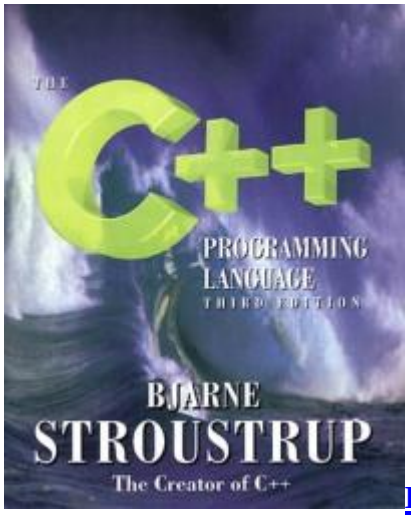
Introdução à linguagem C++



[Bjarne Stroustrup]

Algumas características

- Concebida e implementada por Bjarne Stroustrup entre 1979 e 1985 nos Bell da Labs AT&T, os mesmos laboratórios onde Dennis Ritchie já tinha desenvolvido a linguagem C.
- Uma das linguagens actualmente mais populares tem sido utilizada para escrever todo o tipo de aplicações.
- É uma linguagem padronizada com standards ANSI e ISO.
- É uma linguagem de mais alto nível do que o C, mas continua a oferecer as mesmas facilidades para manipulações de baixo nível, nomeadamente acesso directo à memória.
- A maioria das implementações são muito eficientes, praticamente tão eficientes como o C.
- Aos tipos do C foram adicionadas tipos-objecto, que são definidos usando classes.
- A tipificação mais forte do que no C: o tipo `void *` não é directamente compatível com os outros apontadores; só se podem chamar funções cujo cabeçalho foi previamente declarado.
- O suporte da linguagem para os tipos primitivos (uso de operadores, coerções, etc.) estende-se completamente aos tipos que o programador define usando classes.
- A biblioteca padrão do C++ é mais rica e de mais alto nível do que a do C.
- Tal como o C, o C++ não têm gestão automática de memória. É necessário reservar e libertar memória manualmente, o que é uma conhecida fonte de erros.
- O C++ é uma linguagem híbrida que suporta os paradigmas imperativo (procedimental) e orientado pelos objectos. Programar em C++ não implica necessariamente a utilização dos mecanismos object-oriented.
- A linguagem C++ oferece alternativas de mais alto nível para alguns dos mecanismos do C. Por exemplo oferece as primitivas new/delete como alternativa ao malloc/free do C e oferece a classe string como alternativa ao tipo `char *`. No entanto os mecanismos de mais baixo nível podem continuar a ser usados.

Exemplo1

```
#include <iostream>

int main() {
    std::cout << "Benvindo ao maravilhoso mundo do C++!" << std::endl ;
    return 0 ;
}

// Compilar assim: c++ -o test test.cpp
```

Exemplo2

```
#include <iostream>
using namespace std ;
```

```
int main() {
    cout << "Benvindo ao maravilhoso mundo do C++!" << endl ;
    return 0 ;
}
```

Exemplo 3

```
#include <iostream>    // C++ header file for Input/Ooutput
#include <string>      // C++ header file for strings

int main()
{
    // create two strings
    std::string firstname = "bjarne" ;
    std::string lastname = "stroustrup" ;

    // change strings
    firstname[0] = 'B' ;
    lastname[0] = 'S' ;

    // concatenate strings
    std::string fullname = firstname + " " + lastname ;

    // compare strings
    if( fullname != "" ) {
        // output strings
        std::cout << fullname
                  << " foi o criador do C++" << std::endl ;
    }

    // length of string
    int num = fullname.length() ;
    std::cout << "\"" << fullname << "\" has " << num
              << " characters" << std::endl ;
}
```

Padronizações do C++

- C++ 1.0 - Padrão informal estabelecido em 1985 com a publicação da 1ª edição do livro "The C++ Programming Language".
- C++ 2.0 - Padrão informal estabelecido em 1989 e consolidado em 1990 com a publicação do livro "The Annotated C++ Reference Manual".
- C++ 3.0 - Padrão ISO/ANSI adoptado em 1998 e corrigido em 2003.
- Futuro - C++0x (nome informal) deverá ser aprovado em meados de 2011. Eis o draft corrente e o seu suporte em GCC (mas o GCC nos nossos laboratórias é relativamente antigo e suporta pouco do novo padrão).

O GCC é uma das implementações de C++ actualmente mais usadas. Suporta praticamente todo o padrão ISO/ANSI, mais algumas extensões.

