Capítulo 8

Modularização de alto nível

A modularização é um assunto recorrente em programação. De acordo com [3]:

> **módulo**[1]**.** [Do lat. *modulu*.] S. m. [...] 4. Unidade (de mobiliário, de material de construção, etc.) planejada segundo determinadas proporções e destinada a reunir-se ou ajustar-se a outras unidades análogas, de várias maneiras, formando um todo homogêneo e funcional. [...]

Em programação os módulos são também unidades que se podem reunir ou ajustar umas às outras de modo a formar um todo homogéneo e funcional. Mas normalmente exige-se que os módulos tenham algumas características adicionais:

1. Devem ter um único objectivo bem definido.
2. Devem ser coesos, i.e., devem existir *muitas ligações dentro dos módulos*.
3. Devem ser fracamente ligados entre si, i.e., devem existir *poucas ligações entre os diferentes módulos*.
4. Devem separar claramente a sua interface da sua implementação, i.e., devem distinguir claramente entre o que deve estar visível para o consumidor do módulo e que deve estar escondido no seu interior. Ou seja, os módulos devem respeitar o princípio do encapsulamento.

No Capítulo 3 introduziu-se pela primeira vez a noção de modularização. As unidades básicas de modularização são as rotinas, apresentadas nesse capítulo:

1. Têm um único objectivo bem definido (que no caso das funções é devolver um qualquer valor e no caso dos procedimentos é fazer qualquer coisa).
2. São coesas, pois tipicamente, como têm um objectivo bem definido, podem ser expressos em poucas linhas de código, todas interdependentes. É comum, e a maior parte das vezes desejável, que um módulo utilize outros módulos disponíveis, ou seja, que uma rotina utilize outras rotinas para realizar parte do seu trabalho.

3. São fracamente ligadas. É comum uma rotina usar outra rotina, o que estabelece uma ligação entre as duas, mas fá-lo normalmente recorrendo apenas à sua interface, invocando-a, pelo que as ligações se reduzem à ligação entre argumentos e parâmetros, que desejavelmente são em pequeno número, e ao valor devolvido, no caso de uma função.

4. Separam claramente interface de implementação. O cabeçalho de uma rotina contém (quase[1]) tudo aquilo que quem o utiliza precisa de saber, indicando o nome da rotina, o número e tipo dos parâmetros e o tipo de devolução, i.e., a sua interface. O corpo de uma rotina corresponde à sua implementação, indicando como funciona.

Existem vários níveis de modularização adicionais, que podem ser organizados hierarquicamente. Depois das rotinas, o nível seguinte de modularização corresponde às classes:

1. Têm um único objectivo bem definido que é o de representar um dado conceito, físico ou abstracto. Por exemplo, uma classe `Racional` serve para representar o conceito de número racional, com as respectivas operações, enquanto uma classe `CircuitoLógico` pode servir para representar o conceito de circuito lógico e respectivas operações.

2. São coesas, pois servem para representar uma única entidade, e portanto os seus métodos estão totalmente interligados uns aos outros através dos atributos da classe e, muitas vezes, através de utilizações mútuas.

3. Separam claramente interface de implementação. A interface corresponde aos membros públicos da classe, que tipicamente não incluem atributos. Das funções e procedimentos membro públicos, apenas fazem parte da interface os respectivos cabeçalhos. A implementação está inacessível ao consumidor da classe[2]. Fazem parte da implementação de uma classe todos os membros privados (incluindo normalmente todos os atributos) e o corpo dos métodos públicos da classe.

O nível seguinte de modularização é o nível físico. É porventura o nível com pior suporte pela linguagem e que é mais facilmente mal utilizado pelo programador inexperiente. A modularização física corresponde à divisão dos programas em ficheiros e tem duas utilidades principais: fazer uma divisão lógica das ferramentas de um programa a um nível superior ao das classes e, uma vez que um programa já não precisa de estar concentrado num único ficheiro, mas pode ser dividido por vários ficheiros, facilitar a reutilização de código em programas diferentes.

Finalmente, a linguagem C++ fornece ainda o conceito de espaço nominativo, que se pode fazer corresponder aproximadamente à noção de pacote. Este é o nível mais alto de modularização, e permite fazer uma divisão lógica das ferramentas de um programa a um nível

[1] O cabeçalho de uma rotina diz como ela se utiliza, embora não diga o que faz ou calcula. Para isso, e mesmo que o nome da rotina seja auto-explicativo, acrescenta-se ao cabeçalho um comentário de documentação que indica o que a rotina faz ou calcula e que inclui a sua pré-condição e a sua condição objectivo.

[2] A implementação de um módulo em C++ está muitas vezes visível ao programador consumidor. O programador consumidor de uma função pode, muitas vezes, ver o seu corpo ou implementação. Mas não pode, através do seu programa, afectar directamente essa implementação: o programador consumidor usa a implementação de um módulo sempre indirectamente através da sua interface. É um pouco como se os módulos em C++ fossem transparentes: pode-se ver o mecanismo, mas não se tem acesso directo a ele.

superior ao da modularização física e, simultaneamente, evitar a colisão de nomes definidos em cada parte de um programa complexo. Assim, um espaço nominativo (ou melhor, um pacote) abarca tipicamente ferramentas definidas em vários módulos físicos.

Em resumo, existem os seguintes níveis de modularização:

1. Modularização procedimental: os módulos a este nível chamam-se *rotinas* (*funções* e *procedimentos*).
2. Modularização de dados: os módulos a este nível chamam-se *classes*.
3. Modularização física: os módulos a este nível chamam-se normalmente *módulos físicos* (ou simplesmente *módulos*) e correspondem a ficheiros (normalmente um par ou oum trio de ficheiros, como se verá).
4. Modularização em pacotes: os módulos a este nível chamam-se *pacotes* e correspondem, no C++, a espaços nominativos.

Nas próximas secções serão discutidos em algum pormenor estes dois novos níveis de modularização: ficheiros (modularização física) e pacotes (espaços nominativos).

8.1 Modularização física

Nos capítulos anteriores viu-se que uma forma natural de reaproveitar código no mesmo programa é através da escrita de rotinas. Viu-se também como se criavam novos tipos de dados (i.e., classes), como se equipavam esses novos tipos das respectivas operações, e como se utilizavam esses novos tipos num dado programa. Mas como reutilizar uma ferramenta, e.g., uma rotina ou uma classe, em programas diferentes? Claro está que uma solução seria usar as opções Copiar/Colar do editor de texto. Mas essa não é, decididamente, a melhor solução. Há várias razões para isso, mas a principal talvez seja que dessa forma sempre que se altera a ferramenta copiada, presumivelmente para lhe introduzir correcções ou melhorias, é necessário repetir essas alterações em todas as suas cópias, que ao fim de algum tempo se podem encotrar espalhadas por vários programas. Para obviar a este problema, a linguagem C++ fornece um mecanismo de reutilização mais conveniente: a compilação separada de módulos físicos correspondentes a diferentes ficheiros.

Mesmo que o objectivo não seja a reutilização de código, a modularização física através da colocação de diferentes ferramentas (rotinas, classes, etc.) em ficheiros separados é muito útil, pois permite separar claramente ferramentas com aplicações diversas, agrupando simultaneamente ferramentas com relações fortes entre si. Isto tem vantagens não apenas em termos da estruturação do programa, como também por facilitar a participação de várias equipas no desenvolvimento de um programa: cada equipa desenvolve um conjunto de módulos físicos diferentes, e consequentemente ficheiros diferentes. Além disso a compilação acelera o processo de construção do ficheiro executável pois, como se verá, uma alteração num módulo de um programa não exige normalmente senão a reconstrução de uma pequena parte do programa.

8.1.1 Constituição de um módulo

Um módulo físico de um programa é constituído normalmente por dois ficheiros fonte: o ficheiro de interface (*header file*) com extensão `.H` e o ficheiro de implementação com extensão `.C` (alternativamente `.cpp`, `.cc` ou mesmo `.c++`)[3]. Por vezes um dos ficheiros não existe. É tipicamente o caso do módulo que contém a função `main()` do programa, que não é muito útil para reutilizações noutros programas e que só contém o ficheiro de implementação. Também ocorrem casos em que o módulo contém apenas o ficheiro de interface.

Normalmente cada módulo corresponde a uma unidade de tradução (*translation unit*). Uma unidade de tradução corresponde ao ficheiro de implementação de um módulo já depois de pré-processado, i.e., incluindo todos os ficheiros especificados em directivas `#include` (ver mais abaixo).

O termo módulo por si só refere-se normalmente a um módulo físico, i.e., a uma unidade de modularização física constituída normalmente por um ficheiro de interface e pelo respectivo ficheiro de implementação. Assim, de hora em diante a palavra *módulo* será usada neste sentido mais restrito, sendo utilizações mais latas do termo indicadas explicitamente ou, espera-se, claras pelo contexto.

Os nomes escolhidos para cada um dos ficheiros de um módulo reflectem a sua utilização usual. Normalmente o ficheiro de interface serve para indicar as interfaces de todas as ferramentas disponibilizadas pelo módulo e o ficheiro de implementação serve para implementar essas mesmas ferramentas.

Assim, o ficheiro de interface contém normalmente a declaração de rotinas e a definição das classes disponibilizadas pelo módulo. Cada definição de classe contém a declaração dos respectivos métodos, a definição dos atributos de instância, a declaração dos atributos de classe, e a declaração das classes e rotinas amigos da classe. Por outro lado, o ficheiro de implementação contém normalmente as definições de todas as ferramentas usadas dentro do módulo mas que não fazem parte da sua interface, ou seja, ferramentas de utilidade interna ao módulo, e as definições de todas as ferramentas que fazem parte da interface mas que foram apenas declaradas no ficheiro de interface. Mais à frente se apresentarão um conjunto de regras mais explícito acerca do conteúdo típico destes dois ficheiros.

8.2 Fases da construção do ficheiro executável

A construção de um ficheiro executável a partir de um conjunto de diferentes módulos físicos tem três fases: o pré-processamento, a compilação propriamente dita e a fusão.

Inicialmente um programa chamado *pré-processador* age individualmente sobre o ficheiro de implementação (`.C`) de cada módulo, incluindo nele todos os ficheiros de interface (`.H`) especificados por directivas `#include` e executando todas as directivas de pré-processamento (que correspondem a linhas começadas por um `#`). O resultado do pré-processamento é um ficheiro de extensão `.ii` (em Linux) que contém linguagem C++ e a que se chama uma *unidade de tradução*.

[3] É boa ideia reservar as extensões `.h` e `.c` para ficheiros de interface escritos na linguagem C.

Depois, um programa chamado compilador traduz cada unidade de tradução (`.ii`) de C++ para código relocalizável em linguagem máquina, na forma de um ficheiro objecto de extensão `.o` (em Linux). Os ficheiros objecto, para além de conterem o código em linguagem máquina, contêm também uma lista dos símbolos definidos ou usados nesse ou por esse código e que será usada na fase de fusão. A compilação age sobre cada unidade de tradução independentemente de todas as outras, daí que seja usado o termo compilação separada para referir o processo de construção de um executável a partir dos correspondentes ficheiros fonte.

O termo *compilação* pode ser usado no sentido lato de construção de um executável a partir de ficheiros fonte ou no sentido estrito de tradução de uma unidade de tradução para o correspondente ficheiro objecto. Em rigor só o sentido estrito está correcto, mas é comum (e fazemo-lo neste texto) usar também o sentido lato onde isso for claro pelo contexto.

A última grande fase da construção é a fusão (*linking*). Nesta fase, que é a única que actua sobre todos os módulos, os ficheiros objecto do programa são fundidos (*linked*) num executável. Ao programa que funde os ficheiros objecto chama-se fusor (*linker*).

Note-se que em Linux é o mesmo programa (`c++` ou `g++`) que se encarrega das três fases, podendo-as realizar todas em sequência ou apenas uma delas consoante os casos.

As várias fases da construção de um executável são descritas abaixo com um pouco mais de pormenor.

8.2.1 Pré-processamento

O pré-processamento é uma herança da linguagem C. Não fazendo parte propriamente da linguagem C++, é no entanto fundamental para desenvolver programas em módulos físicos separados. É uma forma muito primitiva de garantir a compatibilidade e a correcção do código escrito em cada módulo. A verdade, porém, é que infelizmente o C++ não disponibiliza nenhum outro método para o fazer, ao contrário de outras linguagens como o Java, onde a modularização física é suportada directamente pela linguagem.

O *pré-processador* age sobre um ficheiro de implementação (`.C`), gerando uma unidade de tradução (`.ii`). Esta unidade de tradução contém código C++, tal como o ficheiro de implementação original, mas o pré-processador faz-lhe algumas alterações. O comando para pré-processar um ficheiro de implementação em Linux é[4]:

```
c++ -E nome.C -o nome.ii
```

onde *`nome`* é o nome do módulo a pré-processar e `-E` é uma opção que leva o programa `c++` a limitar-se a proceder à pré-compilação do ficheiro.

O pré-processador copia o código C++ do ficheiro de implementação para a unidade de tradução[5], mas sempre que encontra uma linha começada por um cardinal (#) interpreta-a. Estas

[4] O comando também pode ser `g++`.

[5] Na realidade o pré-processador faz mais do que isso. Por exemplo, elimina todos os comentários do código substituindo-os por um espaço. A maior parte das operações levadas a cabo pelo pré-processador, no entanto, não são muito relevantes para esta discussão simplificada.

linhas são as chamadas *directivas de pré-processamento*, que, salvo algumas excepções pouco importantes neste contexto, não existem no ficheiro pré-processado, ou seja, na unidade de tradução gerada.

Directivas de pré-processamento

Existem variadíssimas directivas de pré-processamento. No entanto, só algumas são relevantes neste contexto:

- `#include`
- `#define`
- `#ifdef` (`#else`) e `#endif`
- `#ifndef` (`#else`) e `#endif`

A directiva `#include` tem dois formatos:

1. `#include <`*`nome`*`>`
2. `#include "`*`nome`*`"`

Em ambos os casos o resultado da directiva é a substituição dessa directiva, na unidade de tradução (ficheiro pré-processado), por todo o conteúdo pré-processado do ficheiro indicado por *`nome`* (e que pode incluir um *caminho*[6]). A diferença entre os dois formatos prende-se com o local onde os ficheiros a incluir são procurados. Os ficheiros incluídos por estas directivas são também pré-processados, pelo que podem possuir outras directivas de inclusão e assim sucessivamente.

