Programação Orientada para os Objectos

Departamento de Matemática
Faculdade de Ciências e Tecnologia
Universidade de Coimbra

Pedro Quaresma

2021/05/06 (v1037)

Bibliografia Principal

POO Booch, Grady. 1991. *Object Oriented Design with Applications.* Redwood City, USA: The Bejamin/Cummings Publishing Company, Inc.

C++ Stroustrup, Bjarne. 1997. *The C++ Programming Language.* Addison Wesley Longman, Inc.

C++ Rodrigues, Pimenta, Pereira, Pedro, & Sousa, Manuela. 1998. *Programação em C++.* 2 edn. FCA, Editora de Informática LDA.

C Kernighan, Brian, & Ritchie, Dennis. 1988. *The C Programming Language.* 2nd edn. Prentice Hall.

Bibliografia Completar

POO Budd, Timothy. 1996. *An Introduction to Object-Oriented Programming.* 2nd edn. Addison-Wesley. DMAT 68U/BUD.

POO Meyer, Bertrand. 1988. *Object-Oriented Software Construction.* Prentice-Hall International. DMAT 68N/MEY.

POO/C++ Stroustrup, Bjarne. 2009. *Programming: Principles and Practice Using C++.* Addison Wesley Longman, Inc.

C++ Lippman, Stanley B., & Lajoie, José. 1998. *C++ Primer.* 3rd edition edn. Addison-Wesley. DMAT 68N/LIP.

C++ Main, Michael, & Savitch, Walter. 1997. *Data Structures and other Objects Using C++.* Addison-Wesley. DMAT 68P/MAI.

C++ Semgupta, Saumyendra, & Korobkin, Carl Philip. 1994. *C++, Object-Oriented Data Structures.* New-York: Springer-Verlag. DMAT 68N/SEN.

C++ Guerreiro, Pedro. 2003 *Programação com Classes em C++.* FCA. Lisboa. ISBN: 9789727223756.

Páginas de Referência

C++ Standard Library cplusplus.com, `http://www.cplusplus.com/`.

C++ STL C++ STL (Standard Template Library) Tutorial and Examples, `http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html`.

Conteúdo

Lista de Figuras

Lista de Tabelas

Parte I

Metodologia de Programação Orientada aos Objectos

Capítulo 1

Introdução

1.1 Diferentes Metodologias

No prosseguimento do estudo da construção de programas correctos que sejam soluções de problemas típicos das ciências da computação, um factor, que frequentemente é apontado como causa das dificuldades nesta construção, é o próprio paradigma da programação sequencial imperativa.

As características da programação sequencial estão, conceptualmente, muito distantes das noções normalmente usadas na descrição dos problemas. Estas requerem abordagens formais para poderem ser descritas de um modo preciso e, normalmente têm uma natureza estática: isto é, o problema é o mesmo independentemente do instante de tempo em que é observado (existem, obviamente, excepções).

Por seu lado, a programação imperativa não é nada formal, muito pouco precisa e é dinâmica no sentido em que a dimensão tempo procura substituir a dimensão da complexidade: um programa grande é decomposto em problemas simples que são resolvidos um de cada vez, sequencialmente no tempo.

Daqui resulta que a resolução de problemas seria bastante mais simples se a metodologia de construção de programas seguisse de perto a metodologia de descrição dos problemas. Por isso será conveniente programar de um modo: formal, preciso, e em que a estrutura e complexidade do problema se manifeste directamente na estrutura e complexidade do programa.

Algumas metodologias de descrição de problemas que têm vindo a ganhar importância crescente são:

- Modelos, baseados na *teoria dos conjuntos*, para as estruturas de informação e *descrição funcional das operações* sobre essas estruturas.
- *Lógica de primeira ordem* sob várias formas particulares.
- *Regras de simplificação ou substituição.*
- *Teorias algébricas axiomáticas* (a chamada teoria dos tipos abstractos de dados) (Ehrig & Mahr, 1985; Ehrig & Mahr, 1990; Bergstra, 1989).
- Teoria dos *objectos* e das *comunicações entre objectos* (Meyer, 1990).

Existem outras metodologias, designadamente ligadas à noção de processo e comunicação entre processos, mas, como envolvem directamente o tempo como componente intrínseca do problema, não serão tratadas neste curso.

A cada uma das metodologias atrás referenciadas está associada uma (ou mais) metodologias de programação que, em maior ou menor grau, satisfazem as condições já referidas.

A primeira destas metodologias deu origem à metodologia formal mais antiga: a chamada *programação funcional*. Existem imensas abordagens distintas a esta metodologia, muitas delas, porém, muito longe de formais e precisas. Duas das que melhor satisfazem as nossas condições, centram-se nas linguagens *ML* (Paulson, 1991; Wikstrom, 1989) e *Haskell* (Bird, 1998; Thompson, 1996). Para um exemplo de uma implementação de uma pilha genérica em *Haskell* ver o apêndice D, secção D.1.1.

A lógica de primeira ordem foi sempre um veículo para a criação de metodologias de descrição de problemas: uma das primeiras áreas abordadas pelas metodologias formais foi a criação de demonstradores de teoremas.

No entanto a sua complexidade e a indecidibilidade da lógica de primeira ordem não permitia a criação de sistemas computacionais que reproduzissem toda a teoria. Assim procurou-se encontrar um fragmento da lógica de primeira ordem que fosse decidível e bem adaptado à construção de linguagens de programação, a lógica das cláusulas de Horn é um desses fragmentos sendo a base da linguagem de programação em lógica *Prolog* (Lloyd, 1987).

A terceira metodologia está ligada à demonstração de teoremas por simplificação sistemática de formulas lógicas mas também, à descrição de problemas através de regras de transformação.

Este tipo de descrição de problemas deu origem a sistemas computacionais que têm o nome genérico de *sistemas de reescrita*. Um desses sistemas chama-se *ERIL*.

Os tipos abstractos de dados são uma metodologia básica em ciências da computação. A família de linguagens descendentes da linguagem OBJ constituem uma família de linguagens que implementam estes conceitos, temos nomeadamente as linguagens *OBJ3* (Goguen *et al.*, 1992), *CafeOBJ* (Diaconescu & Futatsugi, 1998), e *Maude* (McCombs, 2003; Clavel *et al.*, 1999). Para um exemplo de uma implementação de uma pilha genérica em *CafeObJ* ver o apêndice D, secção D.1.2.

Em relação às metodologias que usam objectos como conceito base tem-se que, infelizmente não existem sistemas computacionais que assentem sobre uma metodologia formal e precisa. Inversamente as (poucas) descrições precisas da noção de objecto não têm suporte em qualquer sistema computacional que seja acessível. Metodologias de objectos com descrição formal mas sem suporte computacional são, por exemplo, *OBLOG* e *FOOPS*. A situação inversa tem muitos mais representantes: *Smalltalk*, *C++* (Stroustrup, 2009; Stroustrup, 1997), *Java*, etc.

1.2 Metodologia de Programação Formal

Programas e problemas são, normalmente, sistemas grandes e complexos. A capacidade humana para entender tais sistemas é limitada e só consegue ter sucesso quando consegue aplicar uma de duas estratégias: a abstracção e a modularidade.

Abstracção A abstracção é a capacidade para agrupar um número elevado de casos concretos diferentes (mas semelhantes) numa única entidade a que podemos dar o nome de *classe*, isto é, as propriedades comuns a todos os casos concretos de modo a que, face a essas propriedades, seja possível:

- prever os efeitos das transformações a que, eventualmente, estejam sujeitos os diferentes casos dentro da classe;

- validar frases lógicas que se apliquem universalmente a todos os casos da classe.

A estratégia de abstracção baseia-se em três técnicas fundamentais:

Instanciação Dada uma classe criar um dos seus casos (uma instância da classe);

Abstracção Dados dois ou mais casos concretos, caracterizar a "menor classe" que os contém como instâncias;

Validação Dada uma classe e um caso concreto verificar se o caso é uma instância da classe
ou
Prever os efeitos de transformações em instâncias de classe
ou
Validar frases lógicas quantificadas universalmente às instâncias da classe.

Modularidade A modularidade é a capacidade para decompor casos complexos numa colecção de casos mais simples (chamados módulos) e em regras de composição de tal modo que:

- A análise (ou síntese) de qualquer um dos módulos seja independente da análise (ou síntese) dos restantes. De preferência cada módulo deve ser uma instância de uma classe de módulos bem conhecida.
- É possível construir o significado do caso complexo por aplicação das regras de composição ao significado dos módulos. Analogamente, se o problema for de síntese, as regras de composição devem permitir construir o sistema global a partir dos módulos.

Uma das características importantes em qualquer metodologia é o seu suporte à abstracção e à modularidade.

O modo mais directo para uma metodologia de programação poder suportar abstracção é o facto de possuir uma teoria de tipos bem desenvolvida.

As linguagens *ML* e *Haskell* (ao contrário do seu ilustre antecessor, *Lisp*) possuem tal teoria de tipos. O mesmo se passa com o *CafeOBJ* e, em certo grau, com o sistema de reescrita *ERIL*. De uma forma geral as linguagens de programação orientada para os objectos suportam (de forma menos formal é certo) a construção de novos tipos e a sua inclusão na estrutura de tipos pré-existente. Apenas o *Prolog* (tal como o *Lisp*) não suporta qualquer teoria de tipos sendo que, dificilmente suporta as técnicas de abstracção.

A mesma observação pode ser feita em relação à modularidade; dada a íntima relação entre estes dois conceitos, não é coincidência que uma teoria de tipos bem desenvolvida esteja ligada a uma boa gestão de módulos.

As linguagens *ML*, *Haskell* e *CafeOBJ*,assim como a maioria das linguagens de programação orientada para os objectos, nomeadamente o *C++*, suportam a modularidade de programas. Porém nem Prolog nem Eril suportam qualquer tipo de modularidade.

A apresentação das várias metodologias terá em vista sempre o modo como estas técnicas podem (ou não) ter um suporte adequado.

Se todas estas metodologias oferecem tantas vantagens quando comparadas com a programação imperativa, qual a razão porque a grande maioria dos programas que correm na generalidade dos sistemas informáticos continuam a ser imperativos?

A resposta é simples: eficiência na execução. A programação imperativa está ligada intimamente à arquitectura da máquina que os executa e, como dissemos, substitui a dimensão

complexidade do problema pela dimensão tempo. De certo modo podemos dizer que um problema grande divide a ocupação da máquina, pelos seus módulos ou componentes, através de fatias de tempo.

Quando a complexidade total do problema é reflectida na estrutura do programa e não partilhada no tempo, é natural que tal programa seja, à partida, mais exigente em termos de recurso da máquina.

Por outro lado, as metodologias de programação formal, fazem automaticamente para qualquer programa um conjunto de serviços genéricos que não existem nas metodologias de programação imperativas; em particular, toda a gestão de memória, é usualmente transparente ao programa que não lida com problemas de detalhe de implementação.

Na programação imperativa os custos da implementação estão sempre presentes e, portanto, é mais simples melhorar a eficiência do programa.

A programação formal, porém, oferece um tipo mais importante de eficiência: a eficiência no desenvolvimento e a garantia de correcção.

Capítulo 2

Metodologia de Programação Orientada para os Objectos

2.1 Introdução

A, assim designada, *Crise da Programação* ("Software Crises") surgiu quando se tornou evidente que os problemas que surgiam no desenvolvimento dos programas não eram algorítmicos mas sim de comunicação e de complexidade.

Como já foi referido a abstracção e a modularidade, e a forma como as diferentes metodologias (e as linguagens que suportam essa metodologia) as implementam são os pontos essenciais a analisar.

Proposição 1 (Princípio de Parmas (David Parmas))

1. *Deve-se dar ao utilizador de um módulo toda a informação necessária para este usar o módulo correctamente, e nada mais.*

2. *Deve-se dar ao programador de um módulo toda a informação necessária para este completar o módulo, e nada mais.*

O conceito de *Tipo Abstracto de Dados* (TAD) vem nesse sentido temos que ao definir-se um tipo abstracto de dados pretende-se:

Definir um novo tipo, com o seu conjunto de elementos e operações internas, mas de forma a que possamos sonegar a informação da forma como a implementação foi efectivada, e de forma a que seja possível ter múltiplas instâncias do tipo de dados que se está a definir.

Os dois conceitos cruzam-se, por um lado estamos a falar da modularidade/abstracção algorítmica, por outra estamos a falar das estruturas de dados e da forma de as implementar também de forma modular e abstracta.

A metodologia da programação orientada para os objectos tenta combinar as duas aproximações definindo os programas como colecções de objectos, entidades que agregam algoritmos e estruturas de dados, e a comunicação entres os objectos, de forma a que possamos ter:

- encapsulamento da informação (como os módulos);
- re-utilização do código através de um mecanismo de instanciação (como os TAD.);

- modularidade, através da definição de um programa como uma colecção de objectos capazes de comunicar entre si.

Temos então que podemos definir a metodologia de programação orientada para os objectos da seguinte forma:

Definição 1 (Metodogia de Programação Orientada aos Objectos) *A metodologia de programação orientada para os objectos (POO) é uma forma de programar na qual os programas estão organizados como uma colecção de objectos que cooperam entre si. Cada objecto representa uma instância de uma dada classe sendo que as classes formam entre si uma hierarquia de classes ligadas por relações de diferentes tipos.*

Uma linguagem de programação é dita uma linguagem de Programação Orientada para os Objectos se:

- Suporta *Objectos* que são abstracções de dados e operações sobre esses dados, com um interface constituído por nomes das operações e um estado interno não visível do exterior;
- Objectos têm um tipo (classe) associado;
- Tipos (classes) podem herdar atributos de super-tipos (super-classes).

Quais são os elementos estruturantes da metodologia de programação orientada para os objectos.

- Abstracção;
- Encapsulação;
- Modularidade;
- Hierarquia;

Também importantes são os conceitos de construção de tipos ("typing"), de concorrência (paralelismo) e de persistência.

Definição 2 (Abstracção) *Por abstracção entende-se o conjunto da(s) características fundamentais de uma dada entidade (classe de objectos), aquelas que a distinguem de todos as outras entidades, e que como tal estabelecem uma fronteira bem definida para o utilizador.*

Relacionado com o conceito de abstracção temos o conceito de instância. A abstracção permite-nos, tal como foi dito, diferenciar uma classe de objectos, por exemplos: os veículos automóveis, em contra-posição com os não automóveis, dentro desta classe podemos ainda identificar algumas sub-classes: os carros; os autocarros; as camionetas, etc. Dentro dos carros podemos ainda identificar diferentes instâncias: os utilitários; os todo-o-terreno; os familiares; as carrinhas. Finalmente podemos ver que para cada uma destas instâncias teremos muitas instâncias, um carro utilitário de uma dada marca/modelo.

Temos aqui (Fig. 2.1) um primeiro exemplo de uma hierarquia (conceito de que falaremos em pormenor mais adiante) de classes e de instâncias.

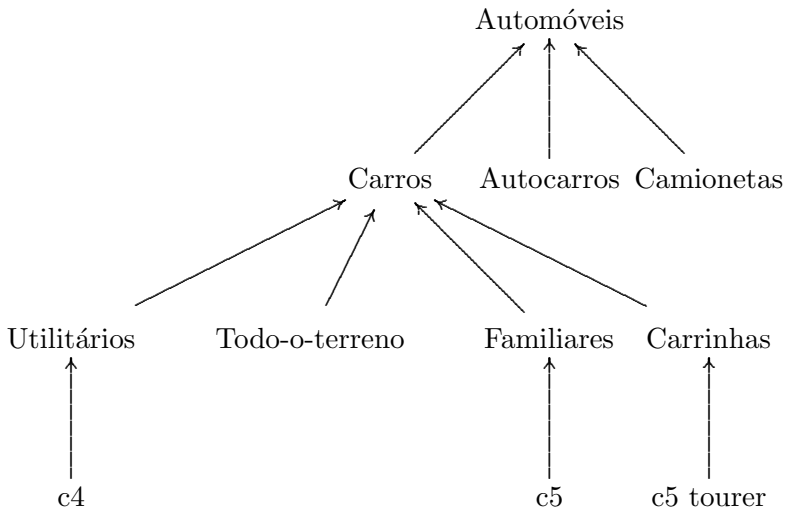


Figura 2.1: Hierarquia de Classes

Definição 3 (Encapsulamento) *Por encapsulamento, ou sonegação da informação, entende-se o acto de esconder (sonegar) os detalhes do objecto que não contribuem para as suas características fundamentais.*

Continuando no exemplo anterior temos que, para o utilizador de um objecto do tipo carro pode não ser relevante o tipo de estrutura que suporta as suas componentes, o facto de ser um "chassis" ou uma "carroçaria" é, para a maior parte dos utilizadores, irrelevante. Como tal essa informação pode ser sonegada ao utilizador, sendo no entanto relevante na classe para caracterizar um carro.

Em *C++* temos os seguinte níveis de visibilidade da informação contida num objecto:

public a informação disponível nesta secção será visível a todas as componentes do programa;

private a informação disponível nesta secção só será visível às componentes internas do objecto. Invisível para o exterior;

protected a informação disponível nesta secção é visível às componentes internas do objecto, assim como para os sub-objectos desta.

Um outro conceito que está muito associado aos dois já descritos é o da modularidade.

Definição 4 (Modularidade) *Por modularidade entende-se a propriedade de um sistema se poder decompor num conjunto de módulos coerentes e fracamente ligados.*

Isto é um sistema em que as abstracções (classes) que estão relacionadas logicamente está agrupadas (coerentemente) e em que as ligações entre módulos são minimizadas ao estritamente necessário.

Definição 5 (Hierarquia) *Por hierarquia de classes entende-se a classificação/ordenação das classes.*

Veja-se o exemplo dado acima 2.1.

Em termos da metodologia da programação orientada aos objectos as relações que se podem estabelecer entre classes são dos seguintes tipos:

Herança ("is a") a relação de herança entre classes é tal que uma das classes partilha a estrutura e/ou o comportamento numa outra classe (herança simples), ou em várias outras classes (herança múltipla).

Por exemplo: na hierarquia 2.1 temos que carros vai herdar de automóveis algumas das suas características, seja em termos de estrutura, seja em termos de comportamento.

Generalização/Especialização quando se tem uma situação de relações entre classes em que uma deles é uma especialização de uma outra, ou dito doutro modo quando uma dada classe generaliza um conjunto de outras classes.

Um exemplo de uma situação de generalização/especialização é dado pela classe *Pilha*. Pretende-se definir a classe genérica *Pilha*, sem mencionar o tipo dos elementos (genéricos), uma *Pilha de Elementos*, posteriormente, aquando da utilização, vai-se instânciar os *Elementos* para, por exemplo, inteiros, obtendo-se a, *Pilha de Inteiros*.

Agregação ("part of") quando se pretende agregar (juntar) numa dada classe um conjunto de outras classes.

Na hierarquia 2.1 não temos nenhum caso de agregação, no entanto se fossemos a pensar numa classe de "peças de um carro", então poderíamos ter o caso de uma agregação em que as diferentes peças se agregam na entidade carro.

Em *C++* vamos poder definir relações de herança múltipla sendo que a hierarquia formada pelas classes é regida pelos atributos (aqui aplicados a classes) `public`, `private` e `protected`.

Uma característica importante numa linguagem de programação, isto porque está intimamente ligada com as suas características de abstracção e de modularidade das estruturas de dados, é a de possuir uma hierarquia de tipo forte ("strongly typed") com suporte ao polimorfismos.

Isto é, deve-se poder criar novas estruturas de dados, ou seja novos tipos de elementos e as operações que os suportam, por exemplo os números complexos, $(\mathbb{C}, \{+, -, \times, /\})$. Este construir de um novo tipos deve ser tal que: defina um novo tipo, isto é, que se possa usar o tipo definido de forma idêntica ao tipos primitivos da linguagem; que os pormenores da implementação sejam totalmente sonegados, isto é, o utilizador pode utilizar os novos elementos e operações entre eles, sem ter, e sem necessitar de ter, conhecimento dos pormenores da implementação. Finalmente o novo tipo deve-se poder "encaixar" na hierarquia de dados já existentes para, por exemplo, se poder multiplicar um real por um complexo, para, por exemplo, se poder ter uma lista ordenada de elementos do novo tipo.

O *C++* é uma linguagem como uma hierarquia de tipo forte ("strongly typed").

Por polimorfismo entende-se a possibilidade de se poder designar por um único nome diferentes objectos em classes diferentes, mas relacionadas por uma super-classe comum.

A possibilidade de ter operações e tipos polimórficos, que de certa forma significa a possibilidade de ter variáveis de tipo, isto é variáveis não sobre uma classe de elementos, mas sim sobre a classe dos tipos de elementos, vai permitir ter definir operações e tipos de dados genéricos acrescentando um outro nível de abstracção à linguagem.

Por exemplo, podemos pensar em definir um tipo de dados "lista de dados de tipo numérico" ou mesmo "lista de elementos". Isto é definiu-se a lista de elementos num dos

nós da hierarquia (elementos internos) de dados e não nas folhas (nós terminais) dessa mesma hierarquia. Para uma hierarquia tal como a 2.2 o polimorfismo permitiria então definir uma "lista de elementos de dados de tipo numérico", que depois poderia ser instanciada em "lista de inteiros", "lista de reais" em vez de se terem que definir listas de inteiros, e depois voltar a definir listas mas desta vez de reais, e mais à frente lista de complexos, etc.

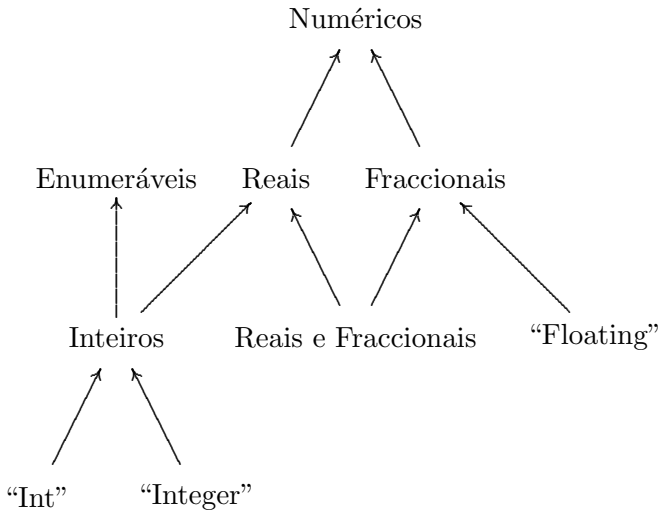


Figura 2.2: Hierarquia (parcial) de Tipos (Haskell)

2.2 Objectos e Classes

Os objectos são as componente fundamentais, de certa forma as componentes atómicas da metodologia de programação orientada aos objectos.

Um objecto é:

- algo tangível e/ou visível;
- algo que pode ser compreendido intelectualmente;
- algo sobre o qual se pode exercer uma acção.

Um objecto é algo que possuí uma fronteira bem definida e que possuí um estado, um comportamento e uma identidade própria.

Definição 6 (Objecto, Classe, Instância (Booch, 1991)) *Um* objecto *é algo que tem um estado um comportamento e uma identidade. A estrutura e o comportamento de objectos similares são definidos na sua* classe *de objectos comuns. Os termos* instância *de um classe e de objecto têm o mesmo significado.*

Na definição de objecto falamos de *estado*, de *comportamento*, de *identidade*, de *classe* e de *instância*. Vejamos em mais pormenor cada um destes conceitos quando aplicados aos objectos.

Um exemplo, uma máquina de venda automática.

Uma máquina de venda automática (bebidas, bilhetes, cigarros, etc.). O comportamento habitual de uma máquina deste tipo é o seguinte: uma pessoa introduz as moedas; faz a selecção do produto que quer; a máquina dá o produto escolhido, assim como o troco.

O que é que acontece se o utilizador não respeitar a ordem das operações; por exemplo tentar seleccionar o produto antes de colocar as moedas? Provavelmente nada acontece, a máquina não está num estado em que aceita as escolhas de produtos, a máquina está num estado de aceitação de moedas.

Definição 7 (Estado (Booch, 1991)) *O estado de um objecto incorpora todas as propriedades, usualmente estáticas, de um objecto assim como os valores, usualmente dinâmicos, de cada uma dessas propriedades.*

Uma propriedade da máquina automática de venda é que aceita moedas (propriedade estática). A quantidade de moedas que contém num dado momento é um valor (dinâmico) dessa propriedade.

Vejamos uma possível estrutura de dados em *C/C++* para uma ficha pessoal.

```
struct RegistoPessoal {
  char nome[100];
  int nContrib;
  char Departamento[20];
  float salario;
};
```

Estamos aqui perante uma classe de objectos, uma instância desta classe, isto é um objecto, seria declarado desta forma:

```
struct RegistoPessoal paulo,maria,jorge;
```

Estritamente falando ainda não estamos perante um objecto dado que não há nenhum comportamento (operações) associado a este registo. Esta é uma definição de uma linguagem procedimental sem ser orientada aos objectos, mais concretamente é uma definição em *C*.

Numa linguagem orientada aos objectos, tal como o C++, pode-se definir um objecto já com sonegação de informação, isto é podemos definir o estado do objecto como sendo interno ao objecto.

```
class RegistoPessoal {
public:
  char *nomeEmpregado() const;
  int  nContribEmpregado() const;
  char *departamentoEmpregado const;
protected:
  void introduzNome(char *nome);
  void introduzNContrib(int nif);
  void introduzDepartamento(char *departamento);
  void introduzSalario(floar salario);
  float salarioEmpregado() const;
private:
  char  nome[100];
  int nContrib;
  char Departamento[20];
  float salario;
};
```

Pode-se afirmar que: todos os objectos num dado programa contêm um dado estado e todo o estado do programa está contido em objectos.

2.2.1 Comportamento

Os objectos de uma linguagem de programação orientada aos objectos distinguem-se das estruturas de dados das linguagens procedimentais no facto de terem associados a si um dado comportamento.

Definição 8 (Comportamento (Booch, 1991)) *O* comportamento *de um objecto é definido pelas acções e reacções de um objecto em termos das suas mudanças de estado e das mensagens que envia aos outros objectos.*

Numa linguagem de programação orientada para os objectos é usual falar-se de *métodos* para designar as operações implementadas num dado objecto. Em *C++* é usual designar tais operações por *funções membro*, como tal vamos considerar tais designações como equivalentes.

Mensagens é a designação dada às operações que um dado objecto cliente requer a um outro objecto, isto é não são mais do que chamadas de funções mas de um objecto para um outro.

Existem diferentes tipos de métodos característicos de uma linguagem programação orientada para os objectos, são eles:

Construtores: criam, e eventualmente inicializam, o estado de um objecto.

Destrutores: destroem, libertando o espaço de memória ocupado, um objecto.

Manipuladores: modificam o estado de um dado objecto.

Selectores: acedem ao estado de um objecto sem no entanto o alterar.

Em *C++* temos que os construtores e os destrutores fazem parte da declaração da classe. Os manipuladores e os selectores podem ser escritos dentro da definição da classe, caso em que se designam por métodos, ou podem ser escritos fora da definição da classe como *sub-programas livres*.

Vejamos agora um exemplo de definição de uma classe, a classe das *pilhas* de caracteres.

Em C++ é usual separar a definição da classe, da definição dos seus sub-programas livres. O programa de chamada é por sua vez independente destes dois fazendo a sua inclusão aquando da diferentes fases da compilação.

O grafo de dependências (ver Figura 2.3), dá-nos uma indicação de um potencial problema: o ficheiro `pilhasChar.hpp` é destino de dois caminhos distintos. Neste caso concreto não há ambiguidade, os pontos de partida são distintos, no entanto em situações mais complexas uma situação de dupla inclusão pode ocorrer, sendo que estas situações de dupla inclusão levariam a uma repetição de declarações, o que é claramente uma situação de erro. A forma de resolver estes problemas é dado pelas directivas de compilação `ifdef` (ver secção 2.2.1). O significado dessas directivas será discutido com mais detalhe no apêndice C.

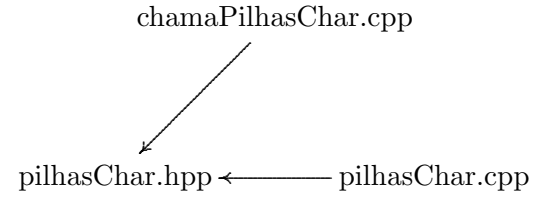


Figura 2.3: Dependências entre ficheiros, ChamaPilhas

Definição da Classe (`pilhaChar.hpp`) A definição da classe é feita num ficheiro `.hpp`, os métodos serão implementados como sub-programas livres, num ficheiro `.cpp`.

Dada o potencial ciclo no grafo das dependências, todo o ficheiro `.hpp` é construído dentro de um condicional (`ifndef`). Deste modo assegura-se que o mesmo é incluído uma só vez aquando da compilação.

```
#ifndef PILHAS
#define PILHAS

class Pilha {
public:
  // construtor
  Pilha(int sz = Tamanho_por_omissao);
  // destrutor
  ~Pilha();
  // manipuladores
  void cria();
  void push(char);
  void pop();
  char top();
  int vazia();
private:
  int tamanho;
  static const int Tamanho_por_omissao = 10;
  char *topo, *pilha;
};

#endif
```

Implementação dos sub-programas livres (`pilhaChar.cpp`) Os manipuladores e selectores são, nesta aproximação, implementados como sub-programas livres. É de notar que, dado que a sua implementação não é feita dentro da estrutura da classe é necessário prefixá-los com o nome da classe.

```
#include <iostream>
#include "pilhasChar.hpp"

/*
 * construtor - cria a Pilha afectando o espaço de memória
 * ->
 * <- ponteiro para o espaço afectado
 */
Pilha::Pilha(int sz){
  // a Pilha vai ser definido como um vector de dimensão "sz"
```

```
  topo = pilha = new char[tamanho=sz];
};

/*
 * destrutor - liberta o espaço ocupado pela pilha
 * -> pilha
 * <- ponteiro com valor indefinido
 */
Pilha::~Pilha(){
  delete [] pilha;
};

/*
 * push - coloca um elemento no topo da pilha
 * -> elemento, pilha
 * <- pilha com o elemento no seu topo
 */
void Pilha::push(char c){
  topo++;
  *topo = c;
};

/*
 * pop - retira um elemento do topo da Pilha
 * -> pilha
 * <- pilha com menos um elemento
 * Nota: a situação de pilha vazia não está salvaguardada
 */
void Pilha::pop(){
  --topo;
};

/*
 * top - devole o elemento do topo da Pilha
 * -> pilha
 * <- elemento (topo da pilha)
 * Nota: a situação de pilha vazia não está salvaguardada
 */
char Pilha::top(){
  return *topo;
};

// ?vazia
int Pilha::vazia(){
  return(pilha==topo);
};
```

O Programa de Chamada (`chamaPilhaChar.cpp`) Um exemplo simples de utilização da classe *Pilha* de caracteres é dado de seguida. Instância-se a classe `Pilha`, criando deste modo o objecto `pilha`, o qual é criado com a dimensão definida por omissão, isto dado que se escolheu utilizar o construtor sem argumentos. Uma utilização equivalente seria `Pilha pilha(10)`.

É de notar, que a exemplo dos programas escritos na linguagem *C*, o programas principal é definido pela função `main`, que, novamente a exemplo dos programas escritos na linguagem *C*, pode aceitar argumentos dados aquando da invocação na linha de comando.

```
#include <iostream>
#include "pilhasChar.hpp"

using namespace std;

/*
 * Pilha de elementos do tipo char
 */
int main() {
  Pilha pilha;   // valor por omissão, 10
  char car;

  cout << "Introduza dois caracteres:\t";
  cin >> car;
  pilha.push(car);
  cin >> car;
  pilha.push(car);
  cout << "\n\nIntroduziu: " << pilha.top() << ", ";
  pilha.pop();
  cout << pilha.top() << endl;
}
```

No caso do exemplo anterior só se definiu (instanciou) um objecto, podemos no entanto definir vários objectos de uma forma em tudo idêntica às definições das variáveis dos tipos pré-definidos. Por exemplo:

```
Pilha p1(10),p(20);
```

Note-se que neste caso optou-se por dar o valor de `tamanho` da pilha de forma explícita. Este valor sobrepõe-se ao valor definido por omissão.

Definição 9 (Identidade (Booch, 1991)) *A* identidade *de um objecto é a propriedade que um objecto possuí que o distingue de todos os outros objectos.*

A criação de um objecto é então feito por recurso aos construtores definidos aquando da declaração da classe. Nesse momento a classe é instanciada criando-se um novo objecto com o nome que se utilizou na declaração.

À semelhança de uma declaração de variáveis numa linguagem procedimental convencional há uma atribuição de um dado espaço de memória, de um tipo pré-definido, ao programa, associando-lhe ao mesmo tempo um nome.

Ao contrário de uma linguagem procedimental convencional, no caso de uma linguagem de programação orientada para os objectos temos que a definição de uma classe define um novo tipo de elementos, com os seus elementos e as suas operações internas, a declaração de um objecto, criando deste modo uma instância da classe, está associado o espaço necessário para guardar o seu estado (atributos), mas também um comportamento (métodos) que esse novo objecto pode ter.

Neste contexto faz sentido falar de atribuição e de igualdade? Isto é, como é que podemos então copiar o estado de um objecto para um outro (atribuição) e como é que podemos comparar objectos?

Tanto a atribuição entre objectos de uma mesma classe, como a relação de igualdade (e de desigualdade) são definidas de forma automática. Para atributos estáticos, isto é não envolvendo ponteiros, a atribuição e a igualdade têm o comportamento esperado: no caso

da atribuição os valores (atributos) de um objecto são copiados para o outro; no caso da igualdade, os valores dos atributos são comparados.

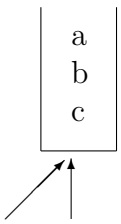
No caso dos objectos cujos atributos sejam definidos de forma dinâmica a questão é semelhante, mas talvez não a esperada. Se nada for dito em contrário tanto a atribuição como a igualdade são vistos em termos de ponteiros.

Isto é, por omissão a atribuição entre objectos não é mais do que a cópia de ponteiros, atribui-se o ponteiro do objecto à direita ao ponteiro do objecto da esquerda da instrução de atribuição, os dois objectos ficam a partilhar o mesmo estado. Em termos da igualdade a situação é semelhante, por omissão, a igualdade é feita em termos de ponteiros e não dos valores apontados, ou seja dois objectos são considerados diferentes se o estado associado a cada um deles não é o mesmo, isto mesmo que os valores que aí se encontrem sejam iguais.

Temos então que, por omissão, em atributos definidos de forma dinâmica (ponteiros) o comportamento da instrução de atribuição e do predicado de igualdade é o seguinte:

```
Pilha p1,p2;

(...)
p1 = p2;
```



p1 = p2;

Uma situação idêntica ocorre com declaração e inicialização de um objecto. A exemplo do que acontece com os outros tipos também é possível declarar e inicializar um objecto, a sintaxe é a seguinte:

```
Pilha p1(10),p2(p1);
```

sendo que o efeito é de uma declaração seguida imediatamente por uma atribuição.

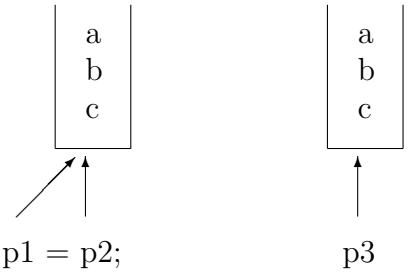
Podemos modificar este comportamento definindo explicitamente o comportamento desta instrução (operador de atribuição), seja através de uma definição vazia, o que o inviabiliza, isto é, deixa de ter efeito.

```
private:
  // inviabilizar a atribuição
  void operator=(const Pilha&) {};

  // inviabilizar a inicialização por cópia
  Pilha(const Pilha&) {};
```

No primeiro caso trata-se da instrução/operador de atribuição, no segundo caso trata-se da atribuição feita aquando da inicialização. A palavra reservada `operator` surge associada à definição de símbolos/palavras reservadas com uma definição polimórfica e não necessariamente em notação pré-fixa.

No caso da igualdade temos:



O comportamento por omissão dar-nos-á que $p_1 = p_2 \neq p_3$, no entanto podemos redefinir o operador `==` de forma a que o conteúdo seja tido em conta e se tenha $p_1 = p_2 = p_3$. Os pormenores de como o fazer estão na parte referente à linguagem *C++* (Parte II).

Retomando o conceito de classe.

Definição 10 (Classe (Booch, 1991)) *Uma classe é um conjunto de objectos que partilham uma estrutura e um comportamento comuns.*

Em *C++* é usual separar a definição de uma classe em duas componentes distintas, sendo que esta separação é em geral física, isto é, as duas componente são colocadas em ficheiros separados.

***header file*, "<nome_ficheiro>.hpp"** O interface da classe dá-nos a visão exterior sobre a classe. Consiste principalmente na definição do estado e nas declarações do todas as operações aplicáveis às instâncias da classe.

***C++ file*, "<nome_ficheiro>.cpp"** A implementação da classe dá-nos a visão interna da classe. Consiste principalmente na implementação das operações como sub-programas livres.

O interface da classe pode-se dividir em três secções:

Secção Pública (`public`) consiste nas declarações que são visíveis a todos os "clientes";

Secção Protegida (`protected`) consiste nas declarações que só são visíveis para as sub-classes (ver-se-à de imediato o que se entende por sub-classe);

Secção Privada (`private`) consiste nas declarações que só são visíveis para a própria classe.

Um programa em programação orientada para os objectos consiste então num conjunto de classes/objectos inter-relacionados, com um ponto de chamada global. No caso da linguagem *C++* ter-se-á, a exemplo dos programas em *C*, a função `main` a qual tem exactamente as mesmas características que tem em *C*.

Antes de começar a secção seguinte vejamos como representar graficamente uma classe. A representação gráfica das classes e, como vamos ver na secção que se segue, das relações entre elas é um instrumento muito útil na concepção de um programa em programação orientada para os objectos.

Para a representação gráficas sistemas de classes e relações entre classes vai-se usar o o formalismo UML (*Unified Modeling Language*,ver apêndice B).

Na figura 2.4 temos múltiplas representações para a mesma entidade mas com um nível de detalhe variável. Momento a momento vai ser necessário decidir qual o nível de detalhe que se quer mostrar, sendo que não é obrigatório que todas as classes num dado diagrama estejam representadas com o mesmo nível de detalhe.

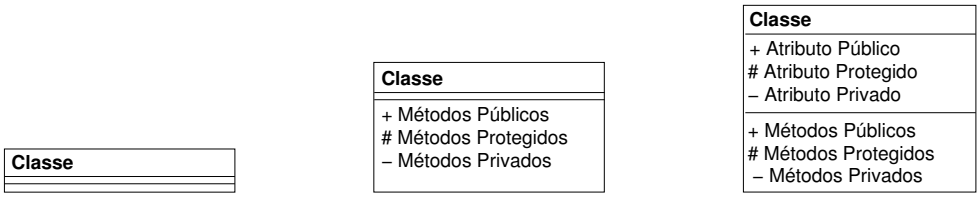


Figura 2.4: UML - Representação de uma Classe

2.3 Relações entre Classes

Consideremos as seguintes classes de objectos *flores, margaridas, túlipas negras, túlipas amarelas, pétalas, caules*. Podemos relacioná-las, podemos dizer que:

- um margarida é um tipo de ("is a/kind of") flor;
- uma rosa é um tipo de flor;
- as túlipas negras e amarelas são ambas túlipas;
- pétalas e caules são partes ("part of") flores.

Isto é, podemos estabelecer uma estrutura de objectos inter-relacionados.

Na metodologia de programação orientada aos objectos é usual considerar as seguintes relações entre objectos:

Herança ("kind of"/"is a")] rosas e túlipas são dois tipos de flores;

Agregação ("part of") pétalas e caules são partes de uma flor;

Generalização algo generalizável com múltiplas e diferentes concretizações;

Associação explicitando uma determinada relação semântica entre objectos, de outra forma não relacionados.

Como exemplo deste último tipo de relação temos que, por exemplo, rosas e velas de iluminação são objectos independentes e sem uma relação evidente, podemos no entanto querer-las relacionar associando-as ambas como elementos decorativos de uma mesa de jantar (ver Fig. 2.5).

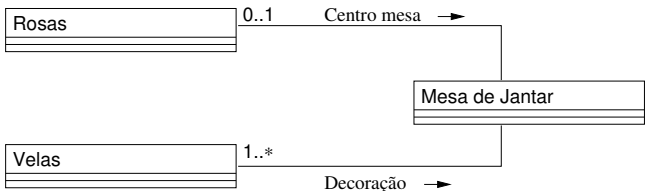


Figura 2.5: Associação

As relações de generalização, agregação e de associação são suportadas de diferentes formas pelas diferentes linguagens de programação orientada para os objectos. Nomeadamente através da implementação de relações de:

- herança (ver Fig. 2.6 e Fig. 2.7);
- agregação (ver Fig. 2.8);
- instanciação (ver Fig. 2.9);
- meta-classe (ver Fig. 2.10).

A relação de herança, na qual uma sub-classe herda, isto é, tem acesso à componente protegida (além da pública, como é óbvio) de uma outra classe. É fácil de ver que podemos ver a relação entre a classe flores e as classes rosas e túlipas como uma relação de herança. As classes rosas e túlipas herdam da classe flores todas as características que são comuns às flores, definindo de seguida aquilo que as diferenciam entre si (ver Fig. 2.6).

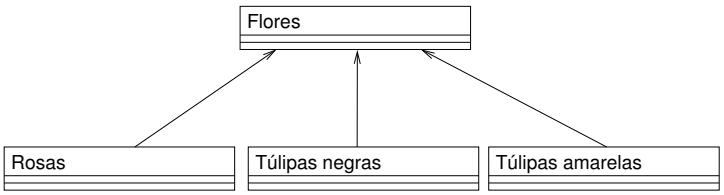


Figura 2.6: Herança Simples

A herança múltipla ocorre sempre que uma dada classe herda algumas das características de mais do que uma classe, veja-se o exemplo dado na figura 2.7.

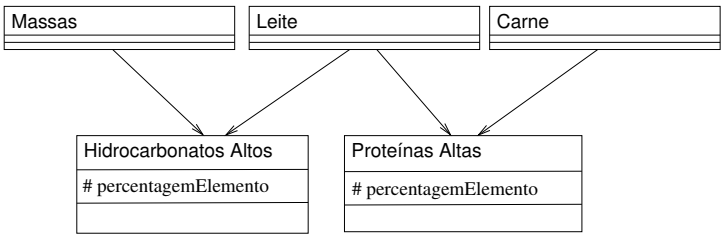


Figura 2.7: Herança Múltipla

A agregação ("part of") ocorre numa situação de utilização de várias classes por uma outra classe, por exemplo, uma flor agrega , entre outras, pétalas e caules. Podemos ter duas situações, a primeira em que a utilização é feita na secção protegida da classe, ou seja a classe que se está a utilizar fica escondida na classe que agrega as várias classes, a outra situação é o oposto, quando a classe utilizada fica visível na classe que a está a utilizar. Estas duas situações são diferenciadas do seguinte modo: se a classe vai ficar visível a terminação não é preenchida, se por acaso a classe vai ficar escondida a terminação é preenchida (ver Fig. 2.8).

As relações de instanciação, por exemplo quando numa classe parametrizável se fornece o parâmetro criando uma classe concreta são representadas por uma seta a tracejado (ver Fig. 2.9).

A relação de herança é a a mais rica semanticamente podendo expressar tanto a generalização como a associação. A relação de herança mais uma relação que possa expressar agregação são suficientes para poder-mos construir um qualquer sistema de classes inter-relacionadas.

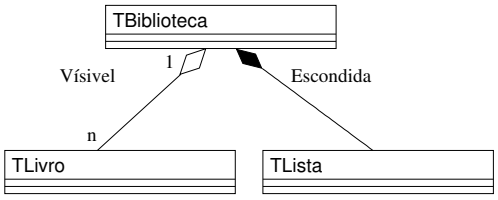


Figura 2.8: Agregação

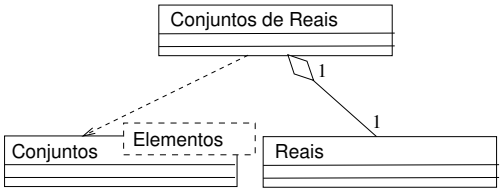


Figura 2.9: Instanciação

A relação de meta-classe é um mecanismo que algumas linguagens de programação orientada para os objectos possuem e que permite considerar as classes como objectos de uma meta-classe, construindo deste modo um novo nível de abstracção (ver Fig. 2.10).

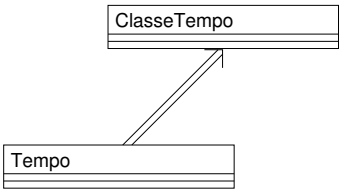


Figura 2.10: Meta-classe

2.3.1 Relação de Herança

Vejamos um exemplo. Quando uma sonda é largada (para o espaço, no fundo do oceano, etc.) espera-se que ela comece a transmitir informação para o posto de controlo, informações sobre o estado dos seus diferentes sub-sistemas.

Podemos então considerar que como dados de de telemetria a sonda enviaria: o estado das baterias; o estado dos motores; o estado dos diferentes sensores, por exemplo colisão, radiação luminosidade, temperatura, pressão, salinidade, entre outros possíveis.

Ao pensar-se na construção de um programa de monitorização das sondas poder-se-ia optar por construir uma estrutura de dados do tipo registo (`struct`) capaz de guardar toda a informação:

Por exemplo, para a componente referente às baterias:

```
struct Tempo {
  int tempoDecorrido;
  int segundo;
}

struct DadosElectricos {
  Tempo tempo;
```

```
  int id;
  float voltagemBat1, voltagemBat2;
  float amperagemBat1,amperagemBat2;
  float corrente;
}
```

As limitações de uma tal aproximação são já conhecidas:

Não encapsulamento da informação. Modificações na estrutura de base vão ter consequências em todo o programa, fazendo com que seja muito difícil, ou mesmo quase impossível, alterar a estrutura sem ter que refazer todo o programa.

Impossibilidade de estabelecer uma hierarquia de sensores. As variações entre sensores complicam a criação de uma estrutura simples, mas capaz de cobrir as diferentes situações. Por exemplo, será que a definição tal como está não levanta já problemas? Como é que podemos lidar com um sensor que necessite de mais do que duas baterias?

Uma aproximação à solução. Uma aproximação (melhor) ao problema é dado pela transformação da estrutura de dados de um registo para uma classe:

```
class DadosElectricos {
public:
  DadosElectricos();
  DadosElectricos(const DadosElectricos &);
  ~DadosElectricos();
  void enviaEstadoBaterias();
  void enviaTempoCorrente() const;
private:
  struct Tempo {
    int tempoDecorrido;
    int segundo;
  }
  Tempo tempo;
  int id;
  float voltagemBat1, voltagemBat2;
  float amperagemBat1,amperagemBat2;
  float corrente;
}
```

Esta solução já permite resolver o problema do encapsulamento da informação, ao utilizar-se um objecto concreto, uma instância da classe, não se teria acesso à sua secção privada, como tal o programa seria independentes desta e qualquer alteração à mesma não o afectaria.

No entanto ainda ficamos com o problema da redundância, muitos tipos de sensores implica muitas classes, similares mas no entanto diferentes.

A relação de herança. A solução passa então por construir uma hierarquia de classes na qual classes mais especializadas herdam a estrutura e o comportamento de classes mais genéricas.

