em LaTeX as figuras são incluídas utilizando o comando `\includegraphics`. Este comando encontra-se definido no pacote `graphicx`, pelo que este pacote deve ser sempre adicionado ao preâmbulo do documento quando se querem usar figuras.

Tarefa 12

Considere o excerto de código seguinte:

```
\usepackage{graphicx} % No Preâmbulo
\begin{document}
\includegraphics[scale=0.5]{ubi_logo.jpg}
\end{document}
```

Crie um novo ficheiro com a estrutura básica de um documento LaTeX e integre o código exibido anteriormente. Antes de compilar, obtenha o logótipo da UBI (*e.g.* https://www.ubi.pt/Ficheiros/Menus/1/logo_ubi_vprincipal.jpg) e insira a imagem no projecto (usando o comando *upload* do *Overleaf*). Mude o nome do ficheiro do logótipo para `ubi_logo` e `compile` .

Q19.: O que tem a dizer relativamente ao extrato de instrução `[scale=0.5]`**?**
☐ Não parece fazer nada.
☐ Permite reduzir o tamanho da figura para 50% do seu tamanho original.
☐ Permite fazer uma imagem à escala de 0 para 5 para o mundo real.
☐ Permite aumentar o tamanho da figura para 50% do seu tamanho original.
☐ Não é obrigatório.

Todo o texto que se encontrar entre [ ] refere-se às opções que podem ser passadas ao comando `\includegraphics`. Na tarefa anterior, foi passada informação relativamente ao tamanho da figura. No entanto, pode também, adicionalmente, especificar a largura e altura de uma figura por esta via.

Q20.: É possível gerar uma lista de figuras?
☐ Tal empreitada não é possível em LaTeX.
☐ Basta utilizar o comando `\tableofcontents` duas vezes.
☐ Após uma pesquisa rápida, cheguei à conclusão que é o comando `\listoffigures`.
☐ Nenhuma das opções anteriores.

Tarefa 13

Altere o ficheiro anterior, introduzindo o comando certo para gerar a lista de figuras, e `compile` .

Observe a lista de figuras gerada. **Q21.: A lista encontra-se vazia?**
☐ Não. Está lá o logótipo da UBI, tal como esperado.
☐ Está vazia, sim, e deduzo (porque tenho um tio que ainda é arraçado do *Sherlock Holmes*) que é preciso fazer algo semelhante ao que foi feito para as tabelas...

Tarefa 14

Modifique o ficheiro de modo a que `\includegraphics[scale=0.5]{ubi_logo.jpg}` fique envolvido num ambiente `figure` e insira a instrução

```
\caption{Logótipo da UBI.}
```

entre `\includegraphics[scale=0.5]{ubi_logo.jpg}` e `\end{figure}`. `Compile` para verificar o resultado.

Observe, novamente, a lista de figuras gerada. **Q22.: A lista continua vazia?**

- ☐ Sim... Não mudou nada!
- ☐ Já não está vazia. Consigo ver a legenda na lista e a indicação da página onde a figura está.
- ☐ Já não está vazia, mas também não está cheia. É uma questão de copo meio cheio ou meio vazio...

Na tarefa anterior fez, novamente, uso da funcionalidade de inserção de legendas (via `\caption{nome_figura}`). A sua utilização e funcionamento em figuras são análogos à utilização e funcionamento em tabelas.

9 Equações

Uma das principais razões para escrever documentos em LaTeX é a sua aplicabilidade na escrita simples e de elevada qualidade de equações matemáticas (muito útil em documentos técnicos e científicos). As equações podem ser escritas em linha (*i.e.*, em contexto com o resto de uma frase) ou em destaque. As fórmulas ou equações matemáticas podem ser introduzidas ao longo do texto, usando o caractere $ no início e fim da fórmula, ou em destaque utilizando o ambiente `equation` (*i.e.*, modo matemático). Observe o excerto de código seguinte:

```
A equação
\begin{equation}
 E=mc^2
\end{equation}
determina a relação da transformação da massa de um objeto em energia e
    vice-versa.
```

Tarefa 15

Integre no corpo de um ficheiro `.tex` o trecho exibido anteriormente. De seguida, crie uma segunda réplica desse excerto, mas substitua o ambiente `\begin{equation}` por `\[` e `\end{equation}` por `\]`. No final, `compile` e analise o resultado.

Q23.: Qual é a diferença entre os dois excertos de códigos criados na tarefa anterior?

- ☐ Eu diria que nenhuma.

☐ Ambos exibem a fórmula, em destaque, mas o ambiente `equation` exibe, adicionalmente, um número junto da equação.

Complementando a informação anteriormente exposta, a exibição de equações ou fórmulas matemáticas, em destaque, tem duas versões: **numerada e não numerada**. Tal como observado na tarefa anterior, para apresentar uma equação de forma numerada deve utilizar o ambiente `equation` enquanto que a forma não numerada é conseguida usando `\[` no início e `\]` no fim da equação.

10 Referências Bibliográficas

É possível produzir a bibliografia de um documento utilizando um ambiente dedicado (`thebibliography`) ou, alternativamente, utilizando um ficheiro externo (`.bib`) que contém todas as referências bibliográficas. Caso se opte pelo uso de um ficheiro externo deve-se incluir este no ficheiro principal `.tex`, dentro do ambiente `document`, usando o comando seguinte: `\bibliography{nome_ficheiro_bibliografia}`. Esta parte do guia apenas se foca na utilização de um ficheiro externo para bibliografia.

Para usar uma referência bibliográfica (do ficheiro externo) é sempre necessário etiquetar a referência bibliográfica e depois utilizar o comando `\cite{etiqueta}`, no documento principal `.tex`.

Tarefa 16

Para perceber melhor esta forma de funcionamento, crie um novo ficheiro `.tex` e integre adequadamente (*i.e.*, no lugar certo) o trecho seguinte:

```
\begin{document}
Conhece o conceito de auto-plágio? Existem artigos \cite{self-plagiarism}
    que o podem ajudar a perceber melhor.
\bibliography{biblio}
\end{document}
```

No final, `compile` .

Q24.: O excerto de código foi compilado com sucesso?

☐ Não, deu imensos erros quando tentei compilar.

☐ Compilou, mas deu-me uns avisos e apareceu-me um ponto de interrogação ali no meio do texto... Será que o LaTeX quer falar comigo?

O excerto de código acima está ainda incompleto. O que falta é o ficheiro com a bibliografia (designado por `biblio.bib`). Por essa razão, o número da referência invocada não apareceu na tarefa anterior sendo colocado, em sua substituição, um ponto de interrogação.

Tarefa 17

Obtenha um ficheiro `.bib` usando esta *hiperligação*. De seguida, introduza este novo ficheiro no projeto, ao lado do ficheiro principal `.tex`. No fim, `compile` .

Q25.: Com a introdução deste novo ficheiro, o excerto de código foi compilado com sucesso?

☐ Agora sim!

☐ Não parece ter mudado nada. Continuo com o ponto de interrogação ali no meio do texto...

Falta ainda algo para que tenhamos um documento com referências. No entanto, observe os seguintes aspectos do excerto de código e das tarefas anteriormente realizadas:

- O texto, entre chavetas, do comando `\bibliography` é idêntico ao nome do ficheiro externo de referências (`biblio.bib`), ignorando a extensão `.bib` deste;
- O texto, entre chavetas, do comando `\cite` tem uma correspondência com um nome de uma referência, dentro de `biblio.bib`. Pode ver o nome da referência na linha 2, no texto, entre chavetas, imediatamente à frente de `@article` . O texto `self-plagiarism` é a etiqueta.

Tarefa 18

Introduza o comando `\bibliographystyle{plain}`, imediatamente acima do comando `\bibliography{biblio}`, e `compile` .

Q26.: Com mais esta mudança, o excerto de código foi (finalmente) compilado com sucesso?

☐ Ainda não foi desta, chiça. E tenho de abandonar, porque tenho uma consulta às 5...

☐ Sim! O ponto de interrogação foi substituído por [1] e já aparece uma secção de referências.

Para produzir bibliografia num documento necessitamos de definir o estilo a utilizar. Por essa razão, utilizamos o comando `\bibliographystyle{plain}` na tarefa anterior. O estilo aqui utilizado foi `plain` mas existem muitos mais [1].

[1] https://www.overleaf.com/learn/latex/bibtex_bibliography_styles/

10 Elaboração de Relatórios Técnicos Usando LaTeX

Sumário

Abordagem à elaboração de relatórios técnicos através de exercícios que procuram focar na sua estruturação e aprimoramento. Alguns exercícios continuam a recorrer ao LaTeX.

Summary

Approaching the preparation of technical reports via exercises that focus on structuring and finetunning aspects. Some of the exercises resort to LaTeX.

Pré-requisitos:

Algumas das tarefas propostas a seguir requerem o acesso a um sistema com acesso à Internet e uma conta na plataforma ***Overleaf*** - *The Online LaTeX Editor*. Se optar por instalar o LaTeX no seu sistema sugere-se, o uso de uma distribuição comum de Linux, onde todas estas condições estarão provavelmente preenchidas.

1 Estrutura de Relatórios Técnicos

A estruturação de um relatório em diferentes capítulos e/ou secções ajuda à sua organização e escrita, por parte dos autores, e, de igual modo, facilita ao leitor a procura de informação no documento. Consulte os apontamentos da aula teórica relativamente à organização do documento.

Só para verificar se estava atento na aula teórica: **Q1.: onde é que deve aparecer o capítulo da *Engenharia de Software*?**

☐ Antes da Introdução, mas depois do Resumo.
☐ Ainda antes da capa!

☐ Depois da Introdução e antes do Estado da Arte.
☐ Antes da Implementação e depois do Estado da Arte.
☐ Depois da Implementação e antes do Estado da Arte.
☐ Por via das dúvidas, bota-se em tudo quanto é lado... e fica resolvido.

Tarefa 1

Assumindo que queremos um documento com os capítulos `Introdução`, `Implementação`, `Conclusão e Trabalho Futuro`, `Engenharia de Software` e `Estado da Arte`, crie um ficheiro `.tex` e coloque os capítulos pela ordem que achar mais adequada. No final, não se esqueça de `compilar` e testar.

Q2.: Relativamente à secção de `Estado da Arte`, podemos citar os seguintes recursos para realizar a pesquisa bibliográfica (selecione todas as que se apliquem):
☐ Artigos científicos em revista.
☐ Artigos científicos em conferência.
☐ Wikipédia.
☐ Livros.
☐ Teses e dissertações.
☐ Relatórios técnicos reconhecidos.
☐ Os meus apontamentos do 12º ano.

Questão bónus → esta sim, não é para todos(as)!
Considere o cenário em que está a ter dificuldades em termos sintáticos e/ou de concordância na redação do seu documento técnico. **Q3.: Que ferramenta(s) pode(m) ajudar nestes aspectos?**
☐ Não há salvação possível. Devia ter estado mais atento nas aulas de Português...
☐ Conheço uma que se chama *LanguageTool*.
☐ *Expresso App* (mas não é a do jornal).
☐ Conheço a ferramenta ideal: *PorFavorCorrigeOMeuDocumento*.

2 Palavras-Chave

As palavras-chave de um documento devem ser relevantes ao tema abordado neste. As palavras escolhidas serão usadas para procurar o trabalho num motor de busca, pelo que é importante uma escolha pertinente das mesmas.

Considere que estava a ler um relatório que identificava as palavras-chaves seguintes, exatamente como se ilustra:

```
Keywords -  Ensino / Universidade / Avaliação Automática
```

Q4.: O que pode comentar relativamente ao uso das palavras-chave?
☐ Estão usadas de forma correta.
☐ Que o jogo vai ser difícil, mas a equipa das palavras-chave tem hipóteses de ganhar.
☐ A separação das palavras-chave não está bem. Deveria ter sido usado vírgula ou ponto-e-vírgula.
☐ As palavras-chave não estão por ordem alfabética.
☐ As palavras-chave deviam estar todas em minúscula.

Tendo em conta que as palavras-chave serão usadas para identificar o documento aquando da sua procura por um motor de busca, poder-se-ia pensar que usar um elevado número de palavras-chave seria uma opção a considerar. **Q5.: O que acha da afirmação anterior?**
☐ Acho muito bem! Quantas mais palavras-chave melhor. Se possível até usamos umas que não se relacionam com o tema.
☐ Um uso equilibrado de palavras-chave é recomendado, dependendo também da quantidade de temas relevantes abordados no documento. Em caso de dúvida, entre três a cinco seria uma boa estimativa.
☐ Se pudesse nem colocava nenhuma.
☐ Palavras-chave? Isso é para abrir cofres?

3 Acrónimos

Os acrónimos são vocábulos formados com as letras ou sílabas iniciais de uma sequência de palavra. São, tipicamente, usados na escrita de um documento técnico com o intuito de evitar o uso repetitivo de terminologias longas. A escolha de quais os acrónimos a utilizar deve ser feita de forma cuidada e a sua definição no texto deve visar a completa compreensão por parte do leitor.

Relembre alguns conceitos abordados na aula teórica relativamente a este tema. **Q6.: Existe algum critério de ordenação para os acrónimos?**
☐ Não existe nenhum critério específico.
☐ Sim, devem estar ordenados por ordem alfabética.
☐ Sim, devem estar ordenados por ordem de aparecimento no documento.
☐ Sim, devem estar ordenados por tamanho de acrónimo.

Tarefa 2

Crie um novo ficheiro `.tex` e escreva nele o seguinte texto

```
A UBI é uma universidade na Covilhã. A UBI é constituída por diversos pó-
los e departamentos. A Universidade da Beira Interior começou por ser um
```

```
politécnico, em 1973. Em 1986, a UBI ficou reconhecida como universidade.
```

Faça as alterações que achar necessárias para que o documento apresente um uso correto de acrónimos e `compile` .

Q7.: Relativamente à tarefa anterior, assinale todos os erros que encontrou:
☐ Acrónimo não foi definido por extenso na primeira utilização.
☐ Uso em demasia de acrónimo.
☐ Utilização de acrónimo por extenso quando já havia sido definido.
☐ Utilização de acrónimos sempre por extenso.

Para utilizar acrónimos em LaTeX deverá incluir um pacote, usando a seguinte sintaxe: `\usepackage[opções]{Nome_pacote}`.

Q8.: Qual é o nome do pacote para utilizar acrónimos?
☐ Chama-se `acrónimos`, de certeza!
☐ Na aula teórica o Professor disse que era `acronym`.
☐ O nome do pacote é `acro`.
☐ Chicha penico! Essa pergunta é muito difícil.