Quando o primeiro formato é usado, o ficheiro é procurado nos locais "oficiais" do sistema onde se trabalha. Por exemplo, para incluir os ficheiros de interface que declaram as ferramentas de entradas e saídas a partir de canais da biblioteca padrão faz-se

```
#include <iostream>
```

pois este é um ficheiro de interface oficial. Neste caso é um ficheiro que faz parte da norma do C++!!citar norma. Isto significa que existe algures no sistema um ficheiro chamado `iostream` (procure-o, em Linux, usando o comando `locate iostream` e veja o seu conteúdo). É típico que os ficheiros de interface oficiais não tenham qualquer extensão no seu nome, embora também sejam comuns as extensões `.H` ou `.h`. É possível acrescentar ficheiros de interface aos locais oficiais, desde que se tenha permissões para isso, ou então acrescentar directórios à lista dos locais oficiais de modo a "oficializar" um conjunto de ferramentas por nós desenvolvido. Nos sistems operativos baseados no Unix há vários directórios com ficheiros de interface oficiais, sendo o mais usual o directório `/usr/include`.

[6]Optou-se por traduzir por *caminho* o inglês *path*.

Quando o segundo formato da directiva de inclusão é usado, o ficheiro é procurado primeiro a partir do directório onde se encontra o ficheiro fonte contendo a directiva em causa e, caso não seja encontrado, é procurado nos locais "oficiais". É típico que os ficheiros de interface não-oficiais tenham a extensão `.H`.

Os ficheiros incluídos são ficheiros de interface com declarações de ferramentas úteis ao módulo em processamento.

!!!Fazer figura com exemplo?

A directiva `#define` tem como objectivo a definição de *macros*, que são como que variáveis do pré-processador. As macros podem ser usadas de formas muito sofisticadas e perigosas, sendo um mecanismo externo à linguagem propriamente dita e que com ela interferem muitas vezes de formas insuspeitas. Por isso recomenda-se cautela na sua utilização. Neste texto ver-se-á apenas a utilização mais simples:

```
#define NOME
```

Depois desta directiva, o pré-processador tem definida uma macro de nome *`NOME`* com conteúdo nulo. Convencionalmente o nome das macros escreve-se usando apenas maiúsculas para as distinguir claramente dos nomes usados no programa C++.

É possível colocar na unidade de tradução código alternativo ou condicional, consoante uma dada macro esteja definida ou não. A isso chama-se compilação condicional ou alternativa e consegue-se usando a directiva condicional `#ifdef` (que significa *if defined*)

```
#ifdef NOME
texto...
#endif
```

ou a directiva de selecção correspondente

```
#ifdef NOME
texto1...
#else
texto2...
#endif
```

ou ainda a directiva negada correspondente, começada por `#ifndef` (que significa *if not defined*).

Quando a directiva condicional é interpretada pelo pré-processador, o texto *`texto`*... só é pré-processado e incluído na unidade de tradução se a macro *`NOME`* estiver definida. Na directiva de selecção, se a macro estiver definida, então o texto *`texto1`*... é pré-processado e incluído na unidade de tradução, caso contrário o texto *`texto2`*... é pré-processado e incluído na unidade de tradução.

Depois do pré-processamento

A unidade de tradução que resulta do pré-processamento de um ficheiro de implementação possui, tipicamente, algumas directivas residuais que são colocadas pelo próprio pré-processador. É o que se passa com o pré-processador da GCC (Gnu Compiler Colection), usado em Linux. A discussão abaixo descreve esse ambiente particular de desenvolvimento.

Para que um programa possa ser executado em modo de depuração deve ser compilado com a opção `-g`. Acontece que, para que o depurador saiba a que linha nos ficheiros fonte corresponde cada instrução (para a poder mostrar no editor, por exemplo o `XEmacs`), a informação acerca da linha e do ficheiro fonte a que corresponde cada instrução em código máquina tem de ficar guardada nos ficheiros objecto e, depois da fusão, no ficheiro executável. Suponha-se o seguinte ficheiro de implementação chamado `olá.C`:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Olá mundo!" << endl;
}
```

Podem-se usar as seguintes instruções para construir o executável:

```
c++ -E olá.C -o olá.ii
c++ -c olá.ii
c++ -o olá olá.o
```

Note-se que se compilou o ficheiro pré-processado, ou seja, a unidade de tradução. Como pode o compilador saber de onde vieram as linhas do ficheiro `olá.ii`, necessárias durante a depuração? Só se o ficheiro pré-processado contiver essa informação! Para isso servem as directivas introduzidas pelo pré-processador (que, como usualmente, começam por `#`).

Não é apenas para a depuração que essa informação é útil. Se essa informação não estivesse no ficheiro pré-processado, os erros de compilação seriam assinalados no ficheiro `olá.ii`, o que não ajudaria muito o programador, que editou o ficheiro `olá.C`. Logo, a informação acerca da origem das linhas da unidade de tradução também é útil para o compilador produzir mensagens de erro (experimente-se executar os comandos acima, mas acrescentando a opção `-P` ao pré-processar, e veja-se o que o compilador diz...).

As únicas directivas presentes nas unidades de tradução (`.ii`) têm o seguinte formato:

`#` *`linha nome`* [*x*...]

onde:

`linha` é número da linha no ficheiro fonte de onde veio a próxima linha da unidade de tradução.

`nome` é o nome do ficheiro fonte de onde veio a próxima linha na unidade de tradução.

`x` são um conjunto de números com os seguintes possíveis valores:

1. Indica o começo de um novo ficheiro fonte.
2. Indica que se regressou a um ficheiro fonte (depois de terminar a inclusão de outro).
3. Indica que as linhas que se seguem vêm de um ficheiro de interface "oficial" (serve para desactivar alguns avisos do compilador).
4. Indica que as próximas linhas contêm linguagem C.

Veja-se um exemplo em Evitando inclusões múltiplas.

Exemplo

Suponha-se que se escreveu uma função `máximoDe()` para calcular o máximo dos primeiros valores contidos num vector de `double`:

```
/** Devolve o maior dos valores dos itens do vector v.
    @pre  PC ≡ 0 < v.size().
    @post CO ≡ (Q j : 0 ≤ j < v.size() : v[j] ≤ máximoDe) ∧
               (E j : 0 ≤ j < v.size() : v[j] = máximoDe). */
double máximoDe(vector<double> const& v)
{
    double máximo = v[0];
    for(vector<double>::size_type i = 1; i != v.size(); ++i)
        if(máximo < v[i])
            máximo = v[i];
    return máximo;
}
```

Durante o desenvolvimento do programa onde a função acima é usada, é conveniente verificar se a pré-condição da função se verifica. Desse modo, se o programador consumidor da função se enganar, o programa abortará imediatamente com uma mensagem de erro apropriada, o que antecipará a detecção do erro e consequente correcção. Assim, durante o desenvolvimento, a função deveria fazer a verificação explícita da pré-condição[7]:

```
/** Devolve o maior dos valores dos itens do vector v.
    @pre  PC ≡ 0 < v.size().
    @post CO ≡ (Q j : 0 ≤ j < v.size() : v[j] ≤ máximoDe) ∧
```

[7] O procedimento `exit()` termina abruptamente a execução de um programa. Não se recomenda a sua utilização em nenhuma circunstância. Por favor leia um pouco mais e encontrará melhores alternativas...

```
              (E j : 0 ≤ j < v.size() : v[j] = máximoDe).  */
double máximoDe(vector<double> const& v)
{
    if(v.size() == 0) {
        cerr << "Erro em máximo()! Vector com dimensão nula!"
             << endl;
        exit(1); // Atenção! Não se advoga o uso de exit()!
                 // Leia um pouco mais, por favor...
    }

    double máximo = v[0];
    for(vector<double>::size_type i = 1; i != v.size(); ++i)
        if(máximo < v[i])
            máximo = v[i];
    return máximo;
}
```

O problema desta solução é que, quando o programa está já desenvolvido, testado e distribuído, a instrução condicional acrescentada à função deixa de ter utilidade, pois presume-se que o programador consumidor já garantiu que todas as suas chamadas desta função verificam a pré-condição, servindo apenas para tornar o programa mais lento. Daí que fosse conveniente que essa instrução fosse retirada depois de depurado o programa e que se voltasse a colocar apenas quando o programa fosse actualizado. Ou seja, o ideal seria que as instruções de verificação produzissem efeito em modo de depuração ou desenvolvimento e não produzissem qualquer efeito em modo de distribuição[8].

Mas pôr e tirar instruções desta forma é uma má ideia, mesmo se para o efeito se usarem comentários: é que pode haver muitas, mas mesmo muitas instruções deste tipo num programa, o que levará a erros e esquecimentos. O problema resolve-se recorrendo a compilação condicional:

```
/** Devolve o maior dos valores dos itens do vector v.
    @pre  PC ≡ 0 < v.size().
    @post CO ≡ (Q j : 0 ≤ j < v.size() : v[j] ≤ máximoDe) ∧
              (E j : 0 ≤ j < v.size() : v[j] = máximoDe).  */
double máximoDe(vector<double> const& v)
{
    #ifndef NDEBUG
    if(v.size() == 0) {
        cerr << "Erro em máximo()! Vector com dimensão nula!"
             << endl;
        exit(1);
    }
```

[8]O modo de distribuição (*release*) é o modo do programa tal como distribuído aos seus utilizadores finais. O modo de depuração (*debug*) ou desenvolvimento é o modo do programa enquanto está em desenvolvimento e teste.

```
    #endif

    double máximo = v[0];
    for(vector<double>::size_type i = 1; i != v.size(); ++i)
        if(máximo < v[i])
            máximo = v[i];
    return máximo;
}
```

Se a macro NDEBUG (que significa *not debug*) não estiver definida, i.e., se se estiver em fase de depuração, a verificação da pré-condição é feita. Caso contrário, i.e., se não se estiver em fase de depuração mas sim em fase de distribuição e portanto a macro NDEBUG estiver definida, então o código de verificação da pré-condição é eliminado da unidade de tradução e nem chega a ser compilado!

Esta macro NDEBUG é usada exactamente da mesma forma que acima pelas instruções de asserção descritas na Secção 3.2.19. De resto, as instruções de asserção permitem escrever o código de uma forma mais clara (onde se aproveitou para colocar uma verificação parcial da condição objectivo)[9]:

```
/** Devolve o maior dos valores dos itens do vector v.
    @pre  PC ≡ 0 < v.size().
    @post CO ≡ (Q j : 0 ≤ j < v.size() : v[j] ≤ máximoDe) ∧
               (E j : 0 ≤ j < v.size() : v[j] = máximoDe). */
double máximo(vector<double> const& v)
{
    assert(0 < v.size());

    double máximo = v[0];
    for(vector<double>::size_type i = 1; i != v.size(); ++i)
        if(máximo < v[i])
            máximo = v[i];

    // Sem ciclos não se pode fazer muito melhor (amostragem em três locais):
    assert(v[0] <= máximo and v[v.size() / 2] <= máximo
           and v[v.size() - 1] <= máximo);

    return máximo;
}
```

Todas as asserções de um módulo podem ser desligadas definindo a macro NDEBUG. Para o fazer não é sequer necessário alterar nenhum ficheiro do módulo! Basta dar a opção de pré-compilação -DNDEBUG. A opção -D*MACRO* define automaticamente a macro *MACRO* em todos os ficheiros pré-processados. Logo, para pré-processar um ficheiro em modo de distribuição pode-se usar o comando

[9] Note-se que assert() é, na realidade, uma macro com argumentos (noção que não se descreve neste texto).

```
c++ -DNDEBUG -E nome.C -o nome.ii
```

ou, se se pretender também compilar o ficheiro, para além de o pré-processar:

```
c++ -DNDEBUG -c nome.C
```

8.2.2 Compilação

O *compilador* age sobre uma unidade de tradução (ficheiro pré-processado) e tradu-lo para linguagem máquina. O ficheiro gerado chama-se *ficheiro objecto* e, apesar de conter linguagem máquina, não é um ficheiro executável, pois contém informação adicional acerca desse mesmo código máquina, como se verá. Os ficheiros objecto têm extensão `.o` (em Linux) e contêm código máquina utilizável (sem necessidade de compilação) por outros programas. O comando para compilar uma unidade de tradução nome em Linux é:

```
c++ -Wall -ansi -pedantic -g -c nome.ii
```

ou, se se quiser pré-processar e compilar usando um só comando:

```
c++ -Wall -ansi -pedantic -g -c nome.C
```

em que a opção `-c` indica que se deve proceder apenas à compilação (e, se necessário, também ao pré-processamento), a opção `-Wall` pede ao compilador para avisar de todos (*all*) os potenciais erros (*warnings*), as opções `-ansi` e `-pedantic` dizem para o compilador seguir tão perto quanto possível a norma da linguagem, e a opção `-g` indica que o ficheiro objecto gerado deve conter informação de depuração.

A compilação de uma unidade de tradução consiste pois na sua tradução para linguagem máquina e é feita em vários passos:

1. Análise lexical. Separa o texto do programa em símbolos, verificando a sua correcção. Equivale à identificação de palavras e sinais de pontuação que os humanos fazem quando lêem um texto, que na realidade consiste numa sequência de caracteres.

2. Análise sintáctica. Verifica erros gramaticais no código, por exemplo conjuntos de símbolos que não fazem sentido como `a / / b`. Equivale à verificação inconsciente da correcção gramatical de um texto que os humanos fazem. Ao contrário dos humanos, que são capazes de lidar com um texto contendo muitos erros gramaticais (e.g., "eu leste texto este e erro nenhum encontrámos"), o compilador não lida nada bem com erros sintácticos. Embora os compiladores tentem recuperar de erros passados e continuar a analisar o código de modo a produzir um conjunto tão relevante quanto possível de erros e avisos, muitas vezes enganam-se redondamente, assinalando erros que não estão presentes no código. Por isso, apenas o primeiro erro assinalado pelos compiladores é verdadeiramente fiável (mesmo que possa ser difícil de perceber).

3. Análise semântica. Verifica se o texto, apesar de sintacticamente correcto, tem significado. Por exemplo, verifica a adequação dos tipos de dados aos operadores, assinalando erros em expressões como `1.1 % 3.4`. Equivale à verificação inconsciente do sentido de frases gramaticalmente correctas. Por exemplo, o leitor humano reconhece como não fazendo sentido os versos[10]

   O ganso, gostou da dupla e fez também qüem, qüem
   Olhou pro cisne e disse assim vem, vem
   Que um quarteto ficará bem, muito bom, muito bem.

4. Optimização. Durante esta fase o compilador elimina código redundante, simplfica expressões, elimina variáveis desnecessárias, etc., de modo a poder gerar código máquina tão eficiente quanto possível.

5. Geração de código máquina (pode ter uma fase intermédia numa linguagem de mais baixo nível).

O resultado da compilação é um ficheiro objecto, como se viu. Este ficheiro não contém apenas código máquina. Simplificando algo grosseiramente a realidade, cada ficheiro objecto tem guardadas duas tabelas: uma é a tabela das disponibilidades, e outra é a tabela das necessidades. A primeira tabela lista os nomes (ou melhor, as assinaturas) das ferramentas definidas pelo respectivo módulo, e portanto disponíveis para utilização em outros módulos. A segunda tabela lista os nomes das ferramentas que são usadas pelo respectivo módulo, mas não são definidas por ele, e que portanto devem ser definidas por outro módulo.

Por exemplo, suponham-se seguintes módulos `A` e `B`:

A.H