O GCC comporta-se como um compilador de C++ quando invocado usando o nome **g++** ou **c++**. Eis um exemplo duma linha de comando em Linux ou Windows:

```
g++ -o myprog myprog.cpp mod1.cpp mod2.cpp mod3.c
```

Como se vê, é possível misturar no mesmo programa ficheiros escritos em C++ e em C. A ordem pela qual ocorrem os ficheiros source e ficheiros objeto não interessa.

Objectivos do C++

Segundo Stroustroup, foram estes os seus objectivos ao criar a linguagem C++:

- Suporte para um amplo espectro de utilizações, mas dar algum destaque à programação de sistemas.
- Suporte de tipos de dados abstractos.
- Suporte de programação orientada pelos objeto.
- Suporte de programação genérica.
- Suporte de programação imperativa (procedimental) tradicional.
- O programador deve poder escolher o estilo de programação a usar, mesmo que seja possível que as suas escolhas sejam erradas.
- Compatibilidade com o C
- Os novos mecanismos não devem piorar a eficiência, relativamente ao C
- A linguagem deve poder ser usada sem a necessidade dum ambiente de desenvolvimento sofisticado.

Comparação do C com C++

A linguagem C++ começou por ser desenvolvida como uma extensão do C e assim o C++ é largamente compatível com o C. No entanto, no standard ISO 99 do C foi decidido que, sendo as linguagens diferentes, devia haver a liberdade de evoluir de forma independente.

Há cerca de 20 situações em que código C válido é considerado código C++ inválido, ou se comporta de maneira diferente. Eis as mais importantes:

- Em C++ só é possível atribuir apontadores de tipo `void *` a outros apontadores usando casts. O mesmo se aplica às atribuições no sentido contrário. Ao usar casts nas atribuições, o código fica compatível, tanto com o C como com o C++.
- O C++ introduz diversas palavras reservadas novas: `new`, `delete`, `namespace`, `template`, `class`, `bool`, etc. Código C que use essas palavras é considerado código C++ inválido.
- O C++ só permite a chamada de funções cujo cabeçalho seja conhecido, ao contrário do C. Além disso o cabeçalho **void f()** significa que a função f aceita qualquer número de argumentos em C, mas nenhum argumento em C++.
- Algumas das extensões do C99 não são suportadas no C++ ou entram em conflito com partes do C++.
- `sizeof('x')` tem valores diferentes nas duas linguagens, porque numa expressão em C 'x' é imediatamente convertido em inteiro enquanto que em C++ 'x' só é convertido em caso de necessidade. O mesmo se passa com os valores enumerados.

Suporte para múltiplos estilos de programação em C++

Stroustrup disse que, em C++, o programador deve poder escolher o estilo de programação a usar, mesmo que seja possível que as suas escolhas sejam erradas.

Esta grande liberdade é muitas vezes uma fonte de confusão em C++ e importa, desde já, esclarecer quais são os principais estilos de programação que se podem usar e, também, saber se há mecanismos específicos associados a cada estilo. Os principais estilos de programação que o C++ suporta são:

- Programação orientada pelos objetos.
- Programação baseada em valores.

Programação orientada pelos objetos

A **programação orientada pelos objetos** é um estilo de programação no qual os dados, denominados por **objetos**, são entidades activas que interagem entre si através de troca de **mensagens**. Um objeto pode reagir a uma mensagem de várias formas: alterando o seu estado interno; criando uma cópia modificada de si próprio; criando outros objetos; enviando mensagens a outros objetos ou a si próprio.

- Abreviadamente, um **objeto** pode ser definido como um elemento de dados **activo** e com **individualidade própria**.

No paradigma orientado pelos objetos, um programa em execução consiste numa comunidade de objetos trocando mensagens entre si. **Quem controla a computação são, pois, os objetos!** O código executável subordina-se a estes, pois só se escreve código para especificar como é que um objeto reage a mensagens provenientes do exterior.

Programação baseada em valores

O estilo de **programação baseada em valores** contrasta radicalmente com o estilo anterior. Neste estilo de programação, os dados, designados por **valores**, são entidades passivas, livremente manipuladas e transferidas entre as diversas partes do programa.

- Abreviadamente, um **valor** pode ser definido como um elemento de dados **passivo** e sem **individualidade própria**.

No estilo de programação baseada em valores, um programa em execução consiste numa colecção de funções e procedimentos que se invocam mutuamente, trocando valores entre si. **Quem controla a computação são pois os procedimentos e funções!** Os dados subordinam-se a estes.