Tarefa 3

Para utilizar um acrónimo no texto deverá primeiro definir um ambiente `acronym` e introduzir o acrónimo a utilizar, dentro do ambiente, usando a seguinte sintaxe:

```
\begin{acronym}[MAIOR_ACRONIMO]
  \acro{acronimo}{acrónimo_por_extenso}
\end{acronym}
```

Defina o acrónimo `UBI`, com base no que foi referido anteriormente, no documento `.tex` da tarefa anterior. No final, `compile` e verifique o resultado.

Q9.: O que colocou como segundo argumento no comando `\acro`?
☐ *Union Bank of India*.
☐ `acrónimo_por_extenso`
☐ Universidade da Beira Interior.
☐ UBI.
☐ Acrónimo.

Definido o acrónimo, este deverá ser utilizado no texto fazendo uso do comando `\ac{acrónimo}`.

Tarefa 4

Usando o mesmo ficheiro `.tex` da tarefa anterior, efetue as alterações necessárias para que todas as instâncias `UBI` e `Universidade da Beira Interior` façam uso do acrónimo

definido e `compile` .

Q10.: O que apareceu no documento final compilado no local onde o acrónimo é utilizado pela primeira vez?

☐ Nada.
☐ Apareceu um ponto de interrogação.
☐ Universidade da Beira Interior (UBI).
☐ UBI.
☐ Universidade da Beira Interior.
☐ (UBI) Universidade da Beira Interior.

4 Capítulos

Os documentos técnicos fazem uso de capítulos, visando manter o seu conteúdo organizado. Dentro de capítulos, em geral, existem secções. Uma correta definição de títulos de capítulos, bem como de secções, passa pelo uso de capitalização coerente, ao longo de todo o documento.

Tarefa 5

Considere o trecho de código seguinte:

```
\begin{document}
\chapter{introdução}
\chapter{Detalhes De Implementação}
\chapter{estado Da Arte}
\chapter{CONCLUSÃO}
\end{document}
```

Crie um novo ficheiro com a estrutura básica de um documento LaTeX e integre o código exibido anteriormente. Promova as alterações necessárias para que o documente apresente uma capitalização coerente de capítulos. Por fim, `compile` .

Considere as alterações que realizou na tarefa anterior. **Q11.: Houve algum capítulo em que não teve que realizar nenhuma alteração, ou seja, já se encontrava corretamente capitalizado?**

☐ Não, tive que fazer alterações a todos os capítulos.
☐ Sim, o capítulo `Detalhes de Implementação` já tinha uma capitalização coerente.

Só para ter a certeza que ficou tudo percebido relativamente à ordenação de capítulos de um documento técnico. **Q12.: O que pode comentar relativamente à ordenação dos capítulos na tarefa anterior?**

☐ Para ficar corretamente ordenado, `Detalhes De Implementação` teria que trocar com `CONCLUSÃO`.
☐ Para ficar corretamente ordenado, `Detalhes De Implementação` teria que trocar com `estado Da Arte`.

☐ Para ficar corretamente ordenado, `introdução` teria que trocar com `CONCLUSÃO`.
☐ A ordenação está correta.

Tarefa 6

Com base nos comentários de aplicação geral da aula teórica, introduza as secções `Introdução` e `Conclusão` nos capítulos que achar conveniente e `compile`.

Q13.: Relativamente à tarefa anterior, em quantos capítulos introduziu as secções referidas?
☐ $-\sqrt{2}$ ☐ zero ☐ um ☐ dois ☐ três ☐ quatro

5 Figuras, Tabelas e Trechos de Código

Os recursos para além do texto, como, por exemplo, figuras, tabelas e trechos de código são elementos importantes para auxiliar o leitor na compreensão de um tema. A sua utilização deve ser pertinente e ter qualidade adequada.

Considere o trecho de código seguinte e a respetiva legenda:

```
#include <stdio.h> int main() {
char ch;
printf("Escreva um char:");scanf("%c",&ch);
printf("O seu char = %c\n",ch);return 0;}
```

Listing 10.1: Programa em C que pede um caractere ao utilizador e apresenta o valor introduzido no ecrã

Q14.: O que pode dizer da qualidade do trecho de código apresentado?
☐ A qualidade é muito boa porque tem cores bonitas.
☐ A qualidade é adequada porque encontra-se devidamente indentado e espaçado.
☐ A qualidade não é adequada porque tem uma paupérrima indentação e espaçamento.
☐ Sei lá, nem sei ler C...

Tarefa 7

Crie um novo documento `.tex`, introduza o trecho de código anterior e proceda a todas as alterações necessárias para que este fique devidamente indentado e espaçado. No final, `compile`.

Observe, atentamente, a legenda do trecho de código anterior. **Q15.: Tendo em conta os comentários de aplicação geral da aula teórica, o que tem a dizer da**

legenda apresentada?
☐ A legenda devia estar em itálico e não está.
☐ A descrição da legenda não é concordante com o trecho de código apresentado.
☐ A legenda diz que o programa é C mas eu sei muito bem que isto é Java!
☐ A legenda é auto-contida, descreve fielmente o conteúdo do trecho de código associado mas não termina com um ponto final.

Recorde também que, caso o trecho de código anterior exceda a meia página de documento, este deve ser inserido em *Anexo*.

Tarefa 8

Considere, novamente os comentários de aplicação geral da aula teórica, e tome especial atenção à referência de elementos no relatório.

Obtenha um ficheiro `.zip` usando esta hiperligação. Faça `unzip` e confirme a existência de um ficheiro `.tex` e de uma figura com extensão `.png`. Crie um novo projeto, no *Overleaf*, e coloque ambos os ficheiros no projeto. Por fim, `compile` .

Q16.: Relativamente à tarefa anterior, todas as figuras e tabelas são referidas, pelo menos uma vez, no texto?
☐ Sim, existe lá uma figura e uma tabela por isso assumo que esteja tudo bem.
☐ Não, falta referir a figura no texto.
☐ Não, falta referir a tabela no texto.
☐ Não, nenhum dos elementos é referido no texto.

Tarefa 9

Promova as alterações necessárias no documento da tarefa anterior para que todos os elementos sejam, de facto, referidos no texto. No fim, `compile` e certifique-se de que está tudo bem.

6 Estilização de Documento

Um documento técnico pode conter diferentes conceitos que necessitam de uma estilização adequada. A título de exemplo, o uso de termos ou referências a códigos devem estar em `monospace`. Neste sentido, considere a seguinte frase:

```
Um upgrade à aplicação ocorrerá em 3 dias.
```

Q17.: Quantos erros se encontram na frase anterior?
☐ zero ☐ um ☐ dois ☐ três ☐ -∞

Tarefa 10

Registe todas as alterações que faria na frase anterior visando uma estilização correta.

No caso de ter dificuldades na interpretação de palavras, em Língua Portuguesa, ou tenha dúvidas sobre que palavras são estrangeirismos, pode recorrer ao dicionário *online* da Porto Editora[1].

[1] http://www.priberam.pt/DLPO/

11 Introdução ao Ambiente de Linha de Comandos

Sumário
Introdução ao ambiente de linha de comandos, com apresentação de principais comandos e funcionalidades.

Summary
Introduction to the command line environment with presentation of the main commands and features.

Pré-requisitos:
Algumas das tarefas propostas a seguir requerem o acesso à Internet e acesso a um **SO Linux**. Se não pretender instalar uma distribuição de Linux na sua máquina pode sempre optar por instalar a distribuição numa máquina virtual. O uso de Subsistema Windows para Linux (*Windows Subsystem for Linux*), para Windows 10, concretiza também uma opção válida.

1 Ambiente de Linha de Comandos

O ambiente de linha de comandos é identificado como uma forma de interagir com o SO, sendo esta interação feita, essencialmente, através de interfaces baseadas em texto.

Em vários SOs com interface gráfica é possível aceder ao ambiente de linha de comandos através de um programa denominado, no caso de SO Unix ou Unix-*like*, `terminal`. **Q1.: Qual é o nome deste programa em Windows?**

☐ Tem o mesmo nome: `terminal`.
☐ Consola.
☐ Janela.
☐ *Notepad*.
☐ *Fortnito*.

Todas as tarefas e questões subsequentes neste documento assumem que é utilizado um

SO Unix ou Unix-*like*.

Tarefa 1

Aceda ao seu ambiente de linha de comandos através de um dos programas disponíveis no seu SO.

Q2.: Análise a sua *prompt* e identifique o seu *user* e *hostname*:

User: _____

Hostname: _____

Q3.: Qual o caractere que se encontra entre o *user* e *hostname*, na sua *prompt*?
☐ `@` ☐ `:` ☐ ∼ ☐ `$` ☐ `#` ☐ `>`

Q4.: Qual o caractere que, numa *prompt* de um sistema Unix ou Unix-*like* pode denotar *utilizador com privilégios de administração*?
☐ `@` ☐ `:` ☐ ∼ ☐ `$` ☐ `#` ☐ α

2 O Sistema de Ficheiros

O sistema de ficheiros é composto pelo conjunto de métodos e estruturas de dados que o SO utiliza para gerir, controlar e indexar todos os ficheiros que se encontram num disco ou partição. Tendo em conta os apontamentos das aulas teóricas responda às seguintes questões:

Q5.: O que é considerado um ficheiro (selecione todas as que se apliquem)?
☐ Diretoria ☐ Teclado ☐ Memória ☐ Caixa de PC ☐ Rato
☐ Impressora ☐ Monitor ☐ Ficheiro ☐ Disco inteiro

Q6.: No ambiente de linha de comandos, quantos e quais são os fluxos de dados que interessa conhecer?
☐ São 2: `stdin` *(standard input),* `stdout` *(standard output).*
☐ São 3: `stdin` *(standard input)*, `stdout` *(standard output)* e `stderr` *(standard error).*
☐ São 4: `stdin` *(standard input),* `stdout` *(standard output),* `stderr` *(standard error)* e `stdnull` *(standard nullifier).*
☐ São demasiados para serem quantificáveis. Diria que para cima de 500... mas à vontadinha!

3 Ajuda, Comentários e Ecos em *Bash*

O ambiente de linha de comandos permite a utilização de diversos comandos, usáveis para diferentes fins. Uma dessas finalidades é a obtenção de ajuda acerca de outros comandos.

Q7.: Qual o comando que permite obter um manual de um determinado comando?
☐ `man` ☐ `manual` ☐ `manuel` ☐ `sos` ☐ `help`

Tarefa 2

Aceda ao seu ambiente de linha de comandos e execute o comando para mostrar o manual do comando `echo`.

Q8.: O que observou ao executar o comando da tarefa anterior?
☐ Não observei nada ou o que observei foi que não aconteceu nada.
☐ O terminal ficou preenchido com informação, aparentemente dividida por secções de *name*, *synopsis* e *description*...
☐ O terminal ficou com uma linha vazia e voltou a aparecer o *prompt* abaixo dessa linha.
☐ O terminal só imprime `echo`, `echo`, `echo`. Parece que fui "hackeado"!

Analise, detalhadamente, o manual exibido: **Q9.: o que faz o comando `echo`?**
☐ Escreve no terminal o que eu gritar para o teclado, como se fosse um eco.
☐ Segundo o manual, exibe uma linha de texto, no terminal.

Q10.: Há alguma opção do comando `echo` que permita imprimir no terminal, mas evitar que se mude de linha no final de imprimir?
☐ Não, não há... mas é pena não haver!
☐ Não, não há... Eu não procurei no manual, mas já sei que não há!
☐ Sim, há. É a opção `-n`.
☐ Sim, há. É a opção `-e`.

Q11.: Qual o último número da versão do comando `man` no seu sistema?
☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9

Utilizando o comando `echo` é possível fazer eco de um texto para uma *stream* chamada *standard output* (`stdout`).

À semelhança do LaTeX é possível colocar comentários no ambiente de linhas de comandos utilizando o caractere `#`.

Tarefa 3

Coloque um comentário descritivo do `echo` na sua linha de comandos utilizando o caractere `#`, *e.g.*, `# O comando echo não parece fazer nada de jeito...` Quando terminar clique `Enter`.

Q12.: O que observou ao executar o comando da tarefa anterior?
☐ Ficou lá o comentário e apareceu um novo *prompt*, pronto a receber *input*. Mas não aconteceu mais nada...
☐ O terminal deu um erro e diz que não consegue interpretar o comando inserido. Diz: `command not found`!!

Tarefa 4

No seu ambiente de linha de comandos, utilize o comando `echo` para fazer eco do seguinte texto no terminal: `Olá Mundo`.

Q13.: No contexto do comando em que foi utilizada, o que constitui a expressão `Olá Mundo`?
☐ Constitui o nome do próprio comando.
☐ Constitui um parâmetro não obrigatório do comando.
☐ Constitui uma opção do comando.
☐ Constitui uma *frase feita*, muito usada no contexto da informática, mas que nem faz assim muito sentido...

4 Criação e Navegação em Directorias

A utilização da linha de comandos torna possível a navegação e manipulação de ficheiros e/ou diretorias do seu SO de uma maneira muito ágil. Quando acede ao ambiente de linha de comandos, encontra-se, tipicamente, na diretoria predefinida para o utilizador atual.

Tarefa 5

No ambiente de linha de comandos, identifique qual é a sua diretoria atual. Utilize um comando que imprima o caminho completo.

Q14.: Qual foi o comando que utilizou para imprimir o caminho completo da diretoria atual?
☐ Utilizei o comando `pwd`.

☐ Utilizei o comando `pwdir`.
☐ Utilizei o comando `%cd%`.
☐ Utilizei o comando `printer`, uma vez que é para imprimir.

Tarefa 6

Liste todas as sub-directorias e ficheiros da sua diretoria atual.

Q15.: Qual foi o comando que utilizou na tarefa anterior?
☐ Utilizei o comando `ls`.
☐ Utilizei o comando `dir`.
☐ Utilizei o comando `please, list`.
☐ Continuo sem entender, o que é para fazer exatamente?

Q16.: O que significa a letra `w` do comando `pwd`?
Note que esta pergunta está propositadamente desfasada da tarefa respetiva para precisamente permitir que responda às perguntas sem interferência e vislumbre da resposta.