```
void meu(); // declaração do procedimento meu() que está definido em A.C.
```

A.C

```
#include "A.H"
#include "B.H"

// Definição do procedimento meu():
void meu()
{
    outro();
}
```

B.H

```
void outro(); // declaração do procedimento outro() que
              // está definido em B.C.
```

[10]"O Pato", de João Gilberto. No entanto, estes versos não estão errados, pois usam a figura da personificação. Os compiladores não têm esta nossa capacidade de entender figuras de estilo, bem entendido...

B.C

```
#include "B.H"

// Definição do procedimento outro():
void outro()
{
}
```

A compilação separada das unidades de tradução destes dois módulos resulta em dois ficheiros objecto `A.o` e `B.o` contendo:

A.o

1. Código máquina.
2. Disponibilidades: `meu()`
3. Necessidades: `outro()`

B.o

1. Código máquina.
2. Disponibilidades: `outro()`
3. Necessidades:

Cada ficheiro objecto, por si só, não é executável. Por exemplo, aos ficheiros `A.o` e `B.o` falta a função `main()` e ao ficheiro `A.o` falta o procedimento `outro()` para poderem ser executáveis. No entanto, mesmo que um ficheiro objecto contenha a definição da função `main()` e de todas as ferramentas usadas, não é executável. O executável é sempre gerado pelo fusor, que utiliza a informação acerca do que cada ficheiro objecto disponibiliza e necessita para fazer o seu trabalho.

8.2.3 Fusão

Quer o pré-processamento quer a compilação, já descritas, agem sobre um único módulo. O pré-processador age sobre o ficheiro de implementação do módulo (`.C`), honrando todas as directivas de `#include` que encontrar, pelo que na realidade o pré-processamento pode envolver vários ficheiros: um de implementação e os outros de interface. O resultado do pré-processamento é um único ficheiro, chamado unidade de tradução. O compilador age sobre uma unidade de tradução de cada vez e independentemente de todas as outras. Por isso se chama muitas vezes à modularização física compilação separada. Mas um programa, mesmo que o seu código esteja espalhado por vários módulos, tem normalmente um único ficheiro executável. Logo, tem de ser a última fase da construção de um executável a lidar com todos os módulos em simultâneo: a fusão.

O *fusor* é o programa que funde todos os ficheiros objecto do programa e gera o ficheiro executável. Em Linux, o comando para fundir os ficheiros objecto num executável é:

```
c++ -o programa módulo1.o módulo2.o ...
```

onde *módulo1*`.o` etc., são os ficheiros objecto a fundir e `programa` é o nome que se pretende dar ao ficheiro executável.

Se se quiser pré-processar, compilar e fundir usando um só comando, o que é pouco recomendável, pode-se usar o comando:

```
c++ -Wall -ansi -pedantic -g -o programa módulo1.C módulo2.C ...
```

Que começa por pré-processar o ficheiros de implementação, depois compila as respectivas unidades de tradução, e finalmente funde os ficheiros objecto resultantes.

O fusor basicamente concatena o código máquina de cada um dos ficheiros objecto especificados. Mas durante esse processo:

1. Verifica se há repetições. Não pode haver dois ficheiros objecto a definir a mesma ferramenta. Se houver repetições, o fusor escreve uma mensagem de erro (com um aspecto um pouco diferente das mensagens do compilador).
2. Verifica se existe uma função `main()`. De outra forma não se poderia criar o executável, cuja execução começa sempre nessa função.
3. Para cada ferramenta listada como necessária na tabela de necessidades de cada ficheiro objecto, o fusor verifica se ela está listada na tabela de disponibilidades de algum ficheiro objecto. Se faltar alguma ferramenta, o fusor assinala o erro.

É de notar que o fusor não coloca no ficheiro executável todo o código máquina de todos os ficheiros objecto. O fusor tenta fazer uma selecção inteligente de quais as ferramentas que são realmente usadas pelo programa. É por isso que os ficheiros executáveis são normalmente bastante mais pequenos que a soma das dimensões das partes fundidas.

Ficheiros fundidos automaticamente

O fusor acrescenta à lista de ficheiros a fundir o ficheiro de arquivo (de extensão `.a`, ver Secção 8.2.4) contendo os ficheiros objecto da biblioteca padrão do C++. É por isso que é possível usar canais de entrada ou saída (`cin` e `cout`), cadeias de caracteres (`string`), etc., sem grandes preocupações: os respectivos ficheiros objecto são fundidos automaticamente ao construir o executável. Por exemplo, o ficheiro executável do famoso primeiro programa em C++:

`olá.C`

```
#include <iostream>

using namespace std;
```

```
int main()
{
    cout << "Olá mundo!" << endl;
}
```

pode ser construído por

```
c++ -Wall -ansi -pedantic -g -c olá.C
c++ -Wall -g -o olá olá.o
```

em que o primeiro comando pré-processa e compila o ficheiro de implementação `olá.C` e o segundo constrói o ficheiro executável `olá`. Estes comandos parecem indicar que o programa depende apenas do ficheiro `olá.C`. Nada mais falso. A inclusão de `iostream` e utilização de `cout`, `endl` e do operador << obrigam à fusão com, entre outros, o ficheiro objecto do módulo `iostream` existente no ficheiro de arquivo da biblioteca padrão da linguagem C++. O arquivo em causa chama-se `libstdc++.a` e encontra-se algures no sistema (procure-o com o comando `locate libstdc++.a` e use o comando `ar t caminho/libstdc++.a` para ver o seu conteúdo, onde encontrará o ficheiro `iostream.o`).

Mesmo o programa mais simples obriga à fusão com um conjunto avulso de outros ficheiros objecto e ficheiros objecto incluídos em arquivos. O comando `c++` especifica esses ficheiros automaticamente, simplificando com isso a vida do programador. Experimente-se passar a opção `-nostdlib` ao comando `c++` ao construir o programa `olá` a partir do ficheiro objecto `olá.o`:

```
c++ -nostdlib -o olá olá.o
```

Como esta opção inibe a fusão automática do ficheiro `olá.o` com o conjunto de ficheiros objecto e arquivos de ficheiros objecto necessários à construção do executável, o comando anterior produz erros semelhantes ao seguinte[11]:

```
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 08048074
olá.o: In function ‘main’:
.../olá.C:7: undefined reference to ‘endl(ostream &)’
.../olá.C:7: undefined reference to ‘cout’
.../olá.C:7: undefined reference to ‘ostream::operator<<(char const *)’
.../olá.C:7: undefined reference to ‘ostream::operator<<(ostream &(*)(ostream &))’
collect2: ld returned 1 exit status
```

O primeiro dos erros apresentados é o menos evidente. Ele indica que o programa não tem sítio para começar. É que os programas em C++, como começam todos na função `main()`, têm de ser fundidos com ficheiros objecto que invocam essa função logo no início do programa. Alguns desses ficheiros objecto estão incluídos no ficheiro de arquivo da biblioteca padrão da linguagem C, que se chama `libc.a`. Se se usar o comando:

[11] Estes erros foram obtidos num sistema Linux, distribuição RedHat 7.0, com o GCC versão 2.96, pelo que em sistemas com outra configuração as mensagens podem ser diferentes.

```
c++ -nostdlib -
o olá /usr/lib/crt1.o /usr/lib/crti.o olá.o /usr/lib/libc.a /usr/lib/gcc-lib/i386-
redhat-linux/2.96/libgcc.a
```

a primeira mensagem de erro desaparece, restando apenas:

```
olá.o: In function ‘main’:
…/olá.C:7: undefined reference to ‘endl(ostream &)’
…/olá.C:7: undefined reference to ‘cout’
…/olá.C:7: undefined reference to ‘ostream::operator<<(char const *)’
…/olá.C:7: undefined reference to ‘ostream::operator<<(ostream &(*)(ostream &))’
collect2: ld returned 1 exit status
```

Para conseguir construir o executável com sucesso, no entanto, é necessário especificar todos os ficheiros objecto e arquivos de ficheiros objecto necessários (e que incluem, entre outros, o arquivo da biblioteca padrão do C++ libstdc++.a):

```
c++ -nostdlib -o olá /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc-lib/i386-redhat-
linux/2.96/crtbegin.o olá.o /usr/lib/gcc-lib/i386-redhat-
linux/2.96/libstdc++.a /usr/lib/gcc-lib/i386-redhat-
linux/2.96/libgcc.a /usr/lib/libc.a /usr/lib/gcc-lib/i386-redhat-
linux/2.96/libgcc.a /usr/lib/gcc-lib/i386-redhat-
linux/2.96/crtend.o /usr/lib/crtn.o
```

Não se preocupe se não sabe para que serve cada um dos ficheiros! Basta retirar a opção `-nostdlib` e tudo isto é feito automaticamente...

Fusão dinâmica

Quase todos os sistemas operativos suportam fusão dinâmica. A ideia é que a fusão com os ficheiros de arquivo das bibliotecas usadas pelo programa pode ser adiada, sendo realizada dinamicamente apenas quando o programa é executado. Esta solução tem como vantagens levar a ficheiros executáveis bastante mais pequenos e permitir a actualização dos ficheiros de arquivo das bibliotecas sem obrigar a reconstruir o executável (note-se que o executável e as bibliotecas, que neste caso se dizem dinâmicas ou partilhadas, podem ser fornecidos por entidades independentes). Os ficheiros de arquivo de bibliotecas dinâmicas ou partilhadas têm extensão `.a`, em Linux, e `.dll` em Windows. Para mais pormenores ver [**?**, Capitulo 7].

8.2.4 Arquivos

É possível colocar um conjunto de ficheiros objecto relacionados num ficheiro de arquivo. Estes ficheiros têm prefixo `lib` e extensão `.a` (de *archive*). É típico que os ficheiros objecto de uma biblioteca de ferramentas sejam colocados num único ficheiro de arquivo, para simplificar a sua fusão com os ficheiros objecto de diferentes programas. Por exemplo, os ficheiros objecto da biblioteca padrão da linguagem C++ (de nome `stdc++`) estão arquivados no ficheiro `libstdc++.a` (que se encontra algures num sub-directório do directório `/usr/lib`).

Para arquivar ficheiros objecto não se usa o fusor: usa-se o chamado *arquivador*. O programa arquivador invoca-se normalmente como se segue:

```
ar ru libbibliteca.a módulo1.o módulo2.o ...
```

Onde `ru` são opções passadas ao arquivador (`r` significa *replace* e `u` significa *update*), *`biblioteca`* é o nome da biblioteca cujo arquivo se pretende criar, e *`módulo1`* etc., são os nomes dos módulos que constituem a biblioteca e cujos ficheiros objecto se pretende arquivar.

Como cada módulo incluído numa biblioteca tem um ficheiro de interface e produz um ficheiro objecto que é guardado no arquivo dessa biblioteca, pode-se dizer que as bibliotecas são representadas por:

1. Ficheiro de arquivo dos ficheiros objecto (um ficheiro objecto por módulo) da biblioteca.
2. Ficheiros de interface de cada módulo (que podem incluir outros ficheiros de interface).

É o que se passa com a biblioteca padrão da linguagem C++, representada em Linux pelo ficheiro de arquivo `libstdc++.a` e pelos vários ficheiros de interface que a compõem (`iostream`, `string`, `cmath`, etc.).

Exemplo

Suponha-se uma biblioteca de nome `saudações` constituída (para já) apenas pelo seguinte módulo:

`olá.H`

```
void olá();
```

`olá.C`

```
#include "olá.H"

#include <iostream>

using namespace std;

void olá()
{
    cout << "Olá mundo!" << endl;
}
```

Para construir o ficheiro de arquivo da biblioteca pode-se usar os seguintes comandos:

```
c++ -Wall -ansi -pedantic -g -c olá.C
ar ru libsaudações.a olá.o
```

em que o primeiro comando pré-processa e compila o ficheiro de implementação do módulo `olá` e o segundo arquiva o ficheiro objecto resultante no arquivo da biblioteca `saudações`, de nome `libsaudações.a`.

Seja agora um programa de nome `saúda` que pretende usar o procedimento `olá()`. Para isso deve começar por incluir o ficheiro de interface `olá.H`, que se pretende seja tomado por um ficheiro oficial (inclusão com <> e não `""`):

`saúda.C`

```
#include <olá.H>

int main()
{
    olá();
}
```

O pré-processamento e compilação do ficheiro de implementação `saúda.C` com o comando