Por exemplo:

```
class DadosTelemetria {
public:
  DadosTelemetria();
```

```
  DadosTelemetria(const DadosTelemetria &);
  ~DadosTelemetria();
  virtual void enviaDados();
  Tempo tempoCorrente() const;
protected:
  int id;
private:
  struct Tempo {
    int tempoDecorrido;
    int segundo;
  }
  Tempo tempo;
}
```

A classe `DadosTelemetria` seria a classe de topo, como se pode ver além do identificador da sonda, `id` e do `tempo` esta classe não especifica mais nada, isso será deixado ao cargo das classes mais especializadas.

```
class DadosElectricos : public DadosTelemetria{
public:
  DadosElecticos(float v1, float v2, float a1, float a2);
  DadosElectricos(const DadosElectricos &);
  ~DadosElectricos();
  virtual void enviaDados();
  float corrente() const;
protected:
  float voltBat1, voltBat2, ampBat1, ampBat2;
};
```

Temos então que a classe `DadosElectricos` deriva da classe `DadosTelemetria`. É também usual dizer que implementa, ou que é um sub-tipo (ver Fig. 2.11).

Adicionaram-se quatro novos elementos à estrutura de dados, uma nova função membro. Além disso o comportamento da função membro `enviaDados` foi redefinido, isto é a função membro terá um comportamento diferente consoante se esteja a falar de objectos, instâncias da classe `DadosElectricos` ou de objectos do tipo `DadosTelemetria`.

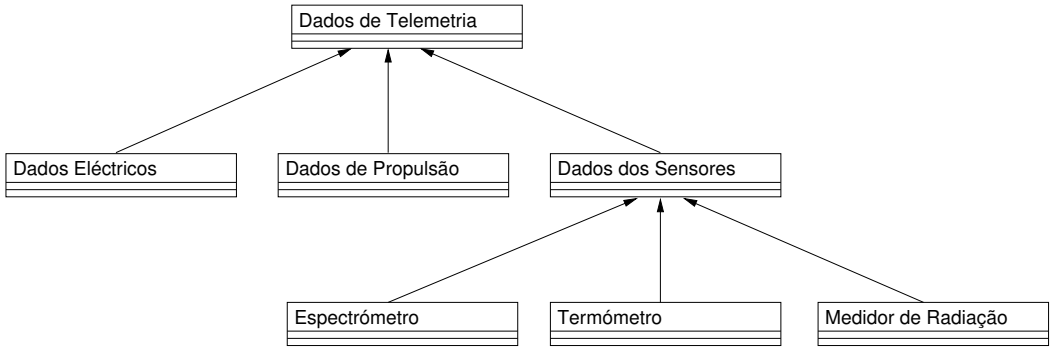
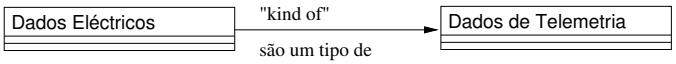


Figura 2.11: Hierarquia Dados de Telemetria

O significado da relação de herança simples.

- a classe *Dados de Telemetria* é uma super-classe (classe base);
- a classe *Dados Eléctricos* é uma sub-classe;

- a classe *Dados de Telemetria* é uma generalização das classes *Dados Eléctricos*, *Dados de Propulsão* e *Dados dos Sensores*;
- a classe *Dados de Propulsão* é uma especialização da classe *Dados de Telemetria*.



As sub-classe (tipicamente) recebem a informação da super-classe aumentando ou redefinindo de algum modo a estrutura e/ou o comportamento da classe mais genérica.

	Especificação do acesso na classe de base		
Acesso à informação da classe base	**Public**	**Protected**	**Private**
Classe de base	$\surd$	$\surd$	$\surd$
Classes derivadas	$\surd$	$\surd$	$\times$
Outras classes	$\surd$	$\times$	$\times$

Tabela 2.1: Especificadores de Acesso e Herança

Pode-se dar o caso de uma dada classe não ter nenhuma instância, isto é, numa dada hierarquia de classes pode existir uma classe da qual não é criado nenhum objecto (instância da classe), esse tipo de classe é dita *abstracta* e o objectivo do programador na sua definição prender-se-á com a construção da hierarquia das classes na qual essa classe tem o seu papel bem definido, por exemplo é o topo da hierarquia permitindo deste modo simplificar a escrita das sub-classes.

Dado um estrutura de classes a classe mais geral é usualmente designada por *classe base*, em *C++* existe uma classe mais geral do que todas as outras cujo propósito é somente o de permitir o fecho da hierarquia global das classes em *C++*. este classe é anónima, não podendo como tal ser invocada.

A relação de herança significa que uma sub-classe herda a estrutura (atributos) da classe base. Em *C++* temos:

- é possível estender a estrutura de uma classe;
- não é possível redefinir elementos da estrutura;
- não é possível reduzir a estrutura, eliminando de alguma forma elementos.

A sub-classe também herda o comportamento da classe base. Em *C++* tem-se:

- as funções membros que são declarada como `virtual` podem ser redefinidas na sub-classe;
- as outras funções membros não podem ser redefinidas;
- podem-se acrescentar funções membro.

Como excepções a estas regras temos que:

- os construtores e os destrutores não são herdados;

- o operador de atribuição não é herdado;
- as funções e as classe «amigas» (*friend*, ver página 38) não são herdadas.

A redefinição das funções membro nas diferentes sub-classes pode ser entendido através do conceito de polimorfismo,

Definição 11 (Polimorfismo) *A palavra polimorfismo vem do grego e significa que pode tomar várias formas. Em programação tem o significado de um dado símbolo (em geral um operador/função) que pode assumir diferentes significados.*

Todas as linguagens de programação usam um dado tipo de polimorfismo, por exemplo os símbolos dos operadores aritméticos, `x+y`, significa a adição de dois inteiros, mas também significa a adição de dois reais.

O conceito de polimorfismo foi primeiro tratado por Christopher Strachey que falava de (Booch, 1991):

- O polimorfismo "ad hoc" (igualmente *sobrecarga* ou em inglês "overloading"), no qual o mesmo símbolo pode designar várias operações;
- O polimorfismo paramétrico, no qual um dado símbolo (identificador) pode designar objectos de diferentes tipos relacionados por um (super)tipo comum.

Sem polimorfismo seria necessário escrever longos registo (eventualmente) com uma parte variável ("variant").

Em *C++* têm-se:

- função `virtual` ("late binding") - polimorfismo;
- função sem `virtual` ("early binding") - monomorfismo.

Sub-classes e sub-tipos Pode-se ver as relações de herança como uma forma de reaproveitamento do código, ou então como uma declaração de que os objectos da sub-classe obedecem à semântica da classe de base de tal forma que uma sub-classe pode ser vista como uma especialização da classe de base, como um sub-tipo.

Em *C++* se se declara que uma classe de base é `public` isso significa que a sub-classe é vista como um sub-tipo. As classes partilham a mesma estrutura e as mesmas operações:

```
class DadosElectricos : public DadosTelemetria {...};
```

A declaração da herança como `private` ou `protected` significa que as classes partilham a estrutura e as operações, mas significa também que a sub-classe não é um sub-tipo dado que a estrutura e o comportamento da classe base passa a ser privado (ou protegido) da sub-classe.

```
class DadosElectricosPrv : private DadosTelemetria {...};
```

Herança Múltipla

Nem sempre a relação de herança simples é a melhor forma de descrever as relações entre classes. Situações como a que são representadas na figure 2.7, situações em que uma dada classe está relacionada com mais do que uma classe, não são convenientemente tratadas pela relação de herança simples.

Por exemplo, na figura 2.12 a classe `Couves` tem uma relação de herança simples com a classe `Frutas e Vegetais`. Já a classe `Leite` tem uma relação de herança múltipla com as classes `HidrocarbonetosAltos`, `ProteinasAltas` e `Productos Lácteos`.

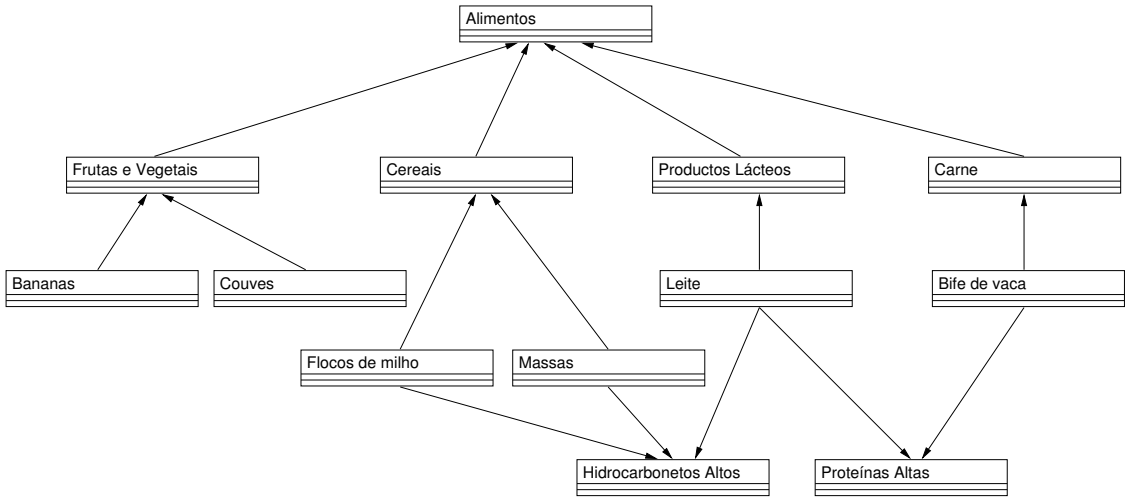


Figura 2.12: Hierarquia Alimentos

Embora seja possível re-escrever a hierarquia de classes de forma a evitar as relações de heranças múltiplas, a possibilidade de as podermos usar para estabelecer uma dada hierarquia de classes permite a especificação de hierarquias de classes mais próximas das situações que se pretende modelar, isto é, temos um maior poder expressivo ao nosso dispor aquando da construção da hierarquia de classes.

Em *C++* é possível especificar heranças múltiplas.

A concepção de uma hierarquia que envolva herança múltipla é delicado. É necessário saber como lidar com dois tipos de problemas que ocorrem nestas situações:

Colisão de Nomes: casos em que, pelo duas das super-classes, usam o mesmo nome para um dos seus elementos;

Repetição de Heranças: casos em que uma dada classe é super-classe de outra de mais do que uma forma.

Vejamos, num caso concreto, como o primeiro destes problemas pode ocorrer e a forma como o resolver. No exemplo acima (ver Figura 2.12) as classes `HidrocarbonetosAltos` e `ProteinasAltas` podem ambas possuir uma variável `percentagemElemento`. Nesse caso a classe `Leite` que herda das duas super-classes referidas vai então receber dois elementos, que devem ser distintos, mas que possuem o mesmo nome.

Diferentes linguagens, têm diferentes formas de resolver este problema, no caso do *C++* a destrinça é feita através do explicitar da classe de origem, ou seja, na sub-classe os dois identificadores têm de ser prefixados com o nome da classe de origem.

O segundo tipo de problemas ocorre em situações tais como a que é representada na figura 2.13.

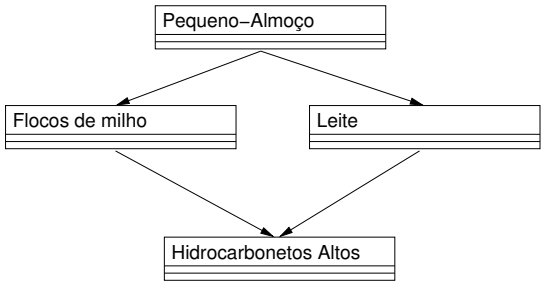


Figura 2.13: Hierarquia Pequeno-Almoço

Novamente diferentes linguagens lidam com este problema de diferentes formas, no caso que nos interessa, o *C++*, temos duas formas de resolver esta situação de herança repetida.

- identificar completamente cada um dos identificadores de forma não ambígua, isto é possível através do prefixar do nome das diferentes super-classes. Ou seja explicita-se a cópia específica que se pretende usar.

- pode-se identificar todas as referências repetidas como sendo referentes à mesma classe. Em *C++* isto é possível desde que se identifique a super-classe como sendo *virtual*.

2.3.2 Relação de Agregação

A relação de herança nem sempre é a mais apropriada para descrever o relacionamento entre classes. Por exemplo a relação entre um livro e uma biblioteca não terá na relação de herança a forma mais apropriada de a descrever. Um livro não herda de uma biblioteca, um livro é parte de uma biblioteca.

Na programação orientada para os objectos existem duas variantes para a relação entre classes "parte de". Pode-se dar o caso de, no interface de uma dada classe, se usar uma outra classe, ou pode acontecer que na implementação de uma classe se use uma outra classe.

No primeiro caso a classe que se esta a usar tem de ser visível para qualquer cliente. Por exemplo todo o código que faça a devolução de um livro tem de ser visível, tanto para uma dada classe `Biblioteca`, isto dado esta implementar um método `entregaLivro`, assim como para a uma classe `Livro`, isto dado que esta declara objectos dessa classe.

No segundo caso a classe usada está embebida, escondida, na secção privada (ou na secção protegida) da classe que a está a usar.

A relação "parte de" coloca também a questão da cardinalidade da relação. Por exemplo pode-se afirmar que:

- 1 Biblioteca contém n Livros, cardinalidade 1:n;

- 1 Carro contém 1 Motor, cardinalidade 1:1;

- m Bibliotecas contém n Livros. cardinalidade m:n

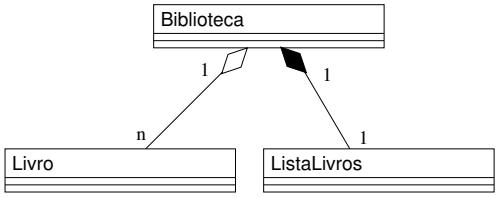


Figura 2.14: Relação "Parte de"

As questões do encapsulamento da informação colidem em alguns aspectos com a relação "parte de". Por exemplo suponhamos que pretendemos criar uma classe `ListaLivrosOrdenada`, a qual necessita de ter acesso à representação interna de `Livro` para poder implementar uma ordenação eficiente dos livros.

Em *C++* é possível relaxar as regras do encapsulamento da informação através da utilização de classes «amigas» (*friends*).

As classe amigas são necessárias sempre que se pretenda declarar métodos que envolvam dois ou mais objectos de diferentes classes e em cuja implementação seja necessário aceder a uma componente privada da outra classe. Note-se que não estamos perante uma relação de herança, caso em que poderíamos fazer uso da secção protegida como forma de sonegar a informação a todas as classes, excepto às classes herdeiras.

Como na vida, os amigos devem ser escolhidos sabiamente.

2.3.3 Relação de Instanciação

Pretende-se, por exemplo, implementar o TAD Conjuntos, mas de forma a que se possa lidar com conjuntos de elementos genéricos, isto é conjuntos de elementos sem especificar o tipo concreto dos elementos. A instanciação do tipo de elementos seria feita somente aquando da criação de uma objecto concreto da classe, criando-se então o, por exemplo, conjuntos dos Reais.

Uma das formas de construir classes genéricas é através dos tipos parametrizados, um mecanismo que permite que se defina uma classe como um modelo (*template*/escantilhão) para um conjunto de classes. No momento da criação dos objectos a classe modelo é então instanciada com um valor concreto para o seu parâmetro.

O *C++* permite implementar classes genéricas através do mecanismo designado por contentores (ou *templates*/escantilhões).

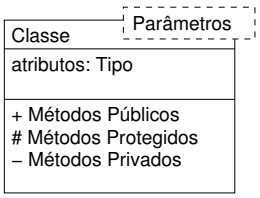


Figura 2.15: Classe Escantilhão

Ter-se ia então que a implementação da classe escantilhão seria

```
template <class Elem> class Conjunto {...
```

sendo que no interior da classe conjunto tem-se acesso ao tipo `Elem`.

Aquando da criação dos objectos ter-se-á de fornecer um valor concreto para o parâmetro. Por exemplo:

```
Conjunto<int> conjZ;
Conjunto<float> conjR;
```

Em termos dos diagramas UML este tipo de relação entre classes é representado da seguinte forma 2.16:

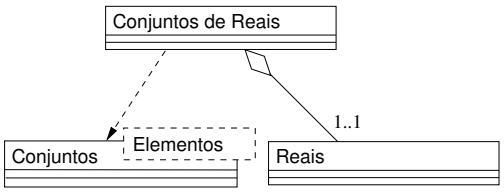


Figura 2.16: Relação de Instanciação com Classe Escantilhão

Em que a classe `Reais` tem um relação de agregação com a classe `Conjunto de Reais`, de um para um, significando que a primeira está a ser usada na segunda. Por outro lado a classe paramétrica *Conjuntos<Elementos>* tem uma relação de instânciação (seta com linha tracejada) com `Conjunto de Reais` querendo significar que a primeira é instânciada na segunda através da utilização da classe `Reais`.

2.3.4 Relação Meta-Classe

O que que acontece se se pretender tratar uma classe como um objecto que pode ser manipulado. Para que isso possa acontecer temos de saber qual é a classe dessa classe, a resposta é imediata, não pode ser uma classe tem de estar a um nível superior, em programação orientada para os objectos é usual designar este tipo de entidade por "meta-classe", isto é uma "classe" cujas instâncias são classes.

Figura 2.17: Relação de Instanciação

A linguagem *C++* não suporta explicitamente o mecanismo de meta-classes, no entanto providência alguns mecanismos deste tipo, especificamente podemos declarar um método, ou um objecto, como sendo estático significando, deste modo, que ele pode ser partilhado por todas as instâncias da classe.

2.4 Exemplos

Exemplo: Sistema Automóvel As relações de agregação entre classes implementam associações do tipo "parte de", isto é, um dado objecto irá ser parte de um todo, ou dito de outra forma irá existir uma classe que vai agregar em si várias classes componentes.

O exemplo da figura 2.18 pretende representar uma situação da especificação de um automóvel, sendo que esta relação entre classes não se enquadra directamente nas situações de herança descritas anteriormente. Os travões não herdam do automóvel (nem vice-versa), o

que podemos dizer é que um automóvel vai conter o sistema de travões, ou doutra forma os travões são parte do automóvel.

Em *C++* esta relação entre classes é-nos dada pela simples utilização de uma classe dentro da outra, ou na sua especificação, ou na sua implementação.

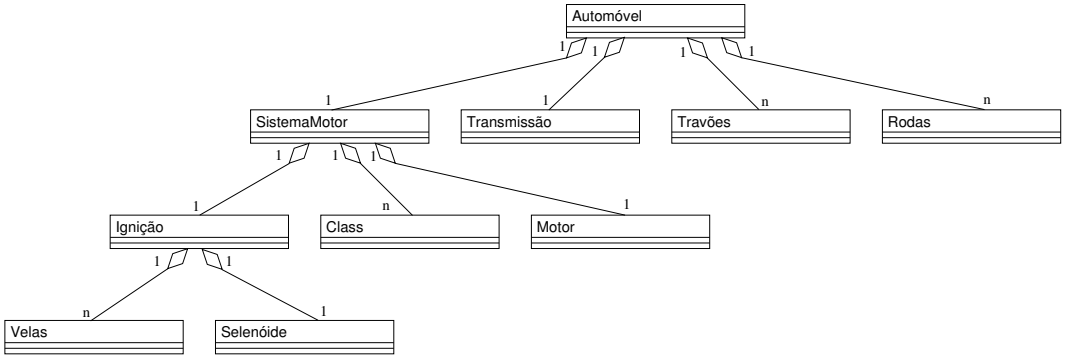


Figura 2.18: Hierarquia Automóvel (relações "parte de")

Exemplo: Sistema de Barragens As hierarquias de classes não estão restritas a hierarquias de relações de herança ou de relações de agregação. Para um dado problema a sua modelização num sistema de classes pode tomar a forma de uma hierarquias com diferentes tipos de relações entre classes e mesmo com algumas classes desconexas, cuja única ligação é dada pelo programa principal (a função `main`).

O exemplo da construção de um sistema de barragens (ver exercício E.12.2) é um desses casos. Pretende-se construir um programa capaz de simular um sistema de múltiplos reservatórios de água ligados entre si de forma arbitrária (em série, em paralelo, ou ambos).

Vejamos então como o modelar este problema (ver figura 2.20).

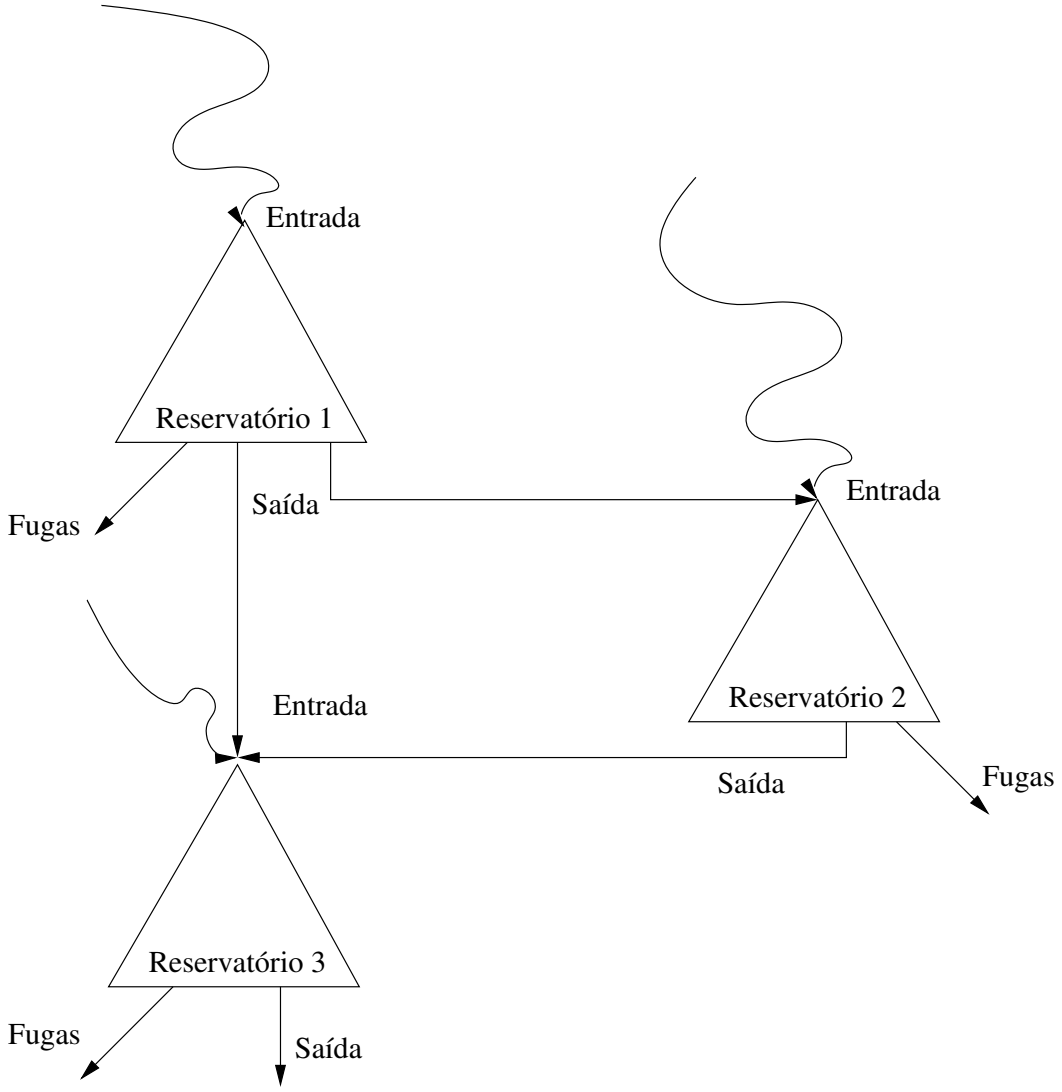


Figura 2.19: Sistema com três reservatórios

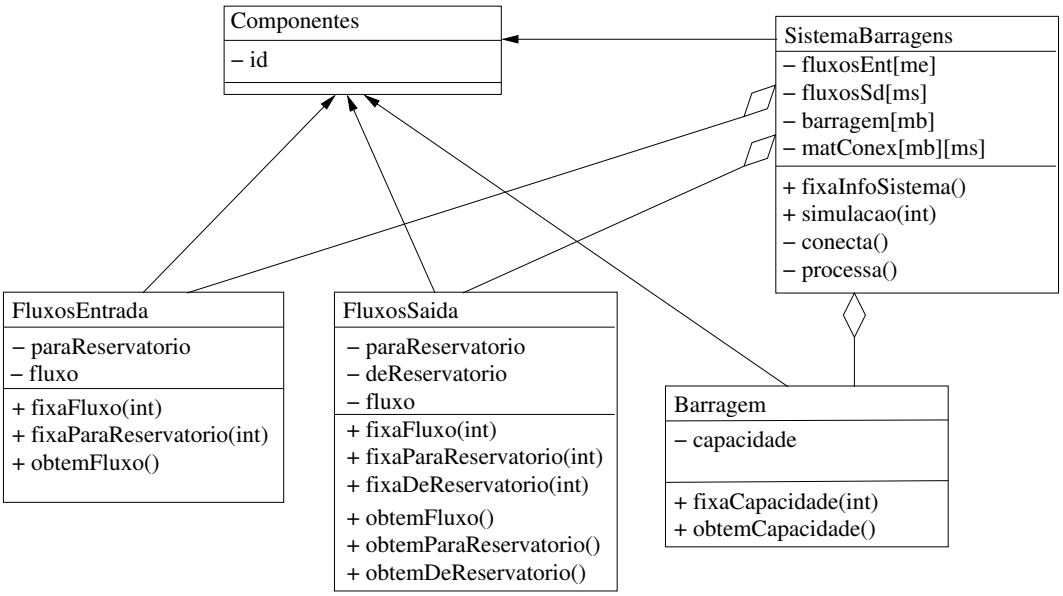


Figura 2.20: Sistema de Barragens

Parte II

A Linguagem C++

Capítulo 3

Documentação

Este capítulo é sobre a importância da documentação do ponto de vista do programador, só de forma breve é que se fala sob a perspectiva do utilizador, e mesmo aí é tendo na ideia uma utilizador/programador. A escrita de manuais para utilizadores não será aqui abordada.

O acto de construir um programa é, em geral, não isolado, nem no tempo, nem na pessoa do programador. Isto é, um programa, mesmo que feito por uma só pessoa, ou por uma só equipa, não é, regra geral, feito num só momento ininterrupto. Por outro a construção de um programa é em geral um trabalho de equipa, sendo que a mesma pode sofrer alterações na sua composição durante a construção do programa.

Por este tipo de razões a documentação, interna, isto é como comentários no próprio código, ou externa, na forma de um relatório externo ao código do programa, é de uma importância extrema.

Neste capítulo dão-se, de forma breve, algumas indicações a seguir na construção de um programa, nomeadamente na documentação que o deve acompanhar.

Como última secção falar-se-á, também de forma breve, numa aproximação, digamos assim, radical ao problema. O programa não é o importante, a documentação é que é o importante. Esta aproximação, conhecida por *programação literária* advoga a mudança de hábitos do programador, este não deve começar por escrever código e mais tarde, eventualmente, escrever a documentação, mas pelo contrário, deve começar pela documentação e só depois é que deve passar para o código.

3.1 Documentação Interna

Através da utilização dos *comentários* a inserir nos ficheiros contendo os programas. A descrição, sucinta, das estruturas de dados (atributos) principais, a descrição em termos de pré e pós condições dos diferentes métodos de cada uma das classes.

Sucinta, mas presente.

No *C/C++* podemos ter ≪**linhas de comentários**≫, tudo o que esteja à direita de um par de barras oblíquas para a direita (≪`//`≫) e até ao fim dessa linha e ≪**blocos de comentários**≫ tudo (várias linhas) o que estiver entre o par de símbolos ≪`/*`≫ e o par ≪`*/`≫.

3.2 Documentação Externa

A descrição do problema, o manual do utilizador, a documentação para o programador: estrutura das classes (UML); descrição geral da estrutura(s) de dados utilizada(s); descrição geral do(s) algoritmo(s) construídos; eventuais detalhes sobre as classes, atributos, métodos, mais significativos.

3.2.1 Documentação Externa em LaTeX

Para incluir código num texto LaTeX existem vários módulos. O módulo *listings* é bastante completo e fácil de usar.

```
usepackage{listingsutf8}

% definição da linguagem de programação
lstset{language=C++,
  extendedchars=true,
  inputencoding=latin1,
  morekeywords={cin,cout,endl,NULL,string},
  basicstyle=\footnotesize}


% para lidar com o UTF8
lstset{literate=
  {á}{{\'a}}1 {é}{{\'e}}1 {í}{{\'i}}1 {ó}{{\'o}}1 {ú}{{\'u}}1
  {Á}{{\'A}}1 {É}{{\'E}}1 {Í}{{\'I}}1 {Ó}{{\'O}}1 {Ú}{{\'U}}1
  {à}{{\`a}}1 {è}{{\`e}}1 {ì}{{\`i}}1 {ò}{{\`o}}1 {ù}{{\`u}}1
  {À}{{\`A}}1 {È}{{\'E}}1 {Ì}{{\`I}}1 {Ò}{{\`O}}1 {Ù}{{\`U}}1
  {ä}{{\"a}}1 {ë}{{\"e}}1 {ï}{{\"i}}1 {ö}{{\"o}}1 {ü}{{\"u}}1
  {Ä}{{\"A}}1 {Ë}{{\"E}}1 {Ï}{{\"I}}1 {Ö}{{\"O}}1 {Ü}{{\"U}}1
  {â}{{\^a}}1 {ê}{{\^e}}1 {î}{{\^i}}1 {ô}{{\^o}}1 {û}{{\^u}}1
  {Â}{{\^A}}1 {Ê}{{\^E}}1 {Î}{{\^I}}1 {Ô}{{\^O}}1 {Û}{{\^U}}1
  {Ã}{{\~A}}1 {ã}{{\~a}}1 {Õ}{{\~O}}1 {õ}{{\~o}}1 {ñ}{{\~n}}1
  {œ}{{\oe}}1 {Œ}{{\OE}}1 {æ}{{\ae}}1 {Æ}{{\AE}}1 {ß}{{\ss}}1
  {ů}{{\H{u}}}1 {Ů}{{\H{U}}}1 {ő}{{\H{o}}}1 {Ő}{{\H{O}}}1
  {ç}{{\c c}}1 {Ç}{{\c C}}1 {«}{{\guillemotleft}}1 {»}{{\guillemotright}}1
  {€}{{\EUR}}1 {£}{{\pounds}}1
}


begin{lstlisting}[frame=single]
struct Data {
  int d,m,a;  // dia, mês, ano
};
end{lstlisting}
```

Para a construção de diagramas UML existem vários programas, por exemplo, `xfig` e `dia`.

3.3 Programação Literária

Sob a designação de *programação literária* define-se uma aproximação à programação em que se pretende que os programas sejam, obras literárias, isto é que possam ser lidas e compreendidas facilmente por todos, em contra-ponto a programas escritos de tal forma que são de difícil leitura (muitas vezes, mesmo pelo próprio programador).

Na programação literária pretende-se que a documentação (legível por humanos) e código fonte (legível pela máquina) estejam num único arquivo (ficheiro) fonte, de modo a manter uma correspondência próxima entre a documentação e o código fonte. A ordem e a estrutura desse arquivo são especificamente projetadas para auxiliar a compreensão humana: código e documentação juntos são organizados em ordem lógica e/ou hierárquica (tipicamente de acordo com um esquema que acomode explicações detalhadas e comentários como necessárias).

Ferramentas externas pegam neste documento e geram a documentação do programa e/ou extraem o o programa propriamente dito para processamento subsequente por compiladores ou interpretadores.

O primeiro ambiente de programação literária publicado foi o sistema *WEB*, introduzido por Donald Knuth em 1981 (Knuth, 1984). Nesse sistema o programa *Weave* produz, a partir do documento original, a documentação a ser posteriormente formatada através do sistema TEX e o programa *Tangle* produz, a partir do documento original, o programa fonte em *Pascal*. Posteriormente o sistema *CWEB* foi introduzido com o intuito de dar suporte a este tipo de programação mas à linguagem de programação *C*.

Capítulo 4

Programação Imperativa

> Programas = Algoritmos + Estruturas de Dados
>
> *Niklaus Wirth*

Do ponto de vista da programação procedimental um computador é uma máquina de estados. De um estado inicial, isto é, os dados de entrada, guardados num conjunto de variáveis que em conjunto definem o estado do programa, pretende-se atingir um dado estado final, definidor do resultado que se pretende obter com o programa.

Um programa é então uma sequência linear de transições de estado. O algoritmo define essa sequência, as estruturas de dados determinam o estado, o qual vai variando temporalmente à medida que se processam as diferentes instruções definidas no algoritmo.

Em *C++* um programa é escrito num, ou mais, ficheiro(s) sendo constituído obrigatoriamente por uma função designada por `main`, a qual determina o início do algoritmo e, eventualmente, por outras funções e/ou classes.

4.1 Estruturas de Dados Básicas em *C++*

As estruturas matemáticas que podemos considerar como básicas na modelização de situações numéricas são:

- $(\mathbb{Z}, +, -, \times, /, 0, 1)$ - corpo dos inteiros;
- $(\mathbb{R}, +, -, \times, /, 0, 1)$ - corpo dos reais; dos Booleanos (lógicos)
- Letras - alfabeto
- Letras* - palavras formadas por sequências de símbolos do alfabeto.

em *C++* temos as seguintes estruturas de dados (os valores concretos são referentes ao compilador *gcc version 4.7.2* e ao sistema operativo *Debian 4.7.2-5* (ver secção D.2):

- `(int,+,-,*,/,%)` - implementação dos Inteiros.
  - representação exacta (conversão entre base 2 e base 10);
  - gama da variação finita:
    - `char`, 8bits/1byte

        - signed: -128 a 127
        - unsigned: 0 a 255
      - `short int`, 16bits/2bytes
        - signed: -32768 a 32767
        - unsigned: 0 a 65535
      - `int`, 32bits/4bytes
        - signed: -2147483648 a 2147483647
        - unsigned: 0 a 4294967295
      - `long int`, 64bits/8bytes
        - signed: -9223372036854775808 a 9223372036854775807
        - unsigned: 0 a 18446744073709551615
      - `long long int`, 64bits/8bytes
        - signed: -9223372036854775808 a 9223372036854775807
        - unsigned: 0 a 18446744073709551615
  - operações com as propriedades usuais;
  - «/» é a divisão inteira, «%» é o resto da divisão inteira;
  - representação usual (na base 10).

- `(float/double,+,-,*,/)` - implementação dos Reais;
  - representação não exacta (conversão entre base 2 e base 10);
  - representação interna na base 2 (em vírgula flutuante);
  - gama de variação finita:
    - `float`, 32bits/4bytes, 1.175494e-38 a 3.402823e+38
    - `double`, 64bits/8bytes, 2.225074e-308 a 1.797693e+308
    - `long double`, 128bits/16bytes, 3.362103e-4932 a 1.189731e+4932
  - operações com as propriedades usuais;
  - representação usual e notação científica (notação inglesa): por exemplo, 3.1415 e 0.31415e1

- `(bool, {&&, ||, !})` - implementação dos Booleanos (*C++*);
  - $\mathscr{V}$ - `true` - qualquer inteiros diferente de zero;
  - $\mathscr{F}$ - `false` - 0.
  - Conectivas lógicas com as propriedades usuais da lógica proposicional.
    - `&&` - conjunção;
    - || - disjunção;
    - `!` - negação.
  - Símbolos relacionais com as propriedades usuais.
    - `==` - igualdades;
    - `!=` - diferença;
    - `<, <=` - menor e menor ou igual;

    - `>, >=` - maior e maior ou igual.
- `(char)` - Alfabeto interno;
    - Aparte o alfabeto *ASCII* não existe uma norma globalmente aceite, este facto tem como consequência que nem sempre é possível assegurar que um carácter acentuado seja representado de forma correcta entre plataformas computacionais diferentes.
    - Como vimos acima o tipo `char` pode ser visto como um sub-tipo dos inteiros, dito de outra forma, internamente o *C++* lida com os caracteres em termos da sua codificação não fazendo distinção entre esses valores e os outros valores do tipo `int`.
- `(string)` - sequências de símbolos do Alfabeto (*C++*).
    - A classe `String` define um conjunto de operações alargado para este tipo, ver-se-á isso mais à frente.
- `<tipo>*` - ponteiros para um dado tipo de dados. Neste caso não temos verdadeiramente um tipo de dados mas sim referências, isto é, o explicitar da ligação entre os identificadores (nomes das variáveis de um dado tipo) e os seus valores (células de memória aonde). Os ponteiros não são mais do que valores naturais (entre 0 e o valor máximo da memória RAM que um dados sistema operativo suporta) que identificam as células de memória aonde os valores das variáveis são guardados (ver Figura 4.1)
    - valores inteiros entre 0 e o valor máximo que o sistema operativo suporta (Linux 64bits, 256GB ($= 256 * 2^{30}$).
    - todas as operações com inteiros. No entanto dado se tratar de ponteiros, isto é referências a células de memória, a sua manipulação directa como valores inteiros deve ser feita com extremo cuidado.

Os tipos `int, float, double, bool` e `char` são tipos atómicos, isto é, os seus elementos não são decomponíveis em partes menores, o tipo `string` é já um tipo composto, os seus elementos são sequências de elementos do tipo `char`, ver-se-á mais à frente como é possível construir outros tipos compostos, sejam em termos de estruturas estáticas (ver secção 4.8), seja em termos da construção de estruturas dinâmicas (ver secção 4.8.2).

O tipo ponteiro é um tipo simples apropriado para criar estruturas compostas, vamos ver já de seguida como manobrar as variáveis deste tipo quando os elementos apontados são de um tipo atómico. A sua utilização para criar estruturas dinâmicas será visto mais à frente (ver secção 4.8.2).

4.2 Declarações de Variáveis

Já tendo visto quais são os tipos básicos disponíveis coloca-se agora a questão de como declarar variáveis desses tipos, isto é, como construir o estado do programa.

Em *C++* a declaração de uma variável pode ser feita em qualquer ponto do programa, a variável irá ser incorporada ao estado do programa a partir desse ponto. Por razões metodológicas é usual agrupar todas as declarações na zona inicial de cada uma das funções constituintes do programa, nomeadamente a função `main`, desse modo é fácil de saber quais são as diferentes componentes definidoras do estado do programa.

Temos então as seguintes variantes:

- Declaração de uma variável:

```
char c;
int i;
```

- Declaração de várias variáveis:

```
float a,aux,delta;
```

- Declaração e inicialização:

```
double x=7.4;
```

- Declaração de uma constante:

```
const double pi=3.14159;
```

De uma forma mais geral tem-se (ver B.2 para uma explicação da meta-sintaxe):

```
DeclaraçãoVariável  ::= [ const ] <tipo> <LstVarInicializações> ;

LstVarInicializações ::= $<$identificador$>$ [  =
$<$valor$>$ ] [ , $<$LstVarInicializações$>$ ]  ;
```

Após as declarações o estado do programa passa a conter os valores das variáveis, os quais estão associados aos nomes das variáveis. No exemplo acima ter-se-ia:

c	???
i	???
a	???
aux	???
delta	???
x	7.4
pi	3.14159

quando uma variável é declarada e não inicializada o seu valor deve ser tido como indeterminado, o assumir de um qualquer valor inicial, definido por omissão, é um erro.

4.2.1 Tipo Ponteiro

Como foi dito no princípio deste capítulo um programa do tipo procedimental é uma máquina de estados. De um estado inicial as diferentes instruções de atribuição vão modificando esse mesmo estado inicial até se obter os resultados desejados.

O estado de um programa é caracterizado por um conjunto de identificadores (nomes de variáveis) e de valores (posições de memória) que lhe estão associados. As variáveis estáticas (sem ponteiros) e as variáveis dinâmicas têm um comportamento diferente na forma como associam estas duas entidades, nomes e posições de memória. No primeiro caso as referências

são geridas automaticamente pelo programa, no segundo essa é já parte das responsabilidades do programador.

Vejamos uma situação em concreto (ver Figura 4.1): a declaração de uma variável estática do tipo inteiro, `x`, e a sua posterior inicialização; e a declaração de uma variável dinâmica, um ponteiro para um inteiro, `z`, a inicialização do espaço e a posterior inicialização do valor apontado.

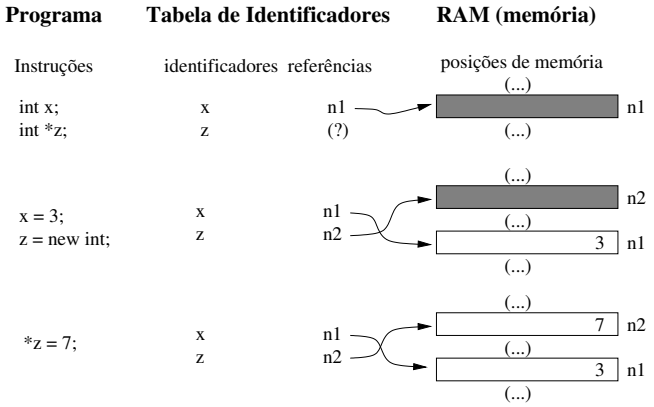


Figura 4.1: Variáveis Estáticas vs Variáveis Dinâmicas

Aquando da declaração de um variável estática temos as seguintes acções:

1. guardar o nome na tabela de identificadores;
2. reservar a memória necessária para guardar um valor do tipo associado ao identificador;
3. inicializar a referência associada (ponteiro, número natural referente a uma posição de memória) com o valor correspondente à posição de memória que foi anteriormente reservada.

o valor associado à variável começa por ser indefinido.

Aquando da declaração de uma variável dinâmica temos a seguinte acção.

1. guardar o nome na tabela de identificadores;

a referência associada a este nome começa por ser indefinida, qualquer tentativa de acesso ao valor (ainda inexistente) associado é um erro de acesso à memória.

Para uma variável dinâmica é necessário fazer uma reserva de memória de forma explícita (através do operador `new`) e só depois é que é possível associar-lhe um valor.

Para uma variável do tipo estático é possível aceder à sua referência (ponteiro) de forma explícita utilizando o operador de "`&`".

declaração	identificador	referência	valor (referenciado)
<tipo> `*x`	`x`	`&x`	`*x`

Para uma variável do tipo dinâmico é possível aceder ao valor que lhe está associado através do operador «`*`»

identificador (do tipo ponteiro)	valor (referenciado)
`x`	`*x`

Retomando o exemplo apresentado na figura 4.1 vejamos um exemplo da declaração, afectação e atribuição de valor a uma variável do tipo ponteiro.

Declaração começa-se por declarar uma variável do tipo ponteiro para um dado tipo específico, por exemplo ponteiro para um inteiro;

```
int *z;
```

neste momento ainda não é possível atribuir um valor (inteiro) a esta variável. Só o espaço para o ponteiro é que foi criado e nem sequer está inicializado. O tentar aceder ao valor de uma variável do tipo ponteiro nesta situação é considerado uma situação de erro (figura 4.1, primeira secção).

Afectação ou seja é necessário criar explicitamente o espaço de memória (para um inteiro) e inicializar o ponteiro para que este aponte para essa posição de memória.

```
z = new(int);
```

a posição de memória em concreto não é inicializada (figura 4.1, secção do meio). Mais à frente (ver secção 4.8.2) falaremos com mais detalhe da funções de gestão de memória (`new` e `delete`).

Atribuição Já tendo o espaço de memória assim como o ponteiro para essa posição, a atribuição e/ou utilização de valores é feita de forma (quase) normal, por exemplo:

```
*x = 7;
```

Como '`x`' é uma variável do tipo ponteiro o acesso ao valor que lhe está associado é feito através do operador `*x` (figura 4.1, última secção).

Como é fácil de perceber a importância das variáveis do tipo ponteiro não está na sua utilização como alternativa às variáveis dos tipos simples. A importância dos ponteiros em *C/C++* prende-se com a necessidade de fazer a passagem, por referência, de valores entre módulos distintos (ver secção 4.6.1), assim como com as estruturas de dados tanto as estáticas como as dinâmicas (ver secção 4.8).

4.3 Entradas/Saídas

Para «dar» valores a um programa e «receber» resultados é necessário considerar as, assim designadas, instruções de entrada e saída.

As instruções de entrada/saída estabelecem uma ligação entre o programa e o sistema operativo e permitem estabelecer canais de comunicação permitindo por essa via transferir valores de e para o programa:

Leitura A leitura é feita associando uma lista de valores a uma lista de variáveis. Estas duas listas devem ser consistentes em termos de número e tipo dos valores e respectivas variáveis. Se forem dados mais valores do que aqueles que são esperados, os valores que estão a mais serão ignorados pela instrução de leitura, se forem dado menos valores do que os requeridos o programa ficará à espera de que os mesmos sejam fornecidos, isto no caso da leitura estar a ser feita através do teclado, ou ocorrerá um erro, para os casos em que a leitura possa estar a ser feita de outros meios, por exemplo ficheiros.

De onde dos canais de entrada, por omissão, a linha de comando e o teclado.

Como sequência de valores separados por um ou mais espaços e de acordo com a representação definida para os tipos das variáveis escolhidas.

Efeito afectação dos valores lidos às variáveis.

Exemplo:

```
int a,b,c;
...
cin >> a >> b >> c;
```

Escrita A escrita efectua-se a partir de uma lista de valores, sejam eles constantes, variáveis, ou mesmo expressões.

Para onde canal de saída, por omissão o ecrã.

Como cálculo dos valores das expressões, conversão dos valores de acordo com a forma normal para os diferentes tipos e sua visualização.

Efeito nenhum (as variáveis não são afectadas).

Exemplos:

```
cout << x;
```

```
cout << "O valor pretendido e" << x+y << endl;
```

Nestes último exemplo temos um valor constante do tipo `string`, uma expressão e a «constante», `endl`, cujo efeito é o de provocar uma mudança de linha no canal de saída.

4.4 Programas em *C++*

Um programa em *C++* é constituído por uma função especial, `main`, ponto de início e fim (em condições normais) do funcionamento do programa e, eventualmente, por outras funções e/ou classes.

Um programa mínimo em *C++* é dado pelo programa vazio:

```
int main() {}
```

sendo que:

`int` é o tipo do resultado (valor de saída) da função.

`main` é o nome da função;

`()` é a lista, neste caso vazia, de argumentos da função;

{ } é o corpo da função, neste caso vazio, que define o funcionamento da função.

A função `main` é um exemplo de uma função em *C++* cuja sintaxe genérica é:

função ::= <tipo> <identificador> (<listaArgumentos>) { <corpoDaFunção> }
listaArgumentos ::= <tipo> <identificador> [, listaArgumentos]
corpoDaFunção ::= <listaInstruções>

sobre a lista de instruções falaremos mais adiante.
Um programa mínimo, mas já com efeitos, `eco.cpp`:

```
#include <iostream>

using namespace std;

/*
 * Programa «eco»
 * -> uma palavra (na linha de comando)
 * <- a mesma palavra ecoada de volta
 */

int main(int argc, char *argv[]) {
  cout << "eco " << argv[1] << endl;
  return(0);
}
```

Para o compilar bastaria (ver Apêndice C):

```
g++ eco.cpp -o eco
```

e para ver o efeito, podemos fazer, por exemplo:

```
./eco olá
```

Neste exemplo temos:

`#include <iostream>` directivas de pré-processamento. Neste caso trata-se da inclusão de uma biblioteca do sistema, a biblioteca `iostream` que trata das entradas e saídas e que é necessária para poder ter acesso aos comandos `cin` e `cout`. Mais pormenores em (ver Apêndice C).