Q17.: O que pode comentar relativamente à opção `-h` do comando `ls`?
☐ Não tenho opinião formada nesse assunto.
☐ Lista as informações dos ficheiros num formato que facilita a leitura humana como, por exemplo, 1K, 234M, 2G, etc.
☐ Tem que ser usada em conjunto com outras opções, nomeadamente, `-l` ou `-s`.
☐ Procurei no manual e essa opção não existe. O docente só cá meteu esta questão para ver se eu estava com atenção! Macaco!

Tarefa 7

No seu ambiente de linha de comandos, utilize o comando `mkdir` para criar a diretoria `Teste`.
Q18.: Qual é o comando que pode utilizar para navegar da sua diretoria atual para a nova diretoria `teste`?.
☐ Estudei arduamente em preparação para esta aula e, neste momento, transbordo de tal maneira de conhecimento que até as minhas vísceras me dizem que é o comando `cd teste`.
☐ Estive a ler as aulas teóricas e, sem dúvida, que é o comando `enter teste`.
☐ É o comando `cd ..`.
☐ É o comando `bamo lá pó teste`.

Admita agora que pretende remover a diretoria criada. **Q19.: Qual é o comando que sugere utilizar?**
☐ Sugiro usar o comando `rm`.
☐ Sugiro usar o comando `rm -r`, no entanto não sei para que serve a opção `-r`.
☐ Dada a minha *expertise* na leitura avançada de manuais, sugeria usar o comando `rmdir`, visto a diretoria estar vazia, mas quem sou eu para ser levad@ em consideação?
☐ Sugeria não usar nada, até porque sou uma pessoa mais orientada para o curtido e as ações destrutivas não são comigo...

Tarefa 8

Remova a diretoria que criou na tarefa anterior, utilizando o comando que achar mais conveniente.

Q20.: Conseguiu remover a diretoria `teste`?
☐ Huh... O terminal diz que a diretoria não existe. Será que não a criei corretamente?
☐ Consegui porque me apercebi que tinha que sair da diretoria para a eliminar.

Caso tenha dúvidas de como regressar à diretoria anterior, consulte os apontamentos da aula teórica.

5 Criação e Edição de Ficheiros

A criação e edição de ficheiros é outra funcionalidade possível num ambiente de linha de comandos.

Tarefa 9

No ambiente de linha de comandos, crie um ficheiro vazio chamado `vazio.txt`.

Q21.: Qual foi o comando que utilizou para criar o ficheiro na tarefa anterior?
☐ `touch` ☐ `create` ☐ `mkfile` ☐ `forge`

Q22.: Tem a certeza de que o ficheiro está mesmo vazio?
☐ Claro que tenho... Quer dizer, eu acho que está vazio, mas tenho medo de o abrir para verificar, não vá ele ficar menos vazio por causa disso...
☐ Tem toda a razão. Devo verificar se, de facto, o ficheiro está vazio, eventualmente com a instrução `ls -a`.

☐ Tem toda a razão. Devo verificar se, de facto, o ficheiro está vazio, eventualmente coma instrução `ls -l`.

Tarefa 10

Utilize o editor de texto orientado para a linha de comandos `nano` para adicionar o texto `"Ficheiro criado via CLI"` ao ficheiro criado anteriormente. De seguida, grave o ficheiro. Findo esta parte, deverá ficar com a *prompt* novamente visível.

Q23.: De que maneira poderia imprimir o conteúdo do ficheiro, criado na tarefa anterior, no terminal?

☐ O comando `nano vazio.txt` poderia ser uma opção.
☐ A opção de usar o comando `show vazio.txt` é a melhor.
☐ Usando o comando `cat vazio.txt`.
☐ Usando o comando `cat cheio.txt`, visto o ficheiro já não estar vazio.
☐ Usando a ajuda do público, e optando de seguida pela segunda opção.

Tarefa 11

Partindo da diretoria onde criou o ficheiro `vazio.txt`, crie uma nova sub-diretoria, denominada `nova_dir`, e mova o ficheiro `vazio.txt` para esta.

Q24.: Relativamente a movimentação do ficheiro, que comando utilizou para realizar a tarefa anterior?

☐ `mv vazio.txt nova_dir/`
☐ `mv nova_dir/ vazio.txt`
☐ `muevete -i vazio.txt -o nova_dir/`
☐ `move -o nova_dir/ -i vazio.txt`

Tarefa 12

Só para mostrar que já começa a dominar o assunto, emita o comando que lhe permite fazer uma cópia do ficheiro que está em `nova_dir` para a diretoria atual.

6 *Pipes* e Redirecionamento

Tipicamente, um comando gera uma ou mais saídas que, por sua vez, podem ser incorporadas (como parâmetros) num outro comando. As saídas deste outro comando podem ser incorporadas noutro, e assim sucessivamente até que o resultado seja mostrado no terminal ou guardado dentro de um ficheiro.

A cadeia referida anteriormente é possível via a utilização de *pipes* e redirecionamentos. Um *pipe* é representado pelo caractere `|` e permite juntar dois ou mais comandos. O redirecionamento de `outputs`, ou dados de saída de um determinado comando, é representado pelo caractere `>`, `2>` ou `&>`.

Tarefa 13

No seu ambiente de linha de comandos, na diretoria onde criou o ficheiro `vazio.txt` (da tarefa 9), execute o comando `cat vazio.txt | grep CLI`. Análise o resultado e considere responder às questões seguintes como forma de solidificar o conhecimento e habilidade adquirida.

Q25.: Para relembrar, qual é a finalidade do caractere `|`?

☐ O caractere `|` representa um *pipe*, que permite ligar o fluxo `stdout` de um comando ao fluxo `stdin` de outro comando.

☐ O caractere `|` representa um redirecionamento, permitindo enviar o fluxo de saída `stdout` para um determinado ficheiro.

☐ Não sei, mas tenho ideia que ainda agora falaram nisso...

Q26.: O que faz o comando `cat`?

☐ Faz a listagem de todos os ficheiros da diretoria atual.

☐ Copia do conteúdo de um ficheiro para outro.

☐ Chama o gato da minha vizinha.

☐ Imprime o conteúdo de um ou mais ficheiros no terminal.

Q27.: O que faz o comando `grep`?

☐ Agarra num ficheiro e coloca-o na diretoria anterior.

☐ Introduz texto num ficheiro.

☐ Imprime linhas que contenham o padrão procurado.

☐ Divide o ficheiro em partes iguais e espalha-as, aleatoriamente, pelo SO.

Q28.: Tendo em conta as respostas das questões anteriores, qual é a função do *pipe* na tarefa anterior?

☐ Introduz a palavra `CLI` no ficheiro `vazio.txt`.

☐ Dado o conteúdo do ficheiro `vazio.txt`, exibe as linhas que contenham a palavra `CLI`.

☐ Substitui o conteúdo do ficheiro `vazio.txt` pela palavra `CLI`.
☐ Modifica o nome do ficheiro `vazio.txt` para `CLI`, mantendo o conteúdo do ficheiro integral.

Tarefa 14

No seu ambiente de linha de comandos, posicione-se (se não estiver já nesse local) na diretoria onde criou o ficheiro `vazio.txt`. Copie o conteúdo deste ficheiro para um novo, denominado `vazio_copia.txt`, mas assumindo que tem que utilizar o comando `cat` e o redirecionamento (`>`). Registe todos os comandos a utilizar.

O operador representado pelos dois caracteres `>>` é também utilizado no contexto dos redirecionamentos. Considere os exemplos `cat file1.txt > file2.txt` e `cat file1.txt >> file2.txt` . **Q29.: Dado os exemplos, qual é a diferença entre os comandos `>` e `>>` ?**

☐ Usando `>>` , o conteúdo do ficheiro `file1.txt` será substituído pelo conteúdo do ficheiro `file2.txt`, enquanto `>` concatena o conteúdo do ficheiro `file2.txt` no final do ficheiro `file1.txt`.
☐ Usando `>>` , o conteúdo do ficheiro `file2.txt` será substituído pelo conteúdo do ficheiro `file1.txt`, enquanto `>` concatena o conteúdo do ficheiro `file1.txt` no final do ficheiro `file2.txt`.
☐ Usando `>` , o conteúdo do ficheiro `file1.txt` será substituído pelo conteúdo do ficheiro `file2.txt`, enquanto `>>` concatena o conteúdo do ficheiro `file2.txt` no final do ficheiro `file1.txt`.
☐ Usando `>` , o conteúdo do ficheiro `file2.txt` será substituído pelo conteúdo do ficheiro `file1.txt`, enquanto `>>` concatena o conteúdo do ficheiro `file1.txt` no final do ficheiro `file2.txt`.

12 *Scripting* em *Bash* e Criação de Makefile

Sumário

Scripting em *bash*. Compilação e execução de programas em `C`. Criação de ficheiro `Makefile`.

Summary

Bash Scripting. Compilation and execution of C programs. `Makefile` *creation.*

Pré-requisitos:

Algumas das tarefas propostas a seguir requerem o acesso à Internet e acesso a um **SO Linux**. Se não pretender instalar uma distribuição de Linux na sua máquina pode sempre optar por instalar a distribuição numa máquina virtual. O uso de Subsistema Windows para Linux (*Windows Subsystem for Linux*), para Windows 10, concretiza também uma opção válida.

1 *Scripts* em *Bash*

Desenvolver *scripts* em *bash* é extremamente útil para a administração e no desenvolvimento de sistemas ou de programas. Estes ficheiros são conjuntos de comandos ordenados e com uma determinada intenção, executados sequencialmente e, potencialmente, adornados por instruções de controlo de fluxo.

Tarefa 1

Crie um ficheiro, denominado `script.sh`, com o trecho de código seguinte. Modifique a variável `APELIDO` para que tenha o seu apelido.

```
#!/bin/bash
APELIDO="Inácio"
```

Q1.: Qual é a utilidade da primeira linha do ficheiro criado na tarefa anterior?
☐ É um comentário por isso não tem utilidade nenhuma.
☐ Serve para indicar que o *script* deve ser interpretado pelo *bash* e não por outro interpretador.

Tarefa 2

Modifique o ficheiro de modo a que, quando executado, imprima, no terminal, a frase: `"Bom dia, Inácio!"`. Para tal deverá utilizar a variável `APELIDO`.

Q2.: Qual foi o comando que usou para imprimir a mensagem no terminal?
☐ `touch` ☐ `print` ☐ `echo` ☐ `grep` ☐ `ls`

Para utilizar o conteúdo da variável `APELIDO` no comando anterior teve que lhe adicionar um símbolo. **Q3.: Qual foi o símbolo utilizado?**
☐ `@` ☐ `:` ☐ ~ ☐ ☠ ☐ `$` ☐ `#` ☐ `>` ☐ ♘

Tarefa 3

Execute o ficheiro `.sh` utilizando o comando `bash script.sh`, no terminal.

Q4.: A execução do ficheiro culminou na impressão da mensagem esperada?
☐ Não. Apareceu `APELIDO` ao invés do meu apelido.
☐ Sim. O terminal deu-me os bons dias e o meu dia até ficou melhor!

Considere que queria executar o *script* usando o comando `./script.sh` **Q5.: É possível?**
☐ Sim. Eu executei e deu o mesmo resultado.
☐ Sim, é possível. Mas para tal é necessário usar o comando `chmod u+r script.sh` antes.
☐ Sim, é possível. Mas para tal é necessário usar o comando `chmod u+x script.sh` antes.
☐ Não. Tal coisa não é possível em terminal. Desisto.

Tarefa 4

Modifique o ficheiro `script.sh` para que receba, como argumento, o seu primeiro nome

e o inclua na mensagem a imprimir no terminal. Por exemplo, caso o seu primeiro nome seja `Pedro`, o *script* deverá emitir a mensagem `"Bom dia, Pedro Inácio!"` quando é executado da seguinte forma: `./script.sh Pedro`

Q6.: O que teve que adicionar ao comando `echo` para que produzisse a mensagem requerida na tarefa anterior?

- ☐ Para usar o primeiro argumento tive que incluir `#1` no comando `echo`.
- ☐ Para usar o primeiro argumento tive que incluir `$1` no comando `echo`.
- ☐ Criei uma nova variável, chamada `NOME` e usei-a no comando `echo` de forma análoga à variável `APELIDO`.
- ☐ Vou confessar. Fui batoteiro e escrevi `Pedro` no comando `echo` porque já não me lembro como se usa os argumentos...

Um cuidado que deve ter, quando espera receber argumentos, é verificar que, efetivamente, estes foram dados ao *script* aquando da sua execução. No exemplo anterior, mesmo que o comando seja executado sem um argumento, o *script* é executado e é exibida a mensagem apenas com o apelido d@ estudante.

Tarefa 5

Execute o seu ficheiro `script.sh`, sem argumentos, e verifique que nenhum erro é exibido no terminal.

Para recordar um conceito apresentado na aula teórica, considere o seguinte exemplo:

```
VAR=101
if [[ $VAR -gt 100 ]]
then
  echo "O número é superior a 100."
else
  echo "O número é inferior a 100."
fi
```

Q7.: Qual é a finalidade de `-gt`, no trecho de código exibido?

- ☐ É um argumento de *if* que aumenta o valor (*give them*) da variável `VAR` em 100 unidades.
- ☐ É uma opção passada à variável `$VAR` para obter o valor desta (*get that*) e, neste exemplo, substituir por 100.
- ☐ É um operador de comparação. Neste caso, é usado para que a condição *if* sirva para verificar se o valor de `VAR` é superior (*greater than*) a 100.
- ☐ Nem vi que estava um `-gt` no trecho de código. Deixe-me só ver melhor e já lhe dou uma resposta.

Tarefa 6

Seguindo a analogia do uso de `-gt` no exemplo anterior, modifique o ficheiro `script.sh` para que seja verificado o número de argumentos e, caso o número recebido **não seja concordante com o esperado**, o ficheiro deverá imprimir, no terminal, a mensagem `"Número incorreto de argumentos!"`. Para verificar o número de argumentos recebidos, use a seguinte combinação de caracteres: `$#` .

Execute novamente o ficheiro *script.sh* sem argumentos. **Q8.: Houve alguma alteração relativamente à execução, deste mesmo ficheiro, comparativamente à tarefa 5?**

☐ Nenhuma. Em ambos os casos apresenta uma mensagem de bom dia no terminal.
☐ Houve, sim senhor! Agora o *script.sh* exige que lhe passe um argumento. Será que se der mais do que um também funciona?