```
c++ -Wall -ansi -pedantic -g -c saúda.C
```

resulta na mensagem de erro

```
saúda.C:1: olá.H: No such file or directory
```

O problema é que, para incluir `olá.H` como um ficheiro de interface oficial, uma de três coisas tem de ocorrer:

1. O ficheiro `olá.H` está num local oficial (e.g., `/usr/include` em Linux).
2. Dá-se a opção `-I.` ao comando `c++` para que ele acrescente o directório corrente (`.`) à lista de directórios onde os ficheiros de interface oficiais são procurados:

   ```
   c++ -Wall -ansi -pedantic -g -I. -c olá.C
   ```

3. Coloca-se o ficheiro `olá.H` num directório criado para o efeito (e.g., `$HOME/include`, onde `$HOME` é o directório "casa" do utilizador corrente em Linux) e coloca-se esse directório na variável de ambiente (*environment variable*) `CPLUS_INCLUDE_PATH`, que contém uma lista de directórios que, para além dos usuais, devem ser considerados como contendo ficheiros de interface oficiais:

   ```
   mkdir $HOME/include
   setenv CPLUS_INCLUDE_PATH $HOME/include
   mv olá.H $HOME/include
   c++ -Wall -ansi -pedantic -g -c saúda.C
   ```

onde o primeiro comando (criar directório) só têm de ser dado uma vez, e o segundo tem de ser dado sempre que se entra no Linux (e só funciona com o interpretador de comandos `/bin/tcsh`), pelo que é um bom candidato a fazer parte do ficheiro `.tcshrc`.

Falta agora fundir o ficheiro `saúda.o` com os ficheiros objecto do ficheiro arquivo `libsaudações.a` (que por acaso é só um: `olá.o`). Para isso podem-se usar um de três procedimentos:

1. Fusão directa com o ficheiro de arquivo:

   ```
   c++ -o saúda saúda.o libsaudações.a
   ```

2. Fusão usando a opção `-l` (*link with*):

   ```
   c++ -o saúda saúda.o -L. -lsaudações
   ```

   É necessária a opção `-L.` para que o comando `c++` acrescente o directório corrente (`.`) à lista de directórios onde os ficheiros de arquivo oficiais são procurados.

3. Fusão usando a opção `-l`, mas convencendo o comando `c++` de que o arquivo em causa é oficial. Para isso:

   (a) coloca-se o ficheiro `libsaudações.a` num local oficial (e.g., `/usr/lib` em Linux); ou

   (b) coloca-se o ficheiro `libsaudações.a` num directório criado para o efeito, e.g., `$HOME/lib`, e coloca-se esse directório na variável de ambiente `LIBRARY_PATH`, que contém uma lista de directórios que, para além dos usuais, devem ser considerados como contendo os arquivos oficiais:

   ```
   mkdir $HOME/lib
   setenv LIBRARY_PATH $HOME/lib
   mv libsaudações.a $HOME/lib
   c++ -o saúda saúda.o -lsaudações
   ```

   onde o primeiro comando (criar directório) só têm de ser dado uma vez, e o segundo tem de ser dado sempre que se entra no Linux (e só funciona com a shell /bin/tcsh), pelo que é um bom candidato a fazer parte do ficheiro .tcshrc.

8.3 Ligação dos nomes

Os exemplos das secções anteriores tinham uma particularidade interessante: rotinas definidas num módulo podiam ser utilizadas noutros módulos, desde que apropriadamente declaradas. Será que é sempre assim? E será assim também para variáveis e constantes globais? E o caso das classes?

Nesta secção abordar-se-á a questão da ligação dos nomes em C++. A ligação de um nome tem a ver com a possibilidade de uma dada entidade (e.g., uma função, uma constante, uma classe, etc.), com um dado nome, poder ser utilizada fora do módulo (ou melhor, fora da unidade de tradução) onde foi definida. Em C++ existem três tipos de ligação possível para os nomes: nomes sem ligação, com ligação interna, e com ligação externa.

Quando um nome não tem ligação, a entidade correspondente não pode ser usada senão no seu âmbito de visibilidade. Uma variável local, por exemplo, não tem ligação, pois pode ser usada apenas no bloco onde foi definida.

Uma entidade cujo nome tenha ligação interna pode ser usada em variados âmbitos dentro da unidade de tradução onde foi definida, desde que o seu nome tenha sido previamente declarado nesses âmbitos, mas não pode ser usada em outras unidades de tradução do mesmo programa. Normalmente há uma unidade de tradução por módulo (que é o seu ficheiro de implementação pré-processado). Assim, pode-se dizer que uma entidade cujo nome tem ligação interna é apenas utilizável dentro do módulo que a define.

Uma entidade cujo nome tenha ligação externa pode ser usada em qualquer âmbito de qualquer unidade de tradução do programa, desde que o seu nome tenha sido previamente declarado nesses âmbitos. Assim, pode-se dizer que uma entidade cujo nome tem ligação externa é utilizável em qualquer módulo do programa.

Em geral as entidades definidas dentro de blocos de instruções, que são entidades locais, não têm ligação. Assim, esta discussão centrar-se-á essencialmente nas entidades com ligação, sendo ela interna ou externa, e que geralmente são as entidades globais, i.e., definidas fora de qualquer função ou procedimento.

A linguagem C++ impõe a chamada *regra da definição única*. Esta regra diz que, no mesmo contexto, não podem ser definidas mais do que uma entidade com o mesmo nome. Assim, resumidamente:

1. Um programa não pode definir mais do que uma rotina ou método que não seja em linha e com ligação externa se estes tiverem a mesma assinatura (podem ter o mesmo nome, desde que variem na lista de parâmetros).

2. Uma unidade de tradução não pode definir mais do que uma rotina não-membro que não seja em linha e com ligação interna se estas tiverem a mesma assinatura.

3. Uma variável ou constante global com ligação externa só pode ser definida uma vez no programa.

4. Uma variável ou constante global com ligação interna só pode ser definida uma vez na sua unidade de tradução.

5. Uma classe ou um tipo enumerado não-local só pode ser definido uma vez em cada unidade de tradução e, se for definido em diferentes unidades de tradução, a sua definição tem de ser idêntica.

8.3.1 Vantagens de restringir a ligação dos nomes

É muito importante perceber que há vantagens em restringir a utilização de determinadas ferramentas a um dado módulo: esta restrição corresponde a distinguir entre ferramentas que fazem apenas parte da implementação do módulo e ferramentas que são disponibilizadas na sua interface. É a possibilidade de distinguir entre o que está restrito e o que não está que faz com que os módulos físicos mereçam o nome que têm...

Suponha-se, por exemplo, um módulo `vectores` com funções e procedimentos especializados em operações sobre vectores de inteiros. Esse módulo poderia consistir nos seguintes ficheiros de interface e implementação:

matrizes.H

```
/** Devolve o maior dos valores dos itens do vector v.
    @pre  PC ≡ 0 < v.size().
    @post CO ≡ (Q j : 0 ≤ j < v.size() : m[j] ≤ máximoDe) ∧
               (E j : 0 ≤ j < v.size() : m[j] = máximoDe). */
double máximoDe(vector<double> const& v);

/** Devolve o índice do primeiro item com o máximo valor do vector v.
    @pre  PC ≡ 1 ≤ v.size().
    @post CO ≡ 0 ≤ índiceDoPrimeiroMáximoDe < v.size()∧
  (Q j : 0 ≤ j < v.size() : v[j] ≤ v[índiceDoPrimeiroMáximoDe]) ∧
  (Q j : 0 ≤ j < índiceDoPrimeiroMáximoDe : v[j] < v[índiceDoPrimeiroMáximoDe]). */
int índiceDoPrimeiroMáximoDe(vector<int> const& v);

/** Ordena os valores do vector v.
    @pre  PC ≡ v = v.
    @post CO ≡ perm(v, v) ∧ (Q i, j : 0 ≤ i ≤ j < v.size() : v[i] ≤ v[j]). */
void ordenaPorOrdemNãoDecrescente(vector<int>& v);
```

matrizes.C

```
#include <cassert>

using namespace std;

/** Verifica se o valor de máximo é o máximo dos valores num vector v.
    @pre  PC ≡ 0 < v.size().
    @post CO ≡ éMáximoDe = ((Q j : 0 ≤ j < v.size() : m[j] ≤ máximo) ∧
                            (E j : 0 ≤ j < v.size() : m[j] = máximo)). */
bool éMáximoDe(int const máximo, vector<int> const& v)
{
    assert(0 < v.size());

    bool existe = false;
    // CI ≡ existe = (E j : 0 ≤ j < i : m[j] = máximo) ∧
    //      (Q j : 0 ≤ j < i : m[j] ≤ máximo).
    for(vector<int>::size_type i = 0; i != v.size(); ++i) {
        if(máximo < v[i])
```

```
            return false;
        existe = existe or máximo == v[i];
    }
    return existe;
}

/** Verifica se os valores do vector v estão por ordem não decrescente.
    @pre PC ≡ V.
    @post CO ≡ éNãoDecrescente =
              (Q j, k : 0 ≤ j ≤ k < v.size() : v[j] ≤ v[k]). */
bool éNãoDecrescente(vector<int> const& v)
{
    if(v.size() <= 1)
        return true;
    // CI ≡ (Q j, k : 0 ≤ j ≤ k < i : v[j] ≤ v[k]).
    for(vector<int>::size_type i = 1; i != v.size(); ++i)
        if(v[i - 1] > v[i])
            return false;
    return true;
}

double máximoDe(vector<double> const& v);
{
    assert(0 < v.size());

    double máximo = v[0];
    // CI ≡ (Q j : 0 ≤ j < i : v[j] ≤ máximo) ∧
    //      (E j : 0 ≤ j < i : m[j] = máximo).
    for(vector<double>::size_type i = 1; i != v.size(); ++i)
        if(máximo < v[i])
            máximo = v[i];

    assert(éMáximoDe(máximo, v));

    return máximo;
}

int índiceDoPrimeiroMáximoDe(vector<int> const& v)
{
    assert(1 <= v.size());

    int i = 0;
    // CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : v[j] ≤ v[i]) ∧
    //      (Q j : 0 ≤ j < i : v[j] < v[i]) ∧ 0 ≤ k ≤ v.size().
    for(vector<int>::size_type int k = 1; k != v.size(); ++k)
        if(v[k] > v[i])
```

```
            i = k;

    assert(0 <= i and i < v.size() and éMáximoDe(v[i], v));

    return i;
}

void ordenaPorOrdemNãoDecrescente(vector<int>& v)
{
    ... // um algoritmo de ordenação eficiente...

    assert(éNãoDecrescente(v));
}
```

No ficheiro de implementação deste módulo existem duas funções de utilidade restrita ao módulo. A primeira, `éMáximoDe()`, serve para verificar se um dado valor é o máximo dos valores contidos num vector. A segunda, `éNãoDecrescente()`, serve para verificar se os itens de um vector estão por ordem crescente (ou melhor, se formam uma sucessão monótona não-decrescente). Ambas são usadas em instruções de asserção para verificar se (pelo menos parte) das condições objectivo das outras rotinas do módulo são cumpridas.

Que estas duas funções não têm qualquer utilidade directa para o programador consumidor desde módulo é evidente. Saber se um valor é o maior dos valores dos itens de um vector não é muito útil em geral. O que é útil é saber qual é o maior dos valores dos itens de um vector. De igual forma não é muito útil em geral saber se um vector está ordenado. É muito mais útil ter um procedimento para ordenar esse vector.

Haverá alguma desvantagem em deixar que essas funções sejam utilizadas fora deste módulo? Em geral sim. Se são ferramentas úteis apenas dentro do módulo, então o programador do módulo pode decidir alterar o seu nome ou mesmo eliminá-las sem "dar cavaco a ninguém". Se isso acontecesse, o incauto programador consumidor dessas funções ficaria numa situação incómoda... Além disso, corre-se o risco de o nome dessas funções colidir com o nome de funções existentes noutros módulos (ver Secção 8.6.1 mais abaixo).

Assim, seria desejável que as duas funções de verificação tivessem ligação interna.

8.3.2 Espaços nominativos sem nome

A melhor forma de restringir a utilização de uma entidade ao módulo em que está definida é colocá-la dentro de um espaço nominativo sem nome (ver Secção 8.6.2). Assim, a solução para o problema apresentado na secção anterior passa por colocar as duas funções de verificação num espaço nominativo sem nome:

`matrizes.C`

```
#include <cassert>
```

```
using namespace std;

namespace { // Aqui definem-se as ferramentas de utilidade restrita a este módulo:

    /** Verifica se o valor de máximo é o máximo dos valores num vector v.
        @pre  PC ≡ 0 < v.size().
        @post CO ≡ éMáximoDe = ((Q j : 0 ≤ j < v.size() : m[j] ≤ máximo) ∧
                               (E j : 0 ≤ j < v.size() : m[j] = máximo)). */
    bool éMáximoDe(int const máximo, vector<int> const& v)
    {
        assert(0 < v.size());

        bool existe = false;
        // CI ≡ existe = (E j : 0 ≤ j < i : m[j] = máximo) ∧
        //      (Q j : 0 ≤ j < i : m[j] ≤ máximo).
        for(vector<int>::size_type i = 0; i != v.size(); ++i) {
            if(máximo < v[i])
                return false;
            existe = existe or máximo == v[i];
        }
        return existe;
    }

    /** Verifica se os valores do vector v estão por ordem não decrescente.
        @pre  PC ≡ V.
        @post CO ≡ éNãoDecrescente =
                  (Q j, k : 0 ≤ j ≤ k < v.size() : v[j] ≤ v[k]). */
    bool éNãoDecrescente(vector<int> const& v)
    {
        if(v.size() <= 1)
            return true;
        // CI ≡ (Q j, k : 0 ≤ j ≤ k < i : v[j] ≤ v[k]).
        for(vector<int>::size_type i = 1; i != v.size(); ++i)
            if(v[i - 1] > v[i])
                return false;
        return true;
    }

}

double máximoDe(vector<double> const& v);
{
    assert(0 < v.size());

    double máximo = v[0];
    // CI ≡ (Q j : 0 ≤ j < i : v[j] ≤ máximo) ∧
```

```
    //      (E j : 0 ≤ j < i : m[j] = máximo).
    for(vector<double>::size_type i = 1; i != v.size(); ++i)
        if(máximo < v[i])
            máximo = v[i];

    assert(éMáximoDe(máximo, v));

    return máximo;
}

int índiceDoPrimeiroMáximoDe(vector<int> const& v)
{
    assert(1 <= v.size());

    int i = 0;
    // CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : v[j] ≤ v[i]) ∧
    //      (Q j : 0 ≤ j < i : v[j] < v[i]) ∧ 0 ≤ k ≤ v.size().
    for(vector<int>::size_type int k = 1; k != v.size(); ++k)
        if(v[k] > v[i])
            i = k;

    assert(0 <= i and i < n and éMáximoDe(v[i], v));

    return i;
}

void ordenaPorOrdemNãoDecrescente(int m[], int n)
{
    assert(n >= 0);

    ... // um algoritmo de ordenação eficiente...

    assert(éNãoDecrescente(m, n));
}
```

Em termos técnicos a definição de entidades dentro de um espaço nominativo sem nome não lhes dá obrigatoriamente ligação interna. O que acontece na realidade é que o compilador atribui um nome único ao espaço nominativo. Em termos práticos o efeito é o mesmo que se as entidades tivessem ligação interna.

8.3.3 Ligação de rotinas, variáveis e constantes

É importante voltar a distinguir entre os conceitos de definição e declaração. Uma definição cria uma entidade. Uma declaração indica a existência de uma entidade, indicando o seu nome e cararacterísticas. Uma definição é sempre também uma declaração. Mas uma declaração no seu sentido estrito não é uma definição, pois indicar a existência de uma entidade e o seu nome não é o mesmo que criar essa entidade: essa entidade foi ou será criada num outro local.

Nas secções seguintes apresentam-se exemplos que permitem distinguir entre declarações e definições de vários tipos de entidades e saber qual a ligação dos seus nomes. Nas tabelas, admite-se sempre que existem dois módulos no programa: o módulo do lado esquerdo (`define`) é o módulo que define a maior parte das entidades enquanto o módulo do lado direito (`usa`) é o módulo que as usa.

Rotinas

A declaração (em sentido estrito) de uma rotina faz-se colocando o seu cabeçalho seguido de `;`. Por exemplo:

```
/** Devolve a potência n de x.
    @pre  PC ≡ n ≥ 0 ∨ x ≠ 0.
    @post CO ≡ potência = xⁿ. */
double potência(double const x, int const n); // função.