`using namespace std` incorpora o espaço de nomes padrão, `std`, permitindo o acesso aos diferentes operadores sem ser necessário explicitar a classe de origem.

Comentários a documentação interna do programa é feita através da escrita de *comentários*. No *C/C++* podemos ter «**linhas de comentários**», tudo o que esteja à direita de um par de barras oblíquas para a direita («`//`») e até ao fim dessa linha. Podemos ter também «**blocos de comentários**» tudo (várias linhas) o que estiver entre o par de símbolos «`/*`» e o par «`*/`».

`argc, argv` são usualmente designados por «argumentos da linha de comandos». `argc` - «argument count» dá-nos o número de argumentos, `argv` «argument values», é uma lista («array») contendo esses mesmos argumentos. Todos estes valores são o resultado da interacção entre o interpretador de comandos do sistema operativo e o programa. O nome do próprio comando também conta, e também é guardado em `argv`. Temos então que no exemplo acima teria-se-ia que `argc` = 2, `argv[0]` = «./eco», `argv[1]` = «olá».

A «lista» (array) `argv` é um exemplo de estrutura de dados composta que será discutida mais à frente (ver secção 4.8).

`return` define o valor de saída que é «passado» ao interpretador de comandos do sistema operativo. Por norma comummente aceite o valor de saída zero indica uma terminação sem erros, um valor diferente de zero, dá-nos um código de erro. Os códigos de erro são próprios de cada programa. Este valor é devolvido ao interpretador de comandos do sistema sendo que, se nada for feito, o código é simplesmente ignorado.

4.5 Instruções Simples

Como já foi dito acima um programa em *C++* é constituído por uma função especial, `main` e, eventualmente, por outras funções e/ou classes. Um programa é uma sequência linear de transições de estado. As variáveis definem o estado do programa.

Já vimos como definir o estado de um programa em termos de estruturas de dados simples (ver-se-á mais à frente as estruturas compostas (ver secção 4.8) e dinâmicas (ver secção 4.8.2). Falta-nos ver: como alterar o estado, como definir a sequência de linear de transições de estado e quais são as instruções básicas que podem ser incluídas na sequência linear de transições de estado.

4.5.1 Sequência Linear de Instruções

O corpo de uma função é então definido por uma lista de instruções separadas por ponto-e-vírgula «;». Algo como $\texttt{I}_1$; ou $\texttt{I}_1; \texttt{I}_2; \ldots; \texttt{I}_n;$.

É também possível agrupar um conjunto de instruções num só "bloco de instruções", o qual passará a contar com uma única instrução (composta). A forma de o fazer é através da utilização dos símbolos de agrupamento, por exemplo $\{\texttt{I}_1; \texttt{I}_2; \ldots; \texttt{I}_n; \}$.

4.5.2 Instrução de Atribuição

A instrução de atribuição é uma instrução fundamental numa linguagem como o *C++* dado que é através dela que se acede ao estado do sistema e/ou se modifica o mesmo.

A sintaxe é a seguinte:

instrução de atribuição ::= <identificador_variável> `=` <expressão> `;`

Por exemplo:

```
x = x * y / 3;
```

A sua leitura tem de ser feita no tempo tendo em conta o estado do programa antes da sua execução e o estado do sistema após a sua execução.

Considere-se que o seguinte excerto de um programa em *C++*.

```
int x,y=3;
x = 7;
x = x * y / 3;
```

A leitura no tempo destas instruções é a seguinte:

Pré	Instrução	Pós
$\emptyset$	`int x,y=3;`	`x` ??? `y` 3
`x` ??? `y` 6	`x = 7;`	`x` 7 `y` 6
`x` 7 `y` 6	$\underbrace{\texttt{x}}_{\text{pós, } 14} = \underbrace{\texttt{x * y / 3;}}_{\text{pré, } 7 \times 6/3}$	`x` 14 `y` 6

É de notar que a as inicializações no caso das instruções de declarações de variáveis não são mais do que uma simplificação de duas acções distintas, uma criação de um variável, a definição do seu valor através de uma instrução de atribuição. De igual modo a instrução de leitura também «esconde» um(ou mais) atribuição(ões) dos valores lidos às variáveis que estão a ser usadas.

Por seu lado a escrita de resultados pode ser vista como o lado direito de uma instrução de atribuição, sendo que o valor obtido não vai alterar o estado mas sim ser «escrito» no dispositivo de saída seleccionado.

Nota importante: é importante ter em conta que nas linguagens derivadas do *C*, nomeadamente no *C++*, a instrução de atribuição tem um valor final (a própria instrução) que é igual ao valor calculado no lado direito e atribuído à variável do lado esquerdo. Este efeito colateral da instrução de atribuição pode ser usado para, por exemplo, fazer uma atribuição em cadeia, `x = y = 23;`, tem como efeito que y toma o valor de 23, como por efeito colateral a instrução `y=23` tem 23 como valor final, então x toma, também o valor de 23;

Em *C++* tem-se acesso a um conjunto vasto de abreviações para instruções de atribuição. A sintaxe é a seguinte:

abreviação de operação/atribuição ::= <identificador_variável> <operador>= <expressão> ;
abreviação de pós-incremento/decremento ::= <identificador_variável><operador><operador>;
abreviação de pré-incremento/decremento ::= <operador><operador><identificador_variável>;

Por exemplo

Abreviação	Instrução
`x += n`	`x = x + n`
`x++`	`x = x + 1`
`++x`	`x = x + 1`

Nos dois últimos casos a ordem pela qual aparecem os dois operadores, à direita ou à esquerda do operando, pode ser significativa. No caso em que estas abreviações são usadas no contexto de uma expressão, então no primeiro caso, `x++`, o valor de `x` no estado actual do sistema é usado para o cálculo é somente após a conclusão do cálculo é que o valor de `x` é

incrementado, no outro caso, `++x`, o valor de `x` é incrementado e é esse o valor que é usado para o cálculo da expressão.

Um exemplo muito simples em que essa distinção altera o resultado final.

```
y = 4;
x = y++;
```

e

```
y = 4;
x = ++y;
```

No primeiro caso os valores finais de `x` e `y` são respectivamente 4 e 5. No segundo caso os valores finais são ambos iguais a 5.

4.5.3 Condicionais

As instruções condicionais dão no a possibilidade de ramificar (dividir) a sequência de instruções em diferentes ramos consoante o valor lógico de uma da proposição.

De manhã podemos dizer, "se as provisões meteorológicas apontarem para chuva levo o guarda-chuva, senão deixo-o em casa", temos que o vou tomar uma de duas (mas não ambas) acções consoante o valor lógico da proposição "as provisões meteorológicas apontarem para chuva", se for verdadeira faço uma acção, se for falsa faço outra.

Nas linguagens de programação as instruções condicionais são, em geral, de dois tipos: a separação binária, o decidir entre uma de duas possibilidades consoante o valor lógico de uma dada proposição; a separação por casos, em que dado o valor de uma expressão numérica, em geral de um tipo enumerável, se vai fazer um de entre múltiplos casos possíveis.

Vejamos como é que a linguagem *C++* implementa este tipo de instruções.

Separação binária

A separação binária em *C++* tem duas variantes "**se** Proposição **então** Instrução" e "**se** Proposição **então** Instrução1 **senão** Instrução2".

instrução condicional ::= `if (` <proposição> `)` <instrução> `;`
instrução condicional ::= `if (` <proposição> `)` <instrução> `else` <instrução> `;`

Que correspondem aos seguintes dois diagramas de fluxo[1] da figura 4.2.

Por exemplo:

```
if (x<0) {
  absX = -x; // x < 0
else
  absX = x;  // ~(x<0) <=> x >= 0
```

Não há restrições ao tipo de instrução que se pode usar na instrução contida nos ramos da instrução condicional.

[1] Os diagramas de fluxo permitem estabelecer o funcionamento, fluxo de instruções, de um programa através da indicação das diferentes ramificações que a sequência de instruções pode tomar.

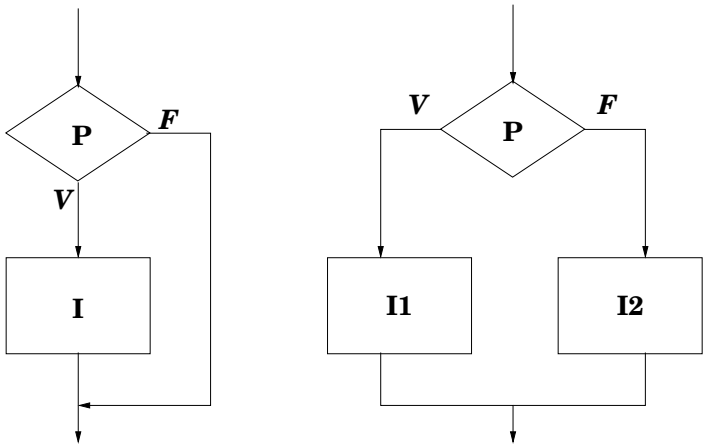


Figura 4.2: Separação binária

```
if (P1) {
  I1; // P1 = V
}
else {
  if (P2) {
    I2; // P1 = F e P2 = V
  }
  else {
    I3; // P1 = F e P2 = F
  }
  I4; // P1 = F
}
```

Na construção de uma instrução condicional deve-se ter o cuidado de não deixar nenhum caso sem tratamento. Por exemplo (exercício E.2), para o cálculo dos valores de uma função definida por ramos:

$$f(x) = \begin{cases} e - e^{\cos x} & \text{se } x \in [0, 2\pi[\\ \log \cos x & \text{se } x \in [-2\pi, -\frac{3}{2}\pi[\bigcup] - \frac{\pi}{2}, 0[\end{cases}$$

ter-se-á a tentação de usar uma instrução condicional somente com dois ramos, correspondentes aos dois ramos da definição da função.

```
if ((x>=0) && (x<(2*M_PI)))
  fx = exp(x)-exp(cos(x));
else if (((-2*M_PI<=x)&&(x<-3/2*M_PI))||((-M_PI/2<x)&&(x<0)))
       fx = log(cos(x));
```

Se do ponto de vista matemático esta definição faz sentido (a função cos é uma função periódica), do ponto de vista da implementação em *C++* ela deixa de fora os valores de x que não pertençam ao intervalos definidos. Devemos acrescentar um terceiro caso que complete a gama de valores possíveis para x.

```
if ((x>=0) && (x<(2*M_PI)))
  fx = exp(x)-exp(cos(x));
  else if (((-2*M_PI<=x)&&(x<-3/2*M_PI))||((-M_PI/2<x)&&(x<0)))
         fx = log(cos(x));
       else
```

```
        cout << "Erro: valor de x incorrecto" << endl;
```

Como se vê pelos exemplos anteriores a proposição P não é necessariamente uma proposição atómica, muito pelo contrário, pode ser constituída por um número (teoricamente) não limitado de proposições combinadas pela conectivas lógicas da negação, $!P$, da conjunção, $P_1 \&\& P_2$, e da disjunção, $P_1 || P_2$. A avaliação do valor lógico final é feita de acordo com as tabelas de verdade da lógica proposicional (Mendelson, 1968).

É de notar que o *C++* adopta um avaliação inteligente, *lazy evaluation*, das expressões lógicas de forma a evitar efectuar cálculos desnecessário, ou seja: a avaliação de uma conjunção pára assim que uma das proposições tenha o valor de verdade «Falso». De forma correspondente a avaliação de uma disjunção pára assim que o valor de uma das proposições tenha o valor de verdade «Verdade». Isto é a avaliação de uma proposição pára assim que o seu valor final é conhecido por se ter obtido o valor absorvente da conectiva lógica que se está a considerar. Deve-se ter em conta este processo de cálculo do *C++*, o qual pode ser útil em algumas situações, como se verá mais à frente (ver secção 4.8).

Além do condicional existe em *C++* uma instrução que nos dá a separação por casos. Esta instrução não é estritamente necessária, no entanto em situações em que há muitos casos para tratar, a sua utilização pode simplificar a escrita de um programa.

Separação por Casos

Para as situações em que se tem uma expressão com valores do tipo inteiro, a qual pode assumir um conjunto alargado de casos, o *C++* possuí uma instrução que permite fazer a separação por casos de uma forma simples.

```
switch (<expressão_inteira>) {
 case <constante1>:
    I1.1; I1.2; ...; I1.n;
    break;
 case <constante2>:
    ...
 default:
    ...
}
```

temos então:

```
switch (<expressão_inteira>) { ... }
```

É a instrução que faz a separação por casos, sendo que os casos são separados consoante o valor, que tem de ser do tipo inteiro, da expressão entre parêntesis.

```
case n: In.1; In.2; ...
```

Define quais as instruções a executar para o caso em que a expressão inteira toma o valor n. No caso do *C++*, a exemplo do que já acontecia na linguagem *C*, após a execução de todas as instruções referentes a este caso a execução do programa continua na instrução imediatamente a seguir ao caso em que se está. Este efeito de continuar no próximo caso («fall through») tem o efeito de se poder especificar o mesmo conjunto de instruções para diferentes casos. Por exemplo:

```
case n1:
case n2:
case n3: In3.1; In3.2; ...
```

para todos os valores da expressão iguais a `n1, n2` e `n3` vão se executar as instruções `In3.1; In3.2; ...`

Para contrair esse efeito de «continuar para o próximo caso», é necessário incluir a instrução `break` no fim da sequência de instruções referentes ao caso em questão. O efeito da instrução `break` é o de «quebrar» a continuidade da instrução `switch` e saltar de imediato para o fim da mesma.

O caso `default` (por omissão) é opcional, o seu efeito, a exemplo do que já foi dito acima para a instrução `if`, é o de fechar a instrução de selecção de casos com um caso que se verifica sempre que nenhuma dos outros casos se verificou. O caso `default` deve ser, por razões óbvias, o último deve ser sempre considerado.

4.5.4 Ciclos

Os ciclos são, como o próprio nome indica, instruções repetitivas e para os quais é importante identificar uma condição de paragem para não se entrar num ciclo (infinito).

Os ciclos são importante sempre que se identifique uma tarefa resolvível por sucessivas aplicações de um dado conjunto de passos. Por exemplo o cálculo do somatório de uma dada função.

$$\sum_{i=1}^{n} f(x_i) = \underbrace{\underbrace{\underbrace{f(x_1)}_{i=1} + f(x_2)}_{i=2} + f(x_3)}_{i=3} + \cdots + f(x_n)}_{i=n}$$

Para cada ciclo deve-se identificar:

O caso inicial neste caso, um somatório, podemos fazer `somatorio = 0`, dado zero ser o elemento neutro da adição.

O invariante isto é a tarefa repetitiva. No caso presente podemos dizer que a cada passagem do ciclo temos o valor do somatório para $i = k - 1$ ao qual se vai somar a parcela de ordem k. `somatorio = somatorio + f(x_k)`.

O caso de paragem o valor final para o qual a variável (ou expressão) de controlo do ciclo vai convergir. No caso do somatório ter-se-ia `i=n` e `somatorio = somatorio + f(x_n)`.

Em *C++* (assim como em *C*) existem três tipos distintos de instruções de ciclo:

`while (P) I;` Enquanto P faz I, isto é enquanto P é verdadeira executa a instrução I (lado esquerdo da Figura 4.3). Com este tipo de ciclo temos que o invariante é executado zero ou mais vezes.

`do I while (P);` Repete I enquanto P, isto é, repete a instrução I enquanto a condição P for verdadeira (lado direito da Figura 4.3). Com este tipo de ciclo temos que o invariante é executado uma ou mais vezes.

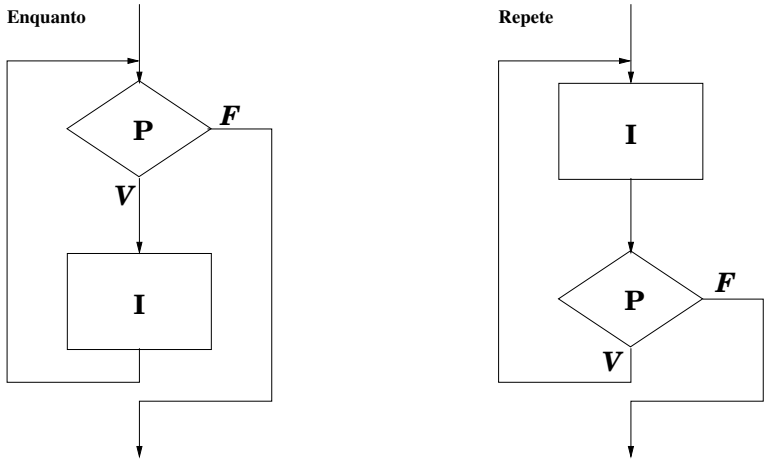


Figura 4.3: Ciclos «`enquanto` P `faz` I» e «`repete` I `até que` P»

`for (Ci;Cf;Inc) I;` Este ciclo é uma versão compacta do ciclo `while`. `Ci` estabelece a condição inicial, `Cf` a condição final; e `Inc` é uma instrução que deve assegurar a convergência desde a condição inicial à condição final. Este ciclo é executado zero ou mais vezes.

É de ter em conta que o ciclo `for` tem em *C++/C* uma semântica bem diferente de muitas outras linguagens, nomeadamente o *Pascal* (Gottfired, 1994; Welsh & Elder, 1979),e o *FORTRAN* (Adams *et al.*, 1997). Nestas linguagens as condições iniciais e finais são avaliadas ao início, dessa avaliação saí o número de iterações a efectuar e o incremento que a variável de controlo (de um tipo enumerável) vai tomar, seguindo-se o executar repetidas vezes da instrução contida no ciclo sem que as condições iniciais e finais e de incremento voltem a ser avaliadas. Como efeito colateral tem-se que a variável que é afectada pelas condições iniciais e finais pode ser usada dentro do ciclo mas a sua modificação não tem nenhum significado no funcionamento do mesmo.

No caso do *C++* não é isso que acontece, a cada iteração a condição de paragem `Pf` é verificada e a instrução de `Inc` executada. Temos então que, ao contrário do que foi dito acima, no caso do *C++* a(s) variável(eis) presentes em `Pf` podem ser usadas e alteradas dentro do ciclo a sua eventual modificação terá uma consequência directa na condição de paragem do ciclo.

Vejamos como calcular o valor da função factorial utilizando as diferentes instruções de ciclo

$$n! = \begin{cases} 1, & n = 0 \\ n * (n-1)!, & n > 0 \end{cases}$$

Este definição por recorrência do factorial não nos serve (ver-se-á mais à frente como tratar estes casos), é necessário poder definir o que se pretende calcular como um processo que se repete de um valor inicial até um valor final.

$$\begin{aligned} 0! &= 1 \\ n! &= \underline{\underline{1 \times 2} \times 3 \times \ldots \times n} = \Pi_{i=1}^{n} i \end{aligned}$$

Temos então:

Caso inicial $0! = 1! = 1$

```
factorial = 1;
```

invariante $i! = (i-1)! \times i$, com $1 \leq i \leq n$

```
factorial = factorial * i;
i = i + 1;
```

caso final (caso de paragem) $i = n$.

Utilizando o ciclo `while`

```
factorial = 1; // caso inicial 0!=1!=1
i = 2;
while (i <= n) { // condição de paragem i=n
  factorial = factorial * i;  //invariante i!=(i-1)*i
  i = i + 1;                  // incremento de i
}
```

Utilizando o ciclo `do while`

```
factorial = 1; // caso inicial 0!=1
i = 1;         // este ciclo é executado pelo menos uma vez
do {
  factorial = factorial * i;  //invariante i!=(i-1)*i
  i = i + 1;                  // incremento de i
} while (i <= n); // condição de paragem
```

Utilizando o ciclo `for`

```
factorial = 1; // caso inicial 0!=1!=1
for (i=2 ; i <=n ; i=i+1) // c. inicial; c. final; incremento de i
  factorial = factorial * i;
```

4.6 Modularidade Procedimental

Como já foi dito atrás (ver secção 1.2) "a modularidade é a capacidade para decompor casos complexos numa colecção de casos mais simples (chamados módulos) e em regras de composição" sendo que o suporte à modularidade é uma característica muito importante numa linguagem de programação.

Um primeiro nível de modularidade é dado pela divisão de um algoritmo em sub-algoritmos. Em *C/C++* esse primeiro nível é dado pela possibilidade que a linguagem dá ao programador de dividir um dado programa em funções independente entre si. As ligações entre funções é efectuada pelos argumentos e pelo resultado das mesmas.

4.6.1 Funções

Como já foi dito acima um programa em *C/C++* é constituído por uma função especial, `main`, ponto de início e fim (em condições normais) do funcionamento do programa e, eventualmente, por outras funções e/ou classes (*C++*). A organização física destas funções e/ou classes é feita em ficheiros sendo que é necessário declarar algo antes de o poder usar, isto é, a ordem pela qual se declaram as funções num dado ficheiro não é irrelevante. Na secção 4.7 falar-se-á desta questão mais em detalhe, nomeadamente da organização de um programa em múltiplos ficheiros.

Em *C++* uma função tem uma semântica (significado) à partida igual às funções matemáticas, isto é:

- n argumentos;
- 1 resultado (associado ao nome da função);
- uma sintaxe de chamada, $f(x)$, igual à usualmente usada em matemática.

Em termos operacionais temos que acrescentar:

- ligação entre os parâmetros (definição) e os argumentos (chamada) feita por cópia do valor;
- é possível definir e usar uma estrutura de dados local (e não global) à função.

Em termos sintácticos temos que considerar a declaração e a utilização.

Declaração, isto é a definição/construção da função através escrita do algoritmo que, dados os valores de entrada, produz o resultado esperado.

O corpo da função é uma sequência de instruções simples, o *C/C++* não admite funções dentro de outras funções, sendo que deve existir pelo menos uma instrução `return(<expressão>)` que defina o valor de saída (resultado) da função.

A utilização de uma função ocorre no âmbito do cálculo de uma expressão. Isoladamente, ou com outros elementos. Por exemplo:

```
x = sin(x)*3;
```

Exemplo de definição e utilização de uma função.

```
/*
 * Potência natural de um real
 * -> x (real), expoente (natural)
 * <- x^expoente
 */
double potencia(double x, int expoente) {
 int i;
 double pot=1; // x^0 = 1
 for (i=1; i<=expoente; i++)
   pot = pot*x;
 return pot;
}
```

É de notar a escrita da instrução `return` sem se utilizarem parêntesis. Isso é possível desde que não haja ambiguidade, um só valor após o identificador `return`, em caso de dúvida pode-se sempre colocar os parêntesis.

É importante acrescentar a especificação da função como comentário, logo após o cabeçalho da função. Uma descrição breve do que a função é suposto fazer, os seus parâmetros de entrada (pré-condição) e o resultado esperado (pós-condição).

A utilização desta função seria feita num outro ponto do programa, ou numa outra função, ou na função `main`. Por exemplo:

```
cout << x << "^" << n << " = " << potencia(x,n) << endl;
```

Parâmetros e Argumentos

A ligação entre módulos é feita pela ligação entre os argumentos (utilização) e os parâmetros (definição). Na linguagem *C/C++* essa ligação é feita por cópia dos valores dos argumentos para os parâmetros.

Para melhor compreender como se processa a ligação entre módulos recuperemos o exemplo da função potência, definindo com ela um programa completo (a numeração à esquerda é só para referência).

```
0 double potencia(double base, int expoente) {
1 int i;
2 double pot=1; // x^0 = 1
3 for (i=1; i<=expoente; i++)
4   pot = pot*base;
5 return pot; }

0 int main() {
1  double x=7.4, res;
2  int n=2;
3  res = potencia(x,n);
4  cout << res;
5  return 0; }
```

Vejamos então, passo a passo, como é que o estado do programa se vai alterando (ver Figura 4.4).

Na linguagem *C/C++* a ligação entre sub-programas (funções) é sempre feita através de uma *passagem por valor* (ver Figura 4.5). Em contraponto a esta forma de ligação temos que no *FORTRAN* as ligações são somente feitas *por referência* e em *Pascal* as ligações podem ser de ambos os tipos.

O que são exactamente estes dois modos de ligação, quais as diferenças e como é que isso afecta a construção de um programa em *C/C++*?

Passagem por Valor vs Passagem por Referência. Na passagem (de valores, ou ligação entre sub-programas) por valor os argumentos e os parâmetros são unidades de memória distintas, aquando da chamada do sub-programa é feita a cópia dos valores dos argumentos, no programa de chamada, para os respectivos parâmetros, no sub-programa que está a ser chamado. Após esta cópia, não existe mais nenhuma ligação entre estes dois conjuntos de posições de memória.

Em contraponto a isso temos a passagem por referência.

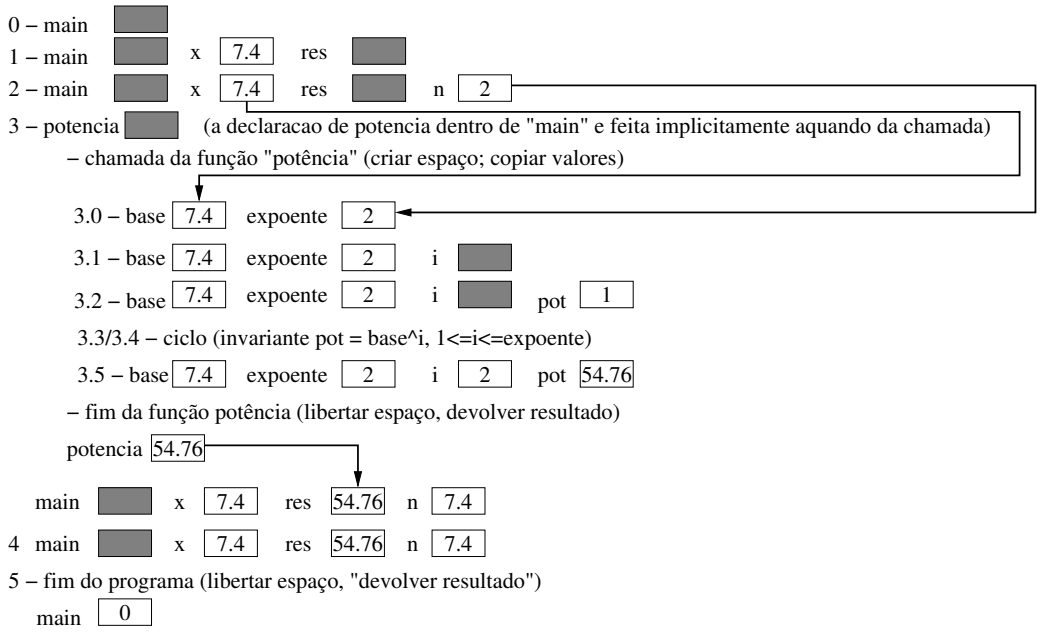


Figura 4.4: Ligação entre Argumentos e Parâmetros

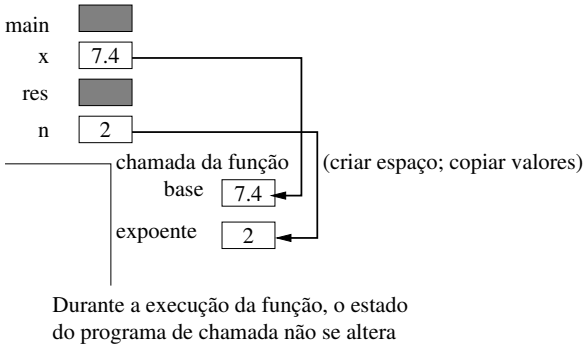


Figura 4.5: Ligação por Valor

Passagem por referência. No caso da passagem por referência os argumentos e os parâmetros são nomes distintos para a mesma posição de memória. Isto é aquando da chamada do sub-programa é feita a cópia das referências para a memória (ponteiros) respeitantes aos argumentos, no programa de chamada, para os respectivos parâmetros, no sub-programa que está a ser chamado. Após esta cópia, os parâmetros não são mais do que um nome que que se referem (apontam) para as posições de memória dos argumentos respectivos (ver Figura 4.6).

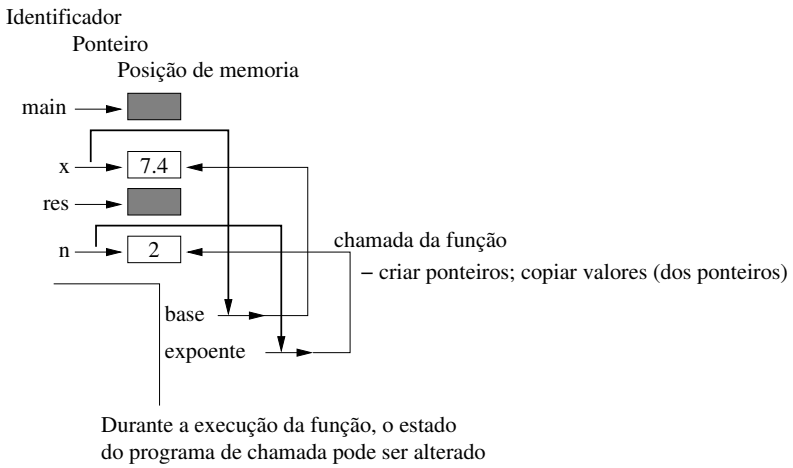


Figura 4.6: Ligação por Referência

Durante a execução da função, o estado do programa de chamada pode ser afectado. Sempre que os parâmetros são usados no sub-programa o estado do programa de chamada é usado ou alterado, conforme as circunstâncias (cálculo de uma expressão, ou instrução de atribuição).

Quais são as implicações, vantagens, desvantagens desta aproximação (só fazer a passagem por valor) da linguagem *C/C++*?

vantagens Dado que os sub-programas em *C/C++* são só do tipo função a ligação por valor é a mais correcta do ponto de vista formal. Após a cópia dos valores, no momento da chamada, não há mais nenhuma interacção com o programa de chamada, que não seja a devolução do valor final da função. Não há efeitos colaterais.

desvantagens Mas, e se se pretender que haja modificações nos argumentos? Por exemplo as funções de leitura de valores, por exemplo uma função que faça a troca dos valores entre os seus dois argumentos. Isto é se se pretender que o sub-programa tenha um comportamento de um módulo que recebe vários argumento e devolve vários resultados, e que permite que alguns desses resultados sejam alterações ao valor dos argumentos. Nestes casos é necessário estabelecer a ligação por referência.

Uma outra desvantagem prende-se com as estruturas de dados do tipo tabela (ver secção 4.8.1). Nestes casos a cópia de uma tabela, com a subsequente criação de uma variável local da mesma dimensão, pode-se revelar muito penalizadora, ou, em última instância, ser impeditiva do funcionamento do programa. Como veremos mais há frente as tabelas, em *C/C++* são sempre passadas por referência.

Como em *C++* não existe o mecanismo de passagem de valores por referência, então, para obter esse efeito, é necessário passar por valor as referências. Isto é em vez de ter

como argumento uma variável `x` do tipo inteiro (por exemplo), coloca-se como argumento a referência respectiva `&x`. Como parâmetro coloca-se uma variável do tipo referência (ponteiro) para um inteiro.

Vejamos um exemplo: um programa para efectuar a troca do valor de duas variáveis do tipo inteiro.

Primeiro a função que faz a troca dos valores:

```
void troca(int *a, int *b) {
 int aux;
 aux = *a;
 *a = *b;
 *b = aux;
}
```

temos então que ter variáveis do tipo ponteiro para inteiros. Para poder manipular os valores associados com as mesmas temos de recorrer ao operador `*`. Como a função não é suposto devolver nenhum resultado o seu tipo de saída é `void` que é um tipo especial em *C++*, o tipo vazio. Não há nenhuma instrução de `return` pela mesma razão.

De seguida o programa de chamada:

```
int main() [
 int x=3,y=7;

 troca(&x,&y);
 cout << x << y;
 return 0;
}
```

aquando da chamada da função `troca` os argumentos têm de ser passados por referência, ou mais correctamente, os argumentos, a ser passados por valor, têm de ser as referências correspondente às variáveis que queremos manipular. Desta forma '`x`' e '`a`' não são mais do que dois nomes referenciando a mesma posição de memória, todas as manipulações que se fizerem sobre '`a`' serão como sendo feitas através de '`x`'.

Como a função `troca` não tem um valor de saída definido a sua utilização é feita independentemente do cálculo de uma qualquer expressão (que é o caso usual de utilização de funções).

Âmbito das Variáveis

No que se disse até aqui sobre sub-programas em *C++* ficou sub-entendido que o estado de um sub-programa é independente do estado do programa de chamada. Isto é parcialmente verdadeiro, iremos ver de seguida as questões relacionadas com o âmbito das variáveis, isto é, quando e como é que podemos aceder aos seus valores, nomeadamente as questões sobre variáveis globais e variáveis locais.

Por âmbito de uma variável entende-se o grupo de instruções aonde se consegue aceder a um determinado valor associado a um identificador (variável).

O *C++* permite que uma variável seja declarada em qualquer ponto do(s) ficheiro(s) que contêm as funções, nomeadamente a função `main`, que constituem o programa. O âmbito da variável vai desse ponto em diante, sujeito às seguintes restrições:

Variáveis Globais as variáveis que são declaradas fora de uma qualquer função têm um âmbito global, isto é, são utilizáveis em qualquer ponto do programa. Como excepção a

esta regra temos a possibilidade de definir uma variável com o mesmo nome mas contida numa função, esta segunda definição, local, sobrepõe-se à global.

Variáveis Locais as variáveis que são declaradas dentro de uma qualquer função têm um âmbito local, isto é, só são válidas na função em que são declaradas.

Não Re-declaração não é possível re-declarar, dentro do mesmo nível (local ou global), uma dada variável.

Parâmetros de Funções (estáticos) os parâmetros de uma função definem variáveis declaradas localmente e cujos valores iniciais são os valores que recebem dos argumentos aquando da chamada. As variáveis locais (parâmetros) são completamente independentes dos argumentos (variáveis de uma outra função).

Parâmetros de Funções (dinâmicos) os parâmetros de uma função que sejam do tipo referência (ponteiro), definem variáveis (do tipo ponteiro) declaradas localmente. As variáveis locais não são mais do que um segundo conjunto de nomes correspondente aos argumentos (variáveis de uma outra função) e seus valores

A este conjunto de regras deve-se acrescentar um conjunto de boas práticas:

Não Declaração de Variáveis Globais a razão de tal atitude é simples, a metodologia da programação orientada para os objectos define os programas como colecções de objectos, entidades que agregam algoritmos e estruturas de dados, e nos quais a comunicação entre os objectos é feita de forma a que possamos ter encapsulamento da informação e re-utilização do código. As variáveis globais vão de contra a estes propósitos.

Declarações no Inicio das Funções embora seja possível declarar variáveis em qualquer ponto de uma função o concentrar-se as declarações logo nas primeiras linhas de cada função permite ter de imediato uma compreensão de qual é o conjunto de variáveis que vai definir o estado desse sub-programa.

Recorrência

Em *C++* é possível a uma função chamar-se a si própria, a este tipo de utilização designa-se chamada recorrente (ou recursiva).

A recorrência define um processo repetitivo, como tal é necessário definir-se:

Caso Inicial conjunto de valores dos argumentos válidos (pré-condição);

Caso Recorrente caso repetitivo, deve assegurar que o cálculo pretendido é efectuado (invariante da recorrência) e que o processo converge para a condição de paragem.

Condição de Paragem caso não repetitivo que deve assegurar que a recorrência para.

A necessidade de que, de um caso de inicial se caminhe em direcção a um caso final (condição de paragem) num número finito de passos, leva a que a recorrência só é possível sob conjuntos de valores (para os argumentos) que possuam uma boa ordenação. Em termos do *C++* isso quer basicamente dizer que o argumento de uma função recorrente que assegura a convergência para o caso final deve ser do tipo inteiro (`int`).

Cada chamada da função implica o criar de um novo conjunto de variáveis locais. É necessário ter isto em conta dado que o mesmo pode ser muito pesado caso haja muitas chamadas por recorrência e/ou as estruturas locais sejam de grande dimensão.

O exemplo clássico de função recorrente é a função factorial.

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

A sua implementação em *C++* é quase imediata:

```
long int factorial(int n) {
  if (n==0) return 1
  else return (n*factorial(n-1));
}
```

como pré-condição temos que o n tem de ser um número natural, isto é, $n \geq 0$.

Um outro exemplo também muito usado é o de um programa que faça a inversão de uma frase lida do terminal. Este é um caso interessante para analisar como se processam as sucessivas chamadas recorrentes com as consequentes declarações de variáveis locais.

Vejamos o programa:

```
#include <iostream>

using namespace std;

void inverte () {
  char aux;
  if (aux=cin.get() && aux != '.') {
    inverte();
    cout << aux;
  }
}

int main() {
  cout << "escreva uma frase (termine com um '.'): ";
  inverte();
  return 0;
}
```

Analisando o funcionamento deste algoritmo do ponto de vista das variáveis locais das sucessivas chamadas recorrentes verifica-se que, a inversão da ordem das letras na palavra decorre automaticamente da forma como a recorrência se processa, primeiro, na fase da leitura, indo do exterior para níveis cada vez mais profundos, depois, na fase da escrita, do nível mais profundo para o exterior (ver Figura 4.7).

É importante notar que a cada chamada recorrente é criada uma nova variável `aux`, no exemplo apresentado isso significou um total de cinco variáveis do tipo `char`. O que aqui não é significativo pode ser determinante noutras situações, não só no número total de chamadas recorrentes (por exemplo, neste caso a escolha de uma frase muito comprida) como nas estruturas de dados a utilizar (neste caso, se em vez de optar pela declaração `char a` se se tivesse optado, por exemplo, pela declaração `char a[1000]`).

Um nota final, a função `main` não é passível de chamadas recorrentes.

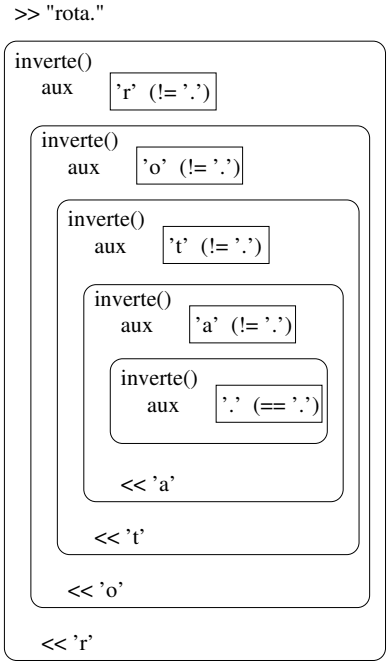


Figura 4.7: Variáveis Locais numa Chamada Recorrente

4.7 Estruturação de um Programa

A metodologia da programação orientada para os objectos caracterizada pela construção de um programa em termos de uma estrutura de classes é apropriada para a construção de programas modulares e de grande dimensão.

A metodologia de programação orientada para os objectos não deve fazer esquecer a metodologia de programação estruturada e descendente, a qual é ainda hoje importante para a construção de programas em que não se utilize a aproximação da programação orientada para os objectos, por exemplo, pela sua dimensão reduzida. Outra razão para não esquecer a metodologia de programação estruturada e descendente passa pela organização do código dentro das classes, o qual pode ser suficientemente complexo para também

> Decompor o problema global em sub-programas específicos (por tarefa bem determinada), provar que se cada sub-problema é resolvido correctamente, e se as várias soluções parciais podem ser combinadas de forma correcta, então o problema global será resolvido correctamente.
>
> Repetir este processo até atingir sub-problemas simples.
>
> S. Alagic e M.A. Arbib (Alagič & Arbib, 1978)

Embora no global a metodologia programação orientada para os objectos tenha surgido como forma de resolver as questões relacionadas com a abstracção e modularidade, questões essas só parcialmente respondidas pela metodologia da programação estruturada e descendente, esta última não deve ser esquecida aquando da construção dos métodos e/ou programas de pequena ou média dimensão.

4.7.1 Separação de um Programa em Múltiplos Ficheiros

Seja a separação de um programa em classes seja em sub-programas, em ambas as aproximações fala-se de separação de algo grande e complexo, em componentes de menor dimensão e/ou complexidade.

Esta separação em estruturas funcionais diferentes tem, em *C++*, uma correspondência física. A separação de um programa em múltiplos ficheiros.

Em primeiro lugar a convenção em *C/C++*é de separar a componente das declarações da componente das implementações.

Declaração/Especificação: por declaração, ou especificação, de uma função entende-se a explicitação dos seus argumentos e do seu resultado, tanto em termos de número e tipo dos argumentos como do tipo do resultado.

Por exemplo.

```
long int factorial(int);
```

Basta-nos para saber que a função factorial necessita de um argumento do tipo inteiro e que produz um resultado também do tipo inteiro (mais especificamente do tipo «inteiro longo»).

A especificação adiciona a isto as pré e pós condições, isto é, quais são os valores aceitáveis para o(s) argumento(s) e qual é o valor do resultado que se espera para um dado argumento. Em *C/C++* as pré e pós condições não fazem parte da linguagem, podemos (devemos) incluí-las como comentários.

```
/*
 * Pré: n >= 0, n do tipo inteiro
 * Pós: factorial(n) = n!
 */
long int factorial(int);
```

Num programa em que haja um uso apropriado das estruturas da linguagem (sonegação da informação, não utilização de efeitos colaterais, não utilização de variáveis globais) a forma concreta como a função é implementada é independente da sua utilização.

Isto é, num dado programa, bastar-no-ia esta informação para utilizar a função factorial num outro ponto do programa.

Por outro lado para o compilador completar a sua análise léxico/gramatical do programa que chama a função factorial, a implementação desta é também irrelevante. Basta saber que os argumentos na chamada correspondem em número e tipo com os parâmetros na declaração, e que o tipo do resultado está de acordo com a variável, ou expressão, que vai receber esse valor.

Esta separação entre declaração e implementação possível em programas em *C/C++* tem uma expressão física.

Por convenção, para um dado programa «exemplo» criar-se-ão dois ficheiros:

Ficheiro das Declarações (*headers file*) ficheiro com extensão «.hpp», `exemplo.hpp` (em *C* seria simplesmente `.h`).

Ficheiro das Implementações (*C/C++ file*) ficheiro com extensão apropriado à linguagem, por exemplo, «.cpp», `exemplo.cpp` (em *C* seria simplesmente `.c`).

Por exemplo, a construção de um programa para o cálculo do factorial de um número natural.

Num dado ficheiro `factorial.hpp` tem-se somente a declarações (cabeçalhos) da função factorial.

```
// factorial.hpp
/*
 * Pré: n >= 0, n do tipo inteiro
 * Pós: factorial(n) = n!
 */
long int factorial(int);
```

No segundo temos a implementação da função factorial.

```
// factorial.cpp
long int factorial(int n) {
  if (n==0) return 1
  else return (n*factorial(n-1));
}
```

Finalmente ter-se-á o programa principal, o qual terá extensão «.cpp» dado se tratar de código e não de declarações, e que terá de conter a função `main`. Podemos designá-lo por `usaFactorial.cpp`.

```
#include <iostream>
#include "factorial.hpp"

using namespace std;

int main() {
  int i;
  cout << "Introduza um inteiro, maior ou igual a zero: ";
  cin >> i;
  cout << i << "! = " << factorial(i) << endl;
}
```

É de notar a inclusão da directiva de compilação `#include factorial.hpp`. Esta é necessária dado que estamos a usar a função «`factorial(i)`», sendo que não temos, neste ficheiro, nada em concreto sobre ela.

Para a fase análise léxico/gramatical só é necessário saber como usar a função, essa informação está contida no ficheiro dos cabeçalhos, e como tal a inclusão é referente ao ficheiro `factorial.hpp`.

Para a fase da criação do código máquina (construção do programa executável, final) é necessário juntar ao nosso programa de chamada a componente da implementação da função factorial, isso é feito pelo compilador em dois passos:

Conversão do código *C++* em código máquina: esta conversão é feita pelo compilador recorrendo à opção «`-c`».

```
cpp -c factorial.c
```

Este procedimento cria um ficheiro `factorial.o`, «`o`» de *object code*, isto é, código máquina.

Junção das diferentes componentes: de forma a criar o programa final (executável) é então necessário juntar todas secções do programa compilado (ficheiros «.o» e programa principal) num só ficheiro. Essa junção (*linking*) é feita pelo compilador explicitando todos os ficheiros a juntar

```
cpp factorial.o usaFactorial.cpp -o usaFactorial
```

A opção «`-o`» é meramente usada para dar um nome específico ao programa final. A convenção em Linux é que este não tenha extensão, outra convenção usual é usar a extensão «`.exe`».

Temos assim um programa dividido em três ficheiros, a divisão em três ou mais ficheiros deve ser ditada por uma organização funcional do código do programa. A forma como lidar com esta explosão no número de ficheiros a considerar pode, e deve, ser feita recorrendo a automatismos próprios do sistema operativo, veja a este propósito o apêndice C.

4.8 Estruturas de Dados Compostas

4.8.1 Estruturas de Dados Estáticas

O *C/C++* possuí dois tipos de estruturas de dados pré-definidas: as tabelas e os registos.

Tabelas (estáticas)

As tabelas (*arrays*) são tabelas uni-dimensionais e homogéneas. Isto é são estruturas com uma só dimensão e de elementos todos do mesmo tipo.

Por exemplo um vector de inteiros:

```
int v[100];
```

neste caso temos uma tabela de inteiros com a capacidade de guardar 100 inteiros. O acesso aos elementos individuais da tabela é feita através de índices (de 0 (zero) até número de elementos-1), por exemplo, `v[4]`, dá-nos o 5º elemento da tabela.

Podemos declarar e inicializar elementos deste tipo.

```
int v[] = {1,2,3,4};
```

Neste caso é usual não especificar o número de elementos da tabela, o mesmo é calculado automaticamente pelo compilador pela simples contagem dos elementos da lista de valores iniciais. Se se optar por explicitar o números de elementos da tabela então esse valor tem de coincidir com o número de elementos na lista de inicialização.

Uma variável do tipo tabela é, sempre, uma variável do tipo ponteiro. Este facto é importante quando se consideram funções. Por exemplo, pretende-se construir uma função que dado um vector de elementos inteiros o ordene (ver §).

```
/*
 * Recebe um vector, assim como o num. de elementos e ordena-o
 * através do método Borbulhagem («Bubble sort»)
 */
int borbulhagem(int nElem, int v[]) {

  int i,aux;
```

```
  bool ordenado;

  do {
    ordenado = true;
    for (i = 0; i < nElem-1 ; i++) { // precorre o vector
      if (v[i] > v[i+1]) { // se necessário troca elementos
        aux = v[i];
        v[i] = v[i+1];
        v[i+1] = aux;
        ordenado = false;
      }
    }
  } while (ordenado); // repete até não haver mais trocas

  return 0;
}
```

Podemos desde já notar alguns pontos que de certa forma parecem colocar em questão o que foi anteriormente dito:

- Dado que o vector está a ser (aparentemente) passado por valor, o seu «valor» (isto é o seu conteúdo) não deveria ser alterado e como tal a ordenação não deveria ter consequências no programa de chamada (não é isso que acontece na realidade);
- A dimensão do vector não está a ser explicitada (`v[]`);

A primeira questão prendem-se com o facto já referido de que uma variável do tipo tabela é sempre (mesmo que isso não seja explícito) do tipo ponteiro. Como tal o vector está a ser passado «por referência» e tudo o que se fizer aos seus elementos reflecte-se no programa de chamada.

A segunda questão é respondida pelo compilador de *C/C++*. Na chamada de uma função não é necessário explicitar a dimensão das tabelas. A dimensão é «dada» pelo programa de chamada. É de notar que se se optar por explicitar a dimensão da tabela (é sempre possível) a mesma tem depois de coincidir com a dimensão declarada no programa de chamada.