A leitura de *inputs* do utilizador e armazenamento destes numa variável é outra funcionalidade que a *bash* disponibiliza. **Q9.: Qual é o comando que pode utilizar para usufruir desta funcionalidade?**

☐ `get` ☐ `scanf` ☐ `nano` ☐ `read` ☐ `stdin`

Tarefa 7

Altere o ficheiro `script.sh` para que, dentro da condição *if* e depois da mensagem de bom dia, imprima a mensagem `"Introduza a sua idade:"`, leia o *input* do utilizador para uma variável, denominada `IDADE`, e imprima uma segunda mensagem, indicando a idade do utilizador. Caso o utilizador introduza o valor `18`, a mensagem a imprimir deverá ser `"Tem 18 anos"`.

Só para verificar que todos os conceitos usados ficaram consolidados: **Q10.: qual o comando que usou para ler o *input* do utilizador e guardá-lo na variável `IDADE`?**

☐ `read $IDADE` ☐ `read IDADE`
☐ `read #IDADE` ☐ `read -IDADE`

E para que não restem mesmo dúvidas nenhumas: **Q11.: qual o comando que usou para imprimir a mensagem com a variável `IDADE`?**

☐ `echo "Tem $IDADE anos!"`
☐ `echo "Tem IDADE anos!"`
☐ `echo "Tem #IDADE anos!"`
☐ `echo "Tem -IDADE anos!"`

2 Edição, Compilação e Execução de Programas

Para o computador executar programas é necessário que os mesmos estejam escritos em linguagem máquina, ou que exista um interpretador que faça a tradução em tempo-real. Assim, é necessário que exista uma tradução entre o que o programador escreveu e o que a máquina compreende. O desenvolvimento de um programa, fazendo uso de linguagens compiladas, passa por três fases essenciais:

1. Implementação do código;
2. Compilação e interligação do código;
3. Execução e teste do código.

Considere o comando seguinte: `cc -o main.exe main.c` . **Q12.: Em qual das fases anteriormente apresentadas se enquadraria o comando apresentado?**
☐ Implementação do código.
☐ Compilação e interligação do código.
☐ Execução e teste do código.
☐ Atenção malta! Pergunta com rasteira! Não pertence a nenhuma das fases.

Q13.: O que pode comentar relativamente ao uso da opção `-o` no comando apresentado?
☐ É uma opção facultativa que permite definir o nome do executável.
☐ É uma opção obrigatória que permite a compilação dos ficheiros `.c` em ficheiro `.o` .

Tarefa 8

Obtenha um ficheiro `.zip` usando esta *hiperligação*. Faça `unzip` e confirme a existência de dois ficheiros `.c` e um ficheiro `.h`. Coloque os três ficheiros numa nova diretoria, denominada `Lab_5_Compile_Make`. Analise os ficheiros e interprete a funcionalidade de cada um deles.

Só para recordar: **Q14.: qual é o significado de ficheiros com extensão `.h`?**
☐ Significa que é ficheiro de ajuda (*helper*).
☐ Significa que é ficheiro de cabeçalho (*header*).
☐ Significa que é ficheiro de ligação (*holder*).
☐ Significa que é ficheiro de excitação (*hype*)!

Observe, atentamente, o ficheiro `main.c`. **Q15.: Qual é a utilidade da inclusão do ficheiro `soma.h`?**
☐ Permite que seja usada uma função que está presente no ficheiro `soma.c`.
☐ É uma gralha. Devia ter sido incluído o ficheiro `soma.c` e não `soma.h`.

Ainda relativamente ao ficheiro `main.c`. **Q16.: Qual é a razão para que `soma.h` tenha sido incluído entre " ", enquanto `stdio.h` foi incluído com < >?**

☐ Foi uma opção do programador. Ambos podem ser usados de forma intercambiável.

☐ Ambos são ficheiros de cabeçalho mas `soma.h` é ficheiro de cabeçalho local enquanto `stdio.h` é do compilador.

Tarefa 9

Registe qual é a finalidade do programa, descriminando o papel desempenhado por cada um dos ficheiros.

Considere agora o processo de compilação do código. **Q17.: Qual o comando que utilizaria para compilar o programa descrito na tarefa anterior?**

☐ `cc` ☐ `gcc` ☐ `g+-` ☐ `java` ☐ `compilador_c`

Q18.: Caso a sua intenção fosse compilar os ficheiros e obter os ficheiros objetos destes (com extensão `.o`) qual seria a opção a usar, aquando do processo de compilação?

☐ `-o` ☐ `-c` ☐ `-v` ☐ `-x` ☐ `-t`

Assuma que pretende compilar os ficheiros fonte fornecidos e obter um executável. **Q19.: Qual, das seguintes opções, permitiria atingir esse objetivo?**

☐ `cc main.c` ☐ `cc main.c soma.h`

☐ `cc soma.c soma.h` ☐ `cc main.c soma.c`

Q20.: Dada a opção escolhida na questão anterior, de que forma executaria o ficheiro executável resultante?

☐ `./main.c` ☐ `./main.exe` ☐ `./a.out` ☐ `./a.exe`

☐ `./nao_faco_ideia.exe`

Tarefa 10

Registe o comando que utilizaria para que a compilação dos ficheiros fonte fornecidos resultasse num ficheiro executável com o nome `eu_e_que_mando.exe`.

Execute o ficheiro resultante e confirme que produz o resultado esperado.

3 Makefile

O `Make` é uma ferramenta de engenharia de *software* que controla a geração de executáveis e outros ficheiros associados a partir dos ficheiros fonte de um programa. A sua utilização ganha maior pertinência com o aumento do número de componentes que contribuem para um programa. Adicionalmente, permite diminuir a probabilidade de erro humano no processo de compilação.

Relembre a sintaxe das regras de um ficheiro `Makefile`, abordada na aula teórica, e considere o trecho de código seguinte:

```
main.o : main.c
    cc -c main.c
```

Q21.: Qual é a designação de `main.o` e `main.c` na nomenclatura dos `Makefiles`, respetivamente, no exemplo anterior?

☐ Objetivo e Comando.
☐ Objetivo e Dependência.
☐ Dependência e Objetivo.
☐ Comando e Objetivo.

E para confirmar que percebeu os conceitos apresentados: **Q22.: qual é a designação de `cc -c main.c` no trecho de código apresentado?**

☐ Objetivo.
☐ Dependência.
☐ Comando.
☐ Não sei e recuso-me a responder, e por isso também não escolho esta opção...

Q23.: Qual é a finalidade da regra apresentada no trecho de código?

☐ É uma regra que necessita de um ficheiro `main.o` e que, caso este exista e esteja acedível, executa um comando que permite obter o ficheiro fonte de `main`, ou seja, `main.c`.
☐ É uma regra que necessita de um ficheiro `main.c` e que, caso este exista e esteja acedível, executa um comando que permite obter o ficheiro objeto de `main`, ou seja, `main.o`.
☐ É uma regra que necessita do comando `cc -c main.c` e que cria o ficheiro `main.o`, independentemente de `main.c` existir ou não.
☐ É uma regra confusa e desnecessária que só está aqui para me confundir e eu já não estou a perceber nada disto...

Tarefa 11

Obtenha um ficheiro `Makefile` usando esta *hiperligação*. Coloque este ficheiro na diretoria `Lab_5_Compile_Make`, criada numa tarefa anterior. Analise o ficheiro e interprete a funcionalidade de cada uma das regras nele contidas.

Q24.: Assumindo que tem os ficheiros `.c`, `.h` e `Makefile` disponibilizados na diretoria `Lab_5_Compile_Make`, o que prevê que aconteça caso execute o comando

`make` ?

☐ Será executado apenas a primeira regra, `all`, que necessita de um ficheiro `main.exe`. No entanto, como este ficheiro não existe, será emitido uma mensagem de erro no terminal.

☐ Tenho uma vaga ideia mas estou só à espera que o Professor faça para depois poder responder com mais certeza.

☐ Serão executados um conjunto de comandos e no final será criado um executável, denominado `main.exe`.

☐ Serão executados um conjunto de comandos mas, seguindo as regras todas, no final teremos a regra `clean` o que significa que toda a diretoria será limpa e ficará sem ficheiros.

Uma boa prática na construção de um ficheiro `Makefile` passa por identificar quais são os *phony targets* e colocá-los numa entrada, denominada `.PHONY`.

Q25.: O que pode comentar relativamente a um *phony target*?

☐ Não produz nenhum ficheiro.

☐ Não tem nenhuma dependência.

☐ Não tem nenhuma funcionalidade.

☐ Não faz bem o que é suposto. É falso!

Tarefa 12

Coloque todos os *phony targets* do ficheiro `Makefile` numa entrada `.PHONY`.

Q26.: Quantos *phony targets* colocou na entrada `.PHONY`?

☐ zero ☐ um ☐ dois ☐ três ☐ quatro ☐ $7 \pm \sqrt{2}$

Tarefa 13

Modifique o ficheiro `Makefile` para que todos os comandos `cc`, de todas as regras, sejam substituídos por uma variável `CC`, cujo conteúdo é `cc`.

Q27.: Qual foi a aparência do comando `cc -c main.c` no `Makefile`, após realizar as alterações pedidas?

☐ `#(CC) -c main.c` ☐ `#(cc) -c main.c`

☐ `$(cc) -c main.c` ☐ `$(CC) -c main.c`

Note que a execução do comando `make` produz o ficheiro `main.exe` e que, para o executar, terá que executar o comando `./main.exe`.

Tarefa 14

Assumindo que percebeu o funcionamento do ficheiro `Makefile`, modifique-o de modo a que, **apenas executando o comando** `make` este compile, faça *linkagem* e execute o ficheiro resultante. Como sugestão, crie uma nova regra, com o objetivo `execute` e atribua as dependências e comandos necessários.

Q28.: Qual foi a dependência que associou à regra criada ou, por outras palavras, do que é que precisa para executar o programa?

☐ `main.exe` ☐ `./main.exe`
☐ `execute` ☐ `main.c`

Q29.: Alterou algo no objetivo `all`?

☐ Não... Era suposto?
☐ Sim, tive que substituir `main.exe` por `execute`.

Q30.: E a entrada `.PHONY`, foi alterada?

☐ Tive dúvidas, mas coloquei lá o `execute`.
☐ Não, pois `execute` não é um *phony target*.

Recorde a utilização de variáveis automáticas, lecionado na aula teórica. Tome particular atenção à *wildcard* `%`, cujo o significado é: *qualquer sequência de caracteres*. Observe também a variável `<`, que, em conjunto com `$`, ou seja, `$<` significa: *nome da primeira dependência*.

Tarefa 15

Altere o ficheiro `Makefile` para que todas as regras que tenham um objetivo com extensão `.o` sejam substituídas por uma única regra que faça uso de `%`, tanto no objetivo como na dependência, e que use `$<` no comando a executar.

Q31.: Como ficou o objetivo e a dependência da nova regra criada?

☐ `%.c: %.o` ☐ `%.o: %.c` ☐ `%.exe: %.o`

Q32.: Quantas regras foram eliminadas, com a inserção desta nova?

☐ $-7 \pm \sqrt{2}$ ☐ zero ☐ uma ☐ duas ☐ três ☐ quatro

Tarefa 16

Introduza o trecho de código seguinte no topo do seu ficheiro `Makefile`

```
SOURCEFILES = $(wildcard *.c) # Forma de obter os nomes de ficheiros .c.
    Neste caso, todos os ficheiros com extensão .c serão armazenados na
    variável SOURCEFILES
```

```
OBJECTS = $(SOURCEFILES:%.c=%.o) # Substitui, em cada ficheiro da lista
    de ficheiros em SOURCEFILES, a extensão .c por .o
```

e analise os comentários.

Q33.: O que ficou armazenado na variável `OBJECTS`?

☐ Todos os ficheiros da diretoria que já tinham extensão `.o` .

☐ Um conjunto de ficheiros com extensão `.o`, cujos nomes correspondem a ficheiros da diretoria que têm extensão `.c` .

Recorde, novamente, a utilização de variáveis automáticas em `Makefile`. A utilização da variável `^` , que, em conjunto com `$` , ou seja, `$^` significa: *conteúdo da lista de dependências.*

Atente, no ficheiro `Makefile`, à regra

```
main.exe : main.o soma.o
  $(CC) -o main.exe main.o soma.o
```

e verifique que, tanto nas dependências como no comando, `main.o` e `soma.o` estão presentes. Na realidade, o conteúdo da variável `OBJECTS` é precisamente estes dois ficheiros.

Tarefa 17

Modifique a regra apresentada, no ficheiro `Makefile`, para que faça uso da variável `OBJECTS` e de `$^` .

Só para verificar que fez as alterações corretas: **Q34.: o que colocou nas dependências da regra alterada?**

☐ `$^` ☐ `$(OBJECTS)` ☐ `$(OBJECTS) $^` ☐ `./$^`

Relativamente à utilização de variáveis automáticas em `Makefile`, a utilização da variável `@` , que, em conjunto com `$` , ou seja, `$@` significa: *nome do objetivo da regra.* Com um olhar mais atento, verifica-se que `main.exe` está no comando da regra apresentada e é, simultaneamente, o objetivo da regra.

Q35.: Acha que conseguiria fazer uso de `$@` no comando da regra apresentada?

☐ Conseguir, conseguia. Mas é melhor não mudar mais não vá isto deixar de funcionar...

☐ Sim, substituiria `main.exe` por `$@` no comando da regra.

Tarefa 18

Se fez todas as tarefas corretamente até agora, irá verificar que `main.exe` aparece três vezes no seu ficheiro `Makefile`.

Tendo em conta todas as opções que lhe foram apresentadas ao longo deste guia, faça todas as alterações que achar conveniente para que `main.exe` apareça apenas uma vez no ficheiro `Makefile`.

13 Introdução a IDEs, Depuração e Perfilagem de Programas

Sumário
Compilação e execução de programas em C, Java e Python. Introdução aos Ambientes de Desenvolvimento Integrados. Depuração e perfilagem de programas.

Summary
Compilation and execution of C, Java, and Python programs. Introduction to Integrated Development Environments. Program debugging and profiling.

Pré-requisitos:
Algumas das tarefas propostas a seguir requerem o acesso à Internet e acesso a um **SO Linux**. Se não pretender instalar uma distribuição de Linux na sua máquina pode sempre optar por instalar a distribuição numa máquina virtual. O uso de Subsistema Windows para Linux (*Windows Subsystem for Linux*), para Windows 10, concretiza também uma opção válida.