/** Troca os valores dos dois argumentos.
    @pre  PC ≡ a = a ∧ b = b.
    @post CO ≡ a = b ∧ b = a. */
void troca(int& a, int& b);                   // procedimento.
```

A definição de uma rotina faz-se indicando o seu corpo imediatamente após o cabeçalho. Por exemplo:

```
double potência(double const x, int const n)
{
    assert(n >= 0 or x != 0.0);

    if(n < 0)
        return 1.0 / potência(x, -n);
    double r = 1.0;
    // CI ≡ r = xⁱ ∧ 0 ≤ i ≤ n.
    for(int i = 0; i != n; ++i)
        r *= x;
    return r;
}

void troca(int& a, int& b)
{
    int const auxiliar = a;
    a = b;
    b = auxiliar;
}
```

Os métodos de uma classe têm ligação externa se a respectiva classe tiver ligação externa. Caso contrário não têm ligação, uma vez que as classes também ou não têm ligação ou têm ligação externa (ver mais abaixo).

Em C++ não é possível definir funções ou procedimentos locais. Sobram, pois, as funções ou procedimentos globais (ou definidos num espaço nominativo), que têm sempre ligação. Por omissão as funções e procedimentos globais têm ligação externa. Para que uma função ou procedimento tenha ligação interna, é necessário preceder a sua declaração do especificador de armazenamento `static`. É um erro declarar uma função ou procedimento como tendo ligação externa e, posteriormente, declará-lo com tendo ligação interna.

Sejam os dois módulos representados na Figura 8.1.

Ao fundir os dois módulos da figura (e admitindo que se eliminaram os erros de compilação assinalados) o fusor gerará erros como os seguintes:

```
.../usa.C:35: undefined reference to 'f3(void)'
.../usa.C:37: undefined reference to 'f5(void)'
```

No primeiro caso, o erro ocorre porque o único procedimento `f3()` que existe tem ligação interna ao módulo `define` (a sua definição é precedida do especificador `static`). No segundo caso, o procedimento `f5()` foi declarado e usado em `usa.C` mas nenhum módulo o define.

O especificador `extern`, embora útil no caso das variáveis e constantes, como se verá mais abaixo, é inútil no caso das rotinas[12]. O que determina a ligação é o especificador `static`. Se uma função ou procedimento for declarado `static`, terá ligação interna. Assim, o seguinte código declara e define um procedimento com ligação interna:

```
static void f7();
extern void f7()
{ ... }
```

Variáveis

As variáveis definidas dentro de um bloco, i.e., as variáveis locais, não têm ligação. As variáveis membro de instância de uma classe têm um esquema de acesso próprio, descrito no Capítulo 6. As variáveis membro de classe (ou estáticas, ver Secção 6.5.5) têm ligação externa se a respectiva classe tiver ligação externa. Caso contrário não terão ligação, uma vez que as classes ou não têm ligação ou têm ligação externa. Os casos mais interessantes de ligação de variáveis dizem respeito a variáveis globais (ou definidas num espaço nominativo), que têm sempre ligação interna ou externa.

[12]Na realidade o especificador `extern` não é totalmente inútil em conjunto com rotinas. Para se poder invocar, a partir de código C++, uma rotina num módulo em linguagem C, é necessário preceder a declaração dessa função ou procedimento de `extern "C"`, como se segue:

```
extern "C" int atoi(char char[]);
```

define.C	usa.C
`void f4(); // externo` `void f1() // externo` `{...}` `void f2() // externo` `{...}` `static void f3() // interno` `{...}` `void f4() // externo` `{ f3(); }`	`void f1(); // externo` `void f3(); // externo` `extern void f2(); // externo` `void f5(); // externo` `void f6(); // externo` `/* Erro de compilação! Declarado externo e depois definido interno: */` `static void f6()` `{...}` `/* Erro de compilação! Não está definido dentro do módulo: */` `static void f7(); // interno.` `static void f8(); // interno.` `extern void f8(); // interno.` `void f8() // interno.` `{...}` `int main()` `{` `    f1();` `    f2();` `    f3();` `    // Erro de compilação, falta declaração:` `    f4();` `    f5();` `    f8();` `}`

Figura 8.1: Exemplo de ligação para rotinas. Declarações em itálico e fontes de erros de compilação a vermelho.

Antes de discutir a ligação de variáveis globais, porém, é conveniente reafirmar o que vem sendo dito: é má ideia usar variáveis globais!

Até este momento neste texto nunca se declararam variáveis globais. Há dois factores que permitem distinguir uma declaração de uma definição no caso das variáveis. Para que uma declaração de uma variável seja uma declaração no sentido estrito, i.e., para que não seja também uma definição, têm de acontecer duas coisas:

1. A declaração tem de ser precedida do especificador `extern`.
2. A declaração não pode inicializar a variável.

Se uma declaração possuir uma inicialização explícita é forçosamente uma definição, mesmo que possua o especificador `extern`. Se uma definição de uma variável global não possuir uma inicialização explícita, será inicializada implicitamente com o valor por omissão do tipo respectivo, mesmo que o seu tipo seja um dos tipos básicos do C++. Ou seja, ao contrário do que acontece com as variáveis locais dos tipos básicos, as variáveis globais dos tipos básicos são inicializadas implicitamente com valor por omissão do respectivo tipo (que é sempre um valor nulo). Se o tipo da variável for uma classe sem construtor por omissão, então uma definição sem inicialização é um erro.

Por exemplo:

```
int i = 10;          // definição e inicialização explícita com 10.
int j;               // definição e inicialização implícita com 0.
extern int k = 0;    // definição e inicialização com 0.
extern int j;        // declaração.

class A {
  public:
    A(int i)
         : i(i) {
    }
  private:
    int i;
};

A a(10); // definição e inicialização com 10.
A b;     // tentativa de definição: falha porque A não tem construtor por omissão.
```

Uma variável global tem, por omissão, ligação externa. Para que uma variável global tenha ligação interna é necessário preceder a sua declaração do especificador `static`. Na realidade, como os especificadores `static` e `extern` não se podem aplicar em simultâneo, a presença do especificador `static` numa declaração obriga-a a ser também uma definição.

Sejam os dois módulos representados na Figura 8.2.

Ao fundir os dois módulos acima (e admitindo que se eliminaram os erros de compilação assinalados) o fusor gerará erros como os seguintes:

<table>
<tr><th>define.C</th><th>usa.C</th></tr>
<tr><td>

extern int v4; // externa

int v1; // externa

int v2 = 1; // externa

static int v3 = 3; // interna

int v4; // externa

</td><td>

extern int v1; // externa

extern int v2; // externa

extern int v3; // externa

extern int v5; // externa

extern int v6; // externa

/* Erro (ou aviso) de compilação! Declarada externa e depois definida interna: */
static int v6 = 12;

static int v7; // interna

extern int v7; // interna

int main()
{
   int i = v1 +
      v2 +
      v3 +
      // Erro de compilação, falta declaração:
      v4 +
      v5 +
      v7;
}

</td></tr>
</table>

Figura 8.2: Exemplo de ligação para variáveis. Declarações em itálico e fontes de erros de compilação a vermelho.

```
.../usa.C:35: undefined reference to 'v3'
.../usa.C:37: undefined reference to 'v5'
```

No primeiro caso, o erro ocorre porque a única variável `v3` que existe tem ligação interna ao módulo `define`, uma vez que a sua definição é precedida do especificador `static`. No segundo caso, a variável `v5` foi declarada em `usa.C` mas nenhum módulo a define.

Constantes

As constantes definidas dentro de um bloco, ou seja, constantes locais, não têm ligação. As constantes membro de instância de uma classe têm um esquema de acesso próprio, descrito no Capítulo 6. As constantes membro de classe (ou estáticas, ver Secção 6.5.5) têm ligação externa se a respectiva classe tiver ligação externa. Caso contrário não terão ligação, uma vez que as classes também ou não têm ligação ou têm ligação externa. Os casos mais interessantes de ligação de constantes dizem respeito a constantes globais (ou definidas num espaço nominativo), que têm sempre ligação interna ou externa.

Para que uma declaração de uma constante seja uma declaração no sentido estrito, i.e., para que não seja também uma definição, têm de acontecer duas coisas:

1. A declaração tem de ser precedida do especificador `extern`.
2. A declaração não pode inicializar a constante.

Se uma declaração possuir uma inicialização explícita, então é forçosamente uma definição, mesmo que possua o especificador `extern`. Se uma definição de uma constante global não possuir uma inicialização explícita, será inicializada implicitamente com o valor por omissão do tipo respectivo, excepto se o seu tipo seja um dos tipos básicos do C++ ou, em geral, um POD (*plain old datatype*). Se o tipo da constante for uma classe sem construtor por omissão, então uma definição sem inicialização é um erro. As regras que definem o que é exactamente um POD são complexas. No contexto deste texto bastará dizer que um agregado, entendido como uma classe só com variáveis membro públicas de tipos básicos ou de outros agregados ou matrizes de tipos básicos e agregados, é um POD. Por exemplo, a classe

```
struct Ponto {
    double x;
    double y;
};
```

é um agregado, e portanto um POD, logo:

```
Ponto const p;
```

é um erro, pois falta uma inicialização explícita.

Outros exemplos:

```
int const i = 10;         // definição e inicialização explícita com 10.
int const j;              // tentativa de definição: falha porque j é de um ti-
po básico.
extern int const k = 0; // definição e inicialização com 0.
extern int const i;       // declaração.

class A {
  public:
    A(int i)
        : i(i) {
    }
  private:
    int i;
};

A const a(10); // definição e inicialização com 10.
A const b;     // tentativa de definição: falha porque A
               // não tem construtor por omissão.
```

Uma constante global tem, por omissão, ligação interna, i.e., o oposto da ligação por omissão das variáveis. Para que uma constante global tenha ligação externa é necessário preceder a sua declaração do especificador `extern`. Assim sendo, a utilização do especificador `static` na declaração de constantes globais é redundante.

Sejam os dois módulos representados na Figura 8.3.

Ao fundir os dois módulos acima (e admitindo que se eliminaram os erros de compilação assinalados a vermelho) o fusor gerará os seguintes erros:

```
.../usa.C:35: undefined reference to 'c3'
.../usa.C:37: undefined reference to 'c5'
```

No primeiro caso, o erro ocorre porque a única constante `c3` que existe tem ligação interna ao módulo `define` (por omissão as constantes globais têm ligação interna). No segundo caso, a constante `c5` foi declarada em `usa.C` mas nenhum módulo a define.

8.3.4 Ligação de classes e tipos enumerados

Não é possível simplesmente declarar um tipo enumerado: é sempre necessário também defini-lo.

A distinção entre a declaração e a definição de uma classe é simples e semelhante sintacticamente distinção entre declaração e definição no caso das rotinas. A declaração no sentido estrito de uma classe termina em `;` logo após o seu nome. Se em vez de `;` surgir um bloco com a declaração os membros da classe, a declaração é também uma definição. Por exemplo:

define.C	usa.C
extern int const c4; // externa extern int const c1 =1;// externa extern int const c2 =1;// externa int const c3 = 3; // interna int const c4 = 33; // externa	*extern int const c1;* // externa *extern int const c2;* // externa *extern int const c3;* // externa *extern int const c5;* // externa *extern int const c6;* // externa /* Erro (ou aviso) de compilação! Declarada externa e depois definida interna: */ static int const c6 = 12; int const c7 = 11; // interna *extern int const c7;* // interna int main() { int i = c1 + c2 + c3 + // Erro de compilação, falta declaração: c4 + c5 + c7; }

Figura 8.3: Exemplo de ligação para constantes. Declarações em itálico e fontes de erros de compilação a vermelho.

```
class Racional; // declaração.

class Racional { // definição.
  public:
    ...

  private:
    ...
};
```

A declaração em sentido estrito de uma classe é útil em poucas situações. Como só a definição da classe inclui a definição dos seus atributos (variáveis e constantes membro) de instância, só com a definição o compilador pode calcular o espaço de memória ocupado pelas instâncias dessa classe. Assim, não é possível definir variáveis de uma classe sem que a classe tenha sido definida antes, embora se possam definir referências e ponteiros (a ver no Capítulo 9) para essa classe. A declaração em sentido estrito usa-se principalmente:

1. Quando se estão a definir duas classes que se usam mutuamente, mas em que nenhuma pode ser considerada mais importante que a outra (e portanto nenhuma é embutida). Note-se que pelo menos uma das classes não pode ter atributos de instância da outra classe. Por exemplo:

```
class B;

class A {
  public:
    A() {
    }
    A(const B&); // Declaração de B essencial!
    ...
};

class B {
  public:
    B() {
    }
    ...
  private:
    A m[10]; // Definição de A essencial!
};
```

2. Quando se está a definir uma classe com outra embutida e se pretendem separar as duas definições por uma questão de clareza. Por exemplo:

```
class ListaInt {
  public:
```

```
        ...
        class Iterador;
        class IteradorConstante;
        ...
    };

    class ListaInt::Iterador {
        ...
    };

    class ListaInt::IteradorConstante {
        ...
    };
```

É importante também perceber que a definição de uma classe não inclui a definição de funções e procedimentos membro da classe, excepto se esta for feita internamente à classe. Assim, pode-se distinguir entre a *definição de uma classe* e a *definição completa de uma classe*, que inclui a definição de todos os seus membros.

Podem-se definir tipo enumerados ou mesmo classes dentro de um bloco de instruções, embora com algumas restrições. Por exemplo, o seguinte código

```
#include <iostream>

using namespace std;

int main()
{
    class Média {
      public:
        Média()
            : soma_dos_valores(0.0), número_de_valores(0) {
        }
        Média& operator += (double const novo_valor) {
            soma_dos_valores += novo_valor;
            ++número_de_valores;
            return *this;
        }
        operator double () const {
            assert(numero_de_valores != 0);
            return soma_dos_valores / numero_de_valores;
        }
      private:
          double soma_dos_valores;
          int numero_de_valores;
    };
```