Duas questões finais sobre tabelas.

Podemos ter tabelas n-dimensionais se se declarar uma tabela com elementos do tipo tabela. Por exemplo `int matriz[10][10]`, dá-nos uma matriz de 10 por 10 de elementos do tipo inteiro;

As tabelas também podem ser declaradas de forma dinâmica, por exemplo, `int *v` pode ser vista como a declaração de uma tabela de elementos do tipo inteiro.

A vantagem deste tipo de definição é que a estrutura assim criada pode depois ter uma dimensão ajustada às necessidades, em contraponto com o outro tipo de declaração que fixa a dimensão da tabela. A desvantagem é que obriga à gestão da afectação da memória por parte do utilizador. Ver-se-à na secção 4.8.2 como proceder nesses casos.

Registos

Ao contrário das tabelas os registos definem uma estrutura de dados não homogénea. Isto é podemos criar um registo que integre um elemento do tipo inteiro e um outro do tipo real e um outro do tipo .... Os registos não definem, ao contrário das tabelas, estruturas com vários elementos todos do mesmo tipo.

Por exemplo: numa dada empresa pretende-se guardar e manipular a informação referente aos seus empregados. Em vez de colocar a informação referente aos empregados em estruturas não relacionas entre si, é vantajoso agrupá-las todas numa só estrutura. Temos então algo como:

```
struct empregado {
 char nome[60]; // nome do empregado
 int numero; // número de empregado
 int cc; // número de identificação (BI ou CC)
 int tipoId; // tipo de identificação (BI=1 ou CC=2 ou ...)
 int telefone;
};
```

Esta declaração cria um novo tipo de dados, o tipo «`empregado`». Caso se pretenda criar uma variável deste novo tipo podemos fazê-lo utilizando a sintaxe habitual:

```
empregado ep1,ep2;
```

Como podemos ver por este exemplo, um registo pode conter uma (ou mais) tabela. O contrário também é verdade. Por exemplo:

```
empregado listaEmpregado[100];
```

podia ser a estrutura aonde toda a informação referente aos empregados era guardada.

As variáveis do tipo registo podem ser declarados e inicializados de uma forma idêntica às variáveis do tipo tabela.

```
empregado eprgds = {"Francisco",23,324543,1,230239239};
```

A forma de aceder a cada um dos membros de uma registo é feita através da utilização do operador '`.`' Por exemplo:

```
cout << ep1.nome << ep1.numero << ep1.cc;
```

Finalmente, no caso em que se queira definir um ponteiro para uma estrutura, por exemplo:

```
empregado* pep;
```

O acesso ao valor dos membros da estrutura tem de respeitar o que está definido para aceder aos valores apontados e o acesso aos diferentes membros da estrutura. Para esta definição ter-se-ia:

```
cout << (*pep).nome << (*pep).numero << (*pep).cc;
```

No caso do *C++* a definição dos objectos e a sua utilização vai levar a que este último situação ocorra muitas vezes. Este facto terá estado na base do desenvolvimento de uma sintaxe alternativa, mais apelativa.

Em vez de `(*pep).nome` podemos escrever `pep->nome`, isto é em vez do operador '`.`' utiliza-se o operador '`->`'. Ter-se-ia então:

```
cout << pep->nome << pep->numero << pep->cc;
```

Os objectos do tipo estrutura podem ser argumentos de funções, assim como o resultado de funções. É também possível fazer atribuições entre variáveis do tipo estrutura.

4.8.2 Estruturas de Dados Dinâmicas

As estruturas de dados estáticas tal como as vistas na secção 4.8.1 levantam duas questões:

- têm uma dimensão (número de elementos) fixa;
- têm uma forma fixa.

Dimensão (número de elementos) Fixa Ao se definir uma tabela com um número de elementos fixos isso leva a que, aquando da utilização do programa esse número máximo de elementos não pode ser ultrapassado, isto é se a utilização do programa revelar que a estimativa para a dimensão da estrutura está errada a única forma de corrigir o problema é alterar a estrutura e recompilar o programa. Por outro lado se a estimativa se revelar muito exagerada há um desperdício de memória que é tento maior quantos mais elementos se utilizarem.

Um caso que pode servir para ilustrar esta situação é a da variável `listaEmpregado` no exemplo da secção 4.8.1. Com esta definição a empresa está limitada a 100 empregados, não é possível guardar a informação para 101, ou mais, empregados.

Por outro lado a definição do membro nome é uma apropriada para nomes portugueses (em geral muito grandes), já é totalmente exagerada para uma companhia que, por exemplo, tivesse a maior parte dos trabalhadores de nacionalidade inglesa (nomes em geral pequenos).

Forma Fixa Uma outra questão tem a ver com a implementação de estruturas cuja forma e dimensão são muito variáveis.

Por exemplo, como representar uma rede rodoviária, com as cidades e as estradas que as ligam? Uma resposta a esta questão é dada por uma estrutura dinâmica, tanto na forma como na dimensão, os grafos.

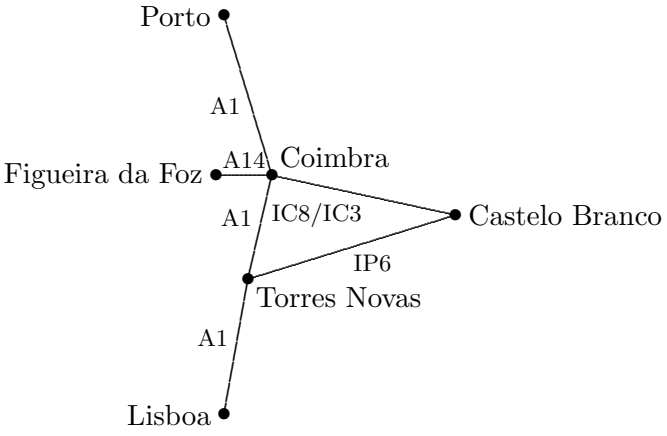


Figura 4.8: Mapa de Estradas (grafo)

Embora seja possível representar esta estrutura utilizando estruturas estáticas a utilização de estruturas dinâmicas capazes de se adaptarem tanto na forma como no número de elementos a guardar é muito vantajosa.

Estruturas Dinâmicas na Dimensão Podemos definir tabelas com um número de elementos não eplicitado à partida. As seguintes declarações são equivalentes:

```
int* tabela1;
int tabela2[];
```

ambas definem ponteiros para inteiros.

Em ambos os casos a possibilidade de associar um série de posições de memória contiguas ao ponteiro assim definido permite-nos forma criar uma tabela de elementos do tipo pretendido (neste caso o tipo `int`).

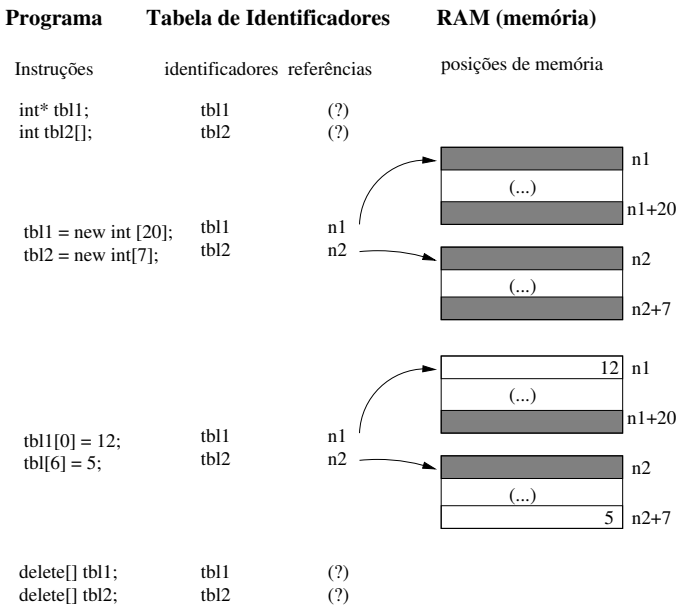


Figura 4.9: Variáveis Estáticas vs Variáveis Dinâmicas

A gestão de memória, a sua afectação e libertação está a cargo de duas dois operadores específicos da linguagem *C++* o operador `new` e `delete`.

De forma a afectar memória usa-se o operador `new`. Este operador tem duas variantes:

```
<ponteiro> = new <tipo>
<ponteiro> = new <tipo> [<número_de_elementos>]
```

No primeiro dos casos está-se a afectar uma só posição de memória, no segundo caso trata-se de um bloco de posições de memória contíguas. Além da afectação da memória o ponteiro que está associado à variável que se está a declarar é inicializado para a primeira (e única, no primeiro dos casos) posição de memória.

Assim como a afectação, a libertação deste espaço de memória é também da responsabilidade do programador, o qual deve ter o cuidado de proceder à libertação do espaço logo que o mesmo deixe de ser necessário. Para este efeito existe o operador `delete` cujo formato é o seguinte:

```
delete <ponteiro>;
delete [] <ponteiro>;
```

No primeiro dos casos está-se a libertar o espaço que foi afectado para um só elemento (através da utilização do operador `new`). No segundo caso liberta-se todo um bloco de memória (que foi previamente afectado como o operador `new[]`).

Em conclusão, no caso das tabelas podemos optar por uma declaração estática, ou por uma declaração dinâmica. Por exemplo:

```
char Estatica[20];

char tblDinamica[];
tblDinamica = new char[20];
```

tendo afectado o mesmo número de posições de memória para as duas variáveis estas são equivalentes do ponto de vista de utilização (excepção feita para a possibilidade/necessidade de libertar o espaço afectado no segundo caso). No entanto a segunda declaração dá-nos a liberdade de só afectar o espaço no momento em que ele é necessário e (eventualmente) já com o conhecimento do espaço exacto que é necessário.

No primeiro caso o espaço a afectar tem de ser estimado aquando da programação, no segundo caso o espaço a afectar pode ser determinado aquando da utilização, ajustando-o às necessidades.

Estruturas Dinâmicas na Forma (e Dimensão) Além das vantajens que advêm do ajustar a dimensão às necessidade, a gestão dinâmica da memória dada pela utilização dos ponteiros permite também a definição de estruturas dinâmicas, não só na sua dimensão mas também na sua forma.

Como exemplos de estruturas dinâmicas muito usadas em programação temos:

Listas: sequências lineares de elementos:

Pilhas: listas com acesso por um só ponto, o topo. Implementam a a disciplina *Last In First Out (LIFO)*, o último a entrar é o primeiro a sair. Entre muitas utilizações temos o cálculo de expressões em notação Polaca inversa.

$$Pilha = (\{pilhaVazia, elemento : Pilha\}, \{\text{cria, push, pop, top, vazia?}\})$$

As pilhas são definidas indutivamente. Temos a $pilhaVazia$ (caso de base) e temos as pilhas não vazias (caso indutivo) que são constituídas por um elemento seguido de uma pilha.

Para a definição das operações internas é necessário explicitar conjunto das pilhas não vazias $PilhaNãoVazia = Pilha \setminus \{pilhaVazia\}$.

As operações têm a seguinte especificação:

$$\begin{array}{rcccc} \text{cria} & : & \mathbf{1} & \longrightarrow & Pilha \\ & & * & \longmapsto & pilhaVazia \end{array}$$

$$\begin{array}{rcccc} \text{push} & : & Pilha \times Elementos & \longrightarrow & Pilha \\ & & (p, e) & \longmapsto & e : p \end{array}$$

$$\begin{array}{rcccc} \text{pop} & : & PilhaNãoVazia & \longrightarrow & Pilha \\ & & e : p & \longmapsto & p \end{array}$$

$$\begin{array}{rcccc} \text{top} & : & PilhaNãoVazia & \longrightarrow & Elementos \\ & & e : p & \longmapsto & e \end{array}$$

$$\begin{array}{rcccl} \text{vazia?} & : & Pilha & \longrightarrow & \mathbb{B} \\ & & p & \longmapsto & \begin{cases} \mathscr{V}, & \text{se p = pilhaVazia} \\ \mathscr{F}, & \text{se p} \neq \text{pilhaVazia} \end{cases} \end{array}$$

Filas listas com acesso apenas pelas extremidades, a entrada e a saída. Implementam a disciplina *First In First Out (FIFO)*, o primeiro a entrar é o primeiro a sair. A simulação de filas de espera, sejam elas caixas num dado hipermercado ou semáforos num dado cruzamento, podem ser modelizadas através deste tipo de estrutura.

$$Fila = (\{filaVazia, elemento{:}Fila\}, \{\text{cria, insere, retira, topo, vazia?}\})$$

A exemplo das pilhas, as filas são também definidas indutivamente. De igual modo é necessário explicitar o conjunto das filas não vazias: $FilaNãoVazia = Fila \setminus \{filaVazia\}$.

As operações têm a seguinte especificação:

$$\begin{array}{rcccl} \text{cria} & : & \mathbf{1} & \longrightarrow & Fila \\ & & * & \longmapsto & filaVazia \end{array}$$

$$\begin{array}{rcccl} \text{insere} & : & Fila \times Elementos & \longrightarrow & Fila \\ & & (f, e) & \longmapsto & e : f \end{array}$$

$$\begin{array}{rcccl} \text{retira} & : & FilaNãoVazia & \longrightarrow & Fila \\ & & f : e & \longmapsto & f \end{array}$$

$$\begin{array}{rcccl} \text{topo} & : & FilaNãoVazia & \longrightarrow & Elementos \\ & & f : e & \longmapsto & e \end{array}$$

$$\begin{array}{rcccl} \text{vazia?} & : & Fila & \longrightarrow & \mathbb{B} \\ & & p & \longmapsto & \begin{cases} \mathscr{V}, & \text{se p = filaVazia} \\ \mathscr{F}, & \text{se p} \neq \text{filaVazia} \end{cases} \end{array}$$

Listas: listas genéricas com acesso livre a uma qualquer posição. Podem ser usadas para modelizar todo o tipo de situações em que se pretende uma lista de elementos todos do mesmo tipo mas em que o número de elementos a considerar varia muito durante o correr do programa.

$$\text{Lista} = (\{listaVazia, elemento{:}lista\}, \{\text{cria, insereN, retiraN, veN, vazia?}\})$$

A exemplo das pilhas e filas são também definidas indutivamente.

As operações têm a seguinte especificação:

$$\begin{array}{rcccl} \text{cria} & : & \mathbf{1} & \longrightarrow & Lista \\ & & * & \longmapsto & listaVazia \end{array}$$

$$\begin{array}{rcccl} \text{insereN} & : & Lista \times Elementos \times \mathbb{N} & \longrightarrow & Lista \\ & & (l, e, i) & \longmapsto & l' = l_1 : \ldots : l_{i-1} : e : l_i : \ldots : l_n \end{array}$$

No caso em que o comprimento da lista é menor do que a posição em que se pretende efectuar a inserção vai-se optar por inserir no fim da lista. A outra opção possível é a de considerar que nesse caso o valor da função não está definido.

$$\begin{array}{rcccl} \text{retiraN} & : & \mathit{Lista} \times \mathbb{N} & \longrightarrow & \mathit{Lista} \\ & & (l, i) & \longmapsto & l' = l_1 : \ldots : l_{i-1} : l_{i+1} : \ldots : l_n \end{array}$$

No caso em que o comprimento da lista é menor do que a posição em que se pretende efectuar a remoção esta não é efectuada. A outra opção possível é a de considerar que nesse caso o valor da função não está definido.

$$\begin{array}{rcccl} \text{veN} & : & \mathit{Lista} \times \mathbb{N} & \longrightarrow & \mathit{Elementos} \\ & & (l, i) & \longmapsto & \begin{cases} e, & \text{se } |l| \geq i \\ \perp, & \text{se } |l| < i \end{cases} \end{array}$$

$$\begin{array}{rcccl} \text{vazia?} & : & \mathit{Lista} & \longrightarrow & \mathbb{B} \\ & & p & \longmapsto & \begin{cases} \mathscr{V}, & \text{se p} = \text{listaVazia} \\ \mathscr{F}, & \text{se p} \neq \text{listaVazia} \end{cases} \end{array}$$

Árvores: estrutura hierárquica.

Árvores Binárias: uma raiz e dois sub-árvores. Dado que é possível representar qualquer tipo de árvore como uma árvore binária, este tipo de estrutura de dados é usada para modelizar todas as situações em que se pretende representar uma estrutura hierárquica.

$$\begin{aligned} \text{AB} = (\{&\mathit{Abvazia}, \mathit{ABesq} : \mathit{elemento} : \mathit{ABdir}\}, \{\text{criaVazia,criaAB,} \\ &\text{procuraElemento,insereElemento,retiraElemento,vazia?}\}) \end{aligned}$$

No caso deste tipo de dados há variantes tanto na forma de definir assim como nas operações de base (Main & Savitch, 1997; Semgupta & Korobkin, 1994; Weiss, 1997; Aho *et al.*, 1983).

Uma operação importante para este tipo de dados é a travessia da árvore, isto é, o «visitar» de todos os nós sem esquecimentos e sem repetições. Podemos ter travessias *em-ordem*, em *pré-ordem* e em *pós-ordem*, consuante a ordem pela qual se visita a raiz em relação aos outros dois elementos, a sub-árvore esquerda e a sub-árvore direita.

De novo temos uma definição indutiva, neste caso com uma dupla indução.

Esta estrutura e a forma como é utilizada é muito mais complexa e variada do que os casos anteriores. Apresenta-se de seguida as definições das operações, é necessário ter em conta que neste caso não há uma uniformidade, estas definições representam uma possível definição para uma utilização genérica.

$$\begin{array}{rcccl} \text{criaVazia} & : & \mathbf{1} & \longrightarrow & \mathit{AB} \\ & & * & \longmapsto & \mathit{ABVazia} \end{array}$$

$$\begin{array}{rcccl} \text{criaAB} & : & \mathit{AB} \times \mathit{Elementos} \times \mathit{AB} & \longrightarrow & \mathit{AB} \\ & & (ab1, e, ab2) & \longmapsto & ab1 : e : ab2 \end{array}$$

Tanto $ab1$ como $ab2$ podem ser árvores binárias vazias. No caso extremo de ambas as sub-árvores serem vazias temos aqui uma operação que transforma um elemento numa árvore binária.

$$\begin{array}{rcccl} \text{procuraElemento} & : & AB \times Elementos & \longrightarrow & \mathbb{B} \\ & & (ab, e) & \longmapsto & \begin{cases} \mathscr{V}, & e \in ab \\ \mathscr{F}, & e \notin ab \end{cases} \end{array}$$

$$\begin{array}{rcccl} \text{insereElemento} & : & AB \times Elementos & \longrightarrow & AB \\ & & (ab, e) & \longmapsto & ab' \end{array}$$

A posição de inserção vai estar dependente da forma como os elementos estão organizados na árvore. No caso da inserção ser feito nos extremos (folhas) da árvore ao elemento serão «adicionadas» duas sub-árvores vazias.

$$\begin{array}{rcccl} \text{retiraElemento} & : & AB \times Elementos & \longrightarrow & AB \\ & & (ab, e) & \longmapsto & ab' \end{array}$$

No caso do retirar de uma das folhas (elementos no extremo da árvore) é só uma questão de colocar no nó da árvore exactamente acima do elemento a retirar a árvore vazia. No caso de se pretender retirar um elemento do meio da árvore, isso vai implicar um re-ordenar da mesma. A forma exacta como isso é feito vai depender da forma como a árvore está organizada.

$$\begin{array}{rcccl} \text{vazia?} & : & AB & \longrightarrow & \mathbb{B} \\ & & p & \longmapsto & \begin{cases} \mathscr{V}, & \text{se p} = \text{ABVazia} \\ \mathscr{F}, & \text{se p} \neq \text{ABVazia} \end{cases} \end{array}$$

Grafos: conjunto de nós e de ramos entre eles (ver figura 4.8). É uma estrutura muito importante quando se quer modelizar situações em rede, isto é, situações em que se tem «locais» e «ligações» entre eles.

Para este tipo de estrutura temos, a exemplo das árvores, diferentes representações. Talvez a mais usual é a representação através de um conjunto de nós e um conjunto de arcos (pares de nós: nó origem, nó destino).

As operações passam pela criação do grafo vazio, a inserção de um novo nó, a inserção de um novo arco, as travessias, e as operações de retirar nós e arcos.

Pelas definições destes tipos de dados acima apresentados é possível reparar que a definição dos elemen tos é feita de forma recursiva. Por exemplo para as *Pilhas*:

$$\text{Pilha} = \begin{cases} \text{Vazia}, & \text{Caso de base} \\ \text{Elemento} : \text{Pilha} & \text{Caso Recursivo} \end{cases}$$

Este tipo de estrutura é possível de representar em *C* da seguinte forma:

```
typedef struct no { // nó de um pilha
  int elems;        // o elemento
  struct no* prox;  // o ponteiro para o próximo elemento
} No;

typedef No* Pilha;  // Pilha como o tipo ponteiro para No
```

Temos então a definição de uma pilha como uma lista de nós. A definição recursiva é possível dado que o elemento não se auto-referência, o que temos é que `prox` é do tipo ponteiro para `no` e não um elemento do tipo `no`.

Para a definição de uma árvore binária ter-se-ia algo de muito idêntico.

```
struct ABno {           // Nó árvore binária
  int elems;            // o elemento na «raiz» da árvore
  struct ABno* ABesq; // ponteiro para a AB esquerda
  struct ABno* ABdir; // ponteiro para a AB direita
}
```

Exemplo: Pilha de Caracteres

Dado que em *C++* este tipo de estrutura vai ser representado de uma forma bem diversa, à custa de objectos não vou aqui aprofundar mais o assunto. A título de exemplo apresenta-se uma possível implementação do tipo abstracto de dados *Pilha* em *C*.

Ficheiro `pilhas.h`

```
/*********************************************/
/*  Pilhas Implementação Dinâmica            */
/*********************************************/
typedef struct no {
  int elems;
  struct no* prox;
} No;

typedef No* Pilha;

void cria(Pilha*);
int push(int, Pilha*);
int top(Pilha);
int pop(Pilha*);
int vazia(Pilha);
```

Ficheiro `pilhas.c`

```
#include "pilha.h"
#include <stdlib.h>
#include <stdio.h>

#define PILHAVAZIA 1

Pilha afectapilha(void) {
  return (Pilha) malloc(sizeof(No));
}

void cria(Pilha* p){
  (*p) = NULL;
}

int push(int elem,Pilha* p){
  Pilha novo;

  if (novo = afectapilha()){
```

```
    novo->elems = elem;
    novo->prox = (*p);
    (*p) = novo;
    return 0;
  }
  else
    return 1;
}

int pop(Pilha* p){
  Pilha aux;

  if ((*p) == NULL)
    return PILHAVAZIA;
  aux = (*p);
  (*p) = (*p)->prox;
  free(aux);
  return 0;
}

int top(Pilha p){
  return(p->elems);
}

int vazia(Pilha p){
  return(p == NULL);
}
```

Capítulo 5

Programação Orientada para os Objectos (em *C++*)

5.1 Classes e Objectos

Uma classe é um tipo definido pelo utilizador, isto é, a definição de um conjunto de elementos e das funções que operam com esses novos elementos.

Considere-se a seguinte definição de um tipo de dados e a declaração de algumas funções sobre esses elementos:

```
struct Data {
  int d,m,a;  // dia, mês, ano
};
```

```
void inic_data(Data& d, int, int, int); // (int,int,int) -> Data
void adicionar_anos(Data& d, int n); // adiciona n anos a d
void adicionar_meses(Data& d, int n); // adiciona n meses a d
void adicionar_dias(Data& d, int n); // adiciona n dias a d
```

Este forma de definição levanta um conjunto de problemas que impedem a concretização pretendida da definição de um novo tipo de dados. Podemos ver que não há ligação entre a estrutura de dados e as funções que a vão manipular. Por outro lado não há forma de esconder a informação da implementação concreta dos novos elementos.

O *C++* providencia então uma nova forma de definir os tipos de dados dos utilizadores que vai permitir responder às questões que já foram referidas aquando do estudo da metodologia (ver capítulo 2), isto é, abstracção, encapsulação da informação e modularidade.

Temos então a seguinte definição:

```
class Data {
private:
   int dia,mes,ano;
public:
   void inic_data(int, int, int); // (int,int,int) -> Data
   void adicionar_anos(int);      // adiciona n anos a Data (ano)
   void adicionar_meses(int);     // adiciona n meses a Data (mes,ano)
   void adicionar_dias(int);      // adiciona n dias a Data (dia,mes,ano)
};
```

Note-se que a referência ao elemento `Data` desapareceu. Ele está subentendido em todos os elementos da classe.

A definição de uma classe (um novo tipo) pode ser sintacticamente muito complexa. No caso mais simples temos a definição de uma classe, com um dado nome, e uma ou mais secções em que se especificam atributos e funções.

Classe ::= `class` <identificador> {
[<tipo de acesso>: [(<declaração_de_elementos>; | <declaração_de_funções>;)]*]*
};

sendo que o «tipo de acesso» pode ser um dos seguintes três: `public`, `protected` e `private`.

A definição da classe `Data` deu-nos um primeiro exemplo deste novo tipo de definição. Como discutido anteriormente os diferentes tipos de acesso determinam: o interface público da classe (acesso `public`), isto é os dados e as funções da classe a que todos podem aceder; a secção privada da classe (acesso `private`), isto é, os elementos que só podem ser acedidos por elementos da própria classe; a secção protegida (acesso `protected`) determina a secção que pode ser acedida pelas funções da classe assim como das classes derivadas da classe. Ver-se-á mais à frente em que consistem as classes derivadas (secção 5.2.1).

É usual separar a declaração de uma classe (num ficheiro de declaração, por exemplo `data.hpp`) da implementação das funções (num ficheiro de implementações, por exemplo `data.cpp`).

No exemplo apresentado acima só se declararam as funções membro da classe, é necessário então proceder à sua definição. Como a implementação está a ser feita fora da classe, como «funções livres», é necessário especificar qual a classe a que as definições dizem respeito. Por exemplo:

```
void Data :: adicionar_anos(int _ano){
  ano += _ano;
};
```

5.1.1 Construtores

Os construtores, e como veremos mais à frente também os destrutores, são métodos especiais dentro de uma classe. Os construtores são responsáveis pela inicialização do estado de um objecto.

- O nome dos construtores é igual ao nome da classe.
- Os construtores sem argumentos, se não forem definidos de forma explícita pelo programador, são criados automaticamente. O seu comportamento, por omissão, será explicado mais abaixo.
- Os construtores não têm tipo de saída.
- Os construtores são, em muitos casos, polimórficos, aceitando um diferente número de argumentos.
- A sua invocação é feita aquando da declaração de uma instância da classe (um objecto).

```
// construtor sem argumentos (cria e inicializa com valores por omissão)
Data :: Data(){
  ano = mes = dia = 1; // não existe a data 0/0/0
};
// construtor com 3 argumentos (cria e inicializa)
Data :: Data(int _ano, int _mes, int _dia) {
  ano = _ano;
  mes = _mes;
  dia = _dia;
};
```

Estes métodos são invocados automaticamente aquando da instanciação da classe, isto é, aquando da criação dos objectos.

```
Data d1, d2(2013,3,12);
```

Em *C++*, a exemplo do que acontece em *C*, é possível declarar e inicializar de imediato uma variável (um objecto) de um dado tipo (de uma dada classe) através da imediata atribuição de um valor concreto.

```
int i = 3; // criação e inicialização em C
```

```
Data d1(2013,3,12),d2(d1); // d2, criação e inicialização em C++
```

O comportamento, por omissão, da cópia entre objectos, seja na inicialização aquando da declaração, seja na atribuição, é o de copiar todos os elementos que definem o estado do objecto.

Se no caso das estruturas de dados estáticas isso é perfeitamente apropriado, no caso das estruturas de dados dinâmicas, isto é que envolvam ponteiros, não é esse o comportamento que se deseja. Neste último caso acaba-se por copiar o ponteiro e não a informação apontada por ele.

No caso das estruturas de dados dinâmicas é então necessário implementar, explicitamente, os construtores por cópia assim como a atribuição por cópia.

Construtor por Cópia

Pode-se dar o caso em que se pretende criar um objecto inicializando-o com o valor de um outro objecto já existente. Isto é pretende-se criar um objecto por cópia do conteúdo de um objecto já existente.

O construtor por cópia é definido de forma a aceitar uma referência para um objecto como argumento.

Considerando-se que estamos a construir a classe `Vector`, ter-se-ia a seguinte definição para o construtor por cópia:

```
Vector(const Vector&);
```

Como argumento tem-se a referência («&») para o objecto e não o objecto, isto dado que se pretende copiar o conteúdo do objecto e não o ponteiro para o mesmo.

O qualificativo `const` é importante, dado que não se pretende modificar o objecto do qual se está a fazer a cópia.

Ter-se-ia então:

```
class Vector {
private:
  int tam;
  double *elems;
public:
  Vector(const Vector&);
  (...)
}
```

O construtor faz a inicialização do espaço para o novo elemento e de seguida copia os elementos.

```
Vector::Vector(const Vector& arg) {
  int i;
  tam = arg.tam;
  elems = new double[tam];
  for (i=0; i < tam; i++)
    elems[i] = arg.elems[i];
}
```

O efeito de um construtor por cópia é o mesmo da declaração e inicialização de uma variável de um dos tipos de dados pré-definidos, unificando desta forma os conceitos de variável de um tipo pré-definido e objecto, como instância de uma classe.

Ter-se-ia:

```
Vector v2=v1;
```

ou, de igual modo

```
Vector v2(v1);
```

Em ambos os casos faz a instanciação de um novo objecto da classe `Vector`, inicializando o seu valor com o valor, previamente definido, do objecto `v1`.

Para podermos considerar um objecto como mais uma variável de um tipo, neste caso uma classe por nós definida, falta-nos ter acesso ao operador de atribuição.

Atribuição por Cópia

O operador/instrução de atribuição entre objectos de uma dada classe é, a exemplo dos construtores, criado automaticamente para cada classe. A exemplo do que se viu acima é possível fazer a atribuição entre diferentes objectos de uma mesma classe, por exemplo:

```
v3 = v2;
```

A atribuição entre objectos de uma mesma classe é definida, por omissão, como sendo uma atribuição de todos os valores que constituem as estruturas de dados do objecto. No caso das estruturas dinâmicas, isto leva a que se copie o ponteiro e não os valores apontados por este último. Num caso como o dos vectores isso não só não é desejável, dado que os ponteiros passam a apontar para o mesmo espaço, sem que isso signifique a cópia dos valores de um objecto para o outro, como além disso leva a uma perda de memória que é sempre indesejável (ver Figura 5.1).

A «solução» para o problema criado pela implementação por omissão do operador de atribuição é semelhante a vista acima para o construtor por cópia. É necessário implementar

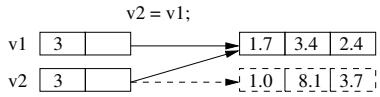


Figura 5.1: Atribuição (por omissão) entre dois Objectos da Classe `Vector`

uma atribuição entre objectos que faça a cópia dos conteúdos e não simplesmente dos ponteiros que definem os objectos.

```
class Vector {
private:
  int tam;
  double *elems;
public:
  Vector(const Vector&); // construtor por cópia
  Vector& operator=(const Vector&); // atribuição por cópia
  (...)
}
```

A implementação passa então por: apagar o espaço antigo; redefinir os valores do objecto; copiar o conteúdo do objecto que é o argumento da atribuição (≪`v1`≫); devolver o próprio elemento (já alterado) (ver Figura 5.2). É de notar que o ≪tamanho≫ do objecto é ajustado ao ≪tamanho≫ do objecto que é o argumento da atribuição.

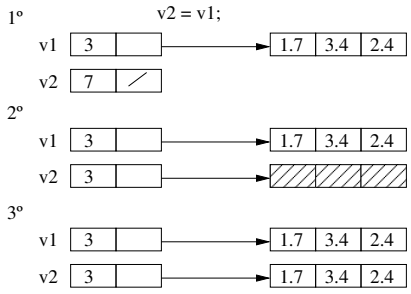


Figura 5.2: Atribuição por Cópia entre dois Objectos da Classe `Vector`

Ou seja ter-se-ia:

```
Vector& Vector::operator=(const Vector& arg) {
  int i;
  if (this != &arg) { // evitar a auto-atribuição arg=arg
    delete[] elem;            // liberta o espaço antigo
    tam = arg.tam;            // copia os elementos do objecto
    elems = new double[tam]; // cria o espaço para os novos elementos
    for (i=0; i<tam; i++)     // copia os elementos
      elems[i] = arg.elems[i];
  }
  return *this;
}
```

Nesta definição da atribuição por cópia recorre-se a duas características do *C++* que ainda não tinham sido abordadas.

`return *this;` o identificador `this` identifica o próprio objecto em que se está a trabalhar,

a sua utilização é aqui necessária dado que de forma explicita, é necessário ter como valor de retorno o próprio objecto.

operator= o identificador `operator` aplica-se a um dado símbolo de operador da linguagem *C++* e permite, deste modo, nomeá-lo e, por aplicação do polimorfismo próprio do *C++*, atribuir-lhe mais um significado, além daqueles que ele já tem. Neste caso permite que o operador «=» possa ser aplicado a objectos da classe `Vector`. Note-se que, dado que um símbolo por si só não constitui um identificador da linguagem *C++*, a prefixação do identificador `operator` é a única forma de podermos referir-nos ao operador de atribuição. Mais à frente falar-se-á mais sobre esta característica da linguagem *C++*.

Tendo definido o operador de atribuição por cópia podemos então usar o operador de atribuição:

```
v2 = v1;
```

Com a certeza de que os elementos do vector `v1` são copiados para `v2`, que os dois vectores são independentes um do outro e que não há nenhuma perda de memória.

Semântica Pré-definida dos Operadores

Como se viu acima tanto no caso da inicialização, como no caso da atribuição, a sua implementação é feita automaticamente sendo o seu significado, por omissão, o da cópia dos valores das estruturas de dados definidas nas classes.

Se não for esse o entendimento do programador podemos bloquear as implementações implícitas criando, de forma explícita, implementações para essas operações, as quais podem ser vazias de significado, e colocando-as na secção privada da classe, tornado-as desta forma existentes, mas inacessíveis.

Por exemplo:

```
class Vector {
private:
  int tam;
  double *elems;
  Vector(const&){}; // construtor por cópia
  Vector& operator=(const Vector&){}; // atribuição por cópia
public:
  (...)
}
```

neste caso a inicialização e a atribuição de objectos da classe `Vector` já não seria possível dado que são ambos métodos existentes, mas privados.

```
(...)
Vector v2(v1);  // erro, método privado
v2 = v1;        // erro, método privado
(...)
```

5.1.2 Destrutores

Á semelhança dos construtores os destrutores são métodos especiais na definição de uma classe. Os destrutores são responsáveis pela libertação do espaço ocupado por um objecto.

- O nome dos destrutores é igual ao nome da classe, prefixado pelo carácter ≪~≫.
- Os destrutores, se não forem definidos de forma explícita pelo programador, são criados automaticamente. O seu comportamento, por omissão, será explicado mais abaixo.
- Os destrutores não têm tipo de saída.
- A sua invocação é feita aquando da destruição de um dado objecto.

Por exemplo, o destrutor para a classe `Data` (secção 5.1)

```
Data :: ~Data(){};
```

Estes métodos são invocados automaticamente no instante em que o objecto deixa de pertencer ao estado presente do programa. Seja pelo terminar de uma função e a consequente destruição de todo o estado local a essa função, seja pelo terminar do programa.

Á semelhança dos construtores o comportamento por omissão de um destrutor é o de destruir todos os elementos (atributos) do objecto, libertando o espaço de memória ocupado pelos mesmos.

No caso de estruturas dinâmicas esse comportamento, por omissão, dos destrutores não é apropriado. Por exemplo o destrutor, por omissão, para a classe `Vector` teria como efeito (ver Figura 5.3) uma perda da ligação às posições de memória que estão ocupadas pelos elementos da estrutura. O espaço correspondente não seria libertado dando origem a ≪uma fuga de memória≫.

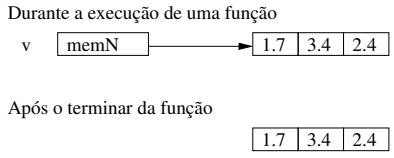


Figura 5.3: Efeito do destrutor (por omissão) para a Classe `Vector`

Para evitar os casos em que se tem uma ≪fuga de memória≫ é necessário implementar de forma explícita o destrutor.

```
Vector :: ~Vector(){
  delete[] elem;
};
```

5.2 Relações entre Classes

O *C++* implementa as relações de:

- herança (≪is a≫), simples e múltipla;
- agregação (≪part of≫);
- instanciação;
- meta-classe.

Vamos de seguida ver em detalhe as primeiras três destes tipos de relações.

5.2.1 Herança Simples

Vejamos através de um exemplo como o *C++* implementa a relação de herança simples. A classe de topo desta hierarquia designa-se por super-classe, as outras classes designam-se por classes derivadas ou sub-classes (ou sub-tipos).

Problema 1 (Herança Simples (exercício E.12.1)) *Pretende-se construir uma aplicação que possa servir para gerir as fichas pessoais dos utilizadores (de diferentes tipos) das residências universitárias de uma dada instituição universitária.*

Começa-se por construir modelizar o problema, isto é, criar um modelo, uma representação do problema, utilizando os diagramas UML para o efeito. O resultado pode ser aquele que é dado pela figura 5.4.

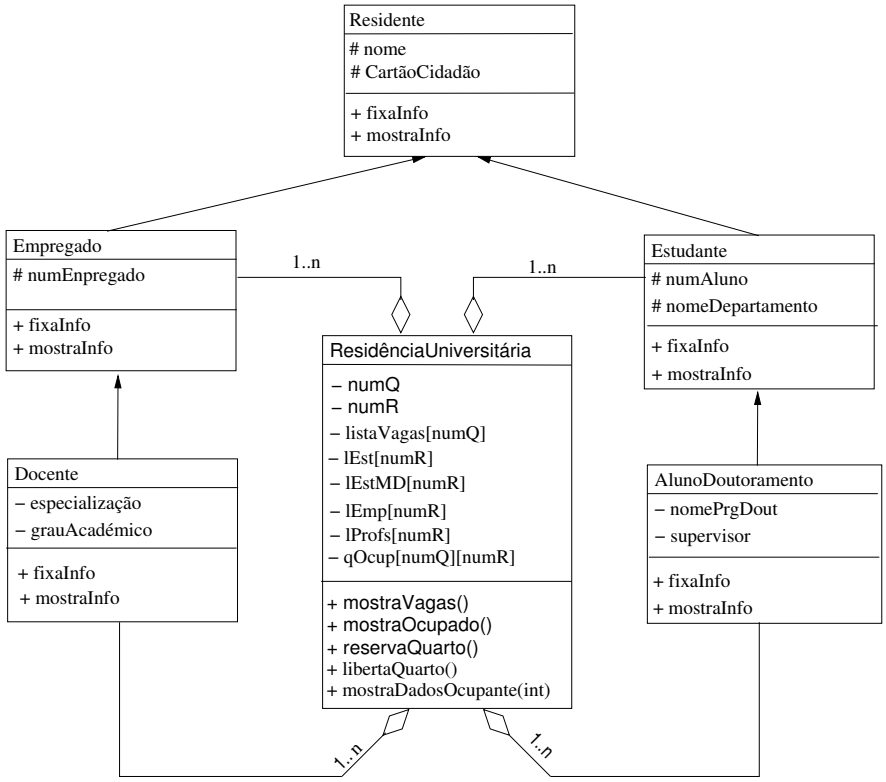


Figura 5.4: UML - Residência Universitária

Vejamos a solução para este problema por estágios: em primeiro lugar só considerando duas classes e sem utilização dos construtores; depois considerando os construtores; finalmente toda a estrutura.

Duas Classes, sem Utilização dos Construtores e num só Ficheiro

Temos então, duas classe `Residente` e `Empregado` e deixando os construtores por especificar, ou seja, os construtores serão os construtores definidos, por omissão, pelo próprio compilador de *C++*.

Ficheiro herancaSemConstrutores.cpp

```
// Classe Empregado derivada da classe de base Residente
// Sem usar construtores
#include <iostream>

#define MAXSTR 9

using namespace std;

class Residente {
public:
  void fixaInfo() {
    //Pre: Sem pré-condições
    //Post: Informação do Residente (nome e número do CC)
    cout << endl << "Forneça o nome (sem espaços): ";
    cin >> nome;
    cout << endl << "Forneça o número do cartão de cidadão: ";
    cin >> cc_num;
  }

  void mostraInfo() {
    //Pre: A informação do Residente já está definida
    //Post: A informação do Residente é visualizada

    cout << endl << "Nome: " << nome;
    cout << endl << "Cartão de Cidadão: " << cc_num;
  }
protected:
  string nome;            // Nome do Residente
  unsigned long cc_num;   // CC do Residente
};

class Empregado : public Residente {
  // Classe derivada Empregado, herança pública de Residente

public:
  void fixaInfo()  {
    //Pre:  Sem pré-condições
    //Post: Informação do Residente (nome e número do CC)
    Residente::fixaInfo();
    cout << "\nForneça o número do Empregado (5 dígitos): ";
    cin >> empr_num;
  }
  void mostraInfo() {
    //Pre: A informação do Empregado já está definida
    //Post: A informação do Empregado é visualizada
    Residente::mostraInfo();
    cout << "\nNúmero de Empregado: " << empr_num << endl;
  }
protected:
  unsigned long empr_num;
};

int main() {
  Empregado e1;

  cout << "Forneça a informação para o Residente (Empregado) 1 " ;
```

```
  e1.fixaInfo();
  cout << "\nDados do Empregado 1: \n";
  e1.mostraInfo();

  return 0;
}
```

Duas Classes e com Utilização dos Construtores

Continuando a considerar somente duas classes, passa-se agora a implementar os construtores. É importante ver que, não havendo em *C++* herança dos construtores, existe uma sintaxe própria que permite a utilização dos construtores da classe de base, aquando da construção do construtor da classe derivada.

Na classe `Empregado` o construtor da classe, e o construtor da classe de base estão ligados de forma a que toda a estrutura de dados possa ser criada aquando da instanciação da classe `Estudante` num dado objecto dessa classe.

```
Empregado() : Residente() {};
```

isto é, a sintaxe é:

<construtor_da_classe_derivada> : <construtor_da_classe_de_base>.

De seguida apresenta-se um excerto da listagem do programa (ver o anexo D, secção D.3.1 para a listagem completa):

```
// Exemplo de relação de herança com utilização dos construtores
#include <iostream>

using namespace std;

class Residente { // classe de base
public:
  Residente(){}; // construtor sem argumentos
  // construtor com dois argumentos
  Residente(string _nome, unsigned long _cc_num) {
    nome = _nome;
    cc_num = _cc_num;
  }
  (...)
};

// Empregado derivação pública, da classe Residente
class Empregado : public Residente {
public:
  Empregado() : Residente() {} // construtor sem argumentos
  Empregado(string _nome, unsigned long _cc_num, unsigned long _empr_num)
    : Residente(_nome, _cc_num) {
    // Construtor com três argumentos, dois deles são passados
    // directamente ao construtor da classe base
    empr_num = _empr_num; // Guarda o número de Empregado
  }
  (...)
};

int main() {
  Residente r1;
```

```
  Empregado e1; // construtor sem argumentos
  (...)
  Empregado e2(nome, numero, empr_num);  // construtor de três argumentos
  (...)
}
```

A Hierarquia Completa e com Utilização dos Construtores

Na implementação da hierarquia completa, isto é, as cinco classes tais como é modelizado em 5.4 e recorrendo aos construtores deve-se fazer, seguindo o diagrama UML, a divisão do programa em unidades funcionais, separando-as em vários ficheiros (ver Figura 5.5).

O código *C++* correspondente está no apêndice D, secção D.3.2.

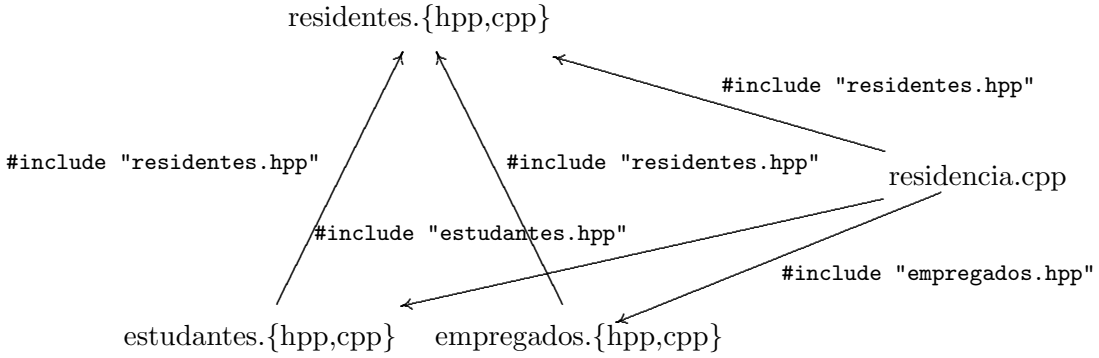


Figura 5.5: Hierarquia de Classes — Ficheiros e Dependências

5.2.2 Herança Múltipla

Vejamos através de um exemplo como o *C++* implementa a relação de herança múltipla.

Problema 2 (Herança Múltipla (exercício E.12.1)) *Um caso de herança múltipla acontece de forma «natural» numa família. Pretende-se construir uma aplicação que permita gerir a informação referente a uma dada família.*

Começando por construir um modelo do problema na forma de um diagrama UML que, de forma grosseira, representa esta situação (ver Figura 5.6).

Na listagem apresentada na secção D.3.3 do apêndice D é de notar que, na classe derivada tem-se acesso a à informação pública e protegida da classe de base, é então possível implementar um método que mostra informação respeitante à classe assim como às classes de que esta deriva. A situação contrária não é verdadeira, podemos dizer que a classe derivada «conhece» a classe da qual deriva, a classe de base não tem acesso às suas classes derivadas.

A utilização da informação/métodos da classe de base pelas classes derivadas fica bem expresso na construção dos construtores da classe `Filho`, utilização da informação da classe de base, assim como na implementação do método `mostraInfoFamilia` em que se utilizam os métodos `mostraInfo` das duas classes de base da qual esta deriva.

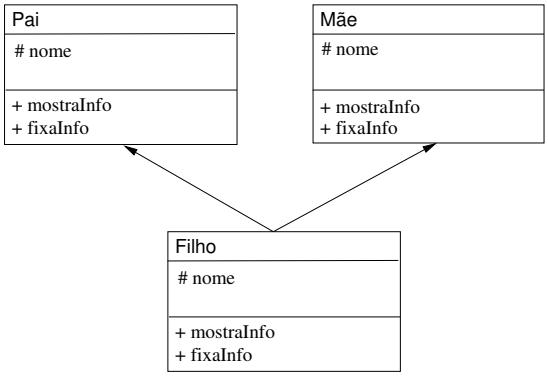


Figura 5.6: UML - Hierarquia Familiar

5.2.3 Relação «Parte de»

As relações de agregação entre classes implementam associações do tipo «parte de», isto é, um dado objecto irá ser parte de um todo, ou dito de outra forma irá existir uma classe que vai agregar em si várias classes componentes.

Relembrando o exemplo da construção de um sistema de barragens e da sua modelização através de um diagrama UML (ver Figura 2.20). Pretende-se construir um programa capaz de simular um sistema de múltiplos reservatórios de água ligados entre si de forma arbitrária (em série, em paralelo, ou ambos). A relação entre o sistema de barragens e, por exemplo, as barragens em si não se enquadra directamente nas situações de herança descritas anteriormente. Neste caso temos uma relação de agregação, uma barragem é *parte de* um sistema de barragens.