Discussão

O C++ suporta tanto programação baseada em objetos como programação baseada em valores, tendo o programador tem de decidir em cada momento sobre qual o estilo a seguir.

Este problema não se coloca em C porque estas linguagens suportam apenas o estilo de programação baseada em valores.

Também não se coloca em Java, pois esta linguagem suporta acima de tudo o estilo orientado pelos objectos.

Considere uma classe `Point` em C++. Os dados criados a partir desta classe costumam ser designados de objectos, mas em C++ eles só serão realmente objectos se for sempre respeitada a sua identidade. A maneira mais radical de conseguir isso, mas nem sempre a mais eficiênte, é criar todos os objectos dinamicamente e aceder sistematicamente a eles através de apontadores, como se faz na seguinte função que imita o estilo de programação do Java:

```
void f() {
    Point *p = new Point(2.3, 5.6) ;   // Objecto criado no heap
    Point *q = p ;       // Respeita a identidade do objecto, e agora q aponta para o
objecto original
    q->Shift(0.1, 0.2) ; // Altera o objecto original
}
```

Mas também é possível deixar os objectos serem criados na pilha de execução para depois tentar manter a sua identidade sem usar apontadores. Este método é mais eficiênte e tem sintaxe mais simples. Este é o estilo de programação preferido em C++, uma linguagem obcecada com a eficiência. Mas manter a identidade requer cuidados especiais, em particular a declaração de algumas referências e a sistemática passagem por referência dos objectos para as funções. Veremos na próxima aula o que são exactamente **referências** em C++.

```
void g() {
    Point p(2.3, 5.6) ; // Objecto criado na pilha de execução, na zona de variáveis
locais
    Point &q = p ;      // Coloca q a referir o objecto guardado em p
    q.Shift(0.1, 0.2) ; // Altera o objecto original
}
```

Nas atribuições directas (as que não envolvem nem apontadores nem referências) temos de ter a consciência que os objectos são normalmente duplicados, o que não respeita a sua identidade.

```
void h() {
    Point p(2.3, 5.6) ; // Objecto criado na pilha de execução, na zona de variáveis
locais
    Point q = p ;       // Efectua uma cópia do objecto, para outra zona da pilha de
execução
```

```
    q.Shift(0.1, 0.2) ; // Altera a cópia
}
```

Conclusão: Se quisermos programar em C++ usando um estilo orientado pelos objectos, temos de nos preocupar com a questão da manutenção da identidade dos objectos. Concretamente temos de ser cuidadosos na forma como passamos os objectos como parâmetro e com as atribuições.

Palavras reservadas do C++

Eis a lista completa das palavras reservadas do C++:

```
   and        const     float         operator static_cast    using
and_eq   const_cast       for               or      struct  virtual
   asm     continue    friend            or_eq      switch     void
  auto      default      goto          private    template volatile
bitand       delete        if        protected        this  wchar_t
 bitor           do    inline           public       throw    while
  bool       double       int         register        true      xor
 break dynamic_cast      long reinterpret_cast         try   xor_eq
  case         else   mutable           return     typedef
 catch         enum namespace            short      typeid
  char     explicit       new           signed    typename
 class       extern       not           sizeof       union
 compl        false    not_eq           static    unsigned
```

O tipo bool

O tipo `bool` existe em C++ e é idêntico ao tipo `bool` do C99. Portanto, tal como em C, o valor de verdade é representado por qualquer valor diferente de zero e o valor `true` não é mais do que um nome alternativo para o inteiro 1.

A única diferença é que o tipo `bool` está embutido na linguagem (é mesmo uma palavra reservada), e por isso nunca se faz a inclusão do ficheiro <stdbool.h>.

As streams padrão cin, cout, cerr

Quando se usa o módulo <iostream> da biblioteca padrão do C++, ficam disponíveis três streams padrão chamadas `cin, cout, cerr`. A primeira pertence à classe `std:istream` e as duas últimas pertencem à classe `std:ostream`.