```
    Média m;
    m += 3;
    m += 5;
    m += 1;
    cout << m << endl;
}
```

define uma classe `Média`[13] localmente à função `main()`[14]. Note-se que não se pretende aqui defender este tipo de prática: são muito raros os casos em que se justifica definir uma classe localmente e este não é certamente um deles!

Uma classe ou tipo enumerado local não tem ligação. A sua utilização restringe-se ao bloco em que foi definido. Assim, a classe `Média` só pode ser usada no programa acima dentro da função `main()`.

Classes definidas dentro de outras classes, as chamadas *classes embutidas*, ou tipos enumerados definidos dentro outras classes, têm a mesma ligação que a classe que as envolve.

Finalmente, classes ou tipos enumerados definidos no contexto global têm sempre ligação externa. Note-se que, ao contrário do que sucede com as funções e procedimentos, a mesma classe ou tipo enumerado pode ser definido em múltiplas unidades de tradução de um mesmo programa: o fusor só verifica as faltas ou duplicações de definições de rotinas globais e de variáveis ou constantes globais. Isto acontece porque a informação presente na definição de uma classe é necessária na sua totalidade durante a compilação, com excepção das definições de rotinas membro que não são em linha, que o compilador não precisa de conhecer. Aliás, a informação presente na definição de uma classe é crucial para o compilador poder fazer o seu papel, ao contrário do que acontece, por exemplo, com as rotinas, que para poderem ser usadas tem apenas de ser declaradas. É por esta razão que as classes não são apenas declaradas nos ficheiros de interface: são também definidas. Desse modo, através da directiva `#include`, a definição de uma classe será incluída em todas as unidades de tradução de dela necessitam.

A regra de definição única tem, por isso, uma versão especial para as classes e enumerados: uma classe ou tipo enumerado com ligação externa pode ser definido em diferentes unidades de tradução desde que essas definições sejam rigorosamente equivalentes. É justamente para garantir essa equivalência que as definições de classes e enumerados são colocadas em ficheiros de interface.

8.4 Conteúdo dos ficheiros de interface e implementação

Que ferramentas se devem colocar em cada módulo? É algo difícil responder de uma forma taxativa a esta pergunta. Mas pode-se dizer que cada módulo deve corresponder a um conjunto

[13] A classe `Média` define um operador de conversão implícita para `double`. Na definição destes operadores de conversão não é necessário (nem permitido) colocar o tipo de devolução, pois este é igual ao tipo indicado após a palavra-chave `operator`. É este conversor que permite inserir uma instância da classe `Média` num canal de saída sem qualquer problema: essa instância é convertida para um `double`, que se pode inserir no canal, sendo para isso calculada a média dos valores inseridos através do operador `+=`.

[14] Esta curiosa possibilidade de definir classes locais permite simular a definição de rotinas locais, que o C++ não suporta. Se está preparado para alguns pontapés na gramática, veja a Secção C.2.

muito coeso de rotinas, classes, etc. É vulgar um módulo conter apenas uma classe e algumas rotinas associadas intimamente a essa classe. Por vezes há mais do que uma classe, quando essas classes estão muito interligadas, quando cada uma delas não faz sentido sem a outra. Ou então várias pequenas classes independentes mas relacionadas conceptualmente. Outras vezes vezes um módulo não definirá classe nenhuma, como poderia suceder, por exemplo, com um módulo de funções matemáticas.

Depois de escolhidas e desenvolvidas as ferramentas que constituem cada módulo físico de um programa, há que decidir para cada módulo o que deve ficar no seu ficheiro de interface (`.H`) e o que deve ficar no seu ficheiro de implementação (`.C`). A ideia geral é que o ficheiro de interface deve conter o que for estritamente necessário para que as ferramentas definidas pelo módulo possam ser usadas em qualquer outro módulo por simples inclusão do correspondente ficheiro de interface. Ou seja, tipicamente o ficheiro de interface declara o que se define no ficheiro de implementação.

Algumas das ferramentas definidas num módulo são de utilidade apenas dentro desse módulo. Essas ferramentas devem ficar inacessíveis do exterior, não dando-lhes ligação interna, conforme explicado nas secções anteriores, mas usando espaços nominativos sem nome, como sugerido na Secção 8.3.2. Todas as outras ferramentas (com excepção de alguns tipos de constantes) deverão ter ligação externa.

As próximas secções apresentam algumas regras gerais acerca do que deve ou não constar em cada um dos ficheiros fonte de um módulo.

8.4.1 Relação entre interface e implementação

Para que o compilador possa verificar se o conteúdo do ficheiro de interface corresponde ao conteúdo do respectivo ficheiro de implementação, o ficheiro de implementação começa sempre por incluir o ficheiro de interface:

***módulo*.C**

```
#include "módulo.H" // ou <módulo.H>
...
```

8.4.2 Ferramentas de utilidade interna ao módulo

Definem-se no ficheiro de implementação (`.C`) dentro de um espaço nominativo sem nome:

***módulo*.C**

```
namespace {
    // Definição de ferramentas internas ao módulo, invisíveis do exterior:
    ...
}
```

8.4.3 Rotinas não-membro

As rotinas não-membro que não sejam em linha devem ser declaradas no ficheiro de interface (`.H`) e definidas no ficheiro de implementação (`.C`):

***módulo*.H**

```
// Declaração de rotinas não-membro e que não sejam em linha:
tipo nome(parâmetros...);
```

***módulo*.C**

```
// Definição de rotinas não-membro e que não sejam em linha:
tipo nome(parâmetros...)
{
    ... // corpo.
}
```

As rotinas não-membro em-linha devem ser definidas apenas no ficheiro de interface (`.H`):

***módulo*.H**

```
// Definição de rotinas não-membro e em linha:
inline tipo nome(parâmetros...) {
    ... // corpo.
}
```

Alternativamente, as rotinas não-membro em linha podem ser definidas num terceiro ficheiro do módulo, como descrito na Secção 8.4.15.

8.4.4 Variáveis globais

As variáveis globais não devem ser usadas.

8.4.5 Constantes globais

As constantes serão tratadas de forma diferente consoante o seu "tamanho". Se o tipo da constante for um tipo básico, uma matriz ou uma classe muito simples, então deverá ser definida com ligação interna no ficheiro de interface (`.H`). Desse modo, serão definidas tantas constantes todas iguais quantos os módulos em que esse ficheiro de interface for incluído. Compreende-se assim que a constante deva ser "pequena", para evitar programas ocupando demasiada memória, e rápida de construir, para evitar ineficiências.

***módulo*.H**

```
// Definição de constantes “pequenas” e “rápidas”:

tipo const nome1; // valor por omissão.

tipo const nome2 = expressão;

tipo const nome3(argumentos);
```

Se o tipo da constante não verificar estas condições, então a constante deverá ser definida com ligação externa no ficheiro de implementação (`.C`) e declarada no ficheiro de interface (`.H`).

***módulo*.H**

```
// Declaração de constantes “grandes” ou “lentas”:

extern tipo const nome1;

extern tipo const nome2;

extern tipo const nome3;
```

***módulo*.C**

```
#include "módulo.H"

...

// Definição de constantes “grandes” ou “lentas”:

tipo const nome1; // valor por omissão.

tipo const nome2 = expressão;

tipo const nome3(argumentos);
```

Note-se que não é necessário o especificador `extern` no ficheiro de implementação porque este inclui sempre o respectivo ficheiro de interface, que possui a declaração das constantes como externas.

8.4.6 Tipos enumerados não-membro

Os tipos enumerados têm de ser definidos no ficheiro de interface (`.H`).

***módulo*.H**

```
// Definições de enumerados:

enum Nome { ... };
```

8.4.7 Classes não-membro

As classes, por razões que já se viram atrás, devem ser definidas apenas no ficheiro de interface (`.H`).

***módulo*.H**

```
// Definição de classes:

class Nome {
    ... // aqui tanto quanto possível só declarações.
};
```

8.4.8 Métodos (rotinas membro)

Os métodos podem ser definidos dentro da definição da classe, o que os torna automaticamente em linha. Mas não é recomendável fazê-lo, pois isso leva a uma mistura de implementação com interface que não facilita a leitura do código. Assim, os métodos, quer de classe quer de instância, são sempre declarados dentro da classe e portanto no ficheiro de interface (`.H`). A sua definição será feita fora da classe. No caso dos métodos que não são em linha, a definição far-se-á no ficheiro de implementação (`.C`). No caso dos métodos em linha, a definição far-se-á no ficheiro de interface.

***módulo*.H**

```
class Nome {
    ...
    // Declarações de métodos:

    tipo nome1(parâmetros...); // membro de instância.

    tipo nome2(parâmetros...); // membro de instância.

    static tipo nome3(parâmetros...); // membro de classe.

    static tipo nome4(parâmetros...); // membro de classe.
    ...
};

// Definições de métodos em linha:

inline tipo Nome::nome2(parâmetros...) { // membro de instância.
    ... // corpo.
}
```

```
inline tipo Nome::nome4(parâmetros...) { // membro de classe.
    ... // corpo.
}
```

módulo.C

```
// Definições de métodos que não são em linha:

tipo Nome::nome1(parâmetros...) // membro de instância.
    ... // corpo.
}

tipo Nome::nome3(parâmetros...); // membro de classe.
    ... // corpo.
}
```

Alternativamente, os métodos em linha podem ser definidos num terceiro ficheiro do módulo, como descrito na Secção 8.4.15.

8.4.9 Variáveis e constantes membro de instância

As variáveis e constantes membro de instância são definidas dentro da própria classe e inicializadas pelos seus construtores, pelo que não requerem qualquer tratamento especial.

8.4.10 Variáveis membro de classe

As variáveis membro de classe declaram-se dentro da classe, e portanto no ficheiro de interface (`.H`), e definem-se no ficheiro de implementação (`.C`).

módulo.H

```
class Nome {
    ...
    // Declarações de variáveis membro de classe:

    static tipo nome;
    ...
};
```

módulo.C

```
// Definições de variáveis membro de classe:

tipo Nome::nome; // valor por omissão.

tipo Nome::nome = expressão;

tipo Nome::nome(argumentos);
```

8.4.11 Constantes membro de classe

As constantes membro de classe tratam-se exactamente como as variáveis membro de classe: declaram-se na classe, e portanto no ficheiro de interface (`.H`), e definem-se no ficheiro de implementação (`.C`).

***módulo*.H**

```
class Nome {
    ...
    // Declarações de constantes membro de classe:

    static tipo const nome;
    ...
};
```

***módulo*.C**

```
// Definições de constantes membro de classe:

tipo const Nome::nome; // valor por omissão.

tipo const Nome::nome = expressão;

tipo const Nome::nome(argumentos);
```

Uma excepção a esta regra são as constantes membro de classe de tipos aritméticos inteiros. Estas podem ser definidas dentro da classe e usadas, por isso, para indicar a dimensão de matrizes membro. Por exemplo:

módulo.H

```
class Nome {
    ...
    // Declarações de constantes membro de classe de tipos aritméticos inteiros:

    static int const limite = 100;
    ...
    int matriz[limite];
    ...
};
```

8.4.12 Classes membro (embutidas)

As classes membro de outras classes, ou seja, embutidas, podem ser definidas dentro da classe que as envolve. Mas é muitas vezes preferível, embora nem sempre possível, defini-las à parte.

Em qualquer dos casos a definição será feita dentro do mesmo ficheiro de interface (`.H`) da classe envolvente. Aos membros de classes embutidas aplicam-se os mesmos comentários que aos membros da classe envolvente.

Forma recomendada:

`módulo.H`

```
class Nome {
    ...
    // Declarações de classes embutidas:

    class OutroNome;
    ...
};

...

class Nome::OutroNome {
    ...
};
```

Alternativa:

`módulo.H`

```
class Nome {
    ...
    // Definições de classes embutidas:

    class OutroNome {
        ...
    };
    ...
};
```

8.4.13 Enumerados membro

Os tipos enumerados membro têm de ser definidos dentro da classe e portanto no ficheiro de interface (`.H`).

`módulo.H`

```
class Data {
    ...
```

```
    // Definições de enumerados embutidos:

    enum DiaDaSemana {
        primeiro,
        segunda-feira = primeiro,
        terça_feira,
        ...
        domingo,
        ultimo = domingo
    };
    ...
};
```

8.4.14 Evitando erros devido a inclusões múltiplas

A implementação da modularização física em C++, sendo bastante primitiva, põe alguns problemas. Suponha-se que se está a desenvolver um programa dividido em três módulos. Dois deles, `A` e `B`, disponibilizam ferramentas. O terceiro, `C`, contém o programa principal e faz uso de ferramentas de `A` e `B`, sendo constituído apenas por um ficheiro de implementação (`C.C`) que inclui os ficheiros de interface dos outros módulos (`A.H` e `B.H`):

`C.C`

```
#include "A.H"
#include "B.H"
...
Restante conteúdo de C.C.
```

Suponha-se ainda que o ficheiro de interface do módulo `B` necessita de algumas declarações do módulo `A`. Nesse caso:

`B.H`

```
#include "A.H"
...
Restante conteúdo de B.H.
```

Suponha-se finalmente que o ficheiro `A.H` define uma classe (o argumento seria o mesmo com um tipo enumerado, uma rotina em linha ou uma constante):

`A.H`

```
class A {
    ...
};
```

Que acontece quando se pré-processa o ficheiro `C.C`? O resultado é a unidade de tradução `C.ii`:

C.ii

```
# 1 "C.C"
# 1 "A.H" 1
class A {
    ...
};
# 2 "C.C" 2
# 2 "B.H" 1
# 1 "A.H" 1
class A {
    ...
};
# 2 "B.H" 2
...
Restante conteúdo de B.H.
# 3 "C.C" 2
...
Restante conteúdo de C.C.
```

Este ficheiro contém duas definições da classe `A`, o que viola a regra da definição única, pelo que não se poderá compilar com sucesso. O problema deve-se à inclusão múltipla do conteúdo do ficheiro `A.H` na unidade de tradução `C.ii`. Como resolver o problema?

O pré-processador fornece uma forma eficaz de evitar este problema recorrendo a compilação condicionada. Para isso basta envolver todo o código dentro dos ficheiros de interface da seguinte forma:

***módulo*.H**

```
#ifndef MÓDULO_H
#define MÓDULO_H

... // Conteúdo do ficheiro de interface.

#endif // MÓDULO_H
```

Desta forma, na segunda inclusão do ficheiro já estará definida a macro *MÓDULO*`_H`, e portanto o código não será colocado segunda vez na unidade de tradução. A macro deve ter um nome único entre todos os módulos, de modo a que este mecanismo não entre em conflito com o mesmo mecanismo noutros módulos também usados. Não é fácil garantir que o nome seja único, o que revela quão primitivo é o modelo de modularização física do C++. Uma das formas de tentar garantir unicidade no nome das macros é anexar-lhes os nomes dos pacotes a que o módulo pertence, como se verá mais à frente.

8.4.15 Ficheiro auxiliar de implementação

Os ficheiros de interface, como se viu, contêm mais informação do que aquela que, em rigor, corresponde à interface das ferramentas disponibilizadas pelo módulo correspondente. Em particular as rotinas e métodos em linha são totalmente definidos no ficheiro de interface. Para o programador consumidor de um módulo, a definição exacta destas rotinas é irrelevante. A sua presença no ficheiro de interface acaba até por ser um factor de distracção. Uma solução comum para este problema passa por utilizar um terceiro tipo de ficheiro em cada módulo: o ficheiro auxiliar de implementação. Sugere-se, pois, que os módulos sejam constituídos por (no máximo) três ficheiros fonte:

1. Ficheiro de interface (`.H`) – Com o conteúdo sugerido nas secções anteriores, mas sem definições de rotinas e métodos em linha e contendo as declarações de todas as rotinas não-membro, independentemente de serem ou não em linha. Este ficheiro deve terminar com a inclusão do respectivo ficheiro auxiliar de implementação (terminado em `_impl.H`).

2. Ficheiro auxiliar de implementação (`_impl.H`) – Com a definição todas as rotinas e métodos em linha.

3. Ficheiro de implementação (`.C`) - com o conteúdo sugerido anteriormente.

Desta forma a complexidade do ficheiro de interface reduz consideravelmente, o que facilita a sua leitura.

A mesma solução pode ser usada para resolver o problema da definição de rotinas e métodos modelo quando se usa compiladores que não suportam a palavra-chave `export`. Ver Capítulo 11 para mais pormenores.

8.5 Construção automática do ficheiro executável

A construção do ficheiro excutável de um programa constituído por vários módulos exige uma sequência de comandos. Em ambientes Unix (como o Linux) é possível automatizar a construção usando o *construtor*, que é um programa chamado `make`, e, se necessário, um ficheiro de construção.

O construtor tem um conjunto de regras implícitas que lhe permitem construir automaticamente pequenos programas. Infelizmente, está especializado para a linguagem C, pelo que são necessários alguns passos de configuração para usar a linguagem C++. Em particular, é necessário indicar claramente quais são os programas e respectivas opções que devem ser usados para pré-processar e compilar e para fundir. Para isso é necessário atribuir valores à variáveis de ambiente `CXX`, que guarda o nome do programa usado para pré-processar e compilar, `CC`, que guarda o nome do programa usado para fundir, e `CXXFLAGS`, que guarda as opções de compilação desejadas. Se se estiver a usar o interpretador de comandos `/bin/tcsh`, tal pode ser conseguido através dos comandos:

```
setenv CC c++
setenv CXX c++
setenv CXXFLAGS "-g -Wall -ansi -pedantic"
```

Estes comandos também podem ser colocados no ficheiro de configuração do interpretador de comandos: `~/.tcshrc`[15].

Depois das configurações anteriores, é fácil construir pequenos programas constituídos apenas por um ficheiro de implementação:

olá.C

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Olá mundo!" << endl;
}
```

Para construir o ficheiro executável basta dar o comando:

```
make olá
```

Em casos mais complicados, quando existem vários módulos num programa, é necessário criar um ficheiro de construção (*makefile*), de nome `Makefile`. Este ficheiro, cujo conteúdo pode ser muito complicado, consiste nas suas versões mais simples num conjunto de regras de dependência. Estas dependências têm o seguinte formato:

```
alvo: dependência dependência...
```

Estas regras servem para indicar ao construtor que dependências existem entre os ficheiros do programa, dependências essas que o construtor não sabe adivinhar.