A relação de agregação é implementada em *C++* através da possibilidade da utilização dos objectos de uma dada classe por outra classe. Os referidos objectos farão parte da definição da nova classe. Ou seja tem-se que para, após definirem-se as classes `Barragem`, `FluxosSaida` e `FluxosEntrada`, as quais herdam de `Componente`, por exemplo:

```
// Classe Barragem
class Barragem : public Componente {
public:
  Barragem() : Componente(){}
  void setstorage(int);
  int getstorage();
private:
  int capacidadeArmazenamento;
};
```

define-se a classe `SistemaBarragens` da seguinte forma:

```
// Classe Sistema de Barragens
class SistemaBarragens : public Componente  {
public:
  SistemaBarragens() : Componente() {}
  void fixaInfoSistema();
private:
  FluxosEntrada fluxoEnt[MAXRES];   // agregação, 1 para n (MAXRES)
  FluxosSaida fluxoSd[MAXSAIDA];   // agregação, 1 para n (MAXSAIDA)
  Barragem barr[MAXRES];        // agregação, 1 para n (MAXRES)
  int matConexoes[MAXRES][MAXSAIDA]; // Matrix de conexões
```

```
  void conecta();
  void processa();
};
```

isto é, o sistema de barragens vai conter (eles são *parte de*) vectores de `FluxosEntrada`, de `Barragens`, assim como de `FluxosSaida`, implementando desta forma uma agregação de n para 1 entre as classes referidas e a classe `SistemaBarragens`.

É de notar que a classe `SistemaBarragens` tem uma herança simples com a classe `Componente`. Um tipo de relação entre classes não invalida os outros.

5.2.4 Relação de Instanciação

Aquando da definição de uma classe fala-se dos seus objectos como instâncias da classe. A relação de instanciação não se refere a este mecanismo entre a classe (objecto genérico) e uma sua instância (objecto particular) mas sim ao mecanismo, que o *C++* possuí, que permite que se possa especificar uma classe tendo uma outra classe como parâmetro. No momento da criação do objecto (instanciação da classe) o parâmetro (que é por sua vez uma classe) é também instanciado, dando origem a um objecto concreto.

Este tipo de mecanismo é muito importante na implementação dos tipos abstractos de dados. relembrando a definição da classe *Pilha* para o caso particular em que os elementos que esta pode conter são do tipo carácter.

```
class PilhaChar {
public:
  // construtor & destrutor
  PilhaChar(int sz = Tamanho_por_omissao);
  ~PilhaChar();
  // manipuladores
  void push(char);
  void pop();
  char top();
  int vazia();
private:
  int tamanho;
  static const int Tamanho_por_omissao = 10;
  char *topo, *pilha;
};
```

Esta definição peca por ser pouco genérica, teríamos que a duplicar para o caso de pilhas de inteiros, de novo para pilhas de palavras, etc. Seria importante poder definir pilhas de elementos, deixando para mais tarde a definição do que se entende por elementos. Esta relação entre classes é caracterizada pela relação de instanciação (ver Figura 5.7).

Em *C++* a implementação das relações de instanciação é feita através do mecanismo das classes escantilhão («template»), as quais permitem a definição da classe genérica por utilização desta classe escantilhão cuja instanciação é só feita aquando da criação de um dado objecto da classe que estamos a definir.

Continuando no exemplo da classe *Pilha*, com este mecanismo é já possível especificar uma pilha genérica.

```
template <class Elementos>
class Pilha {
public:
  // constructor
```

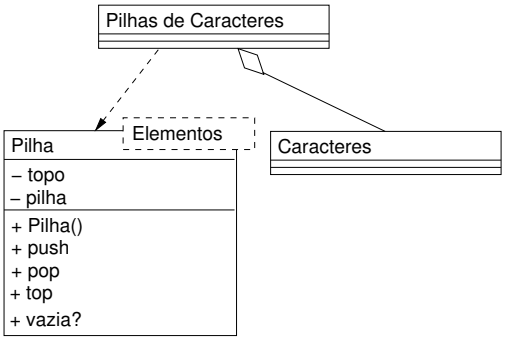


Figura 5.7: Pilha de Caracteres (Instanciação)

```
  Pilha(){
    ptPilha = NULL;
  }
  // destructor
  ~Pilha(){
    while (ptPilha!=NULL) {
     No *aux=ptPilha;
     ptPilha=ptPilha->prox;
     delete aux;
  }
  // push - coloca um elemento na pilha
  void push(Elementos elem){
    // inicializa o novo nó
    No *novo = new No;
    novo->elem=elem;
    novo->prox=ptPilha;
    // inserir à cabeça
    ptPilha = novo;
  }
  // pop - retira um elemento da pilha
  void pop(){
    No *aux=ptPilha;
    ptPilha=ptPilha->prox;
    delete aux;
  }
  // top - "vê" o elemento que está no topo da pilha
  Elementos top(){
    return ptPilha->elem;
  }
  // verifica se a pilha está vazia
  bool vazia() {
    return ptPilha == NULL;
  }
private:
  struct No {
    Elementos elem;
    No *prox;
  };
  No* ptPilha; // apontador para o primeiro elemento
  // define como vazios os constructores por cópia e por atribuição
  Pilha (const Pilha &){}
  void operator=(const Pilha &){}
```

```
};
```

Aquando da criação de objectos desta classe ter-se-á de concretizar, mas só nesse momento, o tipo de elementos que se pretende.

É de notar que neste caso particular não é possível separar a especificação (ficheiro `.hpp`) da implementação (ficheiro `.cpp`). A especificação e implementação da classe tem de ser feita num único ficheiro (e.g. `pilhasGenericas.cpp`).

```
#include "pilhasGenericas.cpp"
(...)
int main() {
  (...)
  Pilha<float> p1; // uma pilha de elementos do tipo float
  Pilha<int> p2;   // uma pilha de elementos do tipo int
  (...)
}
```

A utilização da relação de instanciação é muito importante para a definição de estruturas de elementos genéricos. A definição de funções genéricas, isto é, funções que se possam adaptar a diferentes tipos de elementos, é também possível em *C++*. Por exemplo, a construção de uma função de ordenação passível de ser aplicada aos diferentes tipos de listas. Esse tema será abordado na secção 7.5.

Capítulo 6

A Biblioteca Padrão do *C++*

Uma linguagem cujo objectivo seja a construção de programas grandes e complexos tem de, obrigatoriamente, suportar a construção modular de tais programas. Por outro existe um conjunto de funcionalidades que são básicas à maior parte dos programas não fazendo sentido re-implementá-las sempre que necessário.

Podemos definir as bibliotecas padrão como sendo um agregado de módulos que implementam as funcionalidades mais comummente usadas pelos programadores.

A separação entre a implementação da linguagem e a biblioteca padrão, «Standard Library», está já presente na linguagem *C* e, por maioria de razões está presente na linguagem *C++*.

A biblioteca padrão é algo que todo o implementador de um sistema *C++* tem de providenciar e em que todo o programador pode contar para a construção dos seus programas (Stroustrup, 1997, Capítulo 16).

A biblioteca padrão do *C++*[1] providência:

- suporte para características da linguagem tais como gestão de memória e informação de execução;
- informação acerca de detalhes da implementação da linguagem, tais como os limites das estruturas de dados pré-definidas;
- funções cuja implementação vai estar dependente do sistema computacional usado, tais como `memmove`;
- funcionalidades não pré-definidas na linguagem mas cuja implementação, independente do sistema computacional usado, é muito importante, por exemplo as funcionalidade de entrada/saída;
- uma base de trabalho para possíveis extensões, definindo convenções e suporte de base para a utilização das entradas e saídas com o mesmo estilo que para os tipos pré-definidos;
- uma base de trabalho para outras bibliotecas.

[1] `http://www.cplusplus.com/reference/`

A inclusão das funcionalidades da biblioteca padrão, assim como de outras bibliotecas que se queiram usar faz-se através do mecanismo de inclusão usual do *C/C++* `#include <nome_da_biblioteca>`, sendo que se tem de ter em conta as seguintes convenções:

- os `nome_da_biblioteca`, no caso da biblioteca padrão, não necessitam de ter nenhuma referência ao local (no sistema de ficheiros) aonde as mesmas se encontram. É da responsabilidade do construtor do sistema *C++* que se está a usar providenciar para que o compilador encontre as bibliotecas pretendidas de forma automática.
- as bibliotecas padrão do *C* têm o seu correspondente em *C++* através de bibliotecas cujo nome tem um «c» como prefixo ao nome da biblioteca *C* correspondente, por exemplo `cstdio`.
- outras bibliotecas que não a biblioteca padrão necessitam da referência explícita ao local aonde se encontram:
  - ou por indicação do nome/caminho completo, da mesma forma que se faz usualmente a inclusão dos ficheiros de especificação («header files»), por exemplo `#include ''Filas/filas.hpp''`;
  - ou utilizando as opções de compilação apropriadas (opções `-I` e `-L`).

6.1 Organização da Biblioteca Padrão

A biblioteca padrão define o espaço de nomes «`std`», o que implica que a utilização dos métodos aí definidos têm de ser prefixados com o qualificativo apropriado:

```
std::<nome_do_método>
```

ou então recorrendo a directiva «`using namespace`»:

```
using namespace std;
```

6.1.1 Utilidades Genéricas

Na secção de utilidades genéricas encontram-se, entre outras classes instanciáveis (ver Tabela 6.1), os métodos que nos permitem manipular o datas e as horas, assim como os vários métodos de afectação de memória para as estruturas de dados dinâmicas.

`<utility>`	operadores e pares
`<functional>`	objectos funcionais
`<memory>`	gestão de memória para estruturas de dados dinâmicas
`<ctime>`	data e horas, idêntico à linguagem *C*

Tabela 6.1: Utilidades Genéricas

6.1.2 Algoritmos

A secção dos algoritmos genéricos (ver Tabela 6.2) contém um grande número de métodos aplicáveis a sequências de elementos (Stroustrup, 1997, Capítulo 18).

`<algorithm>`	algoritmos genéricos
`<cstdlib>`	`bsearch()`, `qsort()`

Tabela 6.2: Algoritmos

6.1.3 Diagnósticos

A secção de diagnósticos (ver Tabela 6.3) refere-se aos métodos que permitem incluir o tratamento de erros e de situações de excepção num dado programa em *C++*.

`<exception>`	excepções
`<stdexcept>`	excepções padrão
`<cassert>`	macro «assert»
`<cerrno>`	tratamento de erros, idêntico à linguagem *C*

Tabela 6.3: Diagnósticos

6.1.4 Sequências de Caracteres («Strings»)

A secção das sequências de caracteres contém (ver Tabela 6.4) um conjunto de métodos que permitem afirmar que o *C++*, ao contrário do *C*, tem como tipo de base, o tipo «string». A classe `<string>` contém todo um conjunto de métodos que permitem a manipulação de sequências de caracteres sem que seja necessário considerar a estrutura interna das mesmas.

As classes referentes aos `wchar` («wide chars»), pretendem ser uma resposta às necessidades de codificações que vão além do ASCII, nomeadamente o suporte para línguas como o Português com os caracteres acentuados. No entanto o padrão da linguagem, padrão *ANSI/ISO C*, define a implementação estes caracteres como específica dos compiladores, como tal, a sua compatibilidade não está assegurada entre diferentes plataformas computacionais.

`<string>`	sequências de caracteres do tipo `T`
`<cctype>`	classificação de caracteres
`<cwctype>`	classificação de caracteres com mais de 8bits
`<cstring>`	*C*, sequências de caracteres
`<cwchar>`	*C*, funções para caracteres com mais de 8bits
`<cstdlib>`	*C*, funções para sequências de caracteres

Tabela 6.4: Sequências de Caracteres

6.1.5 Entradas/Saídas («Input/Output»)

Na secção das entradas e saídas (ver Tabela 6.5) tem-se acesso aos métodos usuais, tanto em *C* como em *C++*, para lidar com a manipulação dos fluxos de entrada e saída. Podemos separar os fluxos como vindo dos canais definidos por omissão (teclado/ecrã), `iostream`, de ficheiros `fstream`, e de sequências de caracteres `sstream`.

A classe `<iomanip>` dá-nos um conjunto de manipuladores para os canais de saída, os quais vão-nos permitir formatar as saídas. Entre eles destacam-se os descritos na tabela 6.6.

`<iosfwd>`	redireccionamentos
`<iostream>`	objectos e operações padrão de fluxo de dados
`<ios>`	classe de base para fluxos de dados
`<streambuf>`	memórias tampão de fluxos de dados
`<istream>`	fluxos de entrada
`<ostream>`	fluxos de saída
`<iomanip>`	manipuladores
`<sstream>`	fluxos de/para sequências de caracteres
`<cctype>`	classificação de caracteres
`<fstream>`	fluxos de/para ficheiros
`<cstdio>`	*C*, entrada e saída
`<cwchar>`	*C*, entrada e saída, para caracteres com mais de 8bits

Tabela 6.5: Entradas/Saídas

Manipulador	Descrição
`endl`	Muda de linha e ≪limpa≫ a memória tampão de escrita
`flush`	≪limpa≫ a memória tampão de escrita
`setw(`n`)`	fixa uma largura mínima (n posições) para a escrita
`left`	Justificação do texto à esquerda. Só significativo após a declaração `setw`.
`right`	Justificação do texto à direita. Só significativo após a declaração `setw`.
`setfill(`ch`)`	Faz o preenchimento da saída com o carácter ch, cujo valor por omissão é um espaço em branco. Só significativo após a declaração `setw`.
`setprecision(`n`)`	Fixa o número de casas decimais usados aquando da escrita de um número real.
`fixed`	Usa a notação de vírgula fixa (e.g. 34.56)
`scientific`	Usa a notação vírgula flutuante (e.g. 0.3456e+2)

Tabela 6.6: Manipuladores (Formatadores) das Escritas

Os manipuladores `endl` e `flush` têm um efeito imediato, os manipuladores `setw(`n`)`, `left` e `right` aplicam-se ao elemento que os imediatamente precede, todos os restantes manipuladores referidos na tabela 6.6 aplicam-se a todas as saídas subsequentes.

Vejamos um exemplo de aplicação de tais manipuladores:

```
#include <iostream>
#include <iomanip>

using namespace std;

int main() {
    const int um = 1;
    const float dezpi = 31.416;

    cout << "Exemplos de escrita formatada" << endl << endl;
    cout << setw(60) << right << "Um inteiro com preenchimento: "
```

```
        << setw(8) << setfill('0') << um << endl;
    cout << setw(60) << right << setfill('␣')
        << "Um␣real␣em␣notação␣fixa␣(6␣posições␣decimais):␣␣"
        << setprecision(6) << fixed << dezpi << endl;
    cout << setw(60) << right
        << "Um␣real␣em␣notação␣científica␣(6␣posições␣decimais):␣␣"
        << scientific << dezpi << endl;

    return 0;
}
```

Produz o seguinte resultado [2] :

```
Exemplos de escrita formatada

                      Um inteiro com preenchimento:  00000001
      Um real em notação fixa (6 posições decimais):  31.416000
Um real em notação científica (6 posições decimais):  3.141600e+01
```

6.1.6 Localização («Localization»)

Por internacionalização e localização (O'Donnel, 1994) entende-se a construção de programas que sejam adaptáveis às diferentes particularidades culturais dos diferentes povos (internacionalização) e a posterior adaptação dos programas a essas diferenças (localização).

Isto é, quando se pretende ter um dado programa disponível em diferentes países, o que quer dizer diferentes línguas e convenções culturais (línguas, forma de dizer as horas, dinheiro, etc.) tem-se duas opções:

1. Construir n versões diferentes do mesmo programa, cada uma adaptada à língua/cultura local.

2. Construir uma só versão, internacionalizável, do programa e ter n traduções do programa, uma para cada língua/cultura local.

A primeira das opções é um pesadelo de programação e posterior sincronização/manutenção. A segunda, embora acrescente algum peso à componente de programação, torna a localização do mesmo muito fácil e virtualmente independente de toda e qualquer modificação (novas versões) que se faça ao programa (O'Donnel, 1994).

As linguagens *C* e *C++* têm os mecanismos de base necessários para ajudar no processo de internacionalização (i18n) e posterior localização (l10n) (ver Tabela 6.7).

`<locale>`	informação de localização
`<clocale>`	*C*, informação de localização

Tabela 6.7: Localização

[2] Não inteiramente verdadeiro, no caso de se usar uma codificação *multi-byte*, por exemplo a codificação *UTF-8*, há uma descrepância entre a contagem do números de caracteres no caso do manipulador `setw` e depois a sua escrita. Esse facto leva a um desalinhamento das saídas cujo alinhamento relativo vai depender do número de caracteres acentuados em cada uma das sequências de caracteres.

Falta dizer que estas classes só providenciam os mecanismos de base. A construção de programas internacionalizáveis/localizáveis é feito recorrendo a outras bibliotecas, por exemplo à biblioteca `gettext`[3] (O'Donnel, 1994).

6.1.7 Funções Auxiliares (≪Language Support≫)

Na secção das funções auxiliares (ver Tabela 6.8) tem-se acesso a um conjunto de métodos auxiliares para a linguagem *C++*, tais como a gestão de memória, limites dos vários tipos numéricos implementados pelo compilador e sistema operativo que se está a usar (ver secção D.2), etc.

`<limits>`	limites numéricos
`<climits>`	*C*, limites numéricos
`<cfloat>`	*C*, limites numéricos para tipos vírgula flutuante
`<new>`	gestão dinâmica da memória
`<typeinfo>`	identificação de tipos aquando da execução
`<exception>`	tratamento de erros (e excepções)
`<cstddef>`	*C*, suporte para a linguagem
`<cstdarg>`	funções com uma lista de argumentos
`<csetjmp>`	*C*, manipulação de pilhas
`<cstlib>`	interrupção da execução
`<ctime>`	relógio (≪clock≫) do sistema
`<csignal>`	*C*, gestão de sinalização

Tabela 6.8: Funções Auxiliares

6.1.8 Estruturas e Algoritmos Numéricos

Através da secção de estruturas e algoritmos numéricos (ver Tabela 6.9) tem-se acesso a funções matemáticas, geradores de números pseudo-aleatórios, já presentes nos sistemas *C*, assim como tipos de dados tais como os complexos.

`<complex>`	números complexos
`<valarray>`	vectores numéricos
`<numeric>`	operações numéricas
`<cmath>`	*C*, funções matemáticas
`<cstdlib>`	*C*, gerador de números aleatórios

Tabela 6.9: Estruturas e Algoritmos Numéricos

6.1.9 Estruturas de Dados (≪Containers≫)

A biblioteca padrão implementa um conjunto de estruturas de dados genéricas (ver Tabela 6.10 e Secção E.15). Entre elas temos: vectores, listas duplamente ligadas, filas e filas invertíveis (isto é o início pode passar a ser o fim e vice-versa), pilhas e tabelas associativas e conjuntos.

[3] `http://www.gnu.org/software/gettext/`

As estruturas de dados associativas `multimap` e `multiset` estão contidas em `<map>` e `<set>` respectivamente.

`<vector>`	tabelas uni-dimensionais de elementos do tipo `T`
`<list>`	listas duplamente ligadas de elementos do tipo `T`
`<deque>`	filas invertíveis de elementos do tipo `T`
`<queue>`	filas de elementos do tipo `T`
`<stack>`	pilhas de elementos do tipo `T`
`<map>`	tabelas associativas de elementos do tipo `T`
`<set>`	conjuntos de elementos do tipo `T`
`<bitset>`	tabelas de valores lógicos

Tabela 6.10: Estruturas de Dados

6.1.10 Iteradores («Iterators»)

Os iteradores (ver Tabela 6.11 e Secção E.15) permitem definir um modo de acesso às estruturas de dados. Os iteradores permitem definir métodos para percorrer as referidas estruturas abstraindo da forma particular como as mesmas estão implementadas (Stroustrup, 1997, Secção 16.3.2).

`<iterator>`	iteradores e funcionalidades auxiliares para os mesmos

Tabela 6.11: Iteradores

6.2 *Standard Template Library* (STL)

Nesta secção da biblioteca padrão estão as classes que encaixam no princípio da programação genérica, isto é, poder programar sobre um determinado tipo de problemas e/ou estruturas de dados abstraindo as suas diferenças e o modo como estão implementadas.

Na STL estão definidos os *contentores*, abstracção de diferentes estruturas de dados, por exemplo listas, filas, vectores, etc., assim como os *iteradores*, abstracção da noção de ponteiro de forma a que se possa aceder aos elementos dos contentores sem que seja necessário lidar com os seus detalhes. Finalmente tem-se um conjunto de algoritmos que se aplicam de forma genérica aos contentores, por exemplo, a ordenação dos elementos de um contentor.

6.2.1 Contentores e Iteradores

Os contentores são o mecanismo que providência a abstracção dos diferentes modelos de dados como sequências de objectos.

Os iteradores estão intimamente ligados com os contentores providenciando um mecanismo de abstracção para os ponteiros. Os iteradores permitem manipular os elementos dos contentores, abstraindo os seus detalhes de implementação.

A construção de uma classe tem o duplo objectivo (ver Secção 2) de abstrair, contruíndo a classe dos objectos que partilham um dado comportamento e/ou estrutura e de ajudar à modularidade do programa, sonegando a informação da implementação, dando somente a informação que é necessária para a sua utilização.

Tendo estes objectivos presentes qual deve ser a forma de conceber e implementar uma classe como, por exemplo, uma lista?

Os detalhes da implementação da estrutura: estática, dinâmica, estrutura ligada, ou duplamente ligada, ponteiro para o início e/ou para o fim. Tudo isso deve ficar sonegado na secção privada da classe, abstraindo desse modo os detalhes da implementação.

Por outro lado o interface público deve conter um conjunto completo de operações, isto é, deve ser possível efectuar todo o tipo de manipulações através das operações disponibilizadas na secção pública da classe.

Mas, deve ser este conjunto de operações minimal? Isto é, devem as operações incluídas ser o menor conjunto completo de operações para a referida estrutura de dados?

A resposta é, do ponto de vista metodológico simples: sim o conjunto de operações disponibilizadas para uma dada estrutura de dados abstracta deve ser completo e minimal.

A resposta já não é tão simples quando encarada do ponto de vista da implementação de operações sobre as estruturas de dados, isto sempre que as questões sobre a eficiência da solução encontrada são importantes.

Isto é a sonegação da informação, tão importante para a modularização de um programa, esbarra contra a construção de operações, eficientes, sobre as estruturas de dados. Uma solução será desistir da minimalidade do conjunto de métodos e contruir todos os métodos que sejam necessários para a manipulação da estrutura de dados que se está a usar.

A solução da implementação dos chamados ≪interfaces gordos≫ (*fat interfaces*) não é apropriada quando se está a considerar a construção de uma biblioteca, isto é, de um conjunto de funcionalidades a disponibilizar a outros programadores. Por um lado é virtualmente impossível adivinhar todas as necessidades presentes e futuras que vão ser necessárias pelos diferentes utilizadores, por outro, em vez de um interface simples e ≪limpo≫ ter-se-ia um interface muito longo, complexo em que a principal dificuldade seria descobrir a funcionalidade pretendida e mesmo como a usar de entre todas as variantes e/ou opções que haveria de ter.

É procurando responder a esta questão de conciliação de um interface completo mas minimal com as questões da construção de algoritmos eficientes que a STL providência os contentores e os iteradores sobre os mesmos. Como foi dito acima os contentores constituem o mecanismo que providência a abstracção dos diferentes modelos estruturados, e os iteradores providenciam um mecanismo de abstracção para os ponteiros, permitindo o manipular dos elementos dos contentores, de forma eficiente.

Contentores

Os contentores são então estruturas homogéneas, isto é, definem estruturas de dados capazes de armazenar múltiplos elementos, todos do mesmo tipo.

A biblioteca padrão define dois tipos de contentores: sequências e a contentores associativos. As sequências definem estruturas lineares semelhantes a um vector. Os contentores associativos definem estruturas em que os elementos estão associados, daí o seu nome, a etiquetas (chaves), sendo que o acesso aos elementos é feito através das etiquetas.

Os contentores são estruturas do mesmo tipo, com métodos de acesso do mesmo tipo e que, dentro de certos limites, podem ser trocados entre si num dado programa sem que isso afecte o programa.

Os contentores sequenciais são: `vector`; `list`; `deque`. Os contentores `queue`; `stack`; `priority_queue` são implementados à custa dos primeiros.

Os contentores associativos são: `map`; `multimap`. Os contentores `set` e `multiset` são contentores associativos em que não se associa nenhum elemento às etiquetas, as etiquetas são os elementos.

Como foi dito acima os contentores estão associados à noção de programação genérica, isto é, muitos dos métodos implementados são genéricos para todos os contentores, como tal um programador pode optar (ver Figura 6.1) por um dado tipo específico de contentor no início do programa e, mais tarde, trocar por outro tipo, sem que isso afecte o programa.

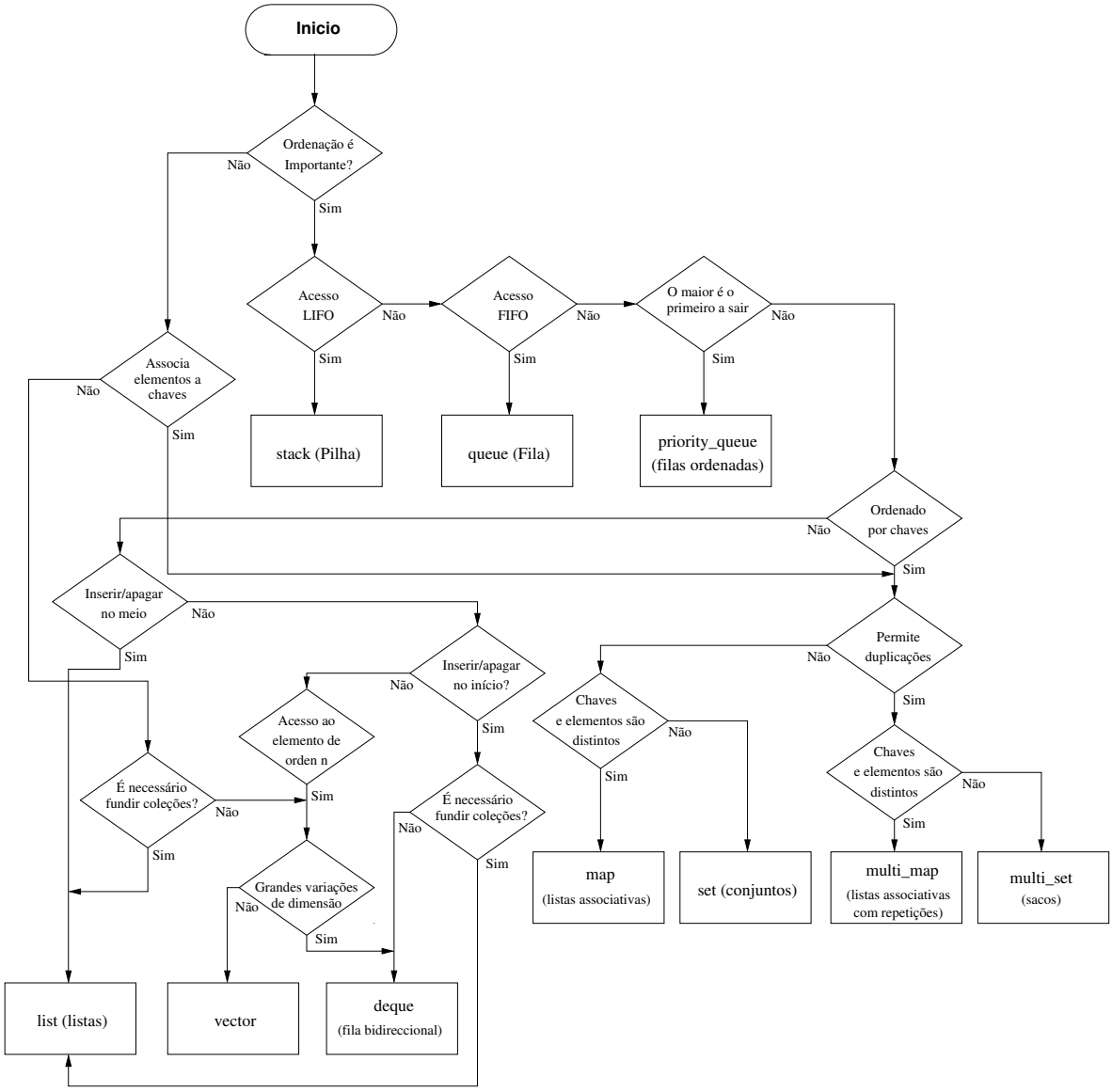


Figura 6.1: Escolha do Tipo do Contentor[4]

Nas tabelas 6.12—6.19 estão alguns dos métodos acessíveis nas diferentes classes que implementam os contentores (para mais detalhes ver (cplusplus.com, 2019; Stroustrup, 1997)).

[4] `http://www.linuxsoftware.co.nz/cppcontainers.html`
[5] `http://www.cplusplus.com/reference/stl/`

Tipo dos Elementos	
`value_type`	tipo dos elementos
`alocator_type`	tipo do gestor de memória
`size_type`	tipo dos índices, contagem elementos, etc.

Tabela 6.12: Tipos dos Elementos

Acesso aos Elementos	
`front()`	primeiro elemento
`back()`	último elemento
`[]`	acesso através de índice (sem verificação) — não disponível em `list`
`at()`	acesso através de índice (com verificação) — `vector` e `deque`

Tabela 6.13: Acesso aos Elementos

Pilhas e Filas	
`push_back()`	acrescentar um elemento (no fim)
`pop_back()`	remover o último elemento
`push_front()`	adicionar um elemento no início — `list` e `deque`
`pop_front()`	remover um elemento no início — `list` e `deque`

Tabela 6.14: Pilhas e Filas

Listas	
`insert(p,x)`	adicionar x antes da posição p
`insert(p,n,x)`	adicionar n cópias de x, antes da posição p
`insert(p,first,last)`	adicionar elementos de $[first, last[$ antes da posição p
`erase(p)`	remover o elemento na posição p
`erase(first,last)`	remover $[first, last[$
`clear()`	remover todos os elementos

Tabela 6.15: Listas

Outras Operações	
`size()`	número de elementos
`empty()`	verificar se o contentor está vazio
`max_size()`	dimensão máxima de um dado contentor
`swap()`	troca de elementos entre dois contentores
`==`	verifica se o conteúdo de dois contentores é igual
`!=`	verifica se o conteúdo de dois contentores é diferente
`<`	verifica se um contentor é lexicograficamente menor do que outro

Tabela 6.16: Outras Operações

Construtores	
container()	contentor vazio
container(n)	n cópias do valor por omissão, não disponível para contentores associativos
container(n,x)	n cópias de x, não disponível para contentores associativos
container(first,last)	elementos iniciais de $[first, last[$
container(x)	construtor por cópia
~container()	destrutor

Tabela 6.17: Construtores

Atribuição	
`operator=(`x`)`	atribuição por cópia dos valores do contentor x
`assign(`n`,`x`)`	atribue n cópias de x, não disponível para contentores associativos
`assign(`*first*`,`*last*`)`	atribuir de $[first, last[$

Tabela 6.18: Atribuição

Operações em Contentores Associativos	
`operator[](`k`)`	aceder ao elemento com etiqueta k (etiquetas únicas)
`find(`k`)`	achar o elemento com etiqueta k
`lower_bound(`k`)`	achar o primeiro elemento que tenha etiqueta k
`upper_bound(`k`)`	achar o primeiro elemento com etiqueta superior a k
`equal_range(`k`)`	achar os dois referidos nas linhas anteriores
`key_comp()`	compara e copia o valor da etiqueta
`value_comp()`	compara e copia o valor associado

Tabela 6.19: Operações Associativas

		Contentores Sequênciais			Contentores Associativos				
Cabeçalhos		<vector>	<deque>	<list>	<set>		<map>		<bitset>
Contentores	complex	vector	deque	list	set	multiset	map	multimap	bitset
iteradores	constructor	*	vector	constructor	constructor	constructor	constructor	constructor	constructor
	destructor	O(n)	~vector	destructor	destructor	destructor	destructor	destructor	
	operator=	O(n)	operator=	operator=	operator=	operator=	operator=	operator=	operators
	begin	O(1)	begin	begin	begin	begin	begin	begin	
	end	O(1)	end	end	end	end	end	end	
	rbegin	O(1)	rbegin	rbegin	rbegin	rbegin	rbegin	rbegin	
	rend	O(1)	rend	rend	rend	rend	rend	rend	
capacidade	size	*	size	size	size	size	size	size	size
	max_size	*	max_size	max_size	max_size	max_size	max_size	max_size	
	empty	O(1)	empty	empty	empty	empty	empty	empty	
	resize	O(n)	resize	resize					
acesso aos elementos	front	O(1)	front	front					
	back	O(1)	back	back					
	operator[]	*	operator[]				operator[]		operator[]
	at	O(1)	at						
modificadores	assign	O(n)	assign	assign					
	insert	*	insert	insert	insert	insert	insert	insert	
	erase	*	erase	erase	erase	erase	erase	erase	
	swap	O(1)	swap	swap	swap	swap	swap	swap	
	clear	O(n)	clear	clear	clear	clear	clear	clear	
	push_front	O(1)	push_front	push_front					
	pop_front	O(1)	pop_front	pop_front					
	push_back	O(1)	push_back	push_back					
	pop_back	O(1)	pop_back	pop_back					
observadores	key_comp	O(1)			key_comp	key_comp	key_comp	key_comp	
	value_comp	O(1)			value_comp	value_comp	value_comp	value_comp	
operations	find	O(log n)			find	find	find	find	
	count	O(log n)			count	count	count	count	count
	lower_bound	O(log n)			lower_bound	lower_bound	lower_bound	lower_bound	
	upper_bound	O(log n)			upper_bound	upper_bound	upper_bound	upper_bound	
	equal_range	O(log n)			equal_range	equal_range	equal_range	equal_range	
métodos únicos	capacity reserve			splice remove remove_if unique merge sort reverse					set reset flip to_ulong to_string test any none

Complexidade: $O(1)$, constante $< O(logn)$, logarítmica $< O(n)$, linear; $*$ = depende do contentor

Tabela 6.20: Métodos dos Contentores[5]

			Contentores Secundários		
Cabeçalhos			`<stack>`	`<queue>`	
Contentores			stack	queue	priority_queue
	constructor	*	constructor	constructor	constructor
capacidade	size	O(1)	size	size	size
	empty	O(1)	empty	empty	empty
acesso aos elementos	front	O(1)		front	
	back	O(1)		back	
	top	O(1)	top		top
modificadores	push	O(1)	push	push	push
	pop	O(1)	pop	pop	pop

Tabela 6.21: Métodos dos Contentores Secundários[6]

Os contentores vão poder ser precorridos através dos iteradores a eles associados. Temos que os diferentes contentores suportam diferentes disciplinas de acesso aos seus elementos:

`list, map, set` suportam iteradores bidireccionais;

`vector, deque` suportam iteradores de acesso aleatório.

Os contentores `stack` e `queue` não são implementações básicas, isto é, estes contentores são versões especiais dos contentores básicos. Tanto um como o outro podem ser implementados usando diferentes tipos de contentores básicos, em ambos os casos, por omissão, o contentor `deque` é usado:

`queue` podem ser implementados através dos contentores `deque` ou `list`:

```
template < class T, class Container = deque<T> > class queue;
```

`stack` podem ser implementados através dos contentores `vector`, `deque` ou `list`:

```
template < class T, class Container = deque<T> > class stack;
```

Iteradores

Iteradores providenciam uma visão abstracta das estruturas de dados de forma a que seja possível aceder às estruturas sem ter que lidar com a multiplicidade de detalhes presentes nas diferentes implementações.

Ao providenciarem um acesso (abstracto) às estruturas de dados vão permitir que se possam implementar, métodos eficientes,

Os iteradores são uma abstracção dos ponteiros, como tal vão ter um modo de operação em tudo semelhante as estes. Isto sem que se seja necessário conhecer os detalhes da implementação.

Temos então:

- `*` e `->` — o elemento que está a ser apontado;

[6] `http://www.cplusplus.com/reference/stl/`

Iteradores	
`begin()`	aponta para o primeiro elemento
`end()`	aponta para o elemento logo após o último elemento
`rbegin`	aponta para o primeiro elemento na sequência inversa
`rend()`	aponta para o elemento logo após o último elemento na sequência inversa

Tabela 6.22: STL Iteradores

- `++` — aponta para o próximo elemento (incremento);
- `--` — aponta para o elemento anterior (decremento), quando a estrutura é bidireccional.
- `==` — igualdade.

Operações com Iteradores e Categorias					
Categoria	saída	entrada	unidireccionais	bidireccionais	acesso aleatório
Abreviatura	Out	In	For	Bi	Ran
Leitura		`=*p`	`=*p`	`=*p`	`=*p`
Acesso		`->`	`->`	`->`	`->`
Escrita	`*p=`		`*p=`	`*p=`	`*p=`
Iteração	`++`	`++`	`++`	`++ --`	`++ -- + - += -=`
Comparação		`== !=`	`== !=`	`== !=`	`== != < > >= <=`

Tabela 6.23: Operações com Iteradores e Categorias

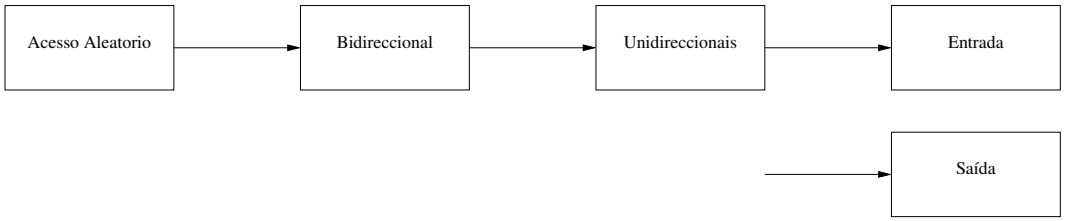


Figura 6.2: Categorias dos Iteradores

ExemplosSTL De seguida apresentam-se dois exemplos (yolinux.com, 2012) de utilização dos contentores, nos casos apresentados um `vector`, e dos iteradores sobre eles.

Exemplo de Utilização do Contentor `vector`. Um exemplo simples da utilização do contentor `vector` para o armazenar de «frases» (*strings*).

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;
```

```
main() {
  vector<string> vs;

  vs.push_back("O número é 10");
  vs.push_back("O número é 20");
  vs.push_back("O número é 30");

  cout << "Ciclo por índices:" << endl;

  int ii;
  for(ii=0; ii < vs.size(); ii++) {
    cout << vs[ii] << endl;
  }

  cout << endl << "Iterador constante:" << endl;

  vector<string>::const_iterator cii;
  for(cii=vs.begin(); cii!=vs.end(); cii++) {
    cout << *cii << endl;
  }

  cout << endl << "Iterator inverso:" << endl;

  vector<string>::reverse_iterator rii;
  for(rii=vs.rbegin(); rii!=vs.rend(); ++rii) {
    cout << *rii << endl;
  }

  cout << endl << "Saídas de Exemplo:" << endl;

  cout << vs.size() << endl;
  cout << vs[2] << endl;

  swap(vs[0], vs[2]);
  cout << vs[2] << endl;
}
```

Exemplo de Iteradores num Vector Bidimensional. Um exemplo da utilização de iteradores para o percorrer de uma estrutura bidimensional.

```
#include <iostream>
#include <vector>

using namespace std;

main() {
    vector< vector<int> > vI2Matrix;     // Matriz bidimensional
    vector<int> a, b;
    vector< vector<int> >::iterator iter_ii;
    vector<int>::iterator iter_jj;

    a.push_back(10);
    a.push_back(20);
    a.push_back(30);
    b.push_back(100);
```

```
    b.push_back(200);
    b.push_back(300);

    vI2Matrix.push_back(a);
    vI2Matrix.push_back(b);

    cout << endl << "Utilizando Iteradores:" << endl;

    for(iter_ii=vI2Matrix.begin(); iter_ii!=vI2Matrix.end(); iter_ii++) {
        for(iter_jj=(*iter_ii).begin(); iter_jj!=(*iter_ii).end(); iter_jj++) {
            cout << *iter_jj << endl;
        }
    }
}
```

Afectadores

Os afectadores (afectar/alocar memória), são o mecanismo de abstracção entre o modelo de baixo-nível dos dados como vectores de *bytes*, na visão de alto-nível dos objectos com a sua estrutura própria (ver (Stroustrup, 1997, Capítulo 19)).

Capítulo 7

Tópicos Diversos (em *C++*)

Nesta capítulo vão-se apresentar alguns pontos desconexos e sem que a ordem de apresentação seja indicativa da sua importância relativa. O ordenamento dos assuntos respeita eventuais precedências entre assuntos.

7.1 Sobrecarga dos Identificadores

Ao definir classes como implementações de novos tipos, com os seus elementos e as suas operações internas, o *C++* levanta um problema. Será possível usar a notação infixa usual para os novos tipos?

Isto é, estamos muito habituados a usar uma notação para a escrita das expressões, aritméticas e outras, usando operadores infixos, em vez de funções e seus argumentos.

Por exemplo, é usual escrever:

```
x+y*z;
```

como forma simplificada de

```
soma(x,multiplicacao(y,z));
```

É tão usual que só a primeira forma nos aparece como "natural", parecendo a segunda algo como o sobre-especificação daquilo que se pretende.

No entanto ao especificar uma nova classe, através da especificação das suas estruturas de dados e das suas funções membro, só se tem acesso ao segundo tipo de notação.

Por exemplo, se se pretender construir a classe dos números complexos ter-se-ia de escrever algo como:

```
class Complexos {
private:
  doube re,im;
public:
  Complexos(double, double); // construtor
  Complexos adicao(Complexos); // adição
  Complexos multiplicacao(Complexos); // multiplicação
```

a sua utilização seria algo como:

```
  (...)
  Complexos a = Complexos(1,3.1);
```

```
Complexos b = Complexos(1.2,3);
Complexos c,d;

c = a.adicao(b);
d = c.multiplicacao(b);
(...)
```

7.1.1 Operadores como Funções

Na tabela 7.1 encontram-se todos os símbolos de operadores do *C++* para os quais é possível: utilizar na sua forma funcional através da palavra reservada `operator` fazer a sobrecarga do seu significado, acrescentando desse modo uma nova funcionalidade aquelas que ele já tem.

+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	–	->*	,
->	[ ]	( )	new	new [ ]	delete	delete [ ]

Tabela 7.1: Operadores como Funções

Para todos estes operadores a sua definição, ou melhor, a sua redefinição com acréscimo de mais uma funcionalidade, é possível.

Podemos deste modo definir uma classe, por exemplo, `Complexos` (ver Secção E.11.2) em que através da (re)definição dos operadores aritméticos usuais, assim como dos operadores de leitura e escrita, obtemos um novo tipo sendo que a sua utilização em tudo se assemelha aos tipos pré-definidos.

```
class Complexos {
private:
  double pre,pim;
public:
  // Construtores
  Complexos(double, double);
  ~Complexos();
  // Selectores
  double real();
  double imag();
  // Operadores Aritméticos
  friend Complexos operator+(Complexos);
  friend Complexos operator-(Complexos);
  friend Complexos operator*(Complexos);
  friend Complexos operator/(Complexos);
  // Igualdade
  friend bool operator==(Complexos,Complexos);
  friend bool operator!=(Complexos,Complexos);
  // Leitura e Escrita
  friend istream& operator>>(istream&, Complexos&);
  friend ostream& operator<<(ostream&, Complexos&);
};
```

Declaradas desta forma (a implementação será feita como funções livres) torna-se necessário declarar estas funções (a versão funcional dos operadores) como funções amigas (`friend`) isto porque se trata de acrescentar significado a funções que são exteriores à classe e que por isso, não têm acesso à secção privada da mesma.

Vejamos a implementação dos operadores `>>` e `<<` (nos outros casos é trivial).