1 Compilação e Execução de Programas em Java

O Java é uma linguagem de programação **orientada a objetos** desenvolvida pela Sun Microsystems, *Inc.* em 1991. Os programas em Java são compilados para um conjunto de instruções interpretadas por uma máquina virtual (Java *Virtual Machine*, designada por JVM).

Tarefa 1

Caso seja necessário, instale o Java *Development Kit* (JDK) utilizando a seguinte *hiperligação* ou instalando o pacote `default-jdk`. Como exemplo, em Ubuntu faça:

```
$ sudo apt install default-jdk
```

Tarefa 2

Crie uma nova diretoria, denominada `Lab_6`. Depois de navegar para a nova diretoria, crie um novo ficheiro, denominado `HelloWorld.java`, com o trecho de código seguinte:

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Finalmente, compile o ficheiro criado.

Q1.: Qual foi o comando que utilizou para compilar?

☐ `java HelloWorld.java` ☐ `javac HelloWorld.java`

☐ `javacc HelloWorld.java` ☐ `gcc HelloWorld.java`

Atente agora à diretoria criada na tarefa anterior. **Q2.: Foi criado algum ficheiro novo, resultante da compilação?**

☐ Sim, foi criado um ficheiro denominado `a.out`.
☐ Sim, foi criado um ficheiro denominado `HelloWorld.java`.
☐ Sim, foi criado um ficheiro denominado `HelloWorld.class`.
☐ Não, neste diretoria apenas existe o ficheiro `HelloWorld.java`.

Q3.: Qual o comando que sugere utilizar para executar o programa que compilou na tarefa anterior?

☐ `java HelloWorld.java` ☐ `java HelloWorld`

☐ `java HelloWorld.class` ☐ `gcc HelloWorld.java`

2 Interpretação e Execução de Programas em Python

A linguagem de programação Python é uma linguagem interpretada (e não compilada). O interpretador Python (alternativamente, Máquina Virtual Python), escrito em C, é responsável por interpretar o código Python.

Verifique se possui na sua máquina a última versão do interpretador Python utilizando o comando `python3 --version`. Caso não tenha uma versão 3.xx do interpretador

de Python instalada deve efetuar a sua instalação utilizando a seguinte *hiperligação* ou instalando o pacote `python3`.

Q4.: O que pode comentar relativamente ao processo de compilação de um ficheiro Python?

☐ É similar a C e Java, em que o código fonte é compilado para um ficheiro *bytecode* e depois executado.

☐ O meu comentário é que o processo é bonito, mas demorado.

☐ Difere de C e Java, pois o ficheiro com o código fonte não é tipicamente compilado, mas sim interpretado.

Tarefa 3

Crie um ficheiro, denominado `helloWorld.py`, com o trecho de código seguinte:

```
print("Hello, World!")
```

Efetue todos os passos necessários para executar (leia-se *interpretar*) o programa em Python.

Q5.: Qual o comando que utilizou para executar o programa da tarefa anterior?

☐ `java helloWorld.py`

☐ `cc helloWorld.py`

☐ `python helloWorld.py`

☐ `python3 helloWorld.py`

Por omissão, nas distribuições Linux terá também uma versão 2.xx do interpretador Python. Para fazer uso deste interpretador deverá usar o comando `python filename.py`, enquanto `python3 filename.py` fará uso do interpretador de Python 3.xx.

Tarefa 4

Verifique a diferença de versões dos seus interpretadores `python` e `python3`, usando a opção que achar mais conveniente, e registe os comandos usados.

---

3 Depuração de Programas

A *depuração* de programas é o processo de localizar e remover erros ou anormalidades (*bugs*) do programa, fazendo uso de ferramentas de depuração.

À medida que os seus programas se tornam cada vez mais complexos será necessário depurar a execução de um programa. Para tal, é muito útil adicionar instruções auxiliares e temporárias para verificação do conteúdo das variáveis, e pontos de paragem ou *break points* ao seu código fonte, que se traduzem em instruções que forcem pausas nos locais assinalados por esses pontos de rutura. Isto é muito útil não só para observar o valores das variáveis em tempo de execução mas também para observar possíveis comportamentos inesperados do programa. Para explorar estas possibilidades, execute as seguintes tarefas.

Tarefa 5

Crie um ficheiro, denominado `Tarefa4.c`, com o trecho de código seguinte:

```
#include <stdio.h>

int f(int a){
  return a-1;
}

int main(){
  int a=1, r, b=f(a);
  r = a/b;
  return 0;
}
```

Compile o ficheiro criado e execute o ficheiro resultante da compilação. Caso queira, pode fazer uso do `Makefile` criado no guia laboratorial anterior e usar o comando `make`.

Q6.: O programa foi executado com sucesso?

☐ Sim, não vejo nada que indique que tenha havido um erro; por outro lado, eu ainda nem sequer o compilei, por isso...

☐ Não, apareceu aqui um erro muito estranho. O *core* foi *dumped*?

Observe com atenção o código. **Q7.: Quais os potenciais pontos onde podem surgir problemas?**

☐ Talvez `b=f(a)`, visto estar a usar a variável `a` na mesma linha onde ela foi declarada e instanciada.

☐ Diria que na divisão, `r=a/b`. Sempre tive problemas com a matemática e o compilador pode também estar a ter algumas dificuldades. Deduzo que o compilador tirou má nota a matemática...

☐ Quase de certeza que o problema vem do `include`.

☐ Creio que retornar zero não é a melhor opção. Eu gosto sempre de retornar valores positivos.

Tarefa 6

Uma abordagem para verificar se o programa chegou a determinada instrução é fazendo *prints* dos valores de variáveis.

Introduza um `printf` para avaliar os valores das variáveis `a` e `b`, na linha imediatamente abaixo da declaração destas, e introduza um `printf`, para avaliar o valor de `r`, na linha imediatamente abaixo da linha da divisão, `r=a/b`.

Q8.: Quais foram os valores observados das variáveis `a, b` e `r`, respetivamente?

☐ Não sei quais são os valores de `a` ou `b` pois o programa encontrou um erro antes deste `printf`.

☐ `a=1, b=0` mas não sei qual é o valor de `r` pois o programa encontrou um erro antes deste `printf`, que engraçado!

☐ `a=1, b=1` e `r=1`.

Q9.: O que pode concluir relativamente à localização do erro no programa?

☐ Este encontra-se na linha de declaração de variáveis.

☐ O erro está dentro da função `f`.

☐ Há um problema na linha da divisão, `r=a/b`.

☐ O programa tem um problema no valor de retorno da função `main`.

☐ O erro está em parte incerta, atualmente em fuga e considerado perigoso.

Tarefa 7

O exemplo apresentado na tarefa 4 era de simplicidade suficiente para que o uso de *prints* de valores de variáveis fosse suficiente para que se pudesse concluir qual a origem do problema no programa. No entanto, para programas de maior dimensão, esta abordagem pode não ser exequível. Neste sentido, existem ferramentas que nos podem auxiliar no processo de depuração de programas.

Obtenha um ficheiro `.c` usando a *hiperligação*. Coloque este ficheiro numa nova diretoria denominada `Lab_6_Debug`. Compile o programa usando os comandos necessários ou o `Makefile` criado no guia laboratorial anterior. No final, execute o programa.

Q10.: O que pode comentar relativamente ao resultado obtido da execução do programa?

☐ O programa foi compilado e executado sem erros e imprimiu o meu número de telemóvel. Como é que isto é possível?

☐ O programa foi compilado e executado sem erros mas o resultado é um pouco estranho...

☐ Não posso comentar muito pois tive um erro a compilar.

Q11.: E se executar várias vezes o programa obtém um resultado diferente?

☐ Não. Claramente terá de dar sempre o mesmo resultado.
☐ Sim. Mas o programa é confuso e existem ali uns `rand()` no meio. Pode ser essa a razão.
☐ Não sei. Continuo sem conseguir compilar e não tenho o `Makefile` do guia laboratorial anterior para me ajudar.

Tarefa 8

Analise detalhadamente os ficheiros obtidos na tarefa anterior e tente perceber qual o fluxo do programa.

Q12.: Consegue formular uma hipótese relativamente à origem da aleatoriedade exibida aquando da execução do programa?
☐ Provavelmente o programa passa por `rand()`. Isso explicaria o resultado apresentado.
☐ O uso de `while()` é um forte candidato. O número de iterações é muito grande e isso pode estar a complicar a execução do programa.
☐ Aquela declaração na linha 24 onde `i = f1(--a)` é altamente duvidosa. Será que `a` é decrementado antes ou depois de ser passado como argumento?
☐ Creio que existem algumas hipóteses válidas mas o melhor mesmo é usar uma ferramenta de depuração para tirar todas as dúvidas!

Tarefa 9

Os procedimentos que se seguem requerem a instalação do editor de código Visual Studio Code. Pode usar esta *hiperligação* para descarregar esta aplicação ou, caso prefira, usar esta *hiperligação* para a instalar nos SOs Linux.

Tarefa 10

Instale a extensão `C/C++`, da Microsoft. Para aceder ao menu de extensões pode usar o comando `Ctrl+Shift+X`. De seguida, realize os seguintes passos:

1. No Visual Studio Code, abra a diretoria que contém o ficheiro com o código fonte;
2. Abra posteriormente o ficheiro `debug.c` no Visual Studio Code;
3. Pressione `F5` ou `Run > Start Debugging` ;
4. No topo da janela será exibido um menu *dropdown* para selecionar o ambiente. Selecione `C++ (GDB/LLDB)` ;

5. No topo da janela será exibido um novo menu *dropdown* para selecionar uma configuração. Selecione `gcc` , sem nenhum número;

Se todo o processo correu sem problemas deverá ter uma nova pasta, denominada `.vscode`, com dois ficheiros `.json`, denominados `launch` e `tasks`.

Nota: caso tenha obtido algum erro, relativamente a `gdb`, instale o mesmo no seu SO. A título de exemplo, para Ubuntu, este pode ser instalado via `sudo apt-get install gdb` .

Tarefa 11

Para avaliar potenciais problemas no programa serão usados pontos de paragem (*breakpoints*). Para tal basta clicar com o botão do lado esquerdo na linha onde deseja que o seu programa pare.

Coloque um ponto de paragem nas linhas 42 e 43 e pressione `F5` (ou `Run > Start Debugging`). Nas tarefas e questões seguintes esta operação será identificada por **depuração**. De seguida, selecione o botão de depuração, no lado esquerdo do editor, ou use a combinação `Ctrl+Shift+D`. O programa irá parar no primeiro ponto de paragem.

Observe a secção de depuração. **Q13.: O que pode concluir em relação ao valor da variável `c` do programa?**

☐ Segundo a subsecção de `Variables`, `c` tem o valor de 3.
☐ Segundo a subsecção de `Watch`, não há variáveis no programa.
☐ Segundo a subsecção de `Call Stack`, está lá uma `main()` mas não sei que valor tem `c`.
☐ Nada. Existe uma subsecção de `Breakpoints` mas não me parece que o valor da variável `c` esteja lá...

Tarefa 12

Clique em `F5` ou no primeiro botão ▷, no topo da janela. O programa deverá prosseguir para o segundo ponto de paragem criado.

Observe, novamente, a secção de depuração. **Q14.: O que pode concluir relativamente à origem da aleatoriedade do resultado do programa?**

☐ Constato que a variável `c` tem, ainda, o valor 3. Logo a aleatoriedade deverá vir do `printf` ou `return 0`.
☐ Reparei, de forma muito perspicaz, que a variável `c` tem já um valor diferente e conclui, de forma ainda mais arisca, que a aleatoriedade deverá ter origem dentro da função `f`.

Termine o processo de depuração clicando no botão □, no topo da janela, ou via `Shift+F5`.

Tarefa 13

Analise o ficheiro e registe as linhas onde colocaria pontos de paragem, visando a avaliação da origem da aleatoriedade do programa.

As tarefas e questões seguintes consideram os pontos de paragem nas linhas 13, 25, 36 e 37.

Tarefa 14

Com o ficheiro `debug.c` selecionado, depure o programa e migre até ao segundo ponto de paragem (linha 13).

Q15.: Tendo em conta a subsecção `Call Stack`, que funções foram chamadas?

☐ Pela ordem apresentada, `main` chamou `f`, que chamou `f1`, e que por sua vez chamou `f2`.

☐ Aparentemente, `main` chamou `f2`.

Q16.: Qual o valor de `a` e, dado este valor, qual a condição que será escolhida?

☐ valor 3 e condição `if` ☐ valor 3 e condição `else`

☐ valor 2 e condição `if` ☐ valor 2 e condição `else`

Tarefa 15

Avalie a sua resposta anterior utilizando a tecla `F10` ou clicando em ↷, no topo da janela.

Q17.: Conseguiu concluir algo relativamente à origem da aleatoriedade do resultado do programa?

☐ Sim! Provém de `rand()`, dentro da função `f2`, como eu inicialmente tinha conjeturado!

☐ Ainda não consegui perceber de onde vem, mas sei que não provém do `rand()` da função `f2`.

Tarefa 16

Prossiga para o próximo ponto de paragem (linha 25).

Q18.: Dado o valor de `a` e `i`, qual será o valor de retorno da função `f2`?

☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 100000002 ☐ ∞

Tarefa 17

Avalie a sua resposta anterior prosseguindo para o próximo ponto de paragem (linha 36).

Q19.: De acordo com a subsecção `Call Stack`, em que função está, neste momento, o processo de depuração?
☐ `fAlone` ☐ `f2` ☐ `f1` ☐ `f` ☐ `main`

Q20.: Qual é a variável que propõe analisar para avaliar a sua resposta à questão 18?
☐ `i` ☐ `soma` ☐ `array` ☐ `tot` ☐ `a`

Tarefa 18

Clique em `F5` (ou no botão ▷) até que chegue ao ponto de paragem na linha 37. A cada clique, tome atenção aos valores das variáveis `soma` e `i`.

Q21.: Este ponto de paragem ajudou-o a concluir algo relativamente à origem da aleatoriedade do resultado do programa?
☐ Creio que sim. O valor de `soma` tinha o somatório de todos os elementos do `array` mas quando saiu do ciclo ficou com um valor aleatório...
☐ Ainda não foi desta. Deduzo que o problema venha do próximo ponto de paragem, ou seja, no `printf`, da função `main`.

Tarefa 19

Registe qual é, na sua opinião, o problema do programa que promove o aparecimento de uma resposta aleatória aquando da sua execução.