A melhor forma de perceber os ficheiros de construção é estudar um exemplo. Suponha-se que um projecto deve construir dois ficheiros executáveis, um para calcular médias de alunos e outro para saber a nota de um dado aluno, e que esse projecto é constituído por três módulos, respectivamente `aluno`, `média` e `procura`. Para simplificar o exemplo não se usará o ficheiro auxiliar de implementação sugerido na secção anterior, pelo que cada módulo terá no máximo dois ficheiros fonte.

O primeiro módulo, `aluno`, define ferramentas para lidar com alunos e respectivas notas, e possui dois ficheiros fonte: `aluno.C` e `aluno.H`. Os outros módulos, `média` e `procura`,

[15]Essas alterações só têm efeito depois de se voltar a entrar no interpretador ou depois de se dar o comando `source ~/.tcshrc` na consola em causa.

definem os programas para cálculo de médias e pesquisa de alunos, e portanto consistem apenas num ficheiro de implementação cada um (`média.C` e `procura.C`). Os ficheiros fonte são[16]:

aluno.H

```
#include <string>
#include <iostream>

bool éPositiva(int nota);

const int nota_mínima_aprovação = 10;
const int nota_máxima = 20;

class Aluno {
  public:
    Aluno(std::string const& nome,
          int número, int nota = 20);
    std::string const& nome() const;
    int número() const;
    int nota() const;

  private:
    std::string nome_;
    int número_;
    int nota_;
};

inline Aluno::Aluno(std::string const& nome,
                    int const número, int const nota)
    : nome_(nome), número_(número), nota_(nota) {
    assert(0 <= nota and nota <= nota_máxima);
}

std::ostream& operator << (std::ostream& saída,
                           Aluno const& aluno);
```

aluno.C

```
#include "aluno.H"

using namespace std;
```

[16]Optou-se por não documentar o código para não o tornar exageradamente longo. Optou-se também por eliminar a verificação de erros do utilizador, pela mesma razão. Finalmente, distinguiu-se algo artificialmente entre métodos e rotinas em linha e não-em linha. A razão foi garantir que o ficheiro de implementação `aluno.C` não ficasse vazio.

```
bool éPositiva(int const nota)
{
    return nota >= nota_mínima_aprovação;
}

string const& Aluno::nome() const
{
    return nome_;
}

int Aluno::número() const
{
    return número_;
}

int Aluno::nota() const
{
    return nota_;
}

ostream& operator << (ostream& saída, Aluno const& aluno)
{
    return saída << aluno.nome() << ' ' << aluno.número()
                 << ' ' << aluno.nota();
}
```

média.C

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

#include "aluno.H"

int main()
{
    cout << "Introduza número de alunos: ";
    int número_de_alunos;
    cin >> número_de_alunos;

    cout << "Introduza os alunos (nome número nota):"
         << endl;
    vector<Aluno> alunos;
```

```
    for(int i = 0; i != número_de_alunos; ++i) {
        string nome;
        int número;
        int nota;
        cin >> nome >> número >> nota;
        alunos.push_back(Aluno(nome, número, nota));
    }

    double soma = 0.0;
    for(int i = 0; i != número_de_alunos; ++i)
        soma += alunos[i].nota();

   cout << "A média é: " << soma / número_de_alunos
        << " valores." << endl;
}
```

procura.C

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

#include "aluno.H"

int main()
{
    vector<Aluno> alunos;
    alunos.push_back(Aluno("Zé", 1234, 15));
    alunos.push_back(Aluno("Zacarias", 666, 20));
    alunos.push_back(Aluno("Marta", 5465, 18));
    alunos.push_back(Aluno("Maria", 1111, 14));

    cout << "Pauta:" << endl;
    for(vector<Aluno>::size_type i = 0;
        i != alunos.size(); ++i)
        cout << alunos[i] << endl;

    cout << "De que aluno deseja saber a nota? ";
    string nome;
    cin >> nome;
    for(vector<Aluno>::size_type i = 0;
        i != alunos.size(); ++i)
        if(alunos[i].nome() == nome) {
            cout << "O aluno " << nome << " teve "
```

```
                    << alunos[i].nota() << endl;
                if(not éPositiva(alunos[i].nota()))
                    cout << "Este aluno reprovou." << endl;
            }
        }
```

Observando estes ficheiros fonte, é fácil verificar que:

1. Sempre que um dos ficheiros fonte do módulo `aluno` for alterado, o respectivo ficheiro objecto deve ser produzido de novo, pré-processando e compilando o respectivo ficheiro de implementação. Como o construtor sabe que os ficheiros objecto dependem dos ficheiros de implementação respectivos, só é necessário indicar explicitamente que o ficheiro objecto depende do ficheiro de interface. Ou seja, deve-se colocar no ficheiro de construção a seguinte linha:

   ```
   aluno.o: aluno.H
   ```

2. O ficheiro de implementação do módulo `média` inclui o ficheiro de interface `aluno.H`. Assim, é necessário indicar explicitamente que o ficheiro objecto `média.o` depende do ficheiro de interface `aluno.H`:

   ```
   média.o: aluno.H
   ```

3. O mesmo se passa para o módulo procura:

   ```
   procura.o: aluno.H
   ```

4. Finalmente, é necessário indicar que os ficheiros executáveis são obtidos, e portanto dependem, de determinados ficheiros objecto:

   ```
   média: média.o aluno.o
   procura: procura.o aluno.o
   ```

Assim, o ficheiro de construção ficará:

`Makefile`

```
média: média.o aluno.o
procura: procura.o aluno.o

aluno.o: aluno.H
média.o: aluno.H
procura.o: aluno.H
```

As três últimas regras podem ser geradas automaticamente pelo compilador se se usar a opção `-MM`. Sugere-se que o ficheiro de construção seja inicializado da seguinte forma:

```
c++ -MM aluno.C média.C procura.C > Makefile
```

O resultado será o ficheiro:

Makefile

```
aluno.o: aluno.C aluno.H
média.o: média.C aluno.H
procura.o: procura.C aluno.H
```

onde faltam as regras de construção dos ficheiros executáveis, que têm de ser acrescentadas à mão.

As regras obtidas automaticamente pelo comando `c++ -MM` indicam explicitamente que os ficheiros objecto dependem dos ficheiros de implementação, o que é redundante, pois o construtor sabe dessa dependência.

Com o ficheiro de construção sugerido, construir o programa `média`, por exemplo, é fácil. Basta dar o comando

```
make média
```

que o construtor `make` se encarregará de pré-processar, compilar e fundir apenas aquilo que for necessário para construir o ficheiro executável `média`. O construtor tenta sempre construir os alvos que lhe forem passados como argumento. Se se invocar o construtor sem quaisquer argumentos, então ele tentará construir o primeiro alvo indicado no ficheiro de construção. Assim, pode-se acrescentar uma regra inicial ao ficheiro de construção de modo a que o construtor tente construir os dois executáveis quando não lhe forem passados argumentos:

Makefile

```
all: média procura

média: média.o aluno.o
procura: procura.o aluno.o

aluno.o: aluno.H
média.o: aluno.H
procura.o: aluno.H
```

Neste caso para construir os dois executáveis basta dar o comando:

```
make
```

8.6 Modularização em pacotes

Já se viu atrás que o nível de modularização acima da modularização física é o nível de pacote. Um pacote é constituído normalmente pelas ferramentas de vários módulos físicos. Em C++ não existe suporte directo para a noção de pacote. A aproximação utilizada consiste na colocação dos vários módulos físicos de um pacote num *espaço nominativo* comum ao pacote. A noção de espaço nominativo é descrita brevemente nas próximas secções.

8.6.1 Colisão de nomes

Um problema comum em grandes projectos é a existência de entidades (classes, variáveis, funções ou procedimentos) com o mesmo nome, embora desenvolvidos por pessoas, equipas ou empresas diferentes. Quando isto acontece diz-se que ocorreu uma colisão de nomes.

Por exemplo, se se está a desenvolver uma aplicação de gestão, pode-se resolver comprar duas bibliotecas de ferramentas, uma com ferramentas de contabilidade, outra de logística. Esta é uma opção acertada a maior parte das vezes: é provavelmente mais rápido e barato comprar as bibliotecas a terceiros do que desenvolver as respectivas ferramentas de raiz.

Suponha-se que os fornecedores dessas bibliotecas são duas empresas diferentes e que estas foram fornecidas no formato de ficheiros de interface e ficheiros de arquivo contendo os ficheiro objecto de cada biblioteca.

Suponha-se ainda que ambas as bibliotecas definem um procedimento com a mesma assinatura[17]:

```
void consolida();
```

Simplificando, os ficheiros fornecidos por cada empresa são:

Biblioteca de logística

`liblogística.a` Ficheiro de arquivo da biblioteca. Contém (entre outros) o ficheiro objecto `produtos.o`, que define (entre outros) o procedimento `consolida()`.

`produtos.H`

```
#ifndef PRODUTOS_H
#define PRODUTOS_H

...
void consolida();
...

#endif // PRODUTOS_H
```

outros ficheiros de interface

[17] Fica a cargo do leitor imaginar o que cada um desses dois procedimentos faz.

Biblioteca de contabilidade

`libcontabilidade.a` Ficheiro de arquivo da biblioteca. Contém (entre outros) o ficheiro objecto `balanço.o`, que define (entre outros) o procedimento `consolida()`.

`balanço.H`

```
#ifndef BALANÇO_H
#define BALANÇO_H

...
void consolida();
...

#endif // BALANÇO_H
```

outros ficheiros de interface

Suponha-se que a aplicação em desenvolvimento contém um módulo `usa.C` com apenas a função `main()` e contendo uma invocação do procedimento `consolida()` da biblioteca de contabilidade:

`usa.C`

```
#include <balanço.H>

int main()
{
    consolida();
}
```

A construção do programa consiste simplesmente em pré-processar e compilar os ficheiros de implementação (entre os quais `usa.C`) e em seguida fundir os ficheiros objecto resultantes com os ficheiros de arquivo das duas bibliotecas (admite-se que algum outro módulo faz uso de ferramentas da biblioteca de logística):

```
c++ -Wall -ansi -pedantic -g -c *.C
c++ -o usa *.o -llogística -lcontabilidade
```

Que sucede? O inesperado: o executável é gerado sem problemas e quando se executa verifica-se que o procedimento `consolida()` executado é o da biblioteca de logística!

Porquê? Porque o fusor não se importa com duplicações nos ficheiros de arquivo. Os ficheiros de arquivo são pesquisados pela ordem pela qual são indicados no comando de fusão e a pesquisa pára quando a ferramenta procurada é encontrada. Como se colocou `-llogística` antes de `-lcontabilidade`, o procedimento encontrado é o da biblioteca de logística...

Mas há um problema mais grave. E se se quiser invocar ambos os procedimentos na função `main()`? É impossível distingui-los.

Como resolver este problema de colisão de nomes? Há várias possibilidades, todas más:

1. Pedir a um dos fornecedores para alterar o nome do procedimento. É uma má solução. Sobretudo porque provavelmente a empresa fornecedora tem outros clientes e não se pode dar ao luxo de ter uma versão diferente da biblioteca para cada cliente. Provavelmente protegeu-se de semelhantes problemas no contrato de fornecimento e recusará o pedido...

2. Esquecer, para cada nome em colisão, a versão da biblioteca contabilidade, e desenvolver essas ferramentas atribuindo-lhes outros nomes. É reinventar a roda, e isso custa tempo e dinheiro.

8.6.2 Espaços nominativos

A solução é escolher fornecedores que garantam um mínimo de possibilidades de colisões de nomes. Cada uma das empresas, se fosse competente, deveria ter colocado o seu código dentro de um espaço nominativo apropriado. Neste caso as escolhas acertadas seriam os espaços nominativos `Logística` e `Contabilidade` (os nomes dos espaços nominativos começam tipicamente por maiúscula, como os das classes). Nesse caso as bibliotecas fornecidas consistiriam em:

Biblioteca de logística

`liblogística.a` Ficheiro de arquivo da biblioteca. Contém (entre outros) o ficheiro objecto `produtos.o`, que define (entre outros) o procedimento `Logística::consolida()`.

`produtos.H`

```
#ifndef PRODUTOS_H
#define PRODUTOS_H

namespace Logística {
    ...
    void consolida();
    ...
}

#endif // PRODUTOS_H
```

outros ficheiros de interface

Biblioteca de contabilidade

`libcontabilidade.a` Ficheiro de arquivo da biblioteca. Contém (entre outros) o ficheiro objecto `balanço.o`, que define (entre outros) o procedimento `Contabilidade::consolida()`.

`balanço.H`

```
#ifndef BALANÇO_H
#define BALANÇO_H
```

```
namespace Contabilidade {
    ...
    void consolida();
    ...
}

#endif // BALANÇO_H
```

outros ficheiros de interface

O efeito prático da construção

```
namespace Nome {
    ...
}
```

é o de fazer com que todos os nomes declarados dentro das chavetas passem a ter o prefixo *Nome*`::`. Assim, com estas novas versões das bibliotecas, passaram a existir dois procedimentos diferentes com diferentes nomes: `Logística::consolida()` e `Contabilidade::consolida()`.