```
// escrita, como x+iy
ostream& operator<<(ostream& oscmplx, const Complexos& a) {
  return(oscmplx << a.pre << "+i" << a.pim << endl);
}
// leitura, como, x +i y
istream& operator>>(istream& iscmplx, Complexos& a) {
  string unidImag;
  return(iscmplx >> a.pre >> unidImag >> a.pim);
}
```

como se pode ver, ao implementar uma nova funcionalidade para estes operadores, passamos a poder ler e escrever os novos elementos de forma em tudo semelhante aos tipos pré-definidos.

7.2 Ficheiros

Os programas manipulam informação. A partir de dados, informação de entrada, produzem resultados, informação de saída. Sempre que haja necessidade de preservar os resultados, ou que os dados de entrada e/ou de saída sejam em número demasiado elevado para um manuseamento imediato, é necessário recorrer aos ficheiros.

Os ficheiros são entidades exteriores aos programas, geridos pelo sistema operativo, e que guardam de forma persistente a informação.

Existem dois tipos de ficheiros: ficheiros de texto e ficheiros binários (i.e. não de texto). De uma forma geral vamos estar interessados em lidar com ficheiros de texto. Para esse caso, e como veremos a seguir, a sua utilização a partir de um programa em *C++* pouco difere da utilização habitual das entradas (via teclado) e saídas (para o ecrã). É quase que um simples redireccionar dos fluxos de informação.

7.2.1 Manipular Ficheiros

Assim como os canais de comunicação «normais» (teclado e ecrã) os ficheiros podem ser de leitura ou de escrita. Para manipular um ficheiro é necessário:

Nome do Ficheiro: nome (identificador) que identifica o ficheiro perante o sistema operativo:

- Leitura — é necessário saber o nome exacto do ficheiro já existente.
- Escrita — é necessário decidir um nome para o ficheiro que vai ser criado, ou no caso de se pretender adicionar informação a um ficheiro já existente, o nome exacto desse ficheiro.

«**Abrir**» **o ficheiro** isto é associar o canal de comunicação (interno ao programa) com o ficheiro (entidade externa ao programa).

- Podemos «abrir» para leitura (desde o início).

- Podemos «abrir» para escrita/criação ou para escrita/adicionar (ao fim).

Ler/Escrever no caso de um ficheiro de texto o procedimento de leitura ou escrita é em tudo idêntico aos procedimentos «normais». No caso de ficheiro binários é necessário respeitar o formato preciso do ficheiro, mais à frente ver-se-á como.

«**Fechar**» **o ficheiro** desassociar a entidade externa do canal de comunicação. Este procedimento é feito automaticamente sempre que a variável associada ao canal de comunicação é destruída, e.g. fim do programa, fim de um dado método aonde esta foi declarada.

As classes associadas com os ficheiros são (ver Figura 7.1) (cplusplus.com, 2019):

- `fstream` — classe que herda de `iostream`, refere-se aos ficheiros de entrada (*input*) e de saída (*output*);
- `ifstrem` — classe que herda de `istrem`, refere-se ficheiros de entrada (*input*);
- `ofstream` — classe que herda de `ostream`, refere-se ficheiros de saída (*output*).

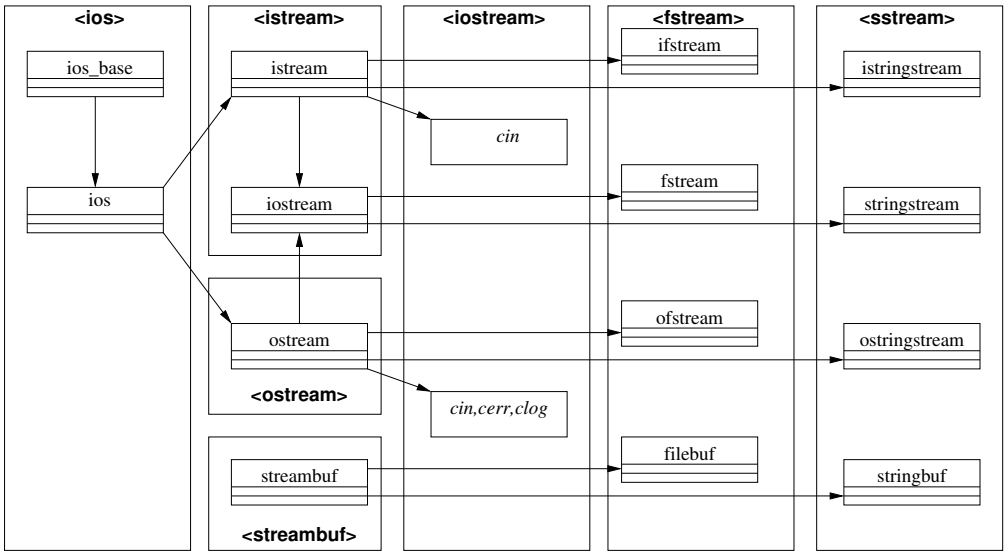


Figura 7.1: Hierarquia de Fluxos de Entrada e Saída

A abertura de um canal para leitura ou escrita e a sua associação a um dado ficheiro é feita através dos construtores apropriados (ou através do método `open`), especificando o nome do ficheiro assim como o modo de abertura. Se o ficheiro não estiver no directório em que o programa está a executar é necessário especificar o «caminho» completo. Por exemplo:

```
ifstream fentrada("dados.txt", ifstream::in);
ofstream fsaida("resultados.txt",ofstream::out);
ofstream fadicionar("maisresultados.txt",ofstream::app);
```

A sintaxe dos construtores é:

```
ifstream <nome_canal_entrada> (<nome_do_ficheiro_leitura>[, <modo>])
ofstream <nome_canal_saída> (<nome_do_ficheiro_escrita>[, <modo>])
```

Notas:

- o nome do ficheiro tem de ser uma sequência de caracteres do tipo `char*`, isto é, uma *string em C* e não um elemento do tipo `string`, isto é, uma *string em* C++. Para tal pode ser necessário o método de conversão `c_str()` que faz a conversão.
- é necessário usar a biblioteca `cstring`, para se ter acesso às *strings* em C.
- os modos mais usuais são: `ifstream::in`, abrir para leitura; `ofstream::out`, abrir para escrita, criando um novo ficheiro ou apagando e escrevendo num ficheiro já existente; `ofstream::app`, abrir para escrita adicionado a nova informação ao fim do ficheiro já existente, ou criando um novo se o ficheiro ainda não existir.

Os valores por omissão dos modos de abertura são `ifstream::in` para os canais de leitura e `ofstream::out` para os canais de escrita.

«Abrir» um Ficheiro para Leitura

Para abrir um ficheiro para leitura temos então de ter algo como:

```
#include <cstring>   // C strings
#include <string>    // C++ strings
#include <fstream>   // fluxos e operações com ficheiros
#include <iostream>  // cin, cout, cerr, etc.

using namespace std;

int main() {
  string nomeFicheiro;
  cout << "Introduza um nome de ficheiro (ASCII, sem espaços): ";
  cin >> nomeFicheiro;
  ifstream fent(nomeFicheiro.c_str());
  if (!fent)
    cout << "\nNão é possível abrir o ficheiro "
         << nomeFicheiro << " para leitura\n\n";
  (...)
}
```

A instrução `ifstream fent(nomeFicheiro.c_str());` utiliza o constructor `ifstream` para declarar um objecto desse tipo, associando um identificador (no programa) com o nome de ficheiro (entidade exterior ao programa). O método `c_str()` é necessário para fazer a conversão de «strings» *C++* para «strings» *C* o que é necessário para a ligação com o sistema operativo.

Os operadores habituais para canais de leitura (nomeadamente o operador `>>` estão disponíveis para o novo objecto do tipo `ifstream`, isto é, para ler do ficheiro, por exemplo dois inteiros, bastar-nos-ia:

```
int i,j;
fent >> i >> j;
```

«Abrir» um Ficheiro para Escrita

O «abrir» para escrita é semelhante ao «abrir» para leitura. Em vez de se criar um objecto do tipo `ifstream`, vai-se criar um objecto do tipo `ofstream`. No caso de se tratar de uma

leitura o ficheiro tem de existir previamente, no caso da escrita existem dois modos: a escrita desde o início; a escrita a partir do fim. No primeiro caso o ficheiro é criado, se não existir, ou no caso em que já existe o seu conteúdo é completamente substituído pelo novo conteúdo. No último caso o conteúdo existente no ficheiro (se ele existir) mantém-se sendo o novo conteúdo acrescentado ao fim do ficheiro.

Temos então algo como (escrita desde o início):

```
#include <cstring>   // C strings
#include <string>    // C++ strings
#include <fstream>   // fluxos e operações com ficheiros
#include <iostream>  // cin, cout, cerr, etc.

using namespace std;

int main() {
  string nomeFicheiro;
  cout << "Introduza um nome de ficheiro (ASCII, sem espaços): ";
  cin >> nomeFicheiro;
  ofstream fsaid(nomeFicheiro.c_str());
  if (!fsaid)
    cout << "\nNão é possível abrir o ficheiro "
         << nomeFicheiro << " para escrita\n\n";
  (...)
}
```

Assim como na leitura, a escrita procede-se da forma habitual através do fluxo de saída criado pelo novo objecto do tipo `ofstream`. Por exemplo:

```
fsaid << i << j;
```

Problemas na Abertura de Ficheiros

Aquando da abertura de um ficheiro podem ocorrer as seguintes situações de erro.

Abertura para leitura: erro caso em que:

- o ficheiro não existe;
- o ficheiro existe, mas está protegido contra leituras.

Abertura para escrita: erro no caso em que:

- não haja espaço em disco;
- o directório aonde se pretende efectuar a escrita está protegido;
- o ficheiro existe e está protegido contra escritas.

Durante a leitura: erro no caso em que:

- se tenta ler após o fim do ficheiro;
- tentar ler num ficheiro aberto para escrita.

Durante a escrita: erro no caso em que:

- se esgota o espaço disponível em disco;

- tentar escrever num ficheiro aberto para leitura.

Pode-se ler e escrever num mesmo ficheiro, no entanto não simultaneamente, isto é, é necessário «fechar» o ficheiro e voltar a abrir no novo modo. O fechar de um ficheiro faz-se através do método `fclose()` aplicado ao objecto do tipo `ifstream` ou `ofstream`.

Vejamos um caso simples em que se vai escrever uma sequência de 10 inteiros num ficheiro, e de seguida (como forma de confirmação) vai-se ler essa mesma informação enviando o resultado para o ecrã.

```
#include <cstring>   // C strings
#include <string>    // C++ strings
#include <fstream>   // fluxos e operações com ficheiros
#include <iostream>  // cin, cout, cerr, etc.

using namespace std;

int main() {
  int i;
  string nomeF;
  // definir o nome do ficheiro de saída/entrada
  cout << "Introduza o nome de ficheiro (ASCII, sem espaços): ";
  cin >> nomeF;
  ofstream fsaid(nomeF.c_str());
  // verificar se não há erros associados à abertura do ficheiro
  if (!fsaid) {
    cout << "\nNão é possível abrir o ficheiro " << nomeF << " para escrita\n";
    return(1);
  }
  // escreve no ficheiro de saída
  for (i=1;i<=100;i++)
    fsaid << i << endl;
  // fecha o
  fsaid.close();
  // volta a abrir mas como ficheiro de entrada
  ifstream fent(nomeF.c_str());
  if (!fent) {
    cout << "\nNão é possível abrir o ficheiro " << nomeF << " para leitura\n";
    return(2);
  }
  // lê do ficheiro e escreve no ecrã (cout)
  while (!fent.eof()) {
    fent >> i;
    cout << i << "\t";
  }
  cout << endl;
  return(0);
}
```

Como podemos ver neste exemplo o fechar de um ficheiro é feito através da aplicação do método apropriado ao canal de comunicação `fsaid.close();`. O método `eof()` permite-nos saber quando, em fase de leitura, se chegou ao fim do ficheiro.

7.3 Espaço de Nomes

Um espaço de nomes ("*namespace*" em Inglês) é um delimitador abstracto que fornece um contexto para os itens que ele armazena (nomes, termos técnicos, conceitos, etc), o que permite uma desambiguação para itens que possuem o mesmo nome mas que residem em espaços de nomes diferentes. Como um contexto distinto é fornecido para cada espaço delimitado, o significado de um nome pode variar de acordo com o espaço de nomes o qual ele pertence.

A utilidade de um tal conceito torna-se evidente quando se pretende construir algo de forma modular, juntando blocos de diferentes proveniências num só programa.

Declaração de um Espaço de Nomes Em *C++*, um espaço de nomes é declarado através de um ou mais blocos.

```
namespace <identificador> { ... }
```

Por exemplo:

```
// complexos.hpp
namespace exerciciosPOO{
  class Complexos {
  public:
    // Construtores
    Complexos (double, double); // x+iy
    Complexos (double); // x+i0
    Complexos (); // 0+i0
...
}
```

Estas declarações têm um efeito cumulativo, isto é, após a declaração inicial que cria o espaço de nomes, incorporando nele os nomes das classes, funções, etc. aí contidos, as declarações posteriores vão adicionar nomes ao espaço de nomes anteriormente criado.

Utilização de um Espaço de Nomes Para utilizar um identificador pertença de um espaço de nomes é necessário prefixar o identificador que se pretende usar com o identificador do espaço de nomes, isto é, o identificador deve ser prefixado com o caminho em profundidade desde o espaço de nomes global até ao espaço de nomes ao qual o identificador pertence (separados por `::`).

Por exemplo:

```
// usaComplexos.cpp
...
// Inicialização do objecto
exerciciosPOO::Complexos z1(x,y);
...
```

Em alternativa podemos "abrir" o espaço de nomes. Por exemplo:

```
// usaComplexos.cpp
// espaço de nomes "standard"
using namespace std;
// espaço de nomes para POO
using namespace exerciciosPOO;
...
```

```
Complexos z1(x,y);
...
```

7.4 Excepções

A grande maioria dos programas tem de lidar com situações de erro, a forma como lidamos com as mesmas vai desde o enviar uma simples mensagem de erro, eventualmente terminado a execução do programa, até à programação de mecanismos que procurem solucionar o problema, permitindo a continuação da execução do programa.

Quando um programa está divido em vários «blocos» (ficheiros, etc.) esta situação torna-se mais problemática. O erro pode ser devido ao código que está num dado bloco e no entanto o momento em que o mesmo ocorre é devido ao código num outro bloco.

Por exemplo um programa que utilize a estrutura de dados *pilha* tem que lidar com a situação dos acessos a uma pilha vazia. O problema ocorre aquando da utilização da pilha no programa principal, no entanto o problema é devido ao código dos métodos `pop` e `top` na classe `Pilha`.

O *C++* providência um modo de lidar com as situações de excepção separando o momento em que a excepção ocorre do momento em que ela é provocada. i

7.4.1 Criar um Elemento Excepção

Podemos considerar que as excepções vão ser elementos adicionais aos tipos existentes. Por exemplo o elemento '`erroPilhaVazia`'. Podemos pensar neste novo «elemento», como um novo elemento do tipo pilha, solucionado do ponto de vista teórico a definição da função '`pop`'.

$$\begin{array}{rcccl} \text{pop} & : & Pilha & \longrightarrow & Pilha \cup \{erroPilhaVazia\} \\ & & p & \longmapsto & \begin{cases} p', & \text{se p} = \text{e:}p' \\ \texttt{erroPilhaVazia}, & \text{se p} = \texttt{pilhaVazia} \end{cases} \end{array}$$

A função '`pop`' passa a estar correctamente definida como sendo uma função total de `Pilha` para `Pilha` $\cup$ {`erroPilhaVazia`}.

A definição de um novo elemento de erro é feita pela definição de uma nova estrutura, um identificador para um novo tipo de dados.

A separação entre diferentes excepções é feita por tipo e não por valor, isto é, podemos levantar uma situação de erro do tipo `int`, a qual vai ser distinta de uma situação do tipo `double`, mas já não podemos dizer que levantamos uma excepção com valor, por exemplo, 5 (do tipo `int`) e esperar que ela seja distinta de uma outra com valor 7 (também do tipo `int`).

Isto leva a que seja normal declarar, antes de mais nada, novos tipos. Um para cada situação de erro que possa ocorrer.

Dois exemplos:

```
// acesso (pop e top) à Pilha Vazia
struct erroPilhaVazia {};
```

```
// acesso a elemento que não existe na Lista
struct indiceIncorrecto {
  int num;
  indiceIncorrecto(int _i) {num = _i;} // Construtor
};
```

No primeiro caso a estrutura não tem elementos, e como tal não transporta informação, é só a declaração do tipo. No segundo caso a estrutura tem um campo que pode ser usado para transportar informação adicional que nos pode ajudar no tratamento do erro. Ver-se-à mais abaixo como o fazer.

7.4.2 Declarar uma Excepção

O comando `throw` permite a declaração de uma excepção (ele «atira/levanta» *throw* uma excepção). A sua sintaxe é a seguinte:

```
throw (<lista_de_identificadores_de_tipo>);
```

em que a lista de identificadores de tipo vai conter um ou mais identificadores de tipos, sejam eles pré-definidos ou definidos pelo programador.

O comando `throw` é usado no método (função) no qual a situação de excepção se encontra. Usando o exemplo da estrutura de dados pilha, ter-se-ia:

```
  /*
   * pop: Pilha -> Pilha + erroPilhaVazia
   * retira o elemento do topo da pilha
   *
   * Excepção: erroPilhaVazia
   */
  void pop();
```

isto aquando da declaração do método (em, por exemplo `pilha.hpp`).

```
// pop, excepção: erroPilhaVazia
void Pilha::pop(){
  No *aux;

  if (pilha!=NULL) {
    aux=pilha;
    pilha=aux->prox;
    delete [] aux;
  }
  else {
    throw Pilha::erroPilhaVazia ();
  }
};
```

aquando da implementação (em, por exemplo `pilha.cpp`).

Na implementação, a declaração da situação de excepção `throw Pilha::erroPilhaVazia ();`. Neste caso o tipo declarado não transporta nenhuma informação adicional.

É de notar que na declaração tem-se `erroPilhaVazia ()`, isto é a criação de uma instância do tipo (criação do objecto) `erroPilhaVazia`, utilizando para tal o construtor, por omissão, da estrutura.

7.4.3 Lidar com Excepções

No programa que vai chamar o método que declara a excepção é necessário usar um mecanismo do *C++* próprio para o lidar com situações de excepção. Esse mecanismo é designado por `try` e `catch`. A sua sintaxe é a seguinte:

```
try {
   <código_que_pode_accionar_a_excepção>
}
[ catch (<declaração_do_tipo_excepção>) {
    <código_que_é_executado_quando_ocorre_a_excepção>
    <relacionado_com_o_tipo_da_excepção>
 }
[ catch (<declaração_do_tipo_excepção>) {
    <excepção_de_outro_tipo>
 }
[ catch (...) {
    <caso_por_omissão>
} ] ] ]
```

Alguma notas:

- As secções `catch` são opcionais. Caso não estejam presentes a execução do programa segue logo a seguir ao bloco `try...catch`;
- Só uma das excepções é invocada, embora semelhante à instrução `switch` esta instrução não necessita da instrução `break`. Não há o efeito de cascata (*fall through*);
- A excepção «`...`», quando usada, associa com um qualquer tipo de excepção.
- Os tipos das excepções são avaliados por ordem descendente, isto significa que a ordem pela qual são escritas é importante. Nomeadamente o caso «`(...)`» tem de ser o último.

Continuando a usar o caso da estrutura de dados pilha. No programa principal (em, por exemplo `usaPilha.cpp`) ter-se-ia:

```
  try {
    cout << s1.top() << endl;
    s1.pop();
    cout << s1.top() << endl;
    s1.pop();
    cout << s1.top() << endl;
  }
  catch(Pilha::erroPilhaVazia) {
    cout << "A pilha está vazia" << endl;
  }
```

Caso a pilha tivesse menos do que três elementos iria ocorrer pelo menos uma excepção. No momento em que o a excepção é invocada pela primeira vez o código referente à mesma é executado, neste caso a mensagem «A pilha está vazia» seria escrita, e a execução do programa continua após o bloco `try...catch`.

No caso em que ocorra uma excepção mas que essa não se enquadre em nenhum dos casos considerados para lidar com as excepções, então a execução do programa termina de imediato.

7.4.4 Excepções que Transportam Informação Adicional

No caso anterior o tipo da excepção é definida por uma estrutura vazia. Podemos ter casos em que seja útil poder usar informação adicional. Voltando ao caso da excepção `indiceincorrecto` vejamos como é que podíamos usufruir dessa informação adicional.

Relembrando:

```
// acesso a elemento que não existe na Lista
struct indiceIncorrecto {
  int num;
  indiceIncorrecto(int _i) {num = _i;} // Construtor
};
```

Note-se que além da definição dos atributos `num` também se definiu o construtor com um argumento (a definição deste, com um valor específico, já não é feita por omissão).

Aquando do levantar da excepção pode-se acrescentar informação à excepção. Por exemplo, no método `retirar_k`, referente a uma estrutura do tipo lista, sempre que a posição do elemento a retirar não exista podemos levantar a excepção e acrescentar a informação de qual foi o elemento que se tentou eliminar.

```
  /*
   * Se o elemento a retirar não existe - situação de erro
   */
  void retirar_k(int posicao){
    No *ant=NULL, *prox=ptLista;
    int i=1;
    while (i < posicao && prox != NULL) {
      ant = prox;
      prox = prox->proxNo;
      i++;
    };
    if (prox == NULL)
      throw indiceIncorrecto(posicao);
    else {
      if (ant == NULL)
        ptLista = prox->proxNo;
      else
        ant->proxNo = prox->proxNo;
      delete prox;
    };
  };
```

No momento da utilização podemos usar a informação adicional para ajudar o utilizador a melhor compreender a situação. Por exemplo:

```
  cout << "Retirar um elemento. Diga qual: \t";
  cin >> pos;
  try {
    listaInt.retirar_k(pos);
  }
  catch(Lista<int>::indiceIncorrecto ii) {
    cout << "Erro, a lista não possuí a posição: " << ii.num << endl;
    cout << "O comprimento da lista é: " << listaInt.comprimento() << endl;
    return 0;
  }
```

A mensagem obtida, algo como:

```
Retirar um elemento. Diga qual:         7
Erro, a lista não possuí a posição: 7
O comprimento da lista é: 5
```

é bastante mais informativa do que uma simples mensagem de erro, dizendo que não foi possível retirar o elemento.

Uma outra possibilidade de utilização deste tipo de mecanismo é dado pelo recuperar de situações de erro. Por exemplo, dependendo da aplicação em questão e no caso do método «`pop`» da classe Pilha, poder-se-ia devolver a pilha vazia, permitindo deste modo o continuar do programa sem a consequente quebra devido à situação de erro.

7.5 Programação Genérica

Por programação genérica entende-se a construção de programas genéricos, capazes de se adaptar a diferentes situações.

> Programação Genérica: a escrita de programas que funcionam sob uma variedade de tipos de argumentos, desde que os mesmos tipos respeitem restrições sintácticas e semânticas bem definidas.
>
> Bjarne Stroustrup em (Stroustrup, 2009)

Já vimos em 5.2.4 a possibilidade de definir estruturas genéricas, por exemplo, a definição de uma pilha de elementos genéricos. Nesta secção vai-se abordar a questão da definição de funções genéricas, por exemplo, uma função de ordenação que possa ser aplicada a diferentes tipos de listas.

Um exemplo, uma função de ordenação genérica, para listas genéricas:

```
#include <iostream>
#include "listagenerica.cpp"

using namespace std;

template<class Elementos> void ordena(Lista<Elementos>&); // declaração


template<class Elementos> void ordena(Lista<Elementos>& v) {
// borbulhagem - bubble sort
  const int dim = v.comprimento(); // método da classe Lista;
  int i;
  Elementos aux; // elemento da classe parâmetro
  bool jaordenado;

  do {
    jaordenado = true;
    for (i = 1; i < dim ; i++) { // precorre a lista
      if (v.ver_k(i) > v.ver_k(i+1)) { // se necessário troca elementos
        aux = v.ver_k(i); // salvaguarda v_i
        v.retirar_k(i); // apaga v_i ficando em seu lugar o v_i+1
        v.inserir_k(i+1,aux); // inserir o v_i na posição i+1
        jaordenado = false;
      }
    }
  } while (!jaordenado); // repete até não haver mais trocas
}


int main() {
```

```
  Lista<int> listaInt;
  int elem,pos,aux=1;

  cout << "Introduza vários inteiros termine com '0': \t";
  do {
    cin >> elem;
    listaInt.inserir_k(aux,elem);
    aux++;
  } while (elem != 0);
  listaInt.mostrar_elementos();
  cout << "Retirar um elemento. Diga qual: \t";
  cin >> pos;
  try {
    listaInt.retirar_k(pos);
  }
  catch(Lista<int>::indiceincorrecto ii) {
    cout << "Erro, a lista não possuí a posição: " << ii.num << endl;
    cout << "O comprimento da lista é: " << listaInt.comprimento() << endl;
  }
  listaInt.mostrar_elementos();
  ordena(listaInt);
  listaInt.mostrar_elementos();
  return 0;
}
```

A função de ordenação tal como foi apresentada tem uma dependência não explícita, mas determinante para o algoritmo:

```
      if (v.ver_k(i) > v.ver_k(i+1)) { // se necessário troca elementos
```

Temos aqui a comparação entre dois elementos genéricos, isto é, dois elementos dos quais não sabemos, à partida, o tipo dos mesmos. Se, por exemplo, os elementos são do tipo *Complexos*, definidos por uma outra classe previamente construída, qual é então o significado de "`v.ver_k(i) > v.ver_k(i+1)`"?

Dado que o *C++* não é uma linguagem funcional, as funções não são elementos de primeira ordem (básicos) e como tal não é possível passar funções como argumentos (embora se possa passar um ponteiro para uma função, ver (Stroustrup, 1997, §7.7)). Temos então que a classe `Elemento` deve implementar o operador '`<`' para que a função genérica possa funcionar. Ao implementar-se funções genéricas é importante ter em conta este tipo de dependência.

7.6 Interfaces de Programação

Quando se usa uma linguagem de programação orientada para os objectos a construção e utilização de bibliotecas torna-se "natural". Para esse efeito temos de construir/utilizar interfaces de programação.

Um interface de programação (API da designação em Inglês *Application Programming Interface*) é um conjunto de chamadas para métodos e atributos estabelecidos para um dado programa de forma a ser possível a utilização das suas funcionalidades por outros programas sem que estes necessitem, ou mesmo tenham acesso, aos detalhes da implementação.

Isto é, trata-se de uma implementação do Princípio de Parmas (ver Proposição 1) de construção de módulos sonegando informação desnecessária,

> "*Deve-se dar ao utilizador de um módulo toda a informação necessária para este usar o módulo correctamente, e nada mais.*"

De modo geral, uma API é composta por uma série de métodos públicos e a descrição pormenorizada da forma como os mesmos podem ser acedidos. Trata-se de uma interface para programadores, isto é, a especificação de uma interface pública que permite aceder aos métodos e atributos de um dado módulo, acrescentando funcionalidades aos programas, ao mesmo tempo que esconde os detalhes de implementação.

7.6.1 Exemplo de Construção/Utilização de uma API

Pretende-se construir uma aplicação para gerir uma lista de contactos, com as operações usuais de inserção, actualização, remoção e pesquisa por nome.

As estruturas de dados a usar devem ser as existentes na biblioteca STL. A agenda é guardada, entre execuções, numa base de dados MySQL, usando para tal a biblioteca de ligação entre o *C++* o MySQL.

Além dos APIs referentes à biblioteca STL e de ligação à base de dados, vamos querer isolar a aplicação de qualquer referência à base de dados. Dessa forma não só se separam tarefas, entre o especialista em bases de dados e o programador do interface, como além disso permite-se que se mude de opinião acerca da forma de guardar, de forma persistente, a informação referente à lista de contactos.

Na figura 7.2 temos o diagrama UML referente a uma possível solução.

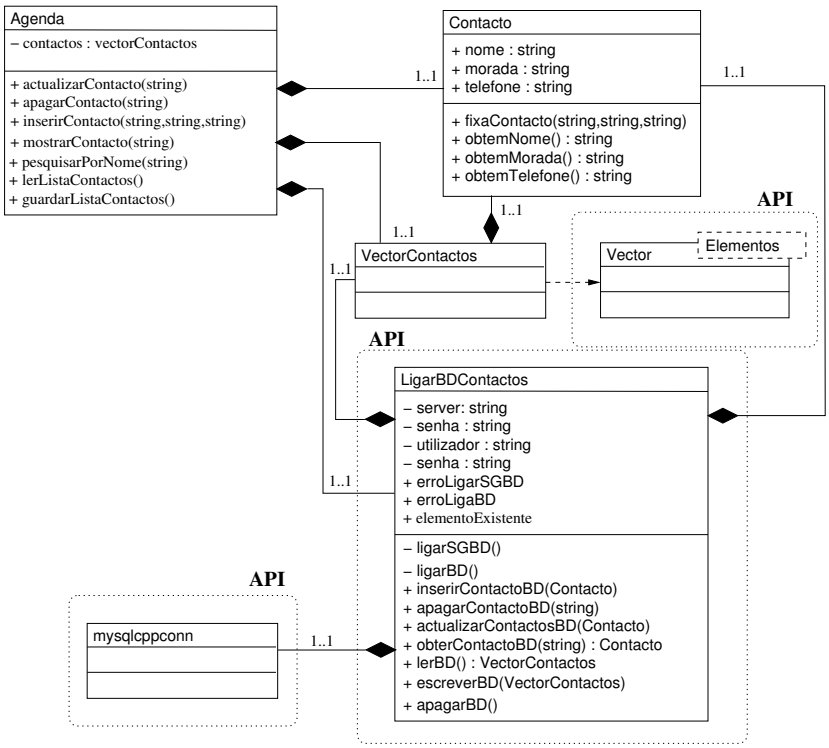


Figura 7.2: Agenda — Lista de Contactos

A escrita do API passa então, além da produção da biblioteca propriamente dita, pela escrita de um manual de referência. Algo como:

Nome da Biblioteca e Objectivos Nome da biblioteca, os seus objectivos e como a utilizar.

LigarBDContactos A biblioteca *LigarBDContactos* providência uma nível de abstracção entre a *Agenda* e a forma de guardar persistentemente a informação da *Agenda*. No caso actual por utilização de uma base de dados.
Após a instalação a sua utilização requer a escrita do `include` apropriado.

```
#include <LigarBDContactos>
```

Além da referência à biblioteca '`-lLigarBDContactos`' e dependendo da instalação a compilação pode requerer a especificação dos caminhos referentes aos ficheiros a incluir '`-I`, assim como o caminho para a biblioteca propriamente dita '`-L`'. O executar dos programas pode requerer a actualização da variável de ambiente apropriada.

Especificação dos Métodos e Atributos Públicos O passo seguinte é a descrição de cada uma das componentes públicas da nova biblioteca.

inserirContactoBD Dado um *Contacto* novo insere o mesmo na base de dados. Caso o *Contacto* já exista é devolvido o código de erro *elementoExistente*.

```
void inserirContactoBD(Contacto novoContacto) throw (elementoExistente)
```

O API deve ser completo no sentido em que ele deve providenciar todos os serviços necessários à sua plena utilização. Algo que seja esquecido, intencionalmente ou não, não ficará disponível ao utilizador/programador.

Por exemplo o método `apagarContactoBD`. Podemos decidir que apagamos um só contacto, através do número de telefone, ou múltiplos contactos através do nome, ou consideramos todas as possibilidades.

Os métodos devem ser simples nos seus objectivos. A sua utilização e os resultados que daí advêm devem ser claros.

O conjuntos de métodos deve ser completo. Se se tem `apagarContactoBD` por número de telefone, então tem de ser possível obter esse mesmo número para uma dada pessoa. Isto é, para a questão do apagar de um contacto temos duas possíveis soluções.

1. Construir diversos métodos de apagar contactos para as diferentes situações: por número de telefone; por nome de contacto; por morada.

2. Um só método para apagar, por exemplo por número de telefone, o utilizador obtêm a informação relevante, o número de telefone, através de um outro método, neste exemplo, `obterContactoBD(string)` e depois usa essa informação para apagar um contacto utilizando para tal o número de telefone.

A situação em que uma dada funcionalidade não pode ser obtida, nem directamente, nem indirectamente, é algo a evitar a todo o custo. Nesse caso o conjunto de funcionalidades fornecido não seria completo.

Apêndice A

A Declaração “const” em C++: Porquê & Como

Por Andrew Hardwick.
Distribuível sob licença freeware GPL.
Escrito em 2001.
Convertido para HTML e aumentado em 2002/3/13.
Atualizado 2002/8/9, 2004/7/19, 2005/6/9 e 2006/5/22.
Correcções ortográficas 2008/3/4.
Correções gramaticais e esclarecimentos 2011/6/16.
Correção de typo 2011/12/29, 2012/4/4 e 2015/11/12.
Disponível on-line em: http://duramecho.com
Traduzido e editado para Português e convertido para LaTeX por:
Pedro Quaresma, 2019/2/28

A declaração `const` é um dos recursos algo confuso do *C++*.

É simples em conceito: variáveis declaradas como `const` tornam-se constantes e não podem ser alteradas pelo programa. No entanto, ele também é usado para substituir um dos recursos ausentes do *C++* e nesse contexto torna-se bastante complicado e às vezes frustrantemente restritivo.

No que se segue tenta-se explicar como `const` é usado e por que ele existe.

A.1 Uso Simples de `const`

O uso mais simples é o de declarar uma constante. Algo já presente na linguagem *C*.

Para fazer isso, declara-se uma constante como se fosse uma variável, mas adicione `const` antes dela. É preciso inicializá-la imediatamente porque, é claro, não é possível definir o valor mais tarde, pois isso seria alterá-lo (o que passa a ser impossível). Por exemplo,

```
const int constante1 = 96;
```

irá criar uma constante inteira, `constante1`, com o valor 96.

Tais constantes são úteis para parâmetros que são usados no programa, mas não precisam ser alterados depois que o programa é compilado. Esta forma tem uma vantagem para os

programadores sobre o comando `#define` do pré-processador do *C*, pois ele é entendido e usado pelo próprio compilador, não apenas substituído no texto do programa pelo pré-processador antes de chegar ao compilador principal. Desta forma as mensagens de erro são muito mais úteis.

Ele também funciona com ponteiros, mas é preciso ter cuidado onde o `const` é colocado para determinar se o ponteiro, ou o que ele aponta, é constante. Por exemplo:

```
const int * constante2
```

declara que `constante2` é um ponteiro variável para um inteiro constante e:

```
int const * constante2
```

é uma sintaxe alternativa que faz o mesmo, enquanto:

```
int * const constante3
```

declara que `constante3` é um ponteiro constante para um inteiro variável e:

```
int const * const constante4
```

declara que `constante4` é um ponteiro constante para um inteiro constante. Basicamente, `const` aplica-se a tudo o que está à sua esquerda imediata (a não ser que não exista nada, caso em que se aplica a qualquer que seja o seu direito imediato).

A.2 Uso de `const` em Valores de Retorno de Funções

Das possíveis combinações de ponteiros e `const`, o ponteiro constante para uma variável é útil para armazenamento que pode ser alterado em valor, mas não movido na memória.

Ainda mais útil é um ponteiro (constante ou não) para um valor `const`. Isso é útil para retornar sequências de caracteres e matrizes constantes de funções que, como elas são implementadas como ponteiros, o programa poderia tentar alterar. Em vez de um alteração não desejada, difícil de localizar, a tentativa de alterar valores inalteráveis será detectada durante a compilação.

Por exemplo, considerando-se a definiçao de uma função que retorna uma constante do tipo *string*, `"Algum texto"`:

```
char * funcao1 () {
   return "Algum texto";
}
```

então o programa poderia falhar se acidentalmente tentasse alterar o valor

```
funcao1 () [1] = 'a';
```

O compilador teria detectado esse erro se a função original tivesse sido escrita

```
const char * funcao1 () {
   return "Algum texto";
}
```

porque o compilador saberia então que o valor era inalterável. É claro que o compilador poderia, teoricamente, ter resolvido isso, em qualquer dos casos, mas os compiladores de *C* não são assim tão inteligentes.

A.2.1 Onde Fica Confuso—na Passagem de Parâmetros

Quando uma sub-rotina ou função é chamada com parâmetros, as variáveis são passadas conforme os parâmetros podem ser lidos a fim de transferir dados para a sub-rotina / função, escrita para transferir dados de volta para o programa de chamada ou ambos, para fazer ambos. Algumas linguagens permitem que se especifique isso directamente, por exemplo, através de definições do tipo `in:`, `out:` e `inout:`, enquanto que em *C* é preciso trabalhar a um nível inferior e especificar o método para passar as variáveis, escolhendo uma que permita a direcção de transferência de dados desejada (o *C* só faz a passagem «in», isto é, passagem por valor).

Por exemplo, uma sub-rotina como:

```
    void subrotina1 (int parametro1) {
       printf("%d", parametro1);
    }
```

aceita o parâmetro passado para ele no modo *C/C++* padrão—que é por cópia. Portanto, a sub-rotina pode ler o valor da variável passada para ela, mas não pode alterá-lo, porque quaisquer alterações feitas são feitas apenas na cópia e são perdidas quando a sub-rotina termina. Por exemplo:

```
    void subrotina2 (int parametro1) {
       parametro1 = 96;
}
```

deixaria a variável com a qual foi chamada inalterada.

A adição de um '`&`' ao nome do parâmetro em *C/C++* faz o efeito de uma passagem por referência, «inout», ao passar o ponteiro para a variável. Desta forma a própria variável real, em vez de uma cópia, é usada como o parâmetro na sub-rotina e, portanto, pode ser usado para passar os dados de volta para a sub-rotina. Assim sendo:

```
    void subrotina3 (int \& parametro1) {
       parametro1 = 96;
    }
```

alteraria a variável com a qual ele foi chamado para 96.

Essa maneira de passar variáveis constitui uma alteração do *C++* em relação ao *C*. Para passar uma variável alterável no *C* original, foi utilizado um método algo complexo, que envolvia o uso de um ponteiro para o parâmetro, em seguida, sendo possível, então, alterar o apontado. Por exemplo

```
    void subroutine4 (int * parametro1) {
       *parametro1 = 96;
    }
```

funciona, mas requer a utilização explicita do ponteiro em todos os pontos da sub-rotina e que a rotina de chamada também seja alterada para passar um ponteiro para a variável. É um pouco complicado.

Mas onde é que `const` entra nisso? Bem, há um segundo uso comum para passar dados por referência ou ponteiro em vez de uma cópia. Isso é quando o copiar da variável tem o efeito de uma grande utilização de memória e/ou muito tempo de execução. Isso é particularmente provável com tipos e variáveis grandes e complexos definidos pelos utilizadores («estruturas» em *C* e «classes» em *C++*). Então uma sub-rotina declarada:

```
void subrotina4 (Tipo_Estrutura_Grande & parametro1);
```

pode estar usando '`&`' porque vai alterar a variável passada para ele, ou pode ser apenas para economizar tempo de cópia e não há como saber qual é a situação no caso de umaa função que é compilada na biblioteca de outra pessoa. Isso pode ser um risco se for necessário confiar na sub-rotina para não alterar a variável.

Para resolver isso, `const` pode ser usado na lista de parâmetros. Por exemplo:

```
void subrotina5 (Tipo_Estrutura_Grande const & parametro1);
```

o que fará com que a variável seja passada sem copiar, mas não possa ser alterada. Isso é confuso porque está essencialmente fazendo um método de passagem constante de um método de passagem variável. Isto é, foi utilizado método de passagem variável somente para enganar o compilador, para fazer alguma optimização.

Idealmente, o programador não deveria precisar especificar com este nível de detalhe exatamente como as variáveis são passadas, apenas dizer em que direção as informações vão e vêem, e deixar o compilador fazer a optimização automaticamente, mas, como o *C* foi projetado para programação básica de baixo nível em computadores muito menos poderosos do que o padrão nos dias de hoje, o programador tem que fazê-lo explicitamente.

A.2.2 Ainda Mais Confuso - na Programação Orientada para os Objetos

Em programação orientada para os objeto, chamar um «método» (forma de designar a chamada de uma função em programação orientada para os objectos) de um objecto fornece uma complicação extra. Assim como as variáveis na lista de parâmetros, o método tem acesso às variáveis do próprio objecto, que são sempre passadas directamente não como cópias. Por exemplo, uma classe trivial, `Classe1`, definida como

```
classe Classe1 {
   void metodo1 ();
   int atributo1;
};
```

não tem parâmetros explícitos para o `metodo1`, mas chamá-lo em um objecto dessa classe pode alterar `atributo1` desse objecto, no caso de se usar o método, `metodo1`, por exemplo:

```
void Classe1 :: metodo1 () {
   atributo1 = atributo1 + 1;
}
```

A solução para isso passa por colocar `const` após a lista de parâmetros como:

```
classe Classe2 {
  void metodo1 () const;
  int atributo1;
}
```

que banirá o `metodo1` na `Classe2` de qualquer coisa que possa tentar alterar qualquer atributo no objeto.

É claro que às vezes é necessário combinar alguns desses diferentes usos de `const`, o que pode ser confuso, como em:

```
const int * const metodo3 (const int * const &) const;
```

onde os cinco usos de `const` significam respectivamente: que a variável apontada pelo ponteiro retornado e o próprio ponteiro retornado não serão alteráveis; e que o método não altera a variável apontada pelo ponteiro dado, o próprio ponteiro dado e o objecto a que o método pertence!

A.3 Inconvenientes de `const`

Além da confusão da sintaxe `const`, existem algumas coisas úteis que o sistema impede que os programas façam.

Uma questão que interfere com optimizações para velocidade. Um método que é declarado `const` não pode nem mesmo fazer alterações nas partes privadas de seu objecto, mesmo que elas não provoquem quaisquer alterações que seriam aparentes do lado de fora. Isso inclui armazenar resultados intermediários de cálculos longos, o que economizaria tempo de processamento nas chamadas subsequentes aos métodos da classe. Em vez disso, ele deve passar esses resultados intermediários de volta para a rotina de chamada para armazenar e passar de volta na próxima vez (confuso) ou recalcular do zero na próxima vez (ineficiente). Em versões mais recentes do C++, a palavra-chave `mutable` foi adicionada, o que permite que `const` seja sobrescrito para este propósito, mas confia totalmente no programador para usá-lo apenas para esse propósito, então se você tiver que escrever um programa usando uma classe escrita por outro programador, que usa `mutable`, você não pode garantir que coisas «mutáveis» sejam realmente constantes, o que torna `const` virtualmente inútil.

Não se pode simplesmente evitar o uso de `const` nos métodos de classe porque `const` é «infeccioso». Um objecto que foi feito `const`, por exemplo, sendo passado como um parâmetro no modo `const &`, pode ter apenas aqueles de seus métodos que são declarados explicitamente `const` (porque o sistema de chamada do *C++* não é capaz de descobrir quais métodos não explicitamente declarados `const` realmente não mudam nada. Portanto, os métodos de classe que não alteram o objecto são melhor declarados como `const` para que não sejam impedidos de serem chamados quando um objecto da classe tiver adquirido o status `const`. Em versões mais recentes do *C++*, um objecto, ou variável que foi declarada `const`, pode ser convertido para alterável pelo uso de `const_cast`, que é uma resposta similar a `mutable` e o uso da mesma forma torna `const` virtualmente inútil.

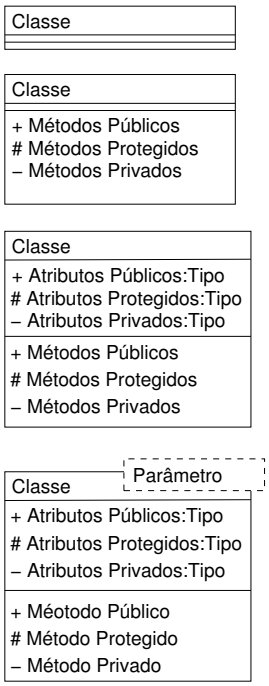
Apêndice B

UML & BNF

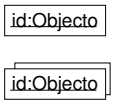
B.1 Unified Modeling Language

Havendo diferentes formas de graficamente expressar um sistema de classes e de associações entre classes o formalismo UML (**U**nified **M**odeling **L**anguage) é comummente usada (com algumas variações menores) e é apropriada para o efeito.

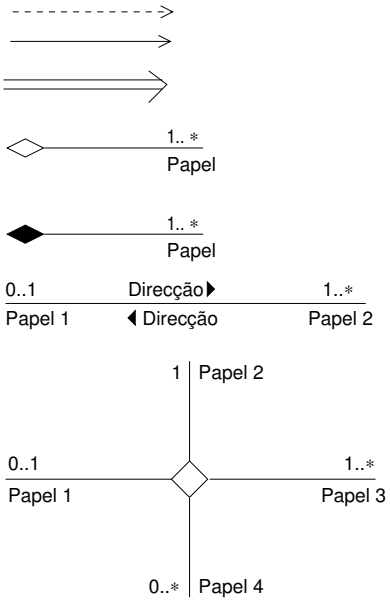
Classe Variações sobre o mesmo tema, úteis em diferentes fases de um dado projecto. A última representação refere-se a classes escantilhão, classes paramétricas para as quais podemos ter diferentes instanciações conforme o parâmetro dado nesse momento.



Objectos No caso dos objectos temos a possibilidade de representar um, ou múltiplos objectos.



Relações Temos, por ordem descendente, a representação para: instanciação; herança; meta-classe, agregação, escondida e visível; associação e associação múltipla.



Exemplos Exemplos de representação de hierarquias de classes: herança simples (B.1); herança múltipla (B.2); agregação (B.3); instanciação (B.4).

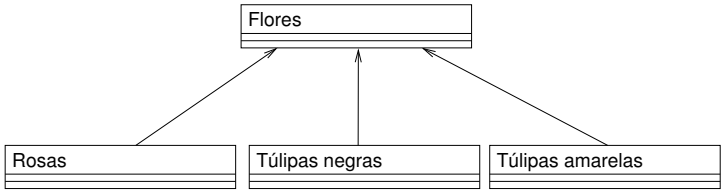


Figura B.1: Herança Simples

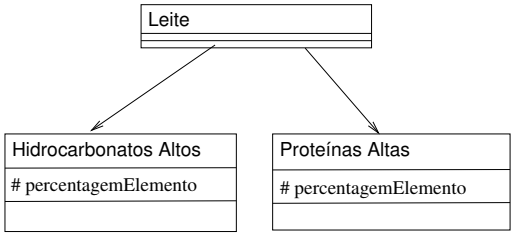


Figura B.2: Herança Múltipla

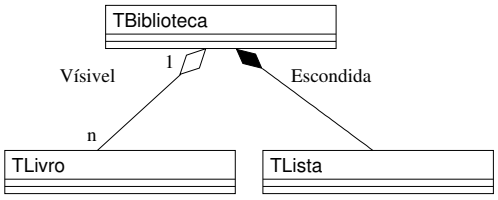


Figura B.3: Agregação

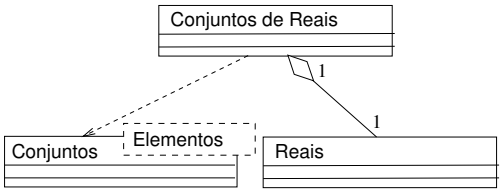


Figura B.4: Instanciação

B.2 Forma Normal Estendida de Backus-Naur

A forma normal Estendida de Backus-Naur (**E**xtended **B**ackus-**N**aur **F**orm) é um simbolismo muito usado na área da descrição das linguagens, nomeadamente na descrição da sintaxe das linguagens computacionais (Pattis, 1980).

As regras de derivação têm um lado direito e um lado esquerdo, separadas pelo símbolo de derivação (::= ou ⇐), por exemplo:

função ::= <tipo> <identificador> (<listaArgumentos>) { <corpoDaFunção> }

Esta regra que define a sintaxe da definição de uma função em *C++* lê-se como *função* é definida por . . . o lado direito da regra de derivação.

Do lado esquerdo aparece sempre um só identificador o qual dá nome à regra e, como já se disse, é o objectivo da definição.

No lado direito, o qual define associada à regra de derivação é uma combinação das formas de controle especificadas na tabela B.1.

Sequência	os elementos aparecem da esquerda para a direita, sendo que a ordem relativa é importante
Alternativa	os elementos são separados por uma barra vertical |; um dos elementos é escolhida dessa lista de opções, a ordem relativa não é importante
Opcional	um elemento opcional é colocado entre parêntesis rectos («[» e «]»), o elemento pode ser considerado ou simplesmente ignorado
Repetição	um elemento é colocado entre chavetas («{», «}») significando que pode ser repetido zero ou mais vezes

Tabela B.1: Formas de Controlo do Lado Direito das Regras EBNF

Sempre que os elementos do lado direito ainda carecem de definição, colocam-se entre os símbolo relacionais de menor e de maior (<, >), a sua definição terá de ser encontrada numa outra regra de derivação. Os *literais*, isto é, os elementos a serem interpretado de forma

literal, são escritos utilizado o tipo de letra «máquina de escrever» («typewriter», tipo de letra mono-espaço) e devem ser escritos exactamente como está na regra.

A forma de aplicar este formalismo na validação da escrita de programas é feita através da associação de padrões, entre o padrão definido por uma definição concreta em, neste caso, da linguagem *C++* e a regra de derivação associada ao elemento que se está a definir.

Por exemplo, será que a seguinte definição de função em *C++* está correcta?