---

4 Perfilagem de Programas

A perfilagem (da expressão inglesa *profiling*) de um programa é uma análise dinâmica do programa que permite avaliar a percentagem de tempo total utilizado por cada função, que funções foram chamadas, qual a árvore de chamada das funções, entre outros aspetos. Neste guia laboratorial iremos usar o *profiler* `gprof`.

Tarefa 20

Utilize o comando `gprof -v` para verificar se possui este *profiler* na sua máquina. Caso verifique que não se encontra instalado, deverá instalá-lo. Para Ubuntu, por exemplo, este poderá ser instalado da seguinte forma: `apt-get install binutils`.

Para obtermos informação de perfilagem do programa, este terá que ser habilitado aquando do processo de compilação. Para tal, terá que ser adicionada a opção `-pg` ao comando de compilação do programa.

Tarefa 21

Compile o ficheiro `.c` disponibilizado com este guia fazendo uso da opção referida anteriormente.

Q22.: Houve alguma alteração na diretoria onde o ficheiro foi compilado?
☐ Houve, sim senhor. Foi criado um novo ficheiro com informação sobre o programa.
☐ Não creio. Parece-me que está tudo igual.

Tarefa 22

Execute o ficheiro resultante do comando usado para compilar na tarefa anterior.

Q23.: E desta vez, verificou alguma alteração na diretoria onde o ficheiro foi executado?
☐ Agora sim! Apareceu um ficheiro `pokemon.out`.
☐ Agora sim! Apareceu um ficheiro `.out`. É suposto executá-lo também?
☐ Não. E eu até fiz `ls -a` para ver se não estava algum ficheiro oculto!

Se realizou todos os passos com sucesso deverá ter um ficheiro denominado `gmon` na diretoria onde executou o programa. Para obter o conteúdo deste ficheiro e ser possível analisá-lo sugere-se a utilização do comando `gprof` que recebe como argumentos o executável do programa e `gmon.out`, por esta ordem.

Q24.: Assumindo que o executável do programa tem o nome `a.out`, qual o comando que permite obter um ficheiro, denominado `profile_output.txt`, com o conteúdo resultante do comando `gprof`?
☐ `gprof a.out gmon.out -o profile_output.txt`
☐ `gprof gmon.out a.out -o profile_output.txt`
☐ `gprof a.out gmon.out > profile_output.txt`
☐ `gprof gmon.out a.out > profile_output.txt`

No ficheiro criado na questão anterior poderá ver inúmeras características, devidamente descritas, resultantes da perfilagem do programa. Este ficheiro divide-se em duas partes: *Flat profile* e *Call graph*. No resto deste guia será analisada uma porção do *Flat profile*.

Atente à primeira tabela do *Flat profile*, no ficheiro `profile_output.txt`, com início na linha 3. **Q25.: Qual foi a função que, em termos percentuais, demorou mais tempo?**

☐ `fAlone` ☐ `f2` ☐ `f1` ☐ `f` ☐ `main`

Q26.: Qual é a razão que encontra para que a função da resposta à questão anterior tenha demorado mais tempo?

☐ A presença de `while` e `rand()` dentro da função é uma boa justificação.
☐ É uma função que tem um `while` a iterar um elevado número de vezes.
☐ A função tem um ciclo `for`.
☐ É a função principal e por isso tem direito a demorar o tempo que quiser.

Q27.: Porque razão a função `fAlone` não aparece na tabela de *Flat profile*?

☐ Porque não demorou praticamente tempo nenhum, logo não se justificava a sua presença.
☐ Porque esta função não foi, em momento algum, invocada ao longo do código. É código morto e deveria ser retirado do ficheiro fonte!

Tarefa 23

Modifique o ficheiro `Makefile` do guia laboratorial anterior para que, quando execute o comando `make profile`, o código fonte seja compilado com a opção `-pg`, executado e o conteúdo de `gmon.out` seja exportado para um ficheiro, denominado `profile_output.txt`. Esta exportação para o ficheiro `.txt` deverá usar o comando `gprof`.

14 Gestão de Versões em Projetos Usando o Sistema Git

Sumário
Gestão de versões em projetos de programação utilizando o sistema Git.

Summary
Version management in programming projects using the Git system.

Pré-requisitos:
As tarefas propostas a seguir presumem o acesso a um SO com o *software* Git ou com permissões para a sua instalação. Todas as tarefas foram testadas com sucesso em ambiente Linux, mas devem funcionar noutros sistemas, assumindo que as ferramentas necessárias estão instaladas.

1 Introdução ao Sistema de Controlo de Versões – Git

O Git, inicialmente desenvolvido por Linus Torvalds, é um sistema de controlo de versões distribuído utilizado para o desenvolvimento de *software*. No entanto, o mesmo pode ser utilizado para manter um registo histórico de qualquer tipo de ficheiros (*e.g.* livros, artigos, *Curriculum Vitae*).

O GitHub é uma plataforma que permite alojar código fonte com suporte ao sistema de controlo de versões Git. Este permite que utilizadores possam contribuir para o desenvolvimento de projetos privados e/ou *open source*.

Tarefa 1

Dirija-se a https://github.com/ e crie uma conta gratuita no GitHub. Escreva nos espaços seguintes o nome do utilizador (*username*) e o endereço de *e-mail* que utilizou no registo:

Username: _____

E-mail: _____

O Git pode ser instalado em SOs Windows, Linux ou macOS. Há instruções detalhadas de instalação em https://git-scm.com/downloads. Por exemplo, na distribuição Ubuntu, o Git pode ser instalado utilizando o comando `sudo apt-get install git`.

Tarefa 2

Verifique o manual do Git utilizando o comando `man git`.

Q1.: De que forma é que o Git é definido no seu manual?
☐ Como uma ferramenta de gestão de versões.
☐ Como um editor de texto.
☐ Como um gestor de conteúdos estúpido.

Tarefa 3

Crie uma diretoria para um novo projeto chamada `Proj1`, e entre nessa diretoria.

Q2.: Para recordar, quais foram os comandos que utilizou para realizar a tarefa anterior?
☐ `mkdir Proj1` e `cd Proj1`
☐ `cd Proj1` e `mkdir Proj1`
☐ `cd_to_dir Proj1` e `make_my_dir Proj1`

Tarefa 4

Inicialize o Git nessa diretoria, utilizando o comando `git init`, e verifique que foi criada uma nova diretoria escondida nessa mesma pasta com `git ls -a`.

Q3.: Para recordar: de que forma é que sistemas Unix ou Unix-*like* simbolizam ficheiros ou diretorias escondidas?

☐ Os nomes desses ficheiros ou diretorias aparecem a vermelho, tal como o meu amado Benfica.
☐ Os nomes desses ficheiros ou diretorias aparecem a verde, tal como o meu amado Sporting.
☐ Os nomes desses ficheiros ou diretorias aparecem a azul, tal como o meu amado Porto.
☐ Os nomes desses ficheiros ou diretorias começam com um **ponto**.
☐ Os nomes desses ficheiros ou diretorias têm todas as letras capitalizadas.

Tarefa 5

Antes de prosseguir, e para estabelecer uma linha base, verifique o estado da gestão de conteúdos na sua diretoria com `git status`.

2 Gestão de Ficheiros e Versões

Tarefa 6

Crie um novo ficheiro chamado `program.c`, abrindo-o imediatamente para escrita utilizando o comando `nano program.c`. Adicione e guarde o seguinte código no seu ficheiro:

```
#include<stdio.h>

int main(){
 printf("Hello World!");
 return 0;
}
```

Tarefa 7

Verifique o estado atual do seu repositório Git utilizando o comando `git status`.

Q4.: Qual foi o *output* do comando realizado na tarefa anterior?
☐ O *status* está mau. Isto está complicado e eu tenho de abandonar, porque tenho uma consulta às 5.
☐ É indicado que existem *untracked files* e que ainda não existem *commits*.
☐ São dadas as indicações do que deve ser feito a seguir.
☐ Não obtive qualquer tipo de *output* utilizando o referido comando.

Tarefa 8

Para **inicializar a monitorização** do ficheiro `program.c` considere emitir o comando `git add program.c`.

Após executar a tarefa anterior, execute novamente o comando `git status`. **Q5.: Qual é agora o seu *output*?**

☐ É indicado que existem *untracked files* e que ainda não existem *commits*.
☐ É indicado que ainda não existem *commits*, mas já não se queixa de *untracked files*.
☐ São dadas as indicações do que deve ser feito a seguir.
☐ Não obtive qualquer tipo de *output* utilizando o referido comando. Apenas um silêncio constrangedor...

Tarefa 9

Se tiver a certeza que o programa está numa forma estável, emita a sua primeira ordem de consolidação, representado por `Commit` ao longo deste guia.

Q6.: Qual foi o comando que utilizou para executar a tarefa anterior?

☐ `git commit -m "My initial commit of program.c"`
☐ `git_my_program commit -m "My initial commit of program.c"`
☐ `commit -m "My initial commit of program.c"`
☐ `cc -o program program.c`

Q7.: Qual foi o *output* do comando que executou na tarefa anterior?

☐ Não me dei conta que era para usar um comando...
☐ Impecável, tudo a funcionar!
☐ Foi me dito que tenho que adicionar uma conta de utilizador.

Tarefa 10

No primeiro `Commit` feito numa máquina, o Git **pede** sempre **informação acerca do utilizador**, nomeadamente o nome de utilizador e *e-mail*. Se já tiver uma conta no GitHub, utilize os mesmos valores. Para ajustar esta informação, terá de utilizar os seguintes comandos:

```
git config user.email my@email.com
git config user.name us3rn4me
```

Pode, alternativamente, guardar estes valores ao nível do sistema, para que os mesmos possam ser utilizados em utilizações futuras e noutros projetos. Para obter esse efeito,

pode adicionar a opção `--global` aos comandos anteriormente referidos, *e.g.*,
`git config --global user.email my@email.com`
`git config --global user.name us3rn4me`

Tarefa 11

Após realizar a tarefa anterior, execute novamente o comando de consolidação para a efetivar:
`git commit -m "My initial commit of program.c"`.

Verifique o estado do sistema de gestão de versões com `git status`. **Q8.: Qual o efeito obtido?**
☐ Escrevi `commit` só com um *m...* e não deu. Mas já vou emendar o erro!
☐ Satisfação: o meu o primeiro *commit* foi feito com sucesso.
☐ Tristeza: foi me dito outra vez que tenho que adicionar uma conta de utilizador. :(

Tarefa 12

Experimente compilar o programa em C que acabou de realizar utilizando o comando `cc program.c -o program.exe`.

Irá verificar que, ao executar o programa, a *string* `"Hello World!"` é escrita no ecrã, mas sem quebra de linha para a *prompt*. Será, portanto, necessário fazer mais uma alteração ao programa.

Tarefa 13

Abra o ficheiro `program.c` para edição (*i.e.* `nano program.c`) e altere a linha do `printf` de:

```
  printf("Hello World!");
```

para

```
  printf("Hello World!\n");
```

No final, saia e salve. Volte a compilar e a executar, verificando se já obtém o efeito esperado.

Tarefa 14

Q9.: Se emitir o comando `git status` irá verificar que o Git reporta dois tipos de ficheiros na diretoria. Que tipos são esses?
☐ *not staged* ☐ *not tracked* ☐ *not pretty* ☐ *not easy*

Repare que não esta interessado em monitorizar o ficheiro `.exe`, já que esse ficheiro é apenas o resultado da compilação, mas está sim interessado em fazer `Commit` do ficheiro `program.c`, já que esse ficheiro é importante para o projeto. A próxima tarefa endereça ambas as situações.

Tarefa 15

Execute os seguintes passos:

1. Crie um ficheiro chamado `.gitignore` e coloque o nome do ficheiro que não quer monitorizar lá dentro. Este procedimento irá evitar que o Git se volte a queixar da existência deste ficheiro;
2. Adicione os dois novos ficheiros modificados (o `program.c` e o `.gitignore` à área de *staging* (encenação) e faça o `Commit` , *i.e.*, execute os comandos seguintes
   ```
   git add program.c .gitignore
   ```
   ```
   git commit -m "Added newline and a new .gitignore file"
   ```

Verifique se as providências tomadas tiveram efeito emitindo novamente o comando `git status` .

Tarefa 16

Antes de terminar esta parte introdutória, experimente ainda o comando que lhe permite ver a história dos *commits* que já fez no seu projeto: `git log`

Q10.: Quantos *commits* já fez até aqui?
☐ Resmas! ☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ Cachalotes! ☐ 4

3 Navegação entre Consolidações – *Reset*

Tarefa 17

Simule uma alteração que vai querer desfazer *a posteriori*. Para isso, abra o ficheiro

`programa.c` e **apague** a linha que declara a função `main`.

Tarefa 18

Após realizar a tarefa anterior, encene (*add*) o ficheiro novamente utilizando o seguinte comando: `git add program.c`. Não se esqueça de realizar `Commit` após encenar/adicionar o ficheiro (*i.e.*, `git commit -m "I think an error was made."`).

Verifique o *log*, e identifique o valor da consolidação **para onde quer voltar** (este deve ser um valor hexadecimal bastante longo). **Q11.: Copie a parte inicial (*e.g.*, 6 primeiros caracteres) desse *commit* para o espaço em baixo:**

Tarefa 19

Realize um *reset* do seu projeto para um *commit* anterior à introdução do erro no seu código. Para este efeito utilize o comando semelhante a
`git reset --hard <id do commit>`.

Tarefa 20

Após realizar a tarefa anterior, verifique se o efeito das alterações no ficheiro desapareceu, e se já desapareceu do *log* o *commit* errado.

Q12.: Confirma que a função `main()` está de volta ao `program.c`?
☐ *Woo Hoo!* Confirmo!
☐ Não confirmo nem desminto.
☐ Usei os comandos `cat program.c` e `git log` para obter a confirmação que me está a pedir: confirmo!

4 Navegação entre Consolidações – *Revert*

O uso do comando `git reset --hard` **é perigoso**. Na verdade, é um dos poucos comandos no Git que causa alterações que não podem ser revertidas mais tarde. Para voltar atrás, sem efeitos destruidores, pode utilizar o comando `git revert <id do commit>`.

Q13.: Em que situações é que se deverá utilizar o comando `git reset --hard`?
☐ Sempre que necessário.

☐ Só na história de um repositório local. Nunca na história de um repositório partilhado.
☐ Para apagar logo que temos a certeza que não queremos que apareça no histórico de versões.