Agora a função `main()` terá de indicar o nome completo do procedimento:

`usa.C`

```
#include <balanço.H>

int main()
{
    Contabilidade::consolida();
}
```

8.6.3 Directivas de utilização

Para evitar ter de preceder o nome das ferramentas da biblioteca de contabilidade de `Contabilidade::`, pode-se usar uma *directiva de utilização*:

`usa.C`

```
#include <balanço.H>

using namespace Contabilidade;

int main()
{
    consolida();
}
```

Uma directiva de utilização injecta todos os nomes do espaço nominativo indicado no espaço nominativo corrente. Note-se que, quando uma entidade não está explicitamente dentro de algum espaço nominativo, então está dentro do chamado *espaço nominativo global*, sendo o seu prefixo simplesmente `::`, que pode ser geralmente omitido. Assim, no código acima a directiva de utilização injecta o nome `consolida()` no espaço nominativo global, pelo que este pode ser usado directamente.

Pode-se simultaneamente usar o procedimento `consolida()` da biblioteca de logística, bastando para isso indicar o seu nome completo:

usa.C

```
#include <balanço.H>
#include <produtos.H>

using namespace Contabilidade;

int main()
{
    consolida();
    Logística::consolida();
}
```

É mesmo possível injectar todos os nomes de ambos os espaços nominativos no espaço nominativo global. Nesse caso os nomes duplicados funcionam como se estivessem sobrecarregados, se forem nomes de rotinas. I.e., se existirem duas rotinas com o mesmo nome, então terão de ter assinaturas (número e tipo dos parâmetros) diferentes. Se tiverem a mesma assinatura, então uma tentativa de usar o nome sem o prefixo apropriado gera um erro de compilação, uma vez que o compilador não sabe que versão invocar. Por exemplo:

usa.C

```
#include <balanço.H>
#include <produtos.H>

using namespace Contabilidade;
using namespace Logística;

int main()
{
    consolida(); // ambíguo!
    Logística::consolida();
}
```

Um possível solução é

usa.C

```
#include <balanço.H>
#include <produtos.H>

using namespace Contabilidade;
using namespace Logística;

int main()
{
    Contabilidade::consolida();
    Logística::consolida();
}
```

onde se usaram nomes completos para discriminar entre as entidades com nomes e assinaturas iguais em ambos os espaços nominativos.

8.6.4 Declarações de utilização

A utilização de directivas de utilização tem a consequência nefasta de injectar no espaço nominativo corrente todos os nomes declarados no espaço nominativo indicado. Por isso, deve-se usar directivas de utilização com conta peso e medida nos ficheiros de implementação e **nunca nos ficheiros de interface**.

Uma alternativa menos drástica que as directivas são as *declarações de utilização*. Se se pretender injectar no espaço nominativo corrente apenas um nome, pode-se usar uma declaração de utilização:

usa.C

```
#include <balanço.H>
#include <produtos.H>

int main()
{
    using Contabilidade::consolida;

    consolida();
    Logística::consolida();
}
```

Nas declarações de utilização apenas se indica o nome da entidade: no caso de uma rotina, por exemplo, é um erro indicar o cabeçalho completo.

8.6.5 Espaços nominativos e modularização física

Podem-se colocar vários módulos num mesmo espaço nominativo[18]. É típico dividir uma biblioteca, por exemplo, num conjunto de pacotes com funções diversas, atribuindo a cada pacote um espaço nominativo, e consistindo cada pacote num conjunto de módulos físicos.

No exemplo apresentado o problema da colisão de nomes não foi totalmente afastado. Foi simplesmente minimizado. É que as duas bibliotecas poderiam, com azar, usar o mesmo nome para um dos seus espaços nominativos. Para reduzir ainda mais essa possibilidade, é conveniente que todos os pacotes de uma empresa pertençam a um super-pacote com o nome da empresa. Suponha-se que a biblioteca de contabilidade foi fornecido pela empresa Verão *Software*, Lda. e que a biblioteca de logística foi fornecido pela empresa Inverno *Software*, Lda.

Outra fonte de possíveis colisões são os nomes das macros usadas para evitar os erros associados a inclusões múltiplas. Como até agora se convencionou que estas macros seriam baseadas apenas no nome do respectivo módulo, podem surgir problemas graves se existirem dois módulos com o mesmo nome nas duas bibliotecas. O problema pode ser resolvido convencionando que as macros incluem também os nomes dos pacotes a que os módulos pertencem.

Se ambas as empresas levando em conta os problemas acima, os ficheiros fornecidos poderiam ser:

Biblioteca de logística

`liblogística.a` Ficheiro de arquivo da biblioteca. Contém (entre outros) o ficheiro objecto `produtos.o`, que define (entre outros) o procedimento `InvernoSoftware::Logística::consolida()`.

`produtos.H`

```
#ifndef INVERNOSOFTWARE_LOGISTICA_PRODUTOS_H
#define INVERNOSOFTWARE_LOGISTICA_PRODUTOS_H

namespace InvernoSoftware {
    namespace Logística {
        ...
        void consolida();
        ...
    }
}

#endif // INVERNOSOFTWARE_LOGISTICA_PRODUTOS_H
```

outros ficheiros de interface

Biblioteca de contabilidade

`libcontabilidade.a` Ficheiro de arquivo da biblioteca. Contém (entre outros) o ficheiro objecto `balanço.o`, que define (entre outros) o procedimento `VerãoSoftware::Contabilidade::consolida()`.

[18] E vice-versa, mas não é tão útil...

balanço.H

```
#ifndef VERAOSOFTWARE_CONTABILIDADE_BALANÇO_H
#define VERAOSOFTWARE_CONTABILIDADE_BALANÇO_H

namespace VerãoSoftware {
    namespace Contabilidade {
        …
        void consolida();
        …
    }
}

#endif // VERAOSOFTWARE_CONTABILIDADE_BALANÇO_H
```

outros ficheiros de interface

Os espaços nominativos podem, portanto, ser organizados hierarquicamente, pelo que um pacote pode conter outros pacotes para além das suas ferramentas.

Neste caso a utilização poderia ter o seguinte aspecto:

usa.C

```
#include <balanço.H>
#include <produtos.H>

int main()
{
    using VerãoSoftware::Contabilidade::consolida;

    consolida();
    InvernoSoftware::Logística::consolida();
}
```

Apresentam-se abaixo algumas regras gerais sobre o conteúdo dos ficheiros fonte de cada módulo quando se usam espaços nominativos.

8.6.6 Ficheiros de interface

Os ficheiros de interface devem ter o conteúdo descrito anteriormente (ver Secção 8.4), excepto que todas as declarações e definições serão envolvidas nos espaços nominativos necessários, como se viu nos exemplos acima. Não se devem fazer inclusões dentro das chavetas de um espaço nominativo, nem mesmo do ficheiro auxiliar de implementação! Por outro lado, a macro usada para evitar erros associados a inclusões múltiplas deve reflectir não apenas o nome do módulo, mas também o nome dos pacotes a que o módulo pertença.

8.6.7 Ficheiros de implementação e ficheiros auxiliares de implementação

Os ficheiros de implementação e auxiliar de implementação, que contêm definições de entidades apenas declaradas no ficheiro de interface, não devem envolver todas as definições nos espaços nominativos a que pertencem. Para evitar erros, os nomes usados na definição devem incluir como prefixo os espaços nominativos a que pertencem, para que o compilador possa garantir que esses nomes foram previamente declarados dentro desses espaços nominativos no ficheiro de interface do módulo.

Por exemplo, os ficheiros de implementação dos módulos `produtos` (da empresa Inverno Software) e `balanço` (da empresa Verão Software), poderiam ter o seguinte aspecto:

produtos.C

```
#include "produtos.H"

... // outros #include.

...
void InvernoSoftware::Logística::consolida()
{
    ...
}
...
```

balanço.C

```
#include "balanço.H"

... // outros #include.

...
void VerãoSoftware::Contabilidade::consolida()
{
    ...
}
```

8.6.8 Pacotes e espaços nominativos

Um pacote é uma unidade de modularização. No entanto, a sua implementação à custa de espaços nominativos tem um problema: não separa claramente interface de implementação. A única separação que existe é a que decorre do facto de os pacotes serem constituídos por módulos físicos e estes por classes, funções e procedimentos, que são níveis de modularização em que esta separação existe. Sendo os pacotes constituídos por módulos físicos, seria útil poder classificar os módulos físicos em módulos físicos públicos (que fazem parte da interface do pacote) e módulos físicos privados (que fazem parte da implementação do pacote). Isso pode-se conseguir de uma forma indirecta colocando os módulos físicos privados num espaço nominativo especial, com um nome pouco evidente, e não disponibilizando para os consumidores do pacote senão os ficheiros de interface dos módulos físicos públicos.

Esta solução é de fácil utilização quando o pacote é fornecido integrado numa biblioteca, visto que as bibliotecas são tipicamente fornecidas como um conjunto de ficheiros de interface (e só se fornecem os que dizem respeito a módulos físicos públicos) e um ficheiro de arquivo (de que fazem parte os ficheiros objecto de todos os módulos, públicos ou privados, mas nos quais as ferramentas dos módulos privados são de mais difícil acesso, pois encontram-se inseridas num espaço nominativo de nome desconhecido para o consumidor).

8.7 Exemplo final

!!No exemplo pôr documentação. Gerar html e pdf com doxygen e colocar disponível na rede. O ideal eram links pdf...

Apresenta-se aqui o exemplo da Secção 8.5, mas melhorado de modo a usar ficheiros auxiliares de implementação e espaços nominativos e a fazer um uso realista de rotinas e métodos em linha.

O exemplo contém um pacote de ferramentas de gestão de uma escola de nome `Escola`. Este pacote, implementado usando um espaço nominativo, contém dois módulos físicos descritos abaixo: `nota` e `aluno`. Módulos:

`nota` Um módulo físico criado para conter ferramentas relacionadas com notas. Pertence ao pacote `Escola`. Colocaram-se todas as ferramentas dentro de um espaço nominativo `Nota` para evitar nomes demasiado complexos para as ferramentas. Pode-se considerar, portanto, que o pacote `Escola` é constituído por um módulo físico `aluno` e por um pacote `Nota`, que por sua vez é constituído por um único módulo físico `nota`. Os ficheiros fonte do módulo `nota` são (não há ficheiro de implementação, pois todas as rotinas são em linha):

`nota.H`

```
#ifndef ESCOLA_NOTA_NOTA_H
#define ESCOLA_NOTA_NOTA_H

/// Pacote que contém todas as ferramentas de gestão da escola.
namespace Escola {

    /// Pacote que contém ferramentas relacionadas com notas.
    namespace Nota {

        /** Devolve verdadeiro se a nota for considerada positiva.
            @pre 0 <= nota e nota <= máxima.
            @post éPositiva =
                      (mínima_de_aprovação <= nota). */
```

```
            bool éPositiva(int nota);

            /// A nota mínima para obter aprovação.
            int const mínima_de_aprovação = 10;

            /// A nota máxima.
            int const máxima = 20;
        }
    }

    #include "nota_impl.H"

    #endif // ESCOLA_NOTA_NOTA_H
```

nota_impl.H

```
    inline bool Escola::Nota::éPositiva(int const nota) {
        return mínima_de_aprovação <= nota;
    }
```

aluno Um módulo físico criado para conter ferramentas relacionadas com alunos. Pertence ao pacote `Escola`. Os ficheiros fonte do módulo `aluno` são (não há ficheiro de implementação, pois todas as funções e procedimentos são em linha):

aluno.H

```
    #ifndef ESCOLA_ALUNO_H
    #define ESCOLA_ALUNO_H

    #include <string>
    #include <iostream>

    #include "nota.H"

    namespace Escola {

        /// Representa o conceito de aluno de uma escola.
        class Aluno {
          public:
            /// Constrói um aluno dado o nome, o número e, opcionalmen-
    te, a nota.
            Aluno(string const& nome,
                  int número, int nota = Nota::maxima);

            /// Devolve o nome do aluno.
            string const& nome() const;

            /// Devolve o número do aluno.
            int número() const;
```

```
        /// Devolve a nota do aluno.
        int nota() const;

      private:
        string nome_;
        int número_;
        int nota_;
    };

    /// Operador de inserção de um aluno num canal.
    ostream& operator << (ostream& saída,
                          Aluno const& aluno);
}

#include "aluno_impl.H"

#endif // ESCOLA_ALUNO_H
```

aluno_impl.H

```
inline Escola::Aluno::Aluno(string const& nome,
                            int const número,
                            int const nota)
    : nome_(nome), número_(numero), nota_(nota) {
    assert(0 <= nota and nota <= Nota::máxima);
}

inline string const& Escola::Aluno::nome() const {
    return nome_;
}

inline int Escola::Aluno::número() const {
    return número_;
}

inline int Escola::Aluno::nota() const {
    return nota_;
}

inline ostream& Escola::operator << (ostream& saída,
                                     Aluno const& aluno) {
    return saída << aluno.nome() << ' ' << aluno.número()
                 << ' ' << aluno.nota();
}
```

`média` e `procura` São os módulos físicos com as funções `main()` dos programas a construir. Não têm ficheiro de interface nem de implementação auxiliar:

média.C

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

#include "aluno.H"

using namespace Escola;

/** Programa que lê informação sobre um conjunto de alunos do teclado e depois mostra a média das suas notas. */
int main()
{
    cout << "Introduza número de alunos: ";
    int número_de_alunos;
    cin >> número_de_alunos;

    cout << "Introduza os alunos "
         << "(nome número nota):" << endl;
    vector<Aluno> alunos;
    for(int i = 0; i != número_de_alunos; ++i) {
        string nome;
        int número;
        int nota;
        cin >> nome >> número >> nota;

        alunos.push_back(Aluno(nome, número, nota));
    }

    double soma = 0.0;
    for(int i = 0; i != número_de_alunos; ++i)
        soma += alunos[i].nota();
    cout << "A média é: " << soma / número_de_alunos
         << " valores." << endl;
}
```

procura.C

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

#include "nota.H"
```

```
#include "aluno.H"

using namespace Escola;
```

/** Programa que mostra informação acerca de um aluno à escolha do utilizador. */

```
int main()
{
    vector<Aluno> alunos;
    alunos.push_back(Aluno("Zé", 1234, 15));
    alunos.push_back(Aluno("Zacarias", 666, 20));
    alunos.push_back(Aluno("Marta", 5465, 18));
    alunos.push_back(Aluno("Maria", 1111, 14));

    cout << "Pauta:" << endl;
    for(vector<Aluno>::size_type i = 0;
        i != alunos.size(); ++i)
        cout << alunos[i] << endl;

    cout << "De que aluno deseja saber a nota? ";
    string nome;
    cin >> nome;
    for(vector<Aluno>::size_type i = 0;
        i != alunos.size(); ++i)
        if(alunos[i].nome() == nome) {
            cout << "O aluno " << nome << " teve "
                 << alunos[i].nota() << endl;
            if(not Nota::éPositiva(alunos[i].nota()))
                cout << "Este aluno reprovou." << endl;
        }
}
```

Makefile

```
all: média procura

procura.o: procura.C nota.H nota_impl.H aluno.H aluno_impl.H
média.o: média.C nota.H nota_impl.H aluno.H aluno_impl.H
```