```
void troca(int *a,int *b) {
  *a = *a*(*b);
  *b = *a/(*b);
  *a = *a/(*b);
}
```

Analisando-o à luz regra definida acima temos

Elemento *C++*	Elemento EBNF	Associação
`void`	<tipo>	sim, `void` é um tipo em *C++*
`troca`	<identificador>	sim, `troca` é um identificador possível em *C++*
`(`	`(`	sim, é um literal
`int *a,int *b`	<listaArgumentos>	sim, ver regra específica
`)`	`)`	sim, é um literal
`{`	`{`	sim, é um literal
*a = *a*(*b); ...	<corpoDaFunção>	sim, ver regra específica
`}`	`}`	sim, é um literal

Tendo havido um «encaixe» perfeito entre a regra de derivação e o fragmento de *C++*, podemos afirmar que a definição da função `troca` está correcta, do ponto de vista da sintaxe do *C++*.

Apêndice C

Construção de um Executável

As linguagens tais como *C/C++* são ditas linguagens compiláveis. Quer se com isto dizer que, através da utilização de um programa próprio, o compilador, o programa escrito na linguagem *C/C++* vai ser transformado num programa em código máquina e autónomo. Isto é o compilador vai transformar o programa em *C/C++* num executável.

A forma como essa transformação é feita, e os programas que podem ser usados para nos ajudar nesse processo são o objecto deste capítulo.

C.1 Compilador de C++

A transformação de um programa em *C/C++* num programa executável é feito pelo compilador respectivo. É possível individualizar três fases distintas no processo de compilação: o pré-processamento; a fase de análise e de geração do código máquina (programa na linguagem da máquina); e agregação e finalização (algumas vezes designado por «linkagem», um aportuguesamento do termo inglês *linking* (agregando/juntando)). Vejamos com um pouco mais de detalhes estas diferentes fases.

pré-processamento: o ficheiro contendo o programa é analisado por completo sendo que as directivas de pré-processamento são cumpridas. É também nesta fase do processamento que todos os comentários são descartados. O resultado desta fase é um programa contendo exclusivamente código *C/C++*. Num processo de compilação normal esta fase não produz nenhum ficheiro de saída.

análise: análise léxica (palavras), sintáctica (frases) e semântica (tipos de dados) do código resultante da fase anterior para avaliar se o mesmo está de acordo com a regras da linguagem *C/C++*.

Se não houver erros, o resultado final desta fase é dado pela conversão do código *C/C++* em código máquina. Nem sempre o resultado desta fase resulta na produção de um ficheiro de saída (ver a secção C.1.2).

agregação (*linkagem*): no caso de se tratar do ficheiro contendo a função `main` («ponto de partida» do programa) o compilador pode proceder à fase final da transformação, agregando todo o código máquina produzido nas fases anteriores, juntamente com as bibliotecas necessárias (incluídas através de directivas de pré-processamento), assim

como o código necessário à execução do código pelo sistema operativo (*runtime routines*), num «bloco» único que é o programa executável.

O resultado desta fase, para o caso em que não há erros, é um ficheiro contendo o executável respeitante ao nosso programa.

É de notar que actualmente é normal o ficheiro resultante do processo de compilação não ser verdadeiramente auto-contido, isto é, o ficheiro não tem em si tudo o que é necessário para o seu funcionamento.

O facto de cada vez mais se usar bibliotecas externas para complementar os programas faz com que a proporção entre o código próprio do programa e o código externo (bibliotecas) seja em muitos casos muito desproporcionado em favor das bibliotecas. Este facto é ainda mais visível numa linguagem de Programação Orientada para os Objectos tal como o *C++* dado que a sua modularidade leva a um uso intensivo de bibliotecas padrão (biblioteca padrão do *C++* bibliotecas gráficas, etc.).

A conclusão do que foi dito acima é que, em geral, a fase de agregação vai «deixar de fora» as bibliotecas (do sistemas) objecto das instruções de inclusão (`#include`), deixando só a indicação que elas são necessárias para o funcionamento do programa. Este processo, designado por agregação dinâmica (*dynamic linking*) tem como grande vantagem gerar programas executáveis com uma dimensão (em «bytes») que é proporcional ao código *C/C++* escrito pelo programador. Tem como desvantagem o facto de que, na ausência de uma das bibliotecas necessárias ao seu funcionamento, o programa não executará.

É possível explicitar que se pretende a inclusão de tudo o que é necessário para a execução do programa obrigando o compilador a proceder a uma, assim designada, agregação estática (*Static Linking*). A desvantagem é que um qualquer programa, mesmo que pequeno, irá gerar um executável de dimensão considerável (vai depender das bibliotecas que se pretendeu incluir). A vantagem é que neste caso o programa será auto-contido, não necessitando de nenhum recurso adicional para o seu funcionamento (ver a secção C.1.2).

C.1.1 Pré-processamento

As directivas de pré-processamento passam pela definição de «macros», da declaração dos nomes dos ficheiros a incluir (bibliotecas e/ou programas auxiliares), assim como a compilação condicional (Kernighan & Ritchie, 1988; Stroustrup, 1997).

Qualquer linha iniciada por '`#`' (eventualmente com espaços em branco antes) será pré-processada. O efeitos das directivas é global para todo o ficheiro em que estão contidas. A declaração de um destes comandos ocorre sempre numa única linha, no entanto uma declaração muito longa pode ser dividida em várias linhas do ficheiro, basta para tal colocar '\' no fim de uma linha para que a próxima linha seja formalmente uma continuação da linha anterior, isto é, do ponto de vista da definição as duas linhas passam a contar como um linha única.

Definição de «Macros»

A sintaxe deste tipo de definição é a seguinte:

```
#define   <identificador> <sequênciaPalavrasReservadas>
#define   <identificador> (<listaIdentificadores>) <sequênciaPalavrasReservadas>
```

A definição de «macros» serve para dar um nome a uma expressão (eventualmente) complexa. Os casos mais usuais prendem-se com a definição de valores constantes que, desta forma, ganham um nome, um significado, assim como uma maior facilidade na sua (eventual) modificação. Por exemplo:

```
#define PI 3.14
#define NUMMAXELEM 30
```

No segundo caso podia significar o número de elementos que se estava a considerar para um dado vector. A declaração, leitura, escrita e cálculo, com os seus correspondentes ciclos seriam escritos todos em função deste valor. A sua alteração seria muito fácil. Por exemplo ter-se-ia:

```
int tabela[NUMMAXELEM];
```

O considerar de um valor mais exacto para π seria também uma questão de alterar uma só linha.

Uma nota: é usual escrever os identificadores associados à definição de macros com todas as letras em maiúsculas. Esta é uma convenção informal na comunidade de programadores em *C/C++*.

A definição:

```
#define VALORABS(a,b) ((a)>(b) ? (a)-(b) : (b)-(a))
```

define uma macro que devolve o valor o valor absoluto da diferença entre os seus argumentos.

Inclusão de Ficheiros

As inclusão de bibliotecas e/ou ficheiros auxiliares é definida do seguinte modo:

```
#include  <<nomeFicheiro>>
#include  "<nomeFicheiro>"
```

No primeiro caso trata-se de inclusão de bibliotecas do sistema, a sua localização concreta varia de acordo com o compilador e/ou sistema operativo usado. A sua utilização está dependente de uma correcta instalação da biblioteca no compilador/sistema operativo em causa.

O segundo caso é usualmente usado para a inclusão de programas auxiliares e/ou bibliotecas definidas pelo próprio programador. O nome do ficheiro pode incluir o «caminho» completo até ao ficheiro.

Ou seja, no primeiro caso a referência ao ficheiro é uma referência relativa, a conversão desta referência relativa para uma localização concreta do ficheiro em questão está a cargo do compilador. No segundo caso estamos a explicitar a localização concreta do ficheiro a incluir.

Compilação Condicional

As directivas de pré-processamento servem também para definir que certas componentes do programa podem ser, ou não, compiladas.

`#if`	<expressãoConstante>	
`#ifdef`	<identificador>	se está definido
`#ifndef`	<identificador>	se não está definido
`#elif`	<expressãoConstante>	else if
`#else`	<expressãoConstante>	else
`#endif`		

As linhas `if`, `elif` e `else` definem um conjunto de opções do qual uma (e só uma) será correcta e como tal processada, sendo os outros casos ignorados.

Uma das utilizações usuais para este tipo de directivas é dado pela inclusão de opções diferenciadas consoante o compilador e/ou sistema computacional em que a compilação está a ser efectuada.

Outra das utilizações é dada pela salvaguarda de uma tentativa (que falharia) de re-declaração de funções e/ou classes aquando da inclusão de um ficheiro auxiliar o qual, pela divisão «normal» de um programa em componentes, pode ser alvo de inclusão por mais do que um ficheiro e, por essa via, de uma dupla (ou mais) inclusão.

Esta situação é evitada através das seguintes directivas de pré-processamento (para o caso concreto do ficheiro `pilhaChar.hpp`, ver 2.2.1).

```
#ifndef PILHAS
#define PILHAS

class Pilha {
(...)
};

#endif
```

A macro `PILHAS` começa por não estar definida, mas aquando da primeira inclusão `PILHAS` passa a estar definida (o seu valor não é relevante). Numa inclusão posterior toda a definição da classe é ignorada não dado azo a erros de re-definição.

C.1.2 Opções de Compilação

Um qualquer compilador de *C/C++* tem um conjunto extenso de opções que permitem modificar a forma como a compilação se vai processar. No que se segue vai-se apresentar algumas das opções mais usuais para o compilador *GNU* C++ *Compiler*[1].

[1] `g++ --help`

Opção	Significado
`-o <nomeFicheiro>`	atribuir o nome ao ficheiro executável.
`-c <nomeFicheiro.cpp>`	cria somente o código máquina, isto é, não faz os passos de agregação e criação de um executável.
`-Wall`	(*Warnings all*) Os avisos sobre construções que alguns utilizadores podem achar questionáveis e que são facilmente evitáveis, passam a ser emitidos. Um conjunto alargado de avisos passa a ser emitido.
`-I<directório>`	Adiciona o *directório* em questão à lista de directórios a serem pesquisados à procura de ficheiro de cabeçalhos (*header files*, `.hpp`).
`-l<biblioteca>`	Agregação de uma dada *biblioteca*.
`-L<directório>`	Adiciona o *directório* em questão à lista de directórios a serem pesquisados à procura de bibliotecas (ver opção `-l`).

C.2 Bibliotecas

Ao utilizar-se o paradigma da programação orientada para os objectos, de forma quase natural surge a necessidade de construir não um programa, entidade fechada, mas uma biblioteca (ou pelo menos um nó, «livro» , de uma biblioteca). Designa-se por biblioteca (*library*) um, ou mais módulos, que implementam um dado conjunto de funcionalidades e que podem ser agregadas aos nossos programas. Além da biblioteca padrão do C++ (*Standard Library*, STL) existem muitas outras à nossa disposição. Por exemplo:

GMP *The GNU Multiple Precision Arithmetic Library.* A GMP é uma biblioteca livre para a aritmética de precisão e gama de variação arbitrárias. Implementa inteiros com sinal, racionais, e reais. As principais aplicações da biblioteca GMP são: sistemas criptográficos; segurança da Rede; sistemas algébricos, entre outras.

`http://gmplib.org/`.

Boost A biblioteca *Boost* é um esforço colectivo para providenciar uma biblioteca livre de código aberto e verificada pela própria comunidade de utilizadores. Providencia um conjunto alargado de funcionalidades.

`http://www.boost.org/`.

NAG *Numerical Algorithms Group Library*, é uma biblioteca comercial com uma vasta gama de algoritmos matemáticos e estatísticos.

`www.nag.co.uk/numeric/CL/CLdescription.asp`.

GTK+ *the GIMP Toolkit*, é uma biblioteca, multi-plataforma para a criação de interfaces gráficos.

`http://www.gtk.org/`.

MySQL Connector A biblioteca *MySQL Connector* providencia os métodos necessários para uma utilização de uma base de dados MySQL.[2]

[2] `http://dev.mysql.com/`

`http://dev.mysql.com/downloads/connector/cpp/`.

SQLite *SQLite*[3] é uma biblioteca, em *C*, que implementa uma base de dados relacional (SQL) auto-contida, isto é, não necessita de um servidor, e como tal é apropriada para ser incorporada num projecto que necessite de um acesso simples a bases de dados.

Possuí um conjunto de métodos necessários a sua utilização por parte de um qualquer programa.

`https://www.sqlite.org/cintro.html`.

Além destas existem muitas outras com diferentes graus de aplicabilidade, funcionalidade e qualidade.

Como é que podemos então usar outras bibliotecas que não a *STL*? Como podemos criar, e usar, a nossa própria biblioteca? Como foi dito acima uma biblioteca é algo que providencia um dado conjunto de funcionalidades e que podemos agregar aos nossos programas (a exemplo do que fazemos com a *STL*).

C.2.1 Como usar

Para podermos usar uma biblioteca externa (que não a *biblioteca padrão do* C++) temos de:

- ter acesso às declarações dos métodos contidos na biblioteca (*header files*, `.hpp`);
- ter acesso ao código, na forma de um ficheiro já pré-compilado;
- adicionar as opções '`-l`' (de ***l****ibrary*), '`-L`' (de caminho para a ***L****ibrary*) e '`-I`' (de caminho para os ficheiros a incluir, ***I****nclude*) ao comando de compilação. A primeira destas opções indica-nos o nome da biblioteca (sem o prefixo '`lib`'). A segunda opção indica a directoria aonde se encontra a biblioteca (se não estiver integrada no sistema). A última opção indica-nos aonde se encontram os ficheiros com as declarações (`.hpp`).

A opção de compilação '`-l`' refere-se ao nome da biblioteca. É no entanto de notar que, por convenção, os nomes das bibliotecas começam todos por '`lib`', por exemplo `libBibPOO.so`, sendo que o prefixo '`lib`' não faz parte do nome da biblioteca, isto é, ter-se-ia `-lBibPOO`.

As bibliotecas podem ser de dois tipos no que diz respeito à sua utilização.

Bibliotecas Dinâmicas consistem de métodos que são pré-compilados e que são carregadas somente aquando da execução do programa. A sua extensão em sistemas *Linux* é a extensão '`so`', de objectos partilhados (***s****hared* ***o****bject*).

Bibliotecas Estáticas consistem de métodos que são pré-compilados e agregados ao programa aquando da compilação deste último. A sua extensão usual em sistemas *Linux* é a extensão '`a`', de arquivo (***a****rchive*).

A grande vantagem das bibliotecas dinâmicas é a sua separação do programa compilado, só na altura da execução é que são chamadas a intervir. As vantagens desta aproximação são múltiplas:

[3] https://www.sqlite.org/

- O ficheiro (executável) do programa não contém o código da biblioteca, em consequência é de menor dimensão do que no caso contrário (bibliotecas estáticas).
- A biblioteca pode sempre ser actualizada sem que isso signifique uma recompilação do programa que a usa.
- A biblioteca pode ser usada por muitos programas, inclusive simultaneamente.

A desvantagem advêm da dependência do programa (para a sua execução) da biblioteca. Num sistema em que a biblioteca dinâmica não esteja instalada os programas que dependem dela não funcionarão.

Isto é, se se quer providenciar uma solução auto-contida, um programa que para funcionar já contém todo o código de que necessita. Então deve-se construir e usar bibliotecas estáticas. No caso contrário devem-se usar bibliotecas dinâmicas. Por omissão as bibliotecas contruídas/usadas são as bibliotecas dinâmicas.

C.2.2 Como Construir

Suponhamos que queríamos construir uma biblioteca que albergasse todos os tipos abstractos de dados construídos durante o decorrer da disciplina de programação orientada para os objectos. para simplificar consideremos somente as Filas e as Pilhas de inteiros.

Ter-se-iam os ficheiros: `filasInt.cpp, filasInt.hpp, pilhasInt.cpp pilhasInt.hpp` e finalmente um ficheiro contendo o programa no qual se pretende usar estas estruturas de dados, por exemplo `usaTADPOO.cpp`.

Bibliotecas Estáticas

No caso de se pretender construir uma biblioteca estática ter-se-ia de seguir os seguintes passos:

Construção

1. ter acessíveis os ficheiros de declarações (`.hpp`), ou no directório aonde se está a proceder à compilação, ou num directório específico;
2. compilar normalmente os ficheiros contendo as classes que vão fazer parte da biblioteca

```
g++ -c filasInt.cpp pilhasInt.cpp
```

3. criar o arquivo que agrega as várias componentes da biblioteca.

```
ar cq libtadpoo.a filasInt.o pilhasInt.o
```

   O programa `ar` é utilizado para criar, modificar e extrair componente de arquivos. No caso em concreto cria a biblioteca estática `libtabpoo.a`.

   A opção '`c`' significa o `c`riar de um novo arquivo e a opção '`q`' (de `q`*ueing*) significa o colar das diferentes componentes umas atrás das outras (em fila).

Inclusão num Programa

1. fazer a inclusão do(s) ficheiro(s) de declarações do(s) «livro(s)» que se pretende usar, por exemplo:

```
#include "pilhasInt.hpp"
```

2. A compilação do programa principal e consequente criação do executável pode então ser feita da seguinte forma:

```
g++ -o usaTADPOO usaTADPOO.cpp -ltabpoo -L. -I.
```

   em que a opção '`-l`' permite-nos identificar a biblioteca que se pretende usar, a opção '`-L`' indica-nos aonde é que a mesma se encontra, e a opção '`-I`' indica-nos aonde é que os ficheiros de declaração se encontram, neste exemplo o directório em que se está a fazer a compilação '`.`'.

O programa resultante da compilação é auto-contido (a menos das bibliotecas do sistema) e como tal a sua utilização, num qualquer sistema computacional, não necessita da existência da biblioteca no referido sistema computacional.

Bibliotecas Dinâmicas

No caso das bibliotecas dinâmicas não só o processo de compilação é um pouco mais complexo como a própria utilização posterior do programa não é auto-contida. No caso dos programas criados com o auxílio de bibliotecas dinâmicas estas (o seu código) têm de estar presentes também na altura da execução. Vejamos para o exemplo acima referido como proceder.

Construção O primeiro passo é idêntico ao caso anterior.

1. ter acessíveis os ficheiros de declarações (`.hpp`), ou no directório aonde se está a proceder à compilação, ou num directório específico;

2. A compilação das componentes que vão fazer parte da biblioteca necessita do explicitar de uma nova opção:

```
g++ -c -fPIC filasInt.cpp pilhasInt.cpp
```

   Como a biblioteca dinâmica é carregada somente no momento da execução o seu código (máquina) tem de ser independente da localização, '`-fPIC`' significa o criar de código independente da posição ***P**osition-**I**ndependent **C**ode*.

3. Criar a biblioteca dinâmica:

```
g++ -shared -o libtadpoo.so.1.0 filasInt.o pilhasInt.o
```

   A opção '`-shared`' é necessária para criar o código partilhável, próprio das bibliotecas dinâmicas.

   Após este ponto podemos remover todos os ficheiros '`.o`' criados até este ponto.

Os números '`1.0`' após a extensão '`.so`' seguem a convenção usual para *versão principal* e *versão secundária*.[4]

4. A compilação e, posteriormente, a execução com bibliotecas dinâmicas necessitam de aceder à biblioteca dinâmica, sendo que a forma de o fazer difere de plataforma para plataforma. A convenção para sistemas *Linux* passa pela criação de duas referências simbólicas (*symbolic links*).

```
ln -sf libtadpoo.so.1.0 libtadpoo.so
ln -sf libtadpoo.so.1.0 libtadpoo.so.1
```

Inclusão num Programa

1. fazer a inclusão do(s) ficheiro(s) de declarações do(s) «livro(s)» que se pretende usar, por exemplo:

```
#include "pilhasInt.hpp"
```

2. A compilação do programa principal é então feita de forma similar à usada para as bibliotecas estáticas (explicitando o caminho para as *header files*):

```
g++ -o usaTADPOO usaTADPOO.cpp -ltadpoo -L. -I.
```

Os ficheiros de declarações, assim como o ficheiro contendo o código da biblioteca, podem estar num directório do utilizador, numa área comum a todos os utilizadores[5], ou no sistema. Para os dois primeiros casos as opções '`-L`' e '`-I`' são estritamente necessárias.

Utilização do Programa Final

1. Para a execução do programa que, ao contrário do caso das bibliotecas estáticas, não é auto-contido, pode ser necessário explicitar aonde se encontra a biblioteca. Faz-se isso através da variável de ambiente '`LD_LIBRARY_PATH`'. A sintaxe ("bash"[6]) para actualizar o seu valor é o seguinte:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

após o qual a execução do programa decorre normalmente.

Dependendo do sistema operativo e/ou compilador usado a nova biblioteca e respectivos ficheiros de declarações (*header files,* `.hpp`) podiam ser instalados de forma a que pudessem ser acedidos a partir de uma qualquer directório do computador.

No caso de ambas as versões da biblioteca estarem presentes (a estática e a dinâmica) pode-se forçar a utilização da biblioteca estática através da opção '`-static`'. Por exemplo:

[4]Embora não seja único esquema possível, é usual usar o seguinte esquema para a numeração de versões de programas computacionais: *l.m.n*, o primeiro número indica que o sistema tem mudanças que o torna incompatível com versões anteriores, o segundo número indica que o sistema tem mudanças compatíveis com versões anteriores, dentro do primeiro número, o terceiro número indica que o sistema tem mudanças menores, como correções de erros e funcionalidades que não prejudicam a compatibilidade com versões anteriores.

[5]Em sistemas *Linux*: `/usr/local/`.

[6]`https://www.gnu.org/software/bash/`

```
g++ -o usaTADPOO usaTADPOO.cpp -L. -static -ltadpoo
```

por omissão as bibliotecas dinâmicas são as utilizadas.

C.3 Makefile

O programa `make` é um utilitário presente em todos os sistemas do tipo *Unix* (*Linux, MacOS*, etc.) assim como em outros sistemas em que se instale um compilador de *C/C++*. Este programa permite automatizar o processo de compilação, sendo possível especificar o que se pretende através da construção de um ficheiro designado `Makefile`. De seguida apresenta-se, de forma sumária, o programa `make` e a forma como construir o ficheiro `Makefile`. Para uma exposição mais completa consulte (Mecklenburg, 2004; Darcano, 2007), ou aceda ao manual do programa[7].

C.3.1 O Programa `make`

O programa `make` é uma maneira muito conveniente de gerir grandes programas ou grupos de programas. Quando se começa a escrever programas cada vez maiores e visível a diferença de tempo necessário para recompilar esses programas em comparação com programas menores. Por outro lado, normalmente trabalha-se apenas em uma pequena parte do programa (tal como uma simples função que se está depurando), e grande parte do resto do programa permanece inalterada.

O programa `make` ajuda na manutenção desses programas observando quais partes do programa foram mudadas desde a última compilação e recompilando apenas essas partes.

Para isso, é necessário que se escreva uma «`Makefile`», que é um arquivo de texto responsável por dizer ao programa make "o que fazer" e contém o relacionamento entre os arquivos fonte, objecto e executáveis.

Outras informações úteis colocadas no `Makefile` são as «`flags`» que precisam ser passados para o compilador e o agregador (*linkador*), como directórios onde encontrar arquivos de cabeçalho (arquivos `.hpp`), com quais bibliotecas o programa deve ser agregado, etc. Isso evita que se precise escrever linhas de comando enormes incluindo essas informações para compilar o programa.

O programa `make` pode ser invocado de duas formas distintas (com '`$>`' a representar a linha de comandos do sistema):

- `$> make`, neste caso o primeiro dos objectivos contidos no ficheiro `Makefile` é executado.
- `$> make <objectivo>`, neste caso especifica-se qual o objectivo que se pretende executar.

Vejamos então como construir um ficheiro `Makefile`.

C.3.2 O Ficheiro `Makefile`

Uma `Makefile` contém essencialmente comentários, «macros» (regras de substituição) e objectivos (*targets*).

[7] `man make`; `http://www.gnu.org/software/make/manual/make.html`

Comentários: Os comentários são delimitados pelo carácter "#" e pelo fim-de-linha respectivo. Por exemplo

```
# Análise sintáctica dos ficheiros de resultados
```

Macros As macros são um simples mecanismo de substituição sintáctica. Permitem ajustar de uma forma simples o processo de compilação a diferentes ambientes de compilação. Permitem também a construção de um ficheiro mais fácil de ler e compreender. Por exemplo: especificar o compilador a usar (`g++`) e a inclusão da biblioteca matemática (`-lm`).

```
CC     = g++
FLAGS  = -lm
```

As macros são especificadas da seguinte forma:

```
<nome> = <valor>
```

sendo que por convenção (usualmente aceite) os nomes das macros são escritos utilizando somente maiúsculas.

Na `Makefile`, expressões da forma `$(<nome>)` ou `${<nome>}` são automaticamente substituídas pelo valor correspondente.

Objectivos Os objectivos determinam a acção a ser efectuada quando se executa o programa `make`. Como foi dito acima se o «chamar» do programa for feito sem argumentos o primeiro dos objectivos é aquele que vai ser processado, caso contrário só o objectivo invocado é que é processado. No caso do objectivo invocado não estar especificado na `Makefile` ocorre uma situação de erro e nada é processado. Por exemplo:

```
all:    exer1 exer2 exer3

clean:
        -rm *.o exer1 exer2 exer3
```

o carácter '-' antes do comando `rm` (*remove*, apagar ficheiros) tem como finalidade o forçar a continuação do comando mesmo na eventualidade da ocorrência de um erro, por exemplo um dos ficheiros que se pretende apagar não existir.

Os objectivos são especificados da seguinte forma:

```
<objectivo1> <objectivo2> ... : <dependência1> <dependência2> ...
<espaço_tabular> <comando1>
<espaço_tabular> <comando2>
 ...
```

é de notar que o espaço tabular (tecla *Tab*) é obrigatório.

Caso uma das linhas respeitantes a um dos comandos seja muito comprida pode-se continuar a mesma (para efeitos de facilitar a sua leitura) por tantas linhas quanto o necessário. Neste casos é obrigatório colocar o carácter '\' (linha de continuação) entre no fim das linhas que tenham continuação.

Um tipo de objectivo especial é o assim designado, falsos objectivos (*phony target*). Este tipo de objectivos não está associado a um nome de ficheiro e é usualmente usado para evitar eventuais confusões entre nomes de ficheiros e nomes de objectivos. Por exemplo é usual ter-se:

```
.PHONY: clean all
```

Macros Especiais O programa `make` tem um conjunto muito grande de macros pré-definidas[8] das quais convêm destacar as seguintes:

- `$@`, é o nome do ficheiro a ser produzido (nome do objectivo);
- `$?`, são os nomes dos dependências que foram alteradas;
- `$<`, é o nome do ficheiro que causou a acção;
- `$*`, é o prefixo comum aos ficheiros resultantes e suas dependências.

Exemplos de `Makefiles` Um primeiro exemplo englobando os exemplos usados anteriormente:

```
FLAGS  = -lm
CC     = g++

.PHONY: all clean

all:    exer1 exer2

clean:
        -rm *.o exer1a exer1b

exer1a: exer1a.cpp
        ${CC} -o $@ $@.cpp ${FLAGS}

exer1b: exer1b.cpp olaMundo.o
        ${CC} -o $@ $@.cpp olaMundo.o ${FLAGS}
```

este exemplo pode ser visto com uma `Makefile` para automatizar a compilação de todos os exercícios, acrescentando objectivos à medida que resolvemos novos exercícios, contidos no apêndice E.

C.4 Ambientes Integrados de Desenvolvimento

Ambientes Integrados de Desenvolvimento, IDE, sigla formada pelas iniciais de *Integrated Development Environment* são sistemas que procuram responder a algumas das questões importantes que se colocam aquando do desenvolvimento de programas:

- a escrita dos programas através de um editor de textos dedicado;
- a compilação do programa, eventualmente envolvendo mais (muito mais) do que um ficheiro;
- a depuração dos erros;
- a documentação.

Vejamos isso ponto a ponto.

[8] Para saber quais fazer `prompt make -p`

C.4.1 Editor de Textos Dedicado

Antes de mais é de esclarecer que estamos a falar de um editor de textos e não de um processadores de texto. Estes últimos manipulam o texto acrescentando muita informação necessária ao processamento do texto, nomeadamente todas as questões relacionadas com a formatação, o que é totalmente inapropriado quando se está a querer construir algo que vai ser submetido ao tratamento automático de um compilador.

Como primeira conclusão: não tente usar um processador de textos para a escrita de um qualquer programa. É necessário usar um editor de textos.

Um editor de texto dedicado é um editor (não acrescenta nada ao ficheiro contendo o código do programa) mas que conhece a linguagem que se está a usar. Isto é um editor deste tipo tem um conhecimento da linguagem que passa por conhecer a sua estrutura léxica e gramatical, o que leva a que possa:

- fazer a verificação sintáctica à medida que se escreve. Em geral isso reflecte-se num código de cores, isto é, a atribuição diferentes cores aos tipos de dados, aos identificadores da linguagem, aos comentários, às sequências de caracteres, etc;

- fazer a verificação gramatical à medida que se escreve, o que se reflecte na forma como a indentação (os diferentes níveis de alinhamento do texto) é feita;

Pode parecer pouco mas, dado que um programa não é mais do que um texto numa dada linguagem, texto esse que vai ser processada automaticamente por um programa, o qual é muito pouco, ou mesmo nada, tolerante aos erros sintácticos e gramaticais, é fácil de perceber que toda a ajuda é muito importante.

Quando algo não está na cor que é suposto estar... algo está sintacticamente errado, por exemplo um identificador da linguagem que está escrito de forma errada.

Quando algo não se ajusta (indenta) para a forma correcta... algo está gramaticalmente errado, por exemplo um esquecimento de um ";" no fim de uma instrução.

C.4.2 Compilação & Makefiles

Desde a compilação simples através da simples escolha de uma opção, até à construção e utilização, com diferentes graus de automatismo, de "makefiles" para a compilação de programas multi-ficheiros.

C.4.3 Depuração de Erros

Desde a indicação e posicionamento automático, da linha aonde os erros ocorrem, até à ligação com programas específicos para o depuramento de erros.

C.4.4 Documentação

No caso do *C++* uma ajuda importante é o de mostrar a estrutura das classes constituintes do programa. Pode passar mesmo pela construção do UML respectivo.

C.4.5 Alguns IDEs para o *C/C++*

É importante que o ambiente de programação escolhido seja:

- Apropriado (talvez mesmo especializado) para o ambiente de trabalho (leia-se empresa) em que se está.
- Multi-plataforma, isto é, esteja disponível em diferentes plataformas computacionais computador/sistema operativo.
- Multi-linguagem, isto é, que seja capaz de se adaptar aos requisitos das diferentes linguagens de programação.

Obviamente que estes dois requisitos aplicam-se a duas situações bem distintas. O primeiro quando se está a trabalhar numa equipa com hábitos e objectivos bem definidos, aí é importante que, para uma maior eficiência, todos utilizem a mesma plataforma de desenvolvimento.

Quando se tem uma situação mais difusa é importante que utilizemos uma plataforma que melhor se adapte às mudanças de plataforma computacional usada e/ou de linguagem de programação usada.

Três aproximações que se adaptam a estes requisitos são:

(X)emacs neste caso estamos perante um editor de textos não exactamente um IDE completo. Dado ser programável (em EmacsLisp) é altamente configurável, o seu modo específico para o *C/C++* é muito bom. Dado não ser um IDE as tarefas de compilação e de depuração de erros não estão integradas. De aprendizagem algo difícil é no entanto um dos (se não o) editor de texto mais flexível e poderoso existente. Multi-plataforma, multi-linguagens de programação, código aberto. `http://www.xemacs.org/`, `http://www.gnu.org/software/emacs/`.

Geany é um IDE bastante completo, o seu editor, embora não tão poderoso como o (X)emacs é mesmo assim muito bom. A construção de Makefiles não é automática, a ligação a um depurador de erros é possível embora necessite configuração. Muito fácil de usar. Multi-plataforma, multi-linguagens de programação, código aberto. `http://www.geany.org/`.

CodeBlocks mais poderoso do que os anteriormente referidos, incorpora a noção de projecto com a automatização da compilação (que não uma Makefile). É um pouco mais complexo que o anterior. Multi-plataforma, específico para a linguagem *C/C++*, código aberto. `http://www.codeblocks.org/`

Eclipse CDT É um Ambiente de Desenvolvimento Integrado para o *C/C++* baseado na plataforma *Eclipse* (para Java). Muito completo, tendo inclusive ferramentas visuais de depuração de erros. Multi-plataforma `http://www.eclipse.org/cdt/`.

Apêndice D

Exemplos

Nesta apêndice apresenta-se o código referente aos vários exemplos descritos nos diferentes capítulos destes apontamentos.

D.1 Exemplos Referentes ao Capítulo 1

Exemplos de programas em diferentes linguagens representativas de diferentes paradigmas de programação.

D.1.1 Haskell

Implementação da pilha genérica possível pela introdução de variáveis de tipo (`Pilha a`). Note-se o tratamento de erros em `top` e `pop`.

```
module Pilha(Pilha,pop,top,push,evazia) where

data Pilha a = Pvazia | Pl (a,Pilha a)
                    deriving (Show)

top :: Pilha a -> a
top Pvazia = error "pilha vazia"
top (Pl(a,p)) = a

pop :: Pilha a -> Pilha a
pop Pvazia = error "pilha vazia"
pop (Pl(a,p)) = p

push :: (a,Pilha a) -> Pilha a
push (a,p) = Pl(a,p)

evazia :: Pilha a -> Bool
evazia Pvazia = True
evazia _ = False

vazia :: () -> Pilha a
vazia() = Pvazia
```

A utilização do módulo `Pilha` instanciando-o para um dado tipo concreto pode de seguida ser feito em um outro qualquer módulo Haskell.

```
module UsaPilhas where

import Pilha(Pilha,pop,top,push,evazia)

-- lineariza pilha
lineariza :: Pilha Int -> [Int]
lineariza p
    | evazia(p) = []
    | otherwise = top(p):lineariza(pop(p))
```

D.1.2 CafeOBJ

Implementação da pilha genérica e sua instanciação para a pilha de naturais. Note-se a definição de uma hierarquia de tipos com a criação dos tipos `PilhaNV` (pilha não vazia) e `Pilha` (pilha).

```
module* ELT { [Elt] }

module! PILHA (X :: TRIV) {
  signature {
      [Elt < PilhaNV < Pilha]
      op vazia : -> Pilha
      op push : Elt Pilha -> PilhaNV
      op pop : PilhaNV -> Pilha
      op top : PilhaNV -> Elt
      op vazia? : Pilha -> Bool
    }
  axioms {
      var E : Elt
      var P : Pilha
      eq pop(push(E,P)) = P .
      eq top(push(E,P)) = E .
      eq vazia?(vazia) = true .
      eq vazia?(push(E,P)) = false .
    }
}

view NAT_AS_ELT from ELT to NAT {
  sort Elt -> Nat
}

module PILHA_NAT {
  protecting (PILHA(NAT_AS_ELT))
}
```

D.2 Exemplos Referentes ao Capítulo 4

A classe `<limits>` dá-nos acesso aos limites númericos para cada um dos tipos númericos disponíveis. O programa a seguir apresentado dá-nos um exemplo de como obter os limites para a versão do compilador e sistema operativo que se está a usar.

```
#include <iostream>
#include <limits>

using namespace std;

int main () {
  // char
  cout << "Número de 'bytes' por 'char': " << sizeof(char) << endl;
  // short
  cout << "Número de 'bytes' por 'short': " << sizeof(short) << endl;
  cout << "Gama de variação: " << numeric_limits<short>::min()
       << " a " << numeric_limits<short>::max() << endl;
  // int
  cout << "Número de 'bytes' por 'int': " << sizeof(int) << endl;
  cout << "Gama de variação: " << numeric_limits<int>::min()
       << " a " << numeric_limits<int>::max() << endl;
  // long int
  cout << "Número de 'bytes' por 'long int': " << sizeof(long) << endl;
  cout << "Gama de variação: " << numeric_limits<long int>::min()
       << " a " << numeric_limits<long int>::max() << endl;
  // long long int
  cout << "Número de 'bytes' por 'long long int': " << sizeof(long long) << endl;
  cout << "Gama de variação: " << numeric_limits<long long int>::min()
       << " a " << numeric_limits<long long int>::max() << endl;
  // float
  cout << "Número de 'bytes' por 'float': " << sizeof(float) << endl;
  cout << "Gama de variação (positivos):" << numeric_limits<float>::min()
       << " a " << numeric_limits<float>::max() << endl;
  cout << "Grau de precisão: " << numeric_limits<float>::epsilon() << endl;
  // double
  cout << "Número de 'bytes' por 'double': " << sizeof(double) << endl;
  cout << "Gama de variação (positivos): " << numeric_limits<double>::min()
       << " a " << numeric_limits<double>::max() << endl;
  cout << "Grau de precisão: " << numeric_limits<double>::epsilon() << endl;
  // long double
  cout << "Número de 'bytes' por 'long double': " << sizeof(long double) << endl;
  cout << "Gama de variação (positivos): " << numeric_limits<long double>::min()
       << " a " << numeric_limits<long double>::max() << endl;
  cout << "Grau de precisão: " << numeric_limits<long double>::epsilon() << endl;

  return(0);
}
```

Para obter a versão do compilador que se está a usar basta usar a opção `-v`. Por exemplo:

```
$> g++ -v
gcc version 4.7.2 (Debian 4.7.2-5)
```

D.3 Exemplos Referentes ao Capítulo 5

Exemplos de implementações de hierarquias de classes.

D.3.1 Exemplo da Secção 5.2.1

Duas Classes e com Utilização dos Construtores Um exemplo de herança simples com a implementação de duas das classes da hieraquia *Residência Universitária*, considerando os respectivos construtores. Um só ficheiro.

Ficheiro `residenciaUniversitaria.cpp`

```
// Exemplo de relação de herança com utilização dos construtores
#include <iostream>

using namespace std;

class Residente { // classe de base
public:
  Residente(){}; // construtor sem argumentos
  // construtor com dois argumentos
  Residente(string _nome, unsigned long _cc_num) {
    nome = _nome;
    cc_num = _cc_num;
  }
  void mostraInfo() {
    // Pré: Informação de Residente já está definida
    // Pós: Mostra a informação do Residente: nome e CC.
    cout << endl << "Nome: " << nome;
    cout << endl << "Cartão de Cidadão: " << cc_num;
  }
protected:
  string nome;       // Nomes dos Residentes
  unsigned long cc_num;  // CCs dos Residentes
};

// Empregado derivação pública, da classe Residente
class Empregado : public Residente {
public:
  Empregado() : Residente() {} // construtor sem argumentos
  Empregado(string _nome, unsigned long _cc_num, unsigned long _empr_num)
    : Residente(_nome, _cc_num) {
    // Construtor com três argumentos, dois deles são passados
    // directamente ao construtor da classe base
    empr_num = _empr_num; // Guarda o número de Empregado
  }
  void mostraInfo() {
    // Pré: A informação do Empregado já foi definida
    // Pós: A informação do Empregado é visualizada
    Residente::mostraInfo();
    cout << "\nNúmero de Empregado: " << empr_num << endl;
  }
protected:
  unsigned long empr_num;
};

int main() {
  Residente r1;
  Empregado e1;   // construtor sem argumentos
  string nome;
  unsigned long numero, empr_num;

  cout << "Forneça a informação para o Residente 2 " ;
  cout << endl << "Nome (sem espaços): ";
  cin >> nome;
  cout << endl << "Número de Cartão de Cidadão: ";
  cin >> numero;
  cout << "\nNúmero de Empregado (5 dígitos): ";
  cin >> empr_num;
```

```
  Empregado e2(nome, numero, empr_num);  // construtor de três argumentos

  cout << "\nInformação do Empregado 1: (objecto não inicializado)\n";
  e1.mostraInfo();
  cout << endl << "\nInformação para o Empregado 2: \n";
  e2.mostraInfo();
  e1 = e2;   // Construtor cópia por omissão
  cout << "\nMostra resultados do construtor por cópia"
       << " definido por omissão\n";
  e1.mostraInfo();

  Residente r2 = e1;     // atribuição de um objecto derivado para um
                         // objecto da classe de base é permitido
  cout << "\nMostra o resultado de uma atribuição de um objecto derivado"
       << "  para um objecto da classe de base\n";
  r2.mostraInfo();
  cout << endl;

  return 0;
}
```

D.3.2 Exemplo da Secção 5.2.1

A Hierarquia Completa e com Utilização dos Construtores Um exemplo de herança simples com a implementação da hierarquia *Residência Universitária*.

Ficheiro `residencia.cpp`

```
/*
 *  Residência Universitária - exemplo de herança simples com dois
 *  níveis (utiliza construtores)
 */
#include <iostream>
#include "residentes.hpp"
#include "estudantes.hpp"
#include "empregados.hpp"

using namespace std;

/*
 * Programa de teste da hierarquia
 *
 * Residentes <- Estudantes <- EstudanteMestDout
 *            <- Empregados <- Professores
 */
int main() {

  string nome,especializacao;
  unsigned long cc,numEmpr;
  unsigned int numOrientacoes;
  Professor prof1;

  cout << "Informação para o Professor 1" ;
  cout << endl << "Nome (uma linha de texto): ";
```

```
  getline (cin,nome); // Extrai caracteres de IS e guarda-os numa
                      // string até atingir um fim-de-linha.
  cout << "Número do Cartão de Cidadão: ";
  cin >> cc;
  cout << "Número de Funcionário: ";
  cin >> numEmpr;
  cout << "Especialização (uma linha de texto): ";
  cin.ignore(10, '\n'); // necessário para "limpar" a entrada e
                        // permitir a leitura de uma nova linha de
                        // texto, ignora caracteres (no máximo 10) até um \n
  getline (cin,especializacao);
  cout << "Número de Estudantes de Mestrado/Doutoramento: ";
  cin >> numOrientacoes;

  prof1.fixaInfo(nome,cc,numEmpr,especializacao,numOrientacoes);
  cout << "\n\nMostra a informação para o Professor 1: \n";
  prof1.mostraInfo();

  return 0;
}
```

Ficheiro `residentes.hpp`

```
#ifndef RESIDENTES
#define RESIDENTES

/*
 * Residentes <- Empregados <- Professores
 */
#include <iostream>

using namespace std;

class Residente {
public:
  // construtor sem argumentos
  Residente();
  /* construtor com dois argumentos
   *  Pré: Sem pré-requisitos
   *  Pós: Definição da informação de um residente (nome e número de
   *  Cartão de Cidadão).
   */
  Residente(string, unsigned long);

  /*
   * Fixa (após) inicialização a informação de um residente.
   *
   * Pré: o objecto já foi inicializado (construtor)
   * Pós: fixa (ou actualiza) a informação de um Residente.
   */
  void fixaInfo(string, unsigned long);

  /*
   * Mostra a informação de um residente.
   *
   * Pré: A informação de um Residente já está definida.
```

```
    * Pós: Mostra a informação de um Residente.
    */
  void mostraInfo();
protected:
  string nome; // Nome do Residente
  unsigned long cc; // CC do Residente
};

#endif
```

Ficheiro `residentes.cpp`

```
// Residentes
#include <iostream>
#include "residentes.hpp"

using namespace std;

Residente::Residente() {};

Residente::Residente(string _nome, unsigned long _cc) {
  nome = _nome; // nome do residente
  cc = _cc;     // número do cartão do cidadão
};

void Residente::fixaInfo(string _nome, unsigned long _cc) {
  nome = _nome;
  cc = _cc;
};

void Residente::mostraInfo() {
    cout << endl << "Nome: " << nome;
    cout << endl << "Cartão de Cidadão: " << cc;
}
```

Ficheiro `empregados.hpp`

```
/*
 * Residentes <- Empregados <- Professores
 */
#include "residentes.hpp"

using namespace std;

// Empregado classe derivada publicamente da classe Residente
class Empregado : public Residente {
public:
  // construtor sem argumentos
  Empregado();
  /*
   * Construtor com três argumentos, dois deles são passados
   * directamente ao construtor da classe
   */
  Empregado(string, unsigned long, unsigned long);
```

```
  /*
   * Fixa (após inicialização) a informação respeitante a um Empregado
   *
   * Pré: o objecto já foi inicializado (construtor)
   * Pós: fixa (ou actualiza) a informação de um Empregado.
   */
  void fixaInfo(string, unsigned long, unsigned long);

  /*
   * Mostra a informação respeitante a um Empregado
   *
   * Pré: A informação de Empregado já está definida.
   * Pós: Mostra a informação de um Empregado.
   */
  void mostraInfo();
protected:
  unsigned long numEmpr;
};

// Professor classe derivada publicamente da classe Empregado
class Professor : public Empregado {
public:
  // construtor sem argumentos
  Professor();
  // Construtor com cinco argumentos, três deles são passados
  // directamente ao construtor da classe base
  Professor(string, unsigned long, unsigned long, string, unsigned int);

  /*
   * Fixa (após inicialização) a informação respeitante a um Professor
   *
   * Pré: o objecto já foi inicializado (construtor)
   * Pós: fixa (ou actualiza) a informação de um Professor
   */
  void fixaInfo(string, unsigned long, unsigned long, string, unsigned int);

  /*
   * Mostra a Informação respeitante a um Professor
   *
   * Pré: A informação de um Professor já está definida.
   * Pós: Mostra a informação de um Professor
   */
  void mostraInfo();
protected:
  string especializacao;
  int numOrientacoes;
};
```

Ficheiro `empregados.cpp`

```
/*
 * Residentes <- Empregados <- Professores
 */
#include <iostream>
#include "residentes.hpp"
```

```
#include "empregados.hpp"

using namespace std;

/*
 * Empregados
 */
Empregado::Empregado() : Residente() {};

Empregado::Empregado(string _nome, unsigned long _cc,
                     unsigned long _numEmpr)
  : Residente(_nome, _cc) {
  numEmpr = _numEmpr; // Guarda o número de empregado
};

void Empregado::fixaInfo(string _nome, unsigned long _cc,
                         unsigned long _numEmpr){
  Residente::fixaInfo(_nome,_cc);
  numEmpr = _numEmpr; // Guarda o número de empregado
};

void Empregado::mostraInfo(){
  Residente::mostraInfo();
  cout << "\nNúmero de Empregado: " << numEmpr;
};

/*
 * Professores
 */
Professor::Professor() : Empregado() {};

Professor::Professor(string _nome, unsigned long _cc, unsigned long _numEmpr,
                     string _especializacao, unsigned int _numOrientacoes)
  : Empregado(_nome, _cc, _numEmpr) {
  especializacao = _especializacao; // Guarda a especialização
  numOrientacoes = _numOrientacoes; // Guarda o número de orientações
};

void Professor::fixaInfo(string _nome, unsigned long _cc, unsigned long _numEmpr,
                         string _especializacao, unsigned int _numOrientacoes ){
  Empregado::fixaInfo(_nome,_cc,_numEmpr);
  especializacao = _especializacao; // Guarda a especialização
  numOrientacoes = _numOrientacoes; // Guarda o número de orientações
};

void Professor::mostraInfo() {
  Empregado::mostraInfo();
  cout << "\nEspecialização:  " << especializacao;
  cout << "\nNúmero de Estudante de Mestrado/Doutoramento: "
       << numOrientacoes << endl;
};
```

Ficheiro `estudantes.hpp`

```
/*
 * Residentes <- Empregados <- Professores
```

```
 */
#include <iostream>
#include "residentes.h"

using namespace std;

//Estudante classe derivada publicamente de Residente
class Estudante : public Residente  {
public:
  // construtor sem argumentos
  Estudante();
  // Construtor com quatro argumentos, dois deles são passados
  // directamente ao construtor da classe base
  Estudante(string, unsigned long, string, unsigned long);

  /*
   * Fixa (após inicialização) a informação respeitante a um Estudante
   *
   * Pré: o objecto já foi inicializado (construtor)
   * Pós: fixa (actualiza) a informação de um Estudante
   */
  void fixaInfo(string, unsigned long, string, unsigned long) ;

  /*
   * Mostra a Informação respeitante a um Estudante
   *
   * Pré: A informação de um Estudante já está definida.
   * Pós: Mostra a informação de um Estudante
   */
  void mostraInfo() ;
protected:
  string nomeDep;
  unsigned long numAluno;
};

/*
 * EstudanteMestDout classe derivada publicamente da classe
 * Estudante
 */
class EstudanteMestDout : public Estudante {
public:
  // construtor sem argumentos
  EstudanteMestDout();
  // Construtor com seis argumentos, três deles são passados
  // directamente ao construtor da classe base
  EstudanteMestDout(string,unsigned long,string,unsigned long,string,string);

  /*
   * Fixa (após inicialização) a informação respeitante a um Estudante
   * de Mestrado/Doutoramento
   *
   * Pré: o objecto já foi inicializado (construtor)
   * Pós: fixa (actualiza) a informação de um Estudante de
   *       Mestrado/Doutoramento
   */
  void fixaInfo(string, unsigned long, string, unsigned long, string, string) ;
```