Tarefa 21

Mais uma vez, simule uma alteração que vai querer desfazer posteriormente. Para isso, abra o ficheiro `program.c` e **apague** a linha que declara a função `main`. Não se esqueça que deve encenar/adicionar o ficheiro e realizar `Commit` . Para este efeito, execute os comandos seguintes (depois de alterar o ficheiro `program.c`):

```
git add program.c
```

```
git commit -m "I think I did it again (by Britney Spears)."
```

Tarefa 22

Execute todos os passos necessários para reverter o seu *commit*. É importante notar que, neste caso, deve indicar o valor da consolidação que quer ver desfeita, e não o valor do *commit* para onde quer voltar (**ao contrário do reset**).

Q14.: Quais foram os comandos que utilizou para efetuar a tarefa anterior?

Q15.: Como evolui a história das consolidações quando se usa `revert`?
☐ Neste caso, os *commits* que estão entre o *commit* mais atual e aquele para onde evoluimos são apagados da história e é criado outro *commit* que substitui o atual.
☐ Neste caso, os *commits* que estão entre o *commit* mais atual e aquele para onde evoluimos são mantidos na história e é criado outro *commit* novo na história.
☐ Neste caso, o ponteiro é ajustado para o *commit* para onde queremos evoluir, e toda a história futura fica intacta. Se consolidarmos, a história é reescrita.

5 Ramificações – Criação e Navegação

É muito comum a situação em que se quer trabalhar numa funcionalidade ou alteração sem querer mexer no que já se encontra feito. Para tal, o Git permite a criação de ramos de desenvolvimento (*branches*), onde se podem fazer alterações sem afetar os ficheiros de outros ramos. Quando se cria um ramo com o Git, este cria automaticamente o equivalente a uma cópia da diretoria de trabalho.

Tarefa 23

Verifique quantos ramos tem definidos no seu projeto. Execute e análise o *output* do seguinte comando: `git branch`.

Q16.: Quantos ramos estão definidos?
☐ 0 ☐ 1 ☐ 1,5 ☐ 2 ☐ n

Q17.: Como se chama o ramo onde se encontra atualmente (e que é definido por defeito ao inicializar o Git)?
☐ `main` ☐ `develop` ☐ `master` ☐ `slave` ☐ `git` ☐ `server`

Tarefa 24

Utilize o comando `git branch teste-funcao` para criar um novo ramo denominado `teste-funcao`. Verifique se a criação do ramo foi bem sucedida (com `git branch`) e evolua para esse ramo utilizando o comando `git checkout teste-funcao`.

Q18.: Quais dos seguintes comandos permitem ver em que ramo se encontra atualmente?
☐ `git set` ☐ `git branch` ☐ `git status`

Dentro do contexto deste ramo experimental pode fazer quaisquer alterações sem prejuízo para o ramo principal. Para explorar as funcionalidades do Git nesta parte, considere as seguintes tarefas/questões.

Tarefa 25

Crie um novo ficheiro chamado `funcao.c` com o seguinte conteúdo:

```
#include<stdio.h>

void imprime(){
 printf("Hello World!");
}
```

Altere também o ficheiro `program.c` de forma a que fique semelhante a:

```
#include<stdio.h>

int main(){
 imprime();
 return 0;
}
```

Após estas alterações, deve compilar e executar o seu programa:

```
cc program.c funcao.c -o program.exe
```

```
./ program.exe
```

Tarefa 26

Se estiver tudo a funcionar corretamente, convém consolidar (*commit*) as alterações para o *branch* atual. Para esse efeito, execute e analise os seguintes comandos (procure perceber todos os detalhes):

```
git status
```

```
git add .
```

```
git commit -m "printf was moved to a new function"
```

Q19.: Qual é a finalidade do comando `git add .`?

☐ Adicionar/encenar todos os ficheiros da diretoria atual.
☐ Claramente vai adicionar/encenar os ficheiros da diretoria anterior.

Ainda antes de prosseguir, responda e armazene a resposta à questão seguinte: **Q20.: quantos ficheiros tem na diretoria de trabalho atual?**

☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ Infinitus!

6 Fusão e Eliminação de Ramos

Volte ao ramo de desenvolvimento principal através da emissão de um comando semelhante a `git checkout master`.

Q21.: O que significa a palavra *checkout* em Português?

☐ Verificar. ☐ Dar uma vista de olhos. ☐ Golo.
☐ É o contrário do *check-in* que se faz no aeroporto.

Q22.: Já verificou quantos ficheiros tem na diretoria de trabalho atual?

☐ Tenho exatamente os mesmos ficheiros que tinha na tarefa anterior.
☐ Que engraçado, tenho mais ficheiros que na tarefa anterior.
☐ Tenho menos ficheiros que na tarefa anterior. Que género de magia é esta?
☐ Como seria de esperar, tenho menos ficheiros que na tarefa anterior.

Procure explicar o que observou:

Tarefa 27

Dado ter implementado e testado com sucesso a funcionalidade nova no ramo `teste-funcao`, pode agorar pensar em **fundir** esse ramo com o principal (`master`). Para fundir um ramo com outro, **deve estar posicionado no ramo que vai receber a fusão** e executar um comando semelhante a `git merge <nome ramo que quer fundir para o atual>`. Construa o comando adequado à fusão e execute-o. Analise o *output*.

Q23.: Qual foi o comando que executou na tarefa anterior?
☐ `git merge teste-funcao`
☐ `git merge main-cuidado`
☐ `git merge slave`
☐ `git merge my-precious`

Depois de emitir o comando anterior, verifique novamente quantos ficheiros tem na diretoria atual. Não se esqueça de verificar também o *log*.
Q24.: Quantos ficheiros tem agora no ramo `master`?
☐ Tenho o mesmo número de ficheiros que tinha no *branch* `teste-funcao`, funcionou!
☐ Fixe, tenho agora exatamente 30 novos ficheiros.
☐ Nada funciona... o que me recomenda?

Tarefa 28

Dado já não necessitar do ramo que criou para testar a nova funcionalidade/forma de implementar, pode apagá-lo. Emita o comando `git branch -d teste-funcao` e tome as providências que achar necessárias para se certificar que o ramo já não existe.

Q25.: Como pode verificar se o ramo foi devidamente eliminado?
☐ `git branch` ☐ `git status` ☐ Todas as anteriores.
☐ Nenhuma das anteriores, exceto a última que diz *Todas as anteriores*... essa não...

Note que o procedimento ilustrado nas tarefas anteriores tem como objetivo dar uma ideia de como deve desenvolver o seu projeto e integrar novas funcionalidades. De uma forma sucinta, pode dizer-se que, para cada funcionalidade que pretenda adicionar, deve:

1. Criar um ramo de desenvolvimento novo e mudar o foco para esse ramo;
2. Desenvolver e testar a nova funcionalidade nesse ramo, mantendo o ramo anterior intato;
3. Mudar o foco para o ramo de desenvolvimento principal, fundir o ramo da nova funcionalidade e apagá-lo.

7 Repositórios e Sincronização

Git é um sistema de controlo de versões distribuído. Neste caso, também significa que cada programador guarda uma versão integral da história de consolidações localmente. A existência de repositórios remotos, contudo, irá permitir guardar uma cópia de segurança (*backup*) do projeto e, mais importante, a colaboração com outros programadores. O GitHub pode ser usado como um desses repositórios.

Tarefa 29

Dirija-se ao GitHub, entre na plataforma com as suas credenciais e selecione `New` .

Tarefa 30

Vai reparar que, após criar o novo repositório, o GitHub lhe faz uma série de sugestões. Por exemplo, é-lhe dito que é uma boa prática que todos os projetos tenham um ficheiro `README.md` e `LICENSE.md`. Sugere-se que colmate estas falhas, selecionando as opções `Add a README file` e `Choose a license` .

Tarefa 31

Crie uma nova diretoria (*e.g.*, chamada `teste-github`) e, colocando nela o contexto de trabalho, criar uma cópia local do repositório remoto utilizando um comando semelhante ao seguinte:

```
git clone https://github.com/<your-githubusername/<your-repository-name>
```

.

Deve alterar o comando, primeiro adicionando o seu nome de utilizador(a) no GitHub e, em segundo, o nome do repositório que acabou de criar. Após executar este comando, serão pedidas as credencias de acesso ao GitHub (*i.e.*, *username* e *password*).

Tarefa 32

Adicione o seu `program.c` e `funcao.c` ao seu repositório Git local (`teste-github`). Execute todos os comandos ou tome todas as providências que achar necessárias para concluir esta tarefa com sucesso.

Q26.: Quais foram os comandos que utilizou para adicionar os ficheiros `program.c` e `funcao.c`?

☐ `git config user.email my@email.com`

☐ `git config user.name us3rn4me`
☐ `git git add .`
☐ `git -m "My first commit to a remote repository"`
☐ Todas as opções anteriores.
☐ Só utilizei os comandos descritos na segunda e terceira opção.

Tarefa 33

O comando `push` é utilizado para transferir ou enviar um *commit*, que foi realizado no *branch* local, para um repositório remoto (*i.e.*, para o GitHub). Experimente agora enviar a consolidação mais recente do seu projeto para o repositório remoto emitindo um comando semelhante a:
`git push origin master`

Deverá conseguir verificar, através de num navegador de Internet, que os novos ficheiros foram agora copiados para o repositório remoto.

8 Colaboração em Projetos

Da execução das tarefas na secção anterior deve ter ficado com uma ideia de como se configura um repositório remoto e de como se enviam os ficheiros para esse repositório. Os comandos que permitem a colaboração de vários programadores no mesmo projeto foram já quase todos abordados, faltando apenas parte daqueles que permitem obter o projeto do repositório remoto, nomeadamente o `git fetch` e o `git pull`. Em baixo apenas se aborda o comando `git pull`, mas fica a indicação de que o comando `git clone https://github.com/<username>/<repositorio>.git` também está incluído neste conjunto, e deve ser usado para obter o conteúdo **e simultaneamente inicializar** um projeto, incluindo a sua história de versões, ramos e consolidações Git. Este é o **comando usado para inicializar um projeto na sua máquina local quando este ainda não existe.**

Tarefa 34

Quando se usa o Git para trabalho colaborativo é necessário ganhar um conjunto de boas práticas. É preciso pensar sempre que outros programadores podem estar a trabalhar no projeto e a publicar as alterações a qualquer momento. **Q27.: Em que momentos considera ser indicado verificar se há alterações no repositório remoto?**
☐ Antes de fazer uma consolidação *commit*.

☐ Sempre que fizer uma alteração.
☐ Quando se prepara para começar a trabalhar no projeto.
☐ Mesmo antes submeter alterações para o repositório remoto.

Finalize o guia, experimentando o comando seguinte, só para ficar com uma ideia de como funciona:

```
git pull origin master
```

15 Geração Automática de Documentação Usando Doxygen

Sumário
Geração automática de documentação a partir de comentários no código fonte utilizando a ferramenta `Doxygen`.

Summary
Automatic generation of documentation from comments in the source code using the `Doxygen` tool.

Pré-requisitos:
As tarefas propostas a seguir presumem o acesso a um SO com o *software* `Doxygen` ou com permissões para a sua instalação. Todas as tarefas foram testadas com sucesso em ambiente Linux, mas devem funcionar noutros sistemas, assumindo que as ferramentas necessárias estão instaladas.

1 Introdução à Geração Automática de Documentação

Apesar da criação de documentação técnica de programas ser uma tarefa onerosa e morosa, facilita sobremaneira a sua interpretação por parte de outras pessoas, e permite a exposição do modo de funcionamento do programa sob a forma de um documento estruturado. Existem várias ferramentas que permitem gerar documentos a partir de um conjunto de comentários adicionados ao código fonte de programas, tornando essa tarefa menos onerosa e simples de gerir. Algumas delas permitem inclusive gerar documentação sem que o código esteja comentado, embora de uma forma mais limitada. Este guia procura explorar precisamente uma dessas ferramentas.

O gerador de documentação automático a utilizar neste guia é o `Doxygen`, que suporta essa funcionalidade para implementações em C, C++, PHP, Python, entre outras.

Tarefa 1

O `Doxygen` pode ser instalado seguindo as instruções de instalação na seguinte *hiperligação*. Por exemplo, nas distribuições baseadas em Debian e Arch Linux, o `Doxygen` pode ser instalado utilizando, respetivamente, os comandos `sudo apt-get install doxygen` e `sudo pacman -S doxygen`

Assim, esta primeira tarefa consiste em verificar se o `Doxygen` está instalado no seu sistema, emitindo o comando `doxygen` no terminal e analisando o seu *output*.

Caso o sistema devolva um erro de `Command not found`, considere a instalação da ferramenta antes de evoluir no guia.

2 Geração de Documentação

O `Doxygen` suporta vários métodos para gerar documentação, partindo de comentários no código. Dois modos de comentários que permitem a sua inclusão na documentação, em C, são:

```
/**
... Comments ...
*/
```

ou

```
/// Comments
```

Repare no asterisco extra (`*`), na primeira linha do primeiro modo, e **na barra extra** (`/`), no segundo modo. Esta sintaxe é utilizada para que o `Doxygen` reconheça que o comentário inserido no código necessita de ser extraído e devidamente processado.

As duas sintaxes previamente descritas devem ser **adicionadas antes de uma entidade**. Uma entidade no `Doxygen` pode ser, por exemplo, uma variável global, uma estrutura ou uma função.

Tarefa 2

Crie uma nova diretoria chamada `program1` com as seguintes sub-diretorias: `docs` e `src`. Na sub-diretoria `src` crie um ficheiro chamado `program1.c` e adicione o seguinte código:

```
/**
  @file program1.c
*/
#include<stdio.h>
```

```c
int a = 20;

int makeSum(int b){
  return(a + b);
}

int main(){
  int b = 20;
  int sum = makeSum(b);
  printf("%d+%d=%d\n", a, b, sum);
  return 0;
}
```

De seguida, crie o ficheiro de configuração que irá conter todos os parâmetros necessários para gerar a documentação do seu programa. Para esse efeito, deverá executar, na diretoria `program1`, o comando `doxygen -g`.

Q1.: Quantos ficheiros foram criados após a execução do comando da tarefa anterior?

☐ Ahh... Adoro perguntas com rasteira. Parece que foi criado só um, mas vou verificar com `ls -la`.

☐ Ahh... Adoro perguntas com rasteira. Parece que foi criado só um, mas vou verificar também dentro da diretoria `docs` que a mim ninguém m'engana!

☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 42

Q2.: Qual foi o nome do ficheiro de configuração gerado pelo `Doxygen`?