```
  /*
   * Mostra a Informação respeitante a um Estudante de Mestrado ou
   * Doutoramento
   *
   * Pré: A informação de um Estudante já está definida.
   * Pós: Mostra a informação de um Estudante
   */
  void mostraInfo();
protected:
  string nomePrg;
  string supervisor;
};
```

Ficheiro `estudantes.cpp`

```
#include <iostream>
#include "residentes.hpp"
#include "estudantes.hpp"

using namespace std;

/*
 * Estudante - classe derivada publicamente de Residente
 */

Estudante::Estudante() : Residente() {};

Estudante::Estudante(string _nome, unsigned long _cc, string _nomeDep,
                     unsigned long _numAluno) : Residente(_nome, _cc) {
    nomeDep = _nomeDep; // guarda o nome do Departamento
    numAluno = _numAluno; // guarda o número de aluno
  };

void Estudante::fixaInfo(string _nome, unsigned long _cc, string _nomeDep,
                         unsigned long _numAluno) {
  Residente::fixaInfo(_nome, _cc);
  nomeDep = _nomeDep; // actualiza o nome do Departamento
  numAluno = _numAluno; // actualiza o número de aluno
}

void Estudante::mostraInfo() {
  Residente::mostraInfo();
  cout << "\nNome do Departmento: " << nomeDep;
  cout << "\nNúmero de Estudante: " << numAluno << endl;
}

/*
 * EstudanteMestDout
 */
EstudanteMestDout::EstudanteMestDout() : Estudante() {} ;

EstudanteMestDout::EstudanteMestDout(string _nome, unsigned long _cc,
                                     string _nomeDep, unsigned long _numAluno,
                                     string _nomePrg, string _supervisor)
  : Estudante(_nome, _cc, _nomeDep, _numAluno){
  nomePrg = _nomePrg; // Guarda o nome do Departamento
```

```
  supervisor = _supervisor; // Guarda o número de aluno
};

void EstudanteMestDout::fixaInfo(string _nome, unsigned long _cc,
                                 string _nomeDep, unsigned long _numAluno,
                                 string _nomePrg, string _supervisor) {
  Estudante::fixaInfo(_nome, _cc, _nomeDep, _numAluno);
  nomePrg = _nomePrg;
  supervisor = _supervisor;
}

void EstudanteMestDout::mostraInfo() {
  Estudante::mostraInfo();
  cout << "\nNome Programa (Mestrado/Doutoramento): " << nomePrg;
  cout << "\nSupervisor: " << supervisor;
}
```

D.3.3 Exemplo da Secção 5.2.2

Herança Múltipla A implementação de uma hierarquia *família* como forma de exemplificar as herança múltipla.

Ficheiro pais.hpp

```
#ifndef PAIS
#define PAIS

using namespace std;

class Pai {
public:
  Pai();       // Construtor sem argumentos
  Pai(string); // Construtor com um só argumento, tipo string
  Pai(int);    // Construtor com um só argumento, tipo int
  Pai(string, int);  // Construtor com um dois argumentos

  /*
   * Insere/actualiza a informação do Pai
   * Pré: objecto já inicializado
   * Pós: actualiza a informação
   */
  void fixaInfo(string, int);

  /*
   * Mostra a informação do Pai
   * Pré: Informação sobre a pai já foi previamente definida
   * Pós: mostra a informação
   */
  void mostraInfo();
protected:
  string nome;
  int anoNascimento;
};

class Mae {
```

```
public:
  Mae();        // Construtor sem argumentos
  Mae(string); // Construtor com um só argumento, tipo string
  Mae(int);     //  Construtor com um só argumento, tipo int
  Mae(string, int);  // Construtor com um dois argumentos

  /*
   * Insere/actualiza a informação da Mãe
   * Pré: objecto já inicializado
   * Pós: actualiza a informação
   */
  void fixaInfo(string, int);

  /*
   * Mostra a informação da Mãe
   * Pré: Informação sobre a mãe já foi previamente definida
   * Pós: mostra a informação
   */
  void mostraInfo();
protected:
  string nome;
  int anoNascimento;
};
#endif
```

Ficheiro pais.cpp

```
#include <iostream>
#include "pais.hpp"

using namespace std;

Pai::Pai() {};
Pai::Pai(string _nome) {
  nome = _nome;
};
Pai::Pai(int _anoNascimento) {
  anoNascimento = _anoNascimento;
};
Pai::Pai(string _nome, int _anoNascimento) {
  nome = _nome;
  anoNascimento = _anoNascimento;
};

void Pai::fixaInfo(string _nome, int _anoNascimento) {
  nome = _nome;
  anoNascimento = _anoNascimento;
};

void Pai::mostraInfo() {
  cout << "\nO nome do Pai é: " << nome;
  cout << "\nA data de nascimento do Pai é: " << anoNascimento;
};

Mae::Mae() {}
Mae::Mae(string _nome) {
```

```
  nome = _nome;
};
Mae::Mae(int _anoNascimento) {
  anoNascimento = _anoNascimento;
};
Mae::Mae(string _nome, int _anoNascimento) {
  nome = _nome;
  anoNascimento = _anoNascimento;
};

void Mae::fixaInfo(string _nome, int _anoNascimento) {
  nome = _nome;
  anoNascimento = _anoNascimento;
};

void Mae::mostraInfo(){
  cout << "\nO nome da Mãe é: " << nome;
  cout << "\nA data de nascimento da Mãe é: " << anoNascimento;
};
```

Ficheiro filho.hpp

```
#ifndef FILHO
#define FILHO

#include "pais.hpp"

class Filho : public Mae, public Pai {
public:
  Filho();
  Filho(string, string, string); // Só nomes
  Filho(string, int, string, int, string); // nomes e anos de nascimento

  /*
   * Insere/actualiza a informação do Filho (e só deste)
   * Pré: objecto já inicializado
   * Post: actualiza a informação
   */
  void fixaInfoFilho(string, string, int, string, int);
  /*
   * Mostra a informação do Filho
   * Pré: Informação sobre a filho já foi previamente definida
   * Post: mostra a informação
   */
  void mostraInfo();

  /*
   * Mostra a informação do Filho
   * Pré: Informação sobre o filho (e dos pais)
   *      já foi previamente definida
   * Post: mostra a informação da família
   */
  void mostraInfoFamilia();
protected:
  string nome;
};
```

```
#endif
```

Ficheiro filho.cpp

```
#include <iostream>
#include "pais.hpp"
#include "filho.hpp"

using namespace std;

Filho::Filho() : Pai() , Mae() {}
Filho::Filho(string nomePai, string nomeMae, string _nome)
  : Pai(nomePai),Mae(nomeMae) {
  nome = _nome;
}
Filho::Filho(string nomePai, int anoNascPai, string nomeMae,
             int anoNascMae, string _nome) : Pai(nomePai,anoNascPai),
                                             Mae(nomeMae,anoNascMae) {
  nome = _nome;
}

void Filho::mostraInfo() {
  cout << "\nO nome do(a) Filho(a) é: " << nome;
}

void Filho::mostraInfoFamilia() {
  Pai::mostraInfo();
  Mae::mostraInfo();
  cout << "\nO nome do(a) Filho(a) é: " << nome;
}
```

Ficheiro familia.cpp

```
/*
 * Família
 */
#include <iostream>
#include "pais.hpp"
#include "filho.hpp"

using namespace std;

int main() {
  Filho filho1;

  cout << "\nUm objecto não inicializado (filho1): ";
  filho1.mostraInfo();

  // inicializar filho2;
  Filho filho2("João",1970,"Joana",1974,"João");

  cout <<  "\nMostra a informação do filho filho2:";
```

```
  filho2.mostraInfo();

  // atribuição de um objecto derivado para um
  // objecto da classe de base é permitido
  Mae m1 = filho2;
  Pai p1 = filho2;

  cout << "\nMostra informação da família (Pai+Mãe+Filho):\n";

  p1.mostraInfo();
  m1.mostraInfo();
  filho2.mostraInfo();
  cout << endl;

  cout << "\nMostra informação da família:\n";
  filho2.mostraInfoFamilia();
  cout << endl;

  return 0;
}
```

Apêndice E

Exercícios Práticos

E.1 Leitura/escrita

1 Escrever a frase "Olá Mundo".

2 Ler dois inteiros e escreve o valor da sua soma.

E.2 Condicionais

3 Ler dois números reais escreva o maior deles.

4 Ler um número inteiro, escreva o número indicando se é par ou ímpar.

5 Ler um valor real, x, calcular e escreva o valor da função:

$$f(x) = \begin{cases} \frac{1}{x}e^x & \text{se } x > 0 \\ e^{|x|} & \text{se } x \leq 0 \end{cases}$$

6 Dado x real, calcule $f(x)$ definida por:

$$f(x) = \begin{cases} e - e^{\cos x} & \text{se } x \in [0, 2\pi[\\ \log \cos x & \text{se } x \in [-2\pi, -\frac{3}{2}\pi[\bigcup]-\frac{\pi}{2}, 0[\end{cases}$$

7 Dado x real, calcule $f(x)$ definida por:

$$f(x) = \begin{cases} x(\cosh x)^{(2/x)} & \text{se } x > 0 \\ 0 & \text{se } x = 0 \\ x^2 \log(\frac{1-x}{x})^2 & \text{se } x < 0 \end{cases}$$

Nota: $\cosh x = \frac{e^x + e^{-x}}{2}$

8 Elabore um programa que calcule as raízes da equação quadrática de coeficientes inteiros:

$$ax^2 + bx + c = 0$$

9 Considere a equação da velocidade no instante t ($t \geq 0$):

$$v(t) = \frac{\log|2 - 2t + t^2| - e}{\cos^2[3(t-1)] + e^{-t+1}}$$

Elabore um programa que indique se, num instante $t0$ dado, o objecto se desloca sobre a respectiva trajectória no sentido positivo ou negativo, ou se muda de sentido.

Nota: quando $v > 0$, o objecto desloca-se no sentido positivo; quando $v < 0$, o objecto desloca-se no sentido negativo; quando $v = 0$, o objecto muda de sentido.

10 Considere a seguinte definição de uma função real de duas variáveis reais:

$$f(x,y) = \begin{cases} e^{\sin x} + e^{\cos y} & \text{se } x \geq 0 \text{ e } y \geq 0 \\ e^{-\sin x} - e^{-\cos y} & \text{se } x < 0 \text{ e } y \geq 0 \\ \log_e |\sin x + \cos y| & \text{se } x < 0 \text{ e } y < 0 \\ \log_e |\cos x - \sin y| & \text{se } x \geq 0 \text{ e } y < 0 \end{cases}$$

Elabore um programa que devolva o valor da função para valores x e y, expressos em *graus*, fornecidos pelo utilizador.

11 Pretende-se um programa que converta entre quilómetros por hora, milhas por hora e nós. O programa deve perguntar ao utilizador pelo valor e pelas unidades. Por exemplo:

```
Qual o valor: 13.7894
Quais as unidades (q = quilómetros por hora, m = milhas por hora, n = nós): m
```

Ao que o programa deve responder com o resultado:

```
Quilómetros por hora = 22.19189
nós = 11.98266
```

Recorde-se que: 1 milha = 1,609344Km e 1 nó = 1,852000Km por hora.

12 Pretende-se um programa que faça conversões entre graus e radianos. O programa deve perguntar ao utilizador qual o valor e as unidades do ângulo. Por exemplo:

```
Qual o valor do ângulo: 37,894
Quais as unidades (g = graus, r = radianos): g
```

Ao que o programa deve responder com o resultado:

```
Radianos = 0,6613751
```

Se o utilizador der um carácter não válido quando o programa pedir as unidades, deverá ser escrita uma mensagem de erro. Por exemplo:

```
Qual o valor do ângulo: 37,894
Quais as unidades (g = graus, r = radianos): w
ERRO: entrada não válida para unidades: w
```

Recorde-se que: π radianos = 180 graus e que $\arctan(1) = (\pi/4)$.

13 Dado um par de valores (número de bilhetes, tipo), referente ao número de bilhetes pretendido e o seu tipo. pretende-se saber qual o valor a pagar tendo em conta a seguinte tabela de preços dos bilhetes:

Tipo	**Preço**
1	100€
2	120€
3,4	200€
5	50€

E.3 Ciclos

14 Dado uma sequência de pares de valores (número de bilhetes, tipo), referentes aos bilhetes vendidos para um dado espectáculo musical, pretende-se saber o valor final apurado pela venda de bilhetes.

Escreva um programa que leia uma sequência de pares de valores (um par de cada vez e terminado com o par (0,0)), por exemplo `10 2 2 3 4 2 12 4` e finalmente `0 0`, faça o cálculo referido acima tendo em conta a seguinte tabela de preços dos bilhetes:

Tipo	**Preço**
1	100€
2	120€
3,4	200€
5	50€

15 O algoritmo de Euclides permite calcular o maior divisor comum de dois números inteiros positivos, baseando-se na seguinte propriedade:

$$\text{mdc}(a,b) = \begin{cases} a & \text{se } b = 0 \\ \text{mdc}(b, a \bmod b) & \text{se } b \neq 0 \end{cases}$$

Elabore um programa seguindo os seguintes passos:

- peça ao utilizador dois números inteiros positivos e leia esses inteiros (a e b)
- enquanto $b \neq 0$
  - atribuir a *resto* o valor do resto da divisão inteira de a por b;
  - faça a tomar o valor de b;
  - faça b tomar o valor de *resto*;
- escreva os valores dados e o respectivo máximo divisor comum, o qual é o valor do último a calculado.

16 Elabore um programa que leia um número inteiro e verifique se é uma capicua, através dos seguintes passos:

- Peça ao utilizador que forneça um número inteiro
- Leia esse número inteiro, n
- Atribua a *ncópia* o valor de n
- Inicialize a variável *inverso* a zero
- Enquanto *ncópia* diferente de zero
  - atribua a *dígito* o valor do algarismo das unidades de *ncópia*
  - atribua a *inverso* o seu valor anterior multiplicado por 10 mais o valor de *dígito*
  - atribua a *ncópia* o valor da sua divisão inteira por 10
- Se o número dado for igual a *inverso*
  então escreve «é capicua»
  senão escreve «não é capicua»

17 Elabore programas capazes de receber do teclado uma sucessão de números inteiros ($x_i, i \in \mathbb{N}$). O último elemento da sucessão não é utilizado nos cálculos, servindo para indicar o fim da sucessão (elementos deste tipo denominam-se «sentinelas»). Para cada alínea estabeleça o valor da sentinela mais adequado e escreva o respectivo programa.

1. Contar o número de elementos de uma sucessão de termos positivos;
2. Calcular o produto dos elementos de uma sucessão;
3. Determinar o maior e o menor de entre os elementos de uma sucessão de termos ímpares;
4. Determinar o maior elemento e o número de vezes que ocorre numa sucessão de termos negativos;
5. Calcular a média dos elementos positivos e a média dos elementos negativos numa sucessão de termos ímpares.

18 Determine a média $\bar{x}$ dos i elementos de uma dada sucessão utilizando a seguinte fórmula:

$$\bar{x}_i = \begin{cases} 0 & \text{se } i = 0 \\ \bar{x}_{i-1} + \frac{x_i - \bar{x}_{i-1}}{i} & \text{se } i \geq 1 \end{cases}$$

19 Elabore um programa para calcular o valor da função cos, por desenvolvimento em série, desprezando termos de valor absoluto inferior a 10^{-8}, sabendo que:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} \ldots$$

20 O valor de π pode ser calculado através do chamado produto de Wallis:

$$\frac{\pi}{2} = \frac{2}{1} \times \frac{2}{3} \times \frac{4}{3} \times \frac{4}{5} \times \frac{6}{5} \times \frac{6}{7} \times \frac{8}{7} \times \ldots$$

Elabore um programa para o efeito, que irá construindo e multiplicando cada um dos factores, até que a diferença entre dois valores consecutivos seja, em módulo, inferior a 10^{-4}.

E.4 Funções

21 Escreva uma função que faça a troca do valor de duas variáveis, mas sem usar variáveis auxiliares, isto é, utilizando somente as operações aritméticas elementares.

22 Implemente as seguintes funções:

1. $\begin{array}{rcl} \log : \mathbb{R}^+ \times A & \to & \mathbb{R} \\ (x,a) & \mapsto & \log_a(x) = \frac{\ln(x)}{\ln(a)} \end{array}$ com $A = \{x \in \mathbb{R} : x > 0 \text{e } x \neq 1\}$
2. $\begin{array}{rcl} \text{senh} : \mathbb{R} & \to & \mathbb{R} \\ x & \mapsto & \text{senh}(x) = \frac{e^x - e^{-x}}{2} \end{array}$
3. $\begin{array}{rcl} \text{f} : \mathbb{R} & \to & \mathbb{R} \\ x & \mapsto & \begin{cases} \sqrt{x} & \text{se } x \geq 0 \\ \sqrt{-x} & \text{se } x < 0 \end{cases} \end{array}$
4. $\begin{array}{rcl} \text{argsech} :]0,1] & \to & \mathbb{R} \\ x & \mapsto & \text{argsech}(x) = \log(\frac{1+\sqrt{1-x^2}}{x}) \end{array}$

23 Números Amigos.

1. Dados dois números inteiros positivos escreva uma função que verifique se os dois números são ou não amigos, isto é, se cada um deles é igual à soma dos divisores próprios do outro.
2. Usando a alínea anterior construa um programa que escreva todos os pares de números amigos existentes entre 1 e 1000. Tente optimizar este programa.

E.5 Recursão

24 Elabore um programa que leia uma palavra e a escreva invertida, por exemplo com a palavra "programa", ter-se-ia o resultado "amargorp".

25 Elabore um programa que calcule os k primeiros termos da série harmónica:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} + \ldots$$

26 Escreva uma função que calcule o factorial de um número. Usando essa função escreva um programa que calcule o número de combinações de m, n a n sabendo que:

$$C_n^m = \frac{m!}{n!(m-n)!}$$

27 Escreva um programa que calcule o m.m.c de dois números inteiros positivos, baseando-se nas seguintes propriedades:

$$\text{mmc}(a,b) = \frac{a \times b}{\text{mdc}(a,b)} \quad \text{e} \quad \text{mdc}(a,b) = \begin{cases} a & \text{se } b = 0 \\ \text{mdc}(b, a \bmod b) & \text{se } b \neq 0 \end{cases}$$

28 O sub-factorial ($!n$) de um número inteiro não negativo é definido por:

$$!n = \begin{cases} 1 & \text{se} \quad n = 0 \\ !(n-1)n + (-1)^n & \text{se} \quad n > 0 \end{cases}$$

Escreva uma função recursiva para calcular $!n$.

29 Implemente uma função recursiva que permita calcular a potência:

$$x^n = \begin{cases} 1 & \text{se} \quad n = 0 \\ x^{n-1} * x & \text{se} \quad n > 0 \\ x^{n+1}/x & \text{se} \quad n < 0 \end{cases}$$

para n inteiro e x real.

30 Seja Comissões(m, n) o número de diferentes comissões de n pessoas que se podem formar por eleição entre m pessoas. Por exemplo, Comissões$(4, 3) = 4$, porque dadas quatro pessoas, **A, B, C** e **D**, há quatro possíveis comissões de três membros, **ABC, ABD, ACD** e **BCD**. É sabido que:

- Comissões$(m, n) = 1$ quando $m = n$;
- Comissões$(m, n) = m$ quando $n = 1$;
- Comissões(m, n) =Comissões$(m-1, n-1)$+ Comissões$(m-1, n)$ quando $m > n > 1$.

Escreva uma função recursiva em C que calcule Comissões(m, n) para $m \geq n \geq 1$.

31 Implemente uma função recursiva que permita escrever, por ordem inversa, os dígitos de um número inteiro.

32 Para uso dos alunos do Ensino Primário, pretende-se imprimir uma tabuada de multiplicar. Por exemplo, para $n = 5$, tal tabuada terá o seguinte formato:

```
1
2   4
3   6   9
4   8   12  16
5   10  15  20  25
```

Elabore uma função recursiva para imprimir uma tabuada de tamanho n, de cabeçalho:

```
void imprimirTabuada (int n);
```

33 Considere a seguinte função, definida em $\mathbb{N}$:

$$P(n) = \prod_{i=1}^{n}(2i-1)$$

1. Elabore uma função recursiva para o seu cálculo.
2. Construa uma versão iterativa da mesma função.

E.6 Tabelas Homógeneas (Matrizes)

34 Escreva um sub-programa que, dados um inteiro positivo k e um vector de n inteiros positivos, calcule a média dos múltiplos de k e a média dos submúltiplos de k, existentes no vector.

35 São dados os n elementos inteiros de um vector x (n constante e igual a 100) e ainda um valor inteiro k. Escreva um programa para imprimir todos os pares de números x_i, x_j, tais que $x_i + x_j = k$.

36 Escreva um programa para:

1. ler o inteiro n;
2. ler os $n \times n$ elementos inteiros de uma matriz A;
3. contar quantos elementos nulos existem acima da diagonal principal
4. se o número de elementos nulos for par, escrever a matriz, caso contrário escrever a sua transposta.

37 Faça sub-programas para:

1. Somar duas matrizes $A(n, m)$ e $B(n, m)$.
2. Multiplicar duas matrizes $A(n, m)$ e $B(m, k)$.
3. Dada uma matriz $A(n, m)$ de elementos reais, determinar a linha cuja soma dos seus elementos é máxima.
4. Calcular o determinante de uma matriz triangular $A(n, n)$ de elementos reais.

38 Escreva um programa que, dada uma matriz A de $m \times n$ elementos reais, determine quantos são superiores ao valor da média de todos os elementos da matriz.

39 Um treinador de atletismo treina 5 atletas e faz 12 sessões de treino por semana. Em cada sessão, cada atleta percorre uma distância que é cronometrada. Os valores dos tempos, em segundos, são registados sob a forma de uma matriz $T(5 \times 12)$, onde cada linha diz respeito a um atleta e cada coluna a uma sessão de treino. Supondo já feita a leitura da matriz, escreva uma secção de programa para:

1. calcular e escrever a média dos tempos realizados em cada sessão de treinos;

2. determinar e escrever o melhor tempo realizado por cada um dos atletas nas 12 sessões.

40 Escreva um programa que, dada uma matriz quadrada de ordem n, $(0 < n \leq 100)$ de elementos inteiros, e dados dois inteiros k e l, devolva a matriz após troca entre si das colunas k e l.

E.7 Estruturas não homogéneas («struct»)

41 Escreva sub-programas para operar com complexos, declarando *complexo* como uma estrutura do tipo `struct complexo`;

42 Como se sabe uma matriz quadrada de elementos complexos diz-se hermítica se for igual à sua associada, isto é se:

$$A = A^*$$

em que A^* é a matriz transposta da matriz conjugada de A.

1. Considere a seguinte definição:

   «Dois números reais dizem-se iguais se a distância entre ambos for inferior a 10^{-30}»

   Elabore uma função para verificar a igualdade entre dois números reais de cabeçalho:

   ```
   int iguais(float x, float y);
   ```

2. Defina o tipo complexo utilizando uma estrutura de `struct complexo`;

3. Considerando as declarações:

   ```
   complexo matriz[20][20];
   ```

   elabore uma função para verificar se uma dada matriz é ou não hermítica, utilizando o conceito de igualdade entre reais definido acima. A função terá como cabeçalho:

   ```
   int hermitica (complexo a[][20], int n);
   ```

   Por razões de eficiência o algoritmo deverá parar logo que encontre um par de elementos correspondentes que não sejam conjugados.

43 Operações com fracções.

1. Escreva sub-programas para operar com fracções (ler, escrever, simplificar, somar, multiplicar, dividir, subtrair, calcular potências de fracções), declarando as fracções como estruturas do tipo `struct` cujos campos são do tipo `integer`;

2. Elabore um programa para escrever os primeiros n termos de uma sucessão associada à *série harmónica*:

$$H = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

sob a forma de fracção. Por exemplo, para $n = 4$, a saída do programa deverá ser:

```
1
3/2
11/6
25/12
```

44 Sendo dados os seguintes tipos:

```
struct peca {
   char  nome[20];
   int   disponivel;
   float precoUnitario;
}

struct tipo{
  int   Codigo;
  peca  Peca;
  peca  Armazem[int];
}
```

que descrevem uma armazém de peças. Escreva um sub-programa de facturação que, recebendo os códigos e quantidades das peças encomendadas escreva, em papel, uma factura cujas *linhas de detalhe*, organizadas por coluna, contêm:

1. Para cada peça *encomendada e disponível*, o código, nome, preço unitário, quantidade encomendada e preço total da quantidade encomendada;

2. Para cada peça *encomendada e não disponível*, o código, nome, preço unitário, quantidade encomendada e a mensagem "não disponível" na coluna correspondente ao preço total;

e cuja *linha de total*, posicionada após todas as *linhas de detalhe*, contêm, na coluna correspondente ao preço total, a soma dos preços totais das quantidades encomendadas de todas as peças disponíveis. Declare os tipos e variáveis de que necessitar para elaborar o sub-programa pedido. Indique como invocaria o dito sub-programa.

45 São dados os seguintes tipos e declaração:

```
struct Planeta {
  char  nome[8];
  int   visivel;
  float raioOrbital;
}

Planeta sistema[9];
```

que descrevem o Sistema Solar. Escreva um sub-programa que leia o seguinte ficheiro:

Mercurio	s	0.39	Saturno	s	9.5
Venus	s	0.72	Urano	n	19.2
Terra	s	1.0	Neptuno	n	30.1
Marte	s	1.5	Plutao	n	39.5
Jupiter	s	5.2			

e que escreva o nome e o raio orbital dos planetas visíveis, da Terra, a olho nu.

E.8 Tipo de Dados Compostos, Ponteiros ("Pointers")

46 Considere a seguinte declarações:

```
struct pessoa {
   char nome[30];
   struct pessoa *prox;
}
```

Para uma dada lista, `esta_lista`, já construída, e considerando o ponteiro `p` para um dos elementos da lista. Escreva sub-programas para:

1. retirar da lista, o elemento seguinte ao apontado por `p`;
2. retirar da lista, o elemento apontado por `p`;
3. inserir o registo apontado por `novo`, entre o registo apontado por `p` e o seguinte;
4. inserir o registo apontado por `novo`, entre o registo apontado por `p` e o anterior a este.

47 Um polinómio em x de coeficientes reais pode ser representado por uma sequência de pares ordenados (coeficientes, grau) de preferência ordenados por ordem decrescente das potências de x.

Por exemplo, o polinómio

$$x^8 + 5x^6 - 7x^5 + 6x + 1$$

pode ser representado por

$$(1, 8)(5, 6)(-7, 5)(6, 1)(1, 0)$$

1. Utilizando ponteiros defina um tipo que representa a sequência anterior como uma lista ligada.
2. Escreva um sub-programa `lerpolinomio`, que leia uma sequência de pares de numeros (coeficientes, grau), dado por ordem decrescente das potências de x.

   Por exemplo o polinómio anterior seria dado por:

   $$1\ 8\ \ 5\ 6\ \ -7\ 5\ \ 6\ 1\ \ 1\ 0$$

   O sub-programa deverá construir a lista ligada representativa do polinómio.
3. Escreva uma função `soma` para somar dois polinómios e uma função produto para multiplicar um número real por um polinómio.
4. Elabore um programa de chamada correspondente.

E.9 Ordenação/Pesquisa

E.9.1 Ordenação

48 Ordene os primeiros n elementos de um vector usando o seguinte algoritmo denominado de **Borbulagem**:

1. precorra o vector trocando pares de elementos que estejam fora de ordem.
2. repita o processo até que, numa das passagens anteriores, não se proceda a nenhuma troca.

49 Ordene os primeiros n elementos de um vector usando o seguinte algoritmo denominado de **Selecção Linear**:

1. encontre o maior dos n elementos;
2. coloque esse elemento na sua posição final trocando-o com o enésimo elemento;
3. recursivamente ordene n-1 primeiras posições do vector.

Note-se que a recursão pode ser facilmente eliminada.

50 Ordene os primeiros n elementos de um vector usando o seguinte algoritmo denominado de **Inserção Linear**:

1. Consideramos o vector como a concatenação de duas sequências: uma ordenada, e a segunda não ordenada.

   No início a primeira sequência contém apenas um elemento.
2. O primeiro elemento da sequência não ordenada é inserido na posição correcta da sequência ordenada (movendo os elementos maiores uma posição para a esquerda).

   Esta operação aumentou a sequência ordenada de um elemento e retira esse elemento à sequência não ordenada.
3. Repetimos até todos os elementos estarem na sequência ordenada.

51 Ordene os primeiros n elementos de um vector usando o seguinte algoritmo denominado de **Quick Sort**:

Dado um vector de n inteiros e um inteiro l existente no vector pretende-se particionar esse vector em duas partes: os elementos inferiores a l e os elementos superiores a l.

Uma maneira de fazer isto é a seguinte:

1. Começando do lado esquerdo do vector procurar um elemento que seja maior ou igual a l.
2. Começando do lado direito do vector procurar um elemento que seja menor ou igual a l.

3. Trocar estes dois elementos.
4. Continuar até as duas procuras se cruzarem.

E.9.2 Pesquisa

52 Dado um vector e um seu possível elemento construa programas para verificar se o mesmo está, ou não, contido no vector. Implemente os métodos a seguir descritos.

1. Método de procura exaustiva. Caso o vector não esteja ordenado o único algoritmo possível é dado pela procura exaustiva desde o início do vector até ao seu fim.
2. Caso o vector esteja ordenado a pesquisa pode ser feita de forma muito eficiente. O algoritmo é designado por **Pesquisa binária**:
   (a) verificar se o elemento é igual à posição média do vector;
   (b) caso seja menor do que a posição média, repetir o processo na primeira metade do vector;
   (c) caso seja maior do que a posição média, repetir o processo na segunda metade do vector.

   O algoritmo para assim que encontrar o elemento, ou quando atinge um vector de dimensão nula.

E.10 Implementação de Novos Tipos de Dados

E.11 Tipos Abstractos de Dados

Ainda sem estar a considerar a relação de abstracção/instanciação uma linguagem de Programação Orientada para os Objectos permite uma implementação de um TAD com sonegação da informação referente à implementação da estrutura.

53 Pilhas de inteiros: Pilhas=({pilhaVazia,Inteiro:Pilha},{cria, push, pop, top, vazia?});

54 Filas de inteiros: Filas=({filaVazia,Inteiro:Fila},{cria, insere, retira, topo, vazia?});

55 Listas de inteiros: Listas=({listaVazia,Inteiro:Lista},{cria, insereI, retiraI, veI, vazia?});

E.11.1 Números Racionais

56 Implemente em *C++* uma classe apropriada para representar os números racionais. Deverá ser possível:

1. declarar (criar) números racionais. Assim como a operação inversa de os «destruir».
2. as operações elementares com números racionais.

3. obter as componentes numerador e denominador de um número racional.
4. simplificar um número racional para a sua versão irredutível.
5. os operadores relacionais de igualdade e de desigualdade.

Elabore um programa para escrever os primeiros n termos de uma sucessão associada à *série harmónica*:

$$H = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

sob a forma de fracção. Por exemplo, para $n = 4$, a saída do programa deverá ser:

```
1
3/2
11/6
25/12
```

E.11.2 Números Complexos

57 Implemente em *C++* uma classe apropriada para representar os números complexos. Deverá ser possível:

1. declarar (criar) números complexos. Assim como a operação inversa de os «destruir».
2. as operações elementares com números complexos.
3. obter as componentes reais e imaginárias de um número complexo.
4. converter um real num número complexo (sem parte imaginária).
5. os operadores relacionais de igualdade e de desigualdade.

Construa um pequeno programa que permita testar a nova classe.

E.11.3 Vectores

58 Implemente em *C++* uma classe apropriada para representar os vectores de números inteiros. Deverá ser possível:

1. declarar (criar) vectores. Assim como a operação inversa de os «destruir».
2. as operações elementares com vectores.
3. a multiplicação escalar.
4. obter um elemento, dado um índice.
5. obter o índice de um dado elemento.
6. os operadores relacionais de igualdade e de desigualdade.

Construa um pequeno programa que permita testar a nova classe.

E.11.4 Matrizes

59 Implemente em *C++* uma classe apropriada para representar os matrizes de números inteiros. Deverá ser possível:

1. declarar (criar) matrizes. Assim como a operação inversa de as ≪destruir≫.
2. as operações elementares com matrizes.
3. obter um elemento, dado um par de índices.
4. obter os índices de um dado elemento.
5. os operadores relacionais de igualdade e de desigualdade.

Construa um pequeno programa que permita testar a nova classe.

E.12 Hierarquia de Classes

E.12.1 Herança Simples e Múltipla

60 Pretende-se construir uma aplicação que possa servir para gerir as fichas pessoais dos utilizadores (de diferentes tipos) das residências universitárias de uma dada instituição universitária.

- Conceba uma hierarquia de classes apropriada descrevendo-a através de um diagrama UML.
- Implemente a referida hierarquia através da relações de herança (simples).

61 Um caso de herança múltipla acontece de forma ≪natural≫ numa família.

- Conceba uma hierarquia de classes apropriada descrevendo-a através de um diagrama UML.
- Implemente a referida hierarquia através da relações de herança (múltipla).

E.12.2 Agregação

62 Construa um programa capaz de simular um sistema de múltiplos reservatórios de água ligados entre si de forma arbitrária (em série, em paralelo, ou ambos).

63 Conceba uma hierarquia de classes capaz de guardar a informação sobre alimentos, por exemplo o seu nome, o tipo, as calorias.

Construa um programa capaz de responder às seguintes questões:

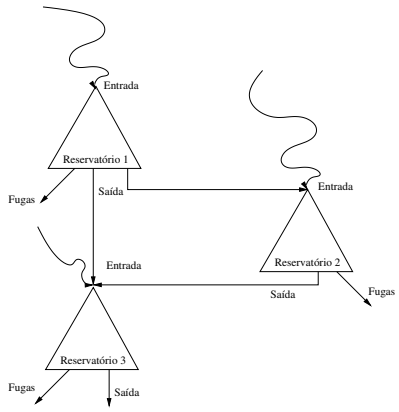


Figura E.1: Sistema com três reservatórios

- Qual é o peso total de uma dada refeição?
- Qual é o seu valor calórico?

64 Construa um programa capaz de simular um sistema de alarme doméstico.

E.12.3 Instanciação

Utilizando o conceito de classe escantilhão (*template*) (re)implemente os seguintes Tipos Abstractos de Dados como tipos de dados paramétricos.
65 Pilhas de elementos genéricos: Pilhas=({pilhaVazia,Elementos:Pilha},{cria, push, pop, top, vazia?});

66 Filas de elementos genéricos: Filas=({filaVazia,Elementos:Fila},{cria, insere, retira, topo, vazia?});

67 Listas de elementos genéricos: Listas=({listaVazia,Elementos:Lista},{cria, insereI, retiraI, veI, vazia?});

68 Árvores Binárias de elementos genéricos:
AB=({ABvazia,ABesq:Elemento:ABdir},{criaVazia, criaAB, procuraElemento, insereElemento, retiraElemento, vazia?});
Implemente as travessias *pré-ordem*, *inordem* e *posordem*.

E.13 Ficheiros

Os ficheiros *CSV* (*Comma separated values*) são um padrão informal, muito usado, para trocas informação entre aplicações distintas (por exemplo, todos os programas o tipo «folha de cálculo», processam este tipo de ficheiro.

É importante então saber como ler e escrever neste formato. Resolva os exercícios seguintes usando o formato *CSV* nos ficheiros.
69 Dado uma lista de valores (do tipo inteiro) contidos no ficheiro `listaNaoOrdenada.dados`, leia os valores, ordene-os e escreva-os no ficheiro `listaOrdenada.dados`.

70 Dado um ficheiro contendo uma série de expressões em NPI (notação polaca inversa), proceda ao seu cálculo e escreva o resultado num outro ficheiro.

O formato de saída, por linha do ficheiro, deverá ser `expressão_em_NPI = resultado`.

E.14 Biblioteca Padrão do C++

Usando os diferentes «livros» da «biblioteca» padrão do *C++*, construa programas para:
71 Construa um programa capaz de obter (e mostrar) o tempo de execução (tempo de CPU) de uma dada função.

72 Para operações geométricas básicas é possível definir ponto através de números complexos. Para um complexo $z = (x + \mathrm{i}y)$ podemos vê-lo como um ponto com coordenadas (x, y).

Temos então as seguintes operações:

- Adição de vectores: $a + b$;
- Multiplicação escalar: $r * a$;
- Produto interno: $(\mathrm{conj}(a) * b).\mathrm{real}()$;
- Producto vectorial: $(\mathrm{conj}(a) * b).\mathrm{imag}()$;
- Distância Euclideana: $\mathrm{abs}(a - b)$

Usando a classe `complex` da STL defina ponto, as funções acima e a função que nos dá a intersecção de duas rectas (não paralelas). Construa um pequeno programa que permita testar as funções que definiu.

E.15 Standard Template Library (STL)

73 Calculadora em Notação Polaca Inversa, usando para tal o «contentor» `stack`, com os «iteradores» correspondentes.

74 Implemente o algoritmo de ordenação «Quick Sort» usando para tal o «contentor» `list`, com os «iteradores» correspondentes.

E.16 Gestão de Erros

O *C++*, através do mecanismo `raise` e `catch` permite uma gestão dos erros, sinalizando-os e gerindo-os aquando da sua ocorrência.

Volte a considerar os seguintes TAD mas agora tenha o cuidado de gerir os casos de erro.
75 Pilhas=({pilhaVazia,Inteiro:Pilha},{cria, push, pop, top, vazia?});
Os métodos `top` e `pop` são ambos problemáticos quando aplicados a uma lista vazia.

76 Filas=({filaVazia,Inteiro:Fila},{cria, insere, retira, topo, vazia?});
Os métodos `retira` e `topo` são ambos problemáticos quando aplicados a uma lista vazia.

77 Listas=({listaVazia,Inteiro:Lista},{cria, insereI, retiraI, veI, vazia?})

Os métodos `insereI`, `retiraI` e `veI` são todos problemáticos para valores do i fora da número de elementos na lista, Além disso os métodos `retiraI` e `veI` são ambos problemáticos quando aplicados a uma lista vazia.

E.17 Problemas — Concepção de UML

78 Pretende-se construir um programa de ajuda à gestão de uma empresa, nomeadamente dos seus funcionários e dos seus diferentes sectores (fabrico, investigação, vendas, contabilidade). Construa um diagrama de classes e suas relações de um tal programa (formato UML).

79 Um concessionário automóvel pretende construir um programa que permita aos eventuais clientes recorrerem a uma simulação (em termos de escolha do modelo+extras e do preço final) antes de decidirem que modelo e que extras é que vão escolher.

Construa um diagrama de classes apropriado à manipulação da informação respeitante a: carros, componentes (tipos de motores, jantes, estofos, etc.), e extras (GPS, RádioMP3, ABS, etc.). Tenha o cuidado de considerar os atributos mais importantes para o problema em questão.

80 Pretende-se simular um sistema de alarme doméstico para um apartamento com cinco divisões. O sistema de alarme deve ser constituído por um conjunto de sensores que detectam perturbações ao normal funcionamento do sistema. Além disso deve possuir uma campainha de alarme que toca sempre que os sensores assinalam uma perturbação sem que haja um consequente desarmar do sistema.

O sistema tem dois modos de funcionamento, armado e desarmado, sendo que a passagem de armado para desarmando está protegido por uma senha (4 dígitos), sendo que esta, tendo um valor por omissão, deve poder ser mudada pelo utilizador.

E.18 Projectos

81 Pretende-se desenvolver uma aplicação capaz de guardar a informação respeitante a uma dada colecção de CDs (musicais).

É necessário guardar a informação respeitante:

- a cada um dos CDs, assim com das colecções de CDs (conjuntos de mais do que um CD mas respeitantes a uma dada colectânea).
- aos artistas que, de alguma forma, estão presentes nos diferentes CDs, assim como as pistas em que actuam, os grupos a que pertencem (ou pertenceram).
- ao estilo musical dos diferentes CDs.
- às empresas discográficas que produziram os diferentes CDs.
- ao local e à de data de compra dos diferentes CDs, considerando também o caso em que o CD foi uma oferta de alguém.

A informação será guardada num ficheiro em formato CSV (*comma-separeted values*).

É necessário também implementar:

- as operações inserir, apagar, e/ou actualizar informação, assim como pesquisar a referida informação.

82 Calculadora em Notação Polaca Inversa - pretende-se desenvolver uma calculadora com as operações aritméticas elementares (adição subtracção, multiplicação e divisão).

Para simplificar a tarefa vai-se considerar que as expressões a introduzir estão em notação pósfixa (notação Polaca Inversa).

- Implemente a classe pilha de palavras (strings);
- O algoritmo de cálculo passa então por:

```
pilha ← cria();
elem ← primeiro_elemento_da_expressão;
enquanto expressão_não_vazia faz
  se (elem é um operando) então
    push(elem,pilha);
  senão
    op2 ← top(pilha);
    pop(pilha);
    op1 ← top(pilha);
    pop(pilha);
    res ← calcula(op1,op2,operador);
    push(res,pilha);
  fimse
  lê_próximo_elemento_da_expressão
fimenquanto
```

83 Pretende-se construir um sistema de leilões «on-line». Temos então um local da Rede («Website») aonde vendedores organizam uma lista de objectos que ele desejam colocar à venda e colocam-a no sistema para ser visualizada pelos potenciais compradores.

Os vendedores, assim como os compradores, necessitam de validar a sua entrada no sistema através de um par (nome de utilizador, senha). No caso de ser a primeira vez que estão a usar o sistema é necessário proceder a um registo.

Aquando dos leilões os compradores que estejam validados podem proceder a ofertas até que o leilão termine. A oferta mais alta é aquela que é aceite pelo vendedor.

O vendedor pode fixar um valor mínimo para cada um dos objectos que coloca à venda. Se ninguém fizer ofertas acima desse valor fixo o objecto não é vendido.

Após o leilão terminar tanto vendedores como compradores têm acesso aos detalhes (vendas/compras) dos seus produtos.

- Construa o diagrama UML que melhor modelize o problema.
- Implemente num programa que permita simular um sistema de leilões (sem a questão da gestão das ligações em rede).

84 Construa um programa que lhe permita jogar Xadrez no modo «Jogador vs. Jogador», sendo que para a especificação das jogadas se deve utilizar a notação algébrica.

Referências

Adams, Jeanne C., Brainerd, Walter S., Martin, Jeanne T., Smith, Brian T., & Wagener, Jerrold L. 1997. *FORTRAN 95 Handbook*. The MIT Press.

Aho, Alfred, Hopcroft, John, & Ullman, Jeffrey. 1983. *Data Structures and Algorithms*. Addison-Wesley Publishing Company.

Alagič, Suad, & Arbib, Michael A. 1978. *The Design of Well-Structured and Correct Programs*. New York: Springer-Verlag.

Bergstra, J. A. 1989. *Algebraic Specification*. Addison-Wesley.

Bird, Richard. 1998. *Introduction to Functional Programming using Haskell*. Prentice Hall.

Booch, Grady. 1991. *Object Oriented Design with Applications*. Redwood City, USA: The Bejamin/Cummings Publishing Company, Inc. Programação Orientada para Objectos.

Budd, Timothy. 1996. *An Introduction to Object-Oriented Programming*. 2nd edn. Addison-Wesley. DMAT 68U/BUD; Programação Orientada para Objectos.

Clavel, Manuel, Durán, Francisco, Eker, Steven, Patrick, Lincoln, Martí-Oliet, Narciso, José, Meseguer, & Quesada, José. 1999 (3). *Maude: Specification and Programming in Rewriting Logic*. Computer Science Laboratory, SRI International.

cplusplus.com. 2019. *C++ Language Library*. `http://www.cplusplus.com/reference/`.

Darcano. 2007. *Tutorial: Aprenda a criar seu próprio Makefile*. Forum Ubuntu Linux - PT. `http://ubuntuforum-pt.org/index.php/topic,21155.0.html`.

Diaconescu, Rŏzvan, & Futatsugi, Kokichi. 1998. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. AMAST series in Computing, vol. 6. World Scientific.

Ehrig, H., & Mahr, B. 1985. *Fundamentals of Algebraic Specification 1*. EATCS Monographs on Theoretical Computer Science, vol. 6. Berlin: Springer-Verlag.

Ehrig, H., & Mahr, B. 1990. *Fundamentals of Algebraic Specification 2*. EATCS - Monographs on Theoretical Computer Science, vol. 21. Berlin: Springer-Verlag.

Goguen, Joseph, Winkler, Timothy, Meseguer, José, Futatsugi, Kokichi, & Jouannaud, Jean-Pierre. 1992 (March). *Introducing OBJ*. Technical Report SRI-CSL-92-03. SRI International. Draft.

Gottfired, Byron. 1994. *Programação em Pascal*. 2nd edn. Lisboa: McGraw-Hill.

Kernighan, Brian, & Ritchie, Dennis. 1988. *The C Programming Language*. 2nd edn. Prentice Hall.

Knuth, Donald E. 1984. Literate Programming. *The Computer Journal*, **27 (2)**, 97–111.

Lippman, Stanley B., & Lajoie, José. 1998. *C++ Primer*. 3rd edition edn. Addison-Wesley. DEPMAT 68N/LIP.

Lloyd, J. W. 1987. *Foundations of Logic Programming*. Berlin: Springer-Verlag.

Main, Michael, & Savitch, Walter. 1997. *Data Structures and other Objects Using C++*. Addison-Wesley. DMAT 68P/MAI.

McCombs, Theodore. 2003 (8). *Maude 2.0 Primer*. ver. 1.0 edn.

Mecklenburg, Robert. 2004. *Managing Projects with GNU Make*. 3rd edn. O'Reilly Media.

Mendelson, Elliott. 1968. *Introduction to Mathematical Logic*. Princeton: D. Van Nostrand Company, Inc.

Meyer, B. 1990. *Introduction to the Theory of Programming Languages*. New York: Prentice Hall International.

Meyer, Bertrand. 1988. *Object-Oriented Software Construction*. Prentice-Hall International. DMAT 68N/MEY; Programaç o Orientada para Objectos.

O'Donnel, Sandra Martin. 1994. *Programming for the World*. Prentice Hall.

Pattis, Richard E. 1980. *EBNF: A Notation to Describe Syntax*. "While developing a manuscript for a textbook on the Ada programming language in the late 1980s, I wrote a chapter on EBNF".

Paulson, Lawrence C. 1991. *ML for the Working Programmer*. Cambridge: Cambridge University Press.

Rodrigues, Pimenta, Pereira, Pedro, & Sousa, Manuela. 1998. *Programação em C++*. 2 edn. FCA, Editora de Informática LDA.

Semgupta, Saumyendra, & Korobkin, Carl Philip. 1994. *C++, Object-Oriented Data Structures*. New-York: Springer-Verlag. DMAT 68N/SEN.

Stroustrup, Bjarne. 1997. *The C++ Programming Language*. Addison Wesley Longman, Inc.

Stroustrup, Bjarne. 2009. *Programming: Principles and Practice Using C++*. Addison Wesley Longman, Inc.

Thompson, Simon. 1996. *Haskell, The Craft of Funcional Programming*. Harlow: Addison-Wesley.

Weiss, Mark Allen. 1997. *Data Structures and Algorithm Analysis in C*. Menlo Park, California: Addison-Wesley.

Welsh, Jim, & Elder, John. 1979. *Introduction to Pascal*. 2nd edition edn. Computer Science. London: Prentice-Hall International, Inc.

Wikstrom, Ake. 1989. *Functional Programming Using Standard ML*. Computer Science. Prentice-Hall International.

yolinux.com. 2012. *C++ STL (Standard Template Library) Tutorial and Examples*. `http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html`.