☐ *Doxygen* ☐ *Doxyfile* ☐ *Doxydoc* ☐ *Doxyconfig* ☐ *Foxy*

Q3.: Qual seria o comando que utilizaria para gerar um ficheiro de configuração com um nome definido por si?

☐ `doxygen <config_file_name>` ☐ `doxygen -u <config_file_name>`

☐ `doxygen -g <config_file_name>` ☐ `javadoc <config_file_name>`

Tarefa 3

Promova as seguintes alterações no ficheiro de configuração gerado pelo `Doxygen` (`nano Doxyfile`):

- Altere o nome do seu projeto, alterando a linha seguinte –
  `PROJECT_NAME = "Sum Values"`

- Altere o caminho da diretoria onde serão colocados os ficheiros de documentação gerados pelo `Doxygen` –
  `OUTPUT_DIRECTORY = "docs/"`

- Defina a diretoria onde se encontram os ficheiros fonte do seu programa – `INPUT = "src/"`
- Caso não esteja definido, especifique o tipo de documentação que será gerado – `GENERATE_HTML = Yes`

Precura rápida: **Q4.: Como se podem fazer pesquisas no editor de texto `nano`?**
☐ Basta fechar os olhos com muita força e pensar no que queremos... o `nano` avança o cursor para esse ponto.
☐ Através da combinação de teclas `CTRL+W`.
☐ Não dá... também já era pedir demais!

Tarefa 4

Execute, na diretoria `program1`, o comando `doxygen <config_file>` para gerar a documentação do programa `program1.c` e procure os ficheiros de documentação gerados. Caso o ficheiro de configuração tenha o nome por omissão (`Doxyfile`), pode usar apenas o comando `doxygen`.

Q5.: Quais foram as sub-diretorias criadas na diretoria `docs`?
☐ `html` ☐ `txt` ☐ `png` ☐ `latex` ☐ `svg`

Abra o ficheiro `index.html`, presente na diretoria `docs/html`, utilizando um navegador (*browser*). **Q6.: Na página *Web*, onde se encontra a documentação do `program1.c`?**
☐ No separador *Files* > *File List* ☐ No separador *Main Page*.
☐ Na página do `Doxygen`. ☐ Na página *preciso d'oxygen*.
☐ Procurei por `program1.c` na barra de pesquisa e encontrei a sua documentação.

Q7.: Que secções são visíveis na documentação do `program1.c`?
☐ `Functions` ☐ `Detailed Description`
☐ `Variables` ☐ `Function Documentation`

Tarefa 5

Adicione o seguinte comentário à função `main` do programa `program1.c`:

```
...
/**
  This is the main function of the program.
*/
int main(){
...
```

Não se esqueça que os comentários seguindo a estrutura acima sugerida devem ser colocados antes das entidades ao qual se referem. Após efetuar as alterações pedidas deverá executar novamente o `Doxygen` na raíz do projeto. **Se não efetuar este passo, a documentação do seu projeto não será devidamente atualizada.**

Tarefa 6

Verifique a documentação gerada, abrindo, novamente, o ficheiro `index.html`. **Q8.: Notou alguma alteração?**

☐ Não existem alterações na documentação HTML.
☐ Surgiu um novo separador em *Files*, denominado *File Members*.
☐ Na página do `program1.c` existe agora uma secção chamada `Function Documentation`. Nesta secção existe o comentário que adicionei, associado à função principal (*i.e.*, `main`).
☐ Procurei por `main` na barra de pesquisa e obtive um resultado, coisa que antes não aconteceu!

Tarefa 7

Altere novamente o `program1.c` e adicione o seguinte comentário à variável `a`:

```
...
/// Global variable with value 20.
int a = 20;
...
```

Verifique, novamente, a documentação gerada. **Q9.: Apercebeu-se de alguma alteração?**

☐ Houve algumas alterações mas nada digno de registo.
☐ Não, pois a tarefa não pediu para executar o `Doxygen`.

Tarefa 8

Execute o `Doxygen` para gerar a documentação do programa e volte a dar uma vista de olhos. **Q10.: E agora, já se deu alguma alteração na documentação gerada?**

☐ Ainda não. Comentários associados a variáveis não são incluídos na documentação.
☐ Agora sim! Na página de `program1.c`, na secção `Variables`, foi agora incluido o comentário que inseri associado à variável `a`. Que MA-RA-VI-*(wait for it)*-LHA!

Tarefa 9

Por uma questão de **completude**, não se esqueça de incluir uma pequena descrição à função `makeSum`, seguindo eventualmente a seguinte sugestão:

```
...
/**
 * This funcition calculates the sum of the parameter with a constant.
 */
int makeSum(int b){
...
```

3 Utilização de Comandos Doxygen

No `Doxygen`, existem dois níveis de descrição para uma entidade: **breve** e **detalhada**. Os comandos `@brief` e `@details` podem ser utilizados para gerar, respectivamente, uma descrição breve e detalhada.

Tarefa 10

Adicione uma descrição breve e detalhada ao seu programa `program1.c`. Para este fim, insira os seguintes comentários no ponto certo da implementação:

```
/**
  @file program1.c
  @brief Pretty nice program.
  @details This program is used to sum two values. One of them is a
      constant.
*/
#include<stdio.h>
...
```

No final, não se esqueça que deve gerar novamente a documentação.

Atente à documentação gerada. **Q11.: Registou alguma alteração?**

- ☐ Estranho, não existem alterações.
- ☐ No separador *Files* > *File List*, aparece a descrição breve.
- ☐ Na página do `program1.c`, na secção `Detailed Description`, existe uma descrição breve e uma detalhada do `program1.c`.
- ☐ Caso de polícia, a diretoria da documentação está *VAZIA*!

Tarefa 11

Existem outros comandos que podem ser utilizados num ambiente `Doxygen`.

Explore a seguinte *hiperligação* e registe quais os comandos a utilizar para adicionar uma **versão** e **autor** ao seu programa. Adicione o seu nome como autor e "1.0" como versão, usando os comandos registados, na mesma região onde definiu a descrição breve e detalhada do programa. No final, execute o `Doxygen`.

Q12.: Após realizar a tarefa anterior, quais foram as alterações que identificou na documentação do programa?

☐ Na secção `Detailed Description` foram adicionados dois novos campos: `The Author Name` e `Version`.

☐ Na secção `Detailed Description` foram adicionados dois novos campos: `Author` e `Version`.

☐ Na secção `Detailed Description` foram adicionados dois novos campos: `Author` e `Final Version`.

☐ Não detetei nenhuma alteração à documentação do programa mas também não me lembro se executei o `Doxygen`...

Os comandos `@file`, `@brief` e `@details`, usados em tarefas anteriores, são exemplos de **comandos especiais** do `Doxygen`. A sintaxe adotada para este tipo de comandos é a utilização do prefixo `@` ou `\`. Ambos podem ser usados de forma permutável. Alguns exemplos de comandos especiais são exibidos na listagem seguinte:

- `@file`: utilizado para definir o nome do ficheiro. Deve sempre existir no cabeçalho do programa. Se **não incluir** este comando, o ficheiro do programa não será processado pelo `Doxygen`.
- `@param`: descreve os parâmetros de uma função.
- `@return`: descreve o(s) valor(es) de retorno de uma função.
- `@mainpage`, `@page` e `@subpage`: definem, na documentação, uma página principal, uma página e uma sub-página. As páginas são geradas a partir de ficheiros Markdown.

Tarefa 12

Modique o ficheiro `program1.c`, adicionando uma descrição dos valores de retorno das funções. **Q13.: Qual foi o comando especial que utilizou?**

☐ `@point` ☐ `@of` ☐ `@no` ☐ `@return`

Tarefa 13

Modique, novamente, o `program1.c`, adicionando uma descrição dos parâmetros da função `makeSum`. No final, execute o `Doxygen`. **Q14.: Qual foi o comando especial do `Doxygen` que utilizou?**

☐ `@return` ☐ `@function` ☐ `@param` ☐ `@voz`

Q15.: Em que secção da documentação foi apresentada a descrição dos parâmetros passados à função `makeSum`?

☐ `Function Documentation`. ☐ `Detailed Description`.
☐ Todas as anteriores. ☐ Nenhuma das anteriores.

4 Páginas de Documentação Auxiliares

O `Doxygen` permite gerar páginas *Web* em HTML a partir de ficheiros Markdown. Estas páginas serão posteriormente integradas na documentação final gerada pelo programa.

A utilização e integração destes ficheiros no `Doxygen` irá permitir adicionar trechos de código, informações relativas à arquitetura adotada pela aplicação ou outras informações que achar pertinente.

O `Doxygen` integra ficheiros Markdown em três categorias: ***main***, ***page*** e ***subpage***.

Tarefa 14

Crie, na diretoria `src`, um novo ficheiro chamado `main.md` e adicione o seguinte código:

```
@mainpage The Sum Program
This is the main page of the documentation of the project. Its contents
    can be summarized as follows:
1. Listing of the files implementing the software;
2. Listing and description of each function of the program;
3. Indication of reference links.
```

A sintaxe do comando especial `mainpage` é a seguinte: `@mainpage [(title)]` .

Q16.: Onde será exibido o conteúdo do ficheiro Markdown da tarefa anterior?

☐ Na página dos ficheiros (*Files* > *File List*) do projeto.
☐ Na página do `program1.c`, na secção `Detailed Description`.
☐ Na página principal (*Main Page*) do projeto.

Tarefa 15

Execute o `Doxygen` para confirmar a resposta à pergunta anterior.

Q17.: Dada a sintaxe do comando `mainpage`, o que pode comentar relativamente ao título?

☐ É obrigatório. ☐ É opcional.

☐ Pode não estar presente mas se não estiver o `Doxygen` não executa. Ou seja, pode não estar mas tem de estar. É uma questão paradoxal!

Tarefa 16

Na diretoria `src`, crie um novo ficheiro chamado `page.md` e insira-lhe o seguinte excerto:

```
@page workflow Page - Description of the Workflow

# The Workflow of the Program.
This is a text explaining the workflow of the program.

# The Details of the Workflow.
This is a text explaining something more.
```

No final, execute o `Doxygen`. Observe que a sintaxe do comando especial `page` é a seguinte: `@page <reference_name/label> (Title)` .

Q18.: Qual é a referência/etiqueta atribuída à página que acabou de criar?

☐ `workflow` ☐ `workflow Page` ☐ `page` ☐ `nao_sei`

Q19.: Verificou alguma alteração na documentação após a criação do ficheiro `page.md`?

☐ A informação na página principal (*Main Page*) foi substituída pela informação presente no ficheiro `page.md`.

☐ Surgiu um novo separador, intitulado `Related Pages`, com uma hiperligação com o título definido em `page.md`.

☐ Foi acrescentado o ficheiro `page.md` na página dos ficheiros (*Files* > *File List*).

☐ Na página do `program1.c`, na secção `Detailed Description`, foi introduzida a informação presente no ficheiro `page.md`.

Tarefa 17

Crie, na diretoria `src`, um novo ficheiro chamado `subpage.md` e adicione o seguinte código:

```
@page sum Subpage - Description of the Sum Function
```

```
# How to Sum Two Numbers
The mathematical explanation of summing two numbers.
```

No final, execute o `Doxygen`.

Q20.: Qual é a referência atribuída à sub-página que acabou de criar?
☐ `sum` ☐ `sum Subpage` ☐ `page` ☐ `nao_respondo`

Atente ao separador `Related Pages`, na documentação gerada. **Q21.: O que pode comentar relativamente à disposição dos ficheiros?**
☐ Apenas é visível uma hiperligação para `Subpage - Description of the Sum Function`.
☐ Apenas é visível uma hiperligação para `Page - Description of the Workflow`.
☐ São visíveis duas hiperligações, ambas com a mesma indentação relativamente à margem esquerda.
☐ São visíveis duas hiperligações, em que `Subpage - Description of the Sum Function` é uma sub-página de `Page - Description of the Workflow`.

Admita que tinha como objetivo a adição de `subpage.md` como sub-página de `page.md`. Para tal, teria que fazer uso do comando `@subpage`, na página `page.md`. **Q22.: Para atingir o seu objetivo, qual era o comando completo a introduzir em `page.md`?**
☐ `@subpage Subpage` ☐ `@subpage sum`
☐ `@subpage workflow` ☐ `@subpage Page`

Tarefa 18

Tendo em conta a resposta anterior, modifique o ficheiro `page.md` e execute o `Doxygen`.

Atente, novamente, ao separador `Related Pages`, na documentação gerada. **Q23.: A tarefa anterior promoveu alguma alteração?**
☐ Nenhuma. A tarefa não produziu nenhum efeito na disposição das páginas.
☐ `Subpage - Description of the Sum Function` é uma sub-página de `Page - Description of the Workflow`.
☐ `Page - Description of the Workflow` é uma sub-página de `Subpage - Description of the Sum Function`.
☐ O separador `Related Pages`, que outrora estava povoado com duas páginas, encontra-se agora desprovido de recursos. Precipitou-se algo de inexplicável e a música dos *X-Files* ocorreu-se-me.

Tarefa 19

O `Doxygen` disponibiliza a opção de adicionar um índice de conteúdos. Registe o comando que usaria para adicionar um índice de conteúdos ao ficheiro `page.md`. Para tal, use o comando que usaria em LaTeX ou pesquise no manual do `Doxygen`, usando a seguinte *hiperligação*.

Adicione o comando ao ficheiro `page.md`, abaixo de `@subpage sum`, e execute o `Doxygen`.

Atente ao conteúdo da hiperligação `Page - Description of the Workflow`, no separador `Related Pages`. **Q24.: Consegue visualizar o índice de conteúdos?**
☐ Sim, sem quaisquer problemas! ☐ Creio que não. Onde deveria estar o dito índice?

Tarefa 20

Modifique o ficheiro de configuração do `Doxygen`, alterando para 1 o conteúdo da variável `TOC_INCLUDE_HEADINGS`. Confirme que a variável `MARKDOWN_SUPPORT` tem como conteúdo `Yes`. No final, execute o `Doxygen`.

Q25.: Face a estas alterações, consegue visualizar o índice de conteúdos?
☐ Agora sim, consegui. ☐ Raios e coriscos, ainda não foi desta!

Tarefa 21

Tendo como exemplo o `Makefile` implementado em aulas anteriores, execute todas as alterações que achar necessárias para automatizar a geração de documentação, após a compilação do programa `program1.c`.