Exemplo de utilização:

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

int main() {
    int i ;
    double d ;
    std::string s ;

    std::cout << "> " ;
    std::cin >> i >> d >> s ;

    std::cout << i << " "
              << d << " "
              << s << " "
              << std::endl ;
    return 0 ;
```

```
}
```

O operador de resolução de escopo ::

O C++ introduz alguns operadores novos, entre os quais se encontra o **operador de resolução de escopo ::**. Usado como operador unário prefixo, permite aceder a variáveis globais, mesmo quando elas foram redefinidas localmente.

Exemplo:

```
#include <iostream>

int x = 100 ;

int main() {
    int x = 23 ;
    std::cout << "Result: " << x + ::x << std::endl ;
    return 0 ;
}
```

O programa anterior escreve "`Result: 123`".

Mas o operador :: é mais normalmente usado como operador binário infixo para aceder explicitamente a um nome definido num namespace ou numa classe. Veja estes dois exemplos:

```
std::cout
std::string::npos
```

Namespaces

Os **namespaces** ajudam a organizar o espaço global de nomes e a evitar conflitos de nomes. Um espaço de nomes pode conter classes, funções, variáveis globais e tipos ou seja quaisquer entidades globais com nome.

O espaço dos nomes da biblioteca padrão do C++ chama-se **std**. Todos os nomes da biblioteca padrão podem ser acedidos usando o prefixo **std::** em cada utilização - como em `std::cout` ou `std::string::npos` - ou então pode usar-se a seguinte diretiva para poupar na escrita de prefixos (veja os exemplos 1 e 2 no início da aula anterior):

```
using namespace std
```

Criar um novo namespace é simples. Basta usar a construção

```
namespace myspace {
...
}
```

em diversos ficheiros ".h" para encapsular a declaração das entidades participantes e em diversos ficheiros ".cpp" para encapsular a implementação dessas mesmas entidades. Para um dado namespace, a construção anterior pode ser usada as vezes que for necessário, pois um namespace é um entidade aberta à qual estamos sempre a tempo de adicionar novos elementos.

O C++ suporta ainda **namespaces anónimos**. Trata-se dum namespace sem nome que se declara dentro dum ficheiro ".cpp" usando a construção

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

Todas as entidades definidas dentro dum namespace anónimo tornam-se privadas ao ficheiro onde aparecem declaradas. Essas entidades também poderiam ser tornadas privadas usando a palavra **static** como em C, mas a utilização dum namespace anónimo é mais elegante.

Um namespace anónimo é *fechado* pois não é possível estendê-lo usando entidades definidas outro ficheiro fonte.

A palavra reservada const

Quando falámos da linguagem C não demos muita atenção à palavra reservada `const`. Vamos agora exemplificar diversos usos da palavra `const`, que se aplicam igualmente ao C e ao C++.

```
int main() {
    const int i = 3 ;      // Constante inteira
    const int *pt ;        // Apontador para inteiro constante
    int const *pt ;        // Apontador para inteiro constante
    int *const pt ;        // Apontador constante para inteiro
    int const *const pt ;  // Apontador constante para inteiro constante
    int *const *pt ;       // Apontador para apontador constante para inteiro
}
```

Quando `const` não é a primeira palavra da declaração da variável, a regra é simples:

- O que ocorrer imediatamente à esquerda de `const` é considerado constante.

Quando `const` é a primeira palavra da declaração da variável, o primeiro elemento do tipo é considerado constante.

Casts

O C++ continua a suportar os casts tradicionais do C

```
(type)expression
```

para mudar explicitamente o tipo duma expressão. Mas esta forma é considerada obsoleta em C++.

Em C++ surgem quatro formas especializadas de casting, cada uma das quais é para ser usada em diferentes situações:

```
static_cast<type>(expression)
const_cast<type>(expression)
dynamic_cast<type>(expression)
reinterpret_cast<type>(expression)
```

O **static cast** é usado para efetuar conversões normais de tipo como no seguinte exemplo:

```
int a = 5, b = 2 ;
double d = static_cast<double>(a) / b ;
```

Com este cast a variável d recebe o valor 2.5. Sem ele o valor seria 2.

O **const cast** é usado para remover o atributo `const` dum tipo. O seguinte pedaço de código:

```
void f(char *s) { }
void g() {
    const char* hello = "Hello world" ;
    f(hello) ;
}
```

dá o seguinte erro de compilação

```
zzz.cpp: In function 'void g()':
zzz.cpp:12: error: invalid conversion from 'const char*' to 'char*'
zzz.cpp:12: error:   initializing argument 1 of 'void f(char*)'
```

Inserindo um const cast o erro desaparece:

```
void f(char *s) { }
void g() {
```

```
    const char* hello = "Hello world" ;
    f(const_cast<char *>(hello)) ;
}
```

Um static cast não funcionaria no caso anterior.

O **dynamic cast** é usado para converter de forma segura um objeto dum tipo mais geral para um tipo mais específico. Exemplo:

```
Animal *a = new Cat ;               // Atribuição polimórfica
Cat *c = a ;                        // DÁ ERRO DE COMPILAÇÃO!
Cat *c = dynamic_cast<Cat *>(a) ;   // Correto, mas pode dar 0
if( c != 0 )
    c->Meou() ;
```

Há dois resultados possíveis para um dynamic cast: em caso de insucesso é devolvido 0 (a forma recomendada de representar NULL em C++); em caso de sucesso é devolvido um apontador (com o tipo desejado) para o objeto.

O **reinterpret_cast** é usado para reinterpretar apontadores, nas situações em que um dynamic cast não é aplicável. Este é o cast de mais baixo nível, e também o mais perigoso de usar. No seguinte exemplo, coloca-se um apontador de tipo char * a apontar para um valor de tipo double, para se poderem analisar os bytes individuais que constituem esse valor.

```
double d ;
char *pt = reinterpret_cast<char *>(&d) ;
```

Sobrecarga (overloading) de funções

Em C++ as funções são distinguidas por assinatura (= nome+lista do tipo dos argumentos) e não apenas por nome. Em Java os métodos também são distinguidos por assinatura. Em C e OCaml as funções são distinguidas apenas por nome.

Portanto, permite-se a definição em C++ de diversas funções com o mesmo nome, desde que elas difiram nos tipos dos argumentos. O atributo `const` também conta para a distinção.

Exemplo:

```
#include <iostream>

void show(int val) {
    std::cout << "Integer: " << val << std::endl ;
}

void show(double val) {
    std::cout << "Double: " << val << std::endl ;
}

void show(char *val) {
    std::cout << "String: " << val << std::endl ;
}

int main() {
    show(21056) ;
    show(2.543) ;
    show("ola'");
    return 0 ;
}
```

Os compiladores implementam funções overloaded usando uma técnica chamada **name mangling**. Para as funções do exemplo anterior, o compilador gera internamente nomes únicos. Esses nomes dependem do compilador. No caso do gcc os nomes são: _Z4showi, _Z4showd, _Z4showPc. Estes nomes podem ser observados aplicando o comando `nm` ao ficheiro objeto que resulta da compilação.

Argumentos por omissão nas funções

Em C++ é possível fornecer argumentos por omissão quando se escrevem uma função. Nas chamadas da função, o compilador preenche automaticamente os argumentos que não forem fornecidos pelo programador. O programador só tem a liberdade de omitir os argumentos mais à direita, ou seja no final da lista de argumentos.

Exemplo:

```
#include <iostream>

int add(int a = 1, int b = 2, int c = 3, int d = 4, int e = 5) {
    return a + b + c + d + e ;
}

int main() {
    std::cout << "Result: " << add(0, 0) << std::endl ;
    return 0 ;
}
```

O programa anterior escreve "12".

Se uma função com argumentos por omissão for exportada por um módulo, então os valores por omissão devem ser colocados no respetivo protótipo, no ficheiro ".h" e não na sua implementação, no ficheiro ".cpp". Exemplo:

```
// header file

int add(int a = 1, int b = 2, int c = 3, int d = 4, int e = 5) ;

// implementation file

#include <iostream>

int add(int a, int b, int c, int d, int e) {
    return a + b + c + d + e ;
}

int main() {
    std::cout << "Result: " << add(0, 0) << std::endl ;
    return 0 ;
}
```

Mais um exemplo uso de argumentos por omissão:

```
void read(std::istream &input = std::cin) ;
void write(std::ostream &output = std::cout) ;
```

Tipos referência

Um mecanismo muito útil em C++ é o mecanismo das referências. Uma **referência** é simplesmente sinónimo (alias) para uma variável. O tipo duma referência tem a forma geral `T&` onde `T` é um tipo qualquer.

Uma referência têm de ser inicializada no ponto da declaração. Depois de inicializada, não pode mais ser alterada.

O seguinte exemplo mostra a criação duma referência 'rv' para uma variável 'v'. Depois de criada a referência, usar o nome 'v' ou 'rv' não faz absolutamente diferença nenhuma pois são sinónimos.

```
int main() {
    int v ;
```

```
    int &rv = v ;
    v = 5 ;       // atribui 5 à variável v
    rv = 5 ;      // tem rigorosamente o mesmo efeito do comando anterior
    return 0 ;
}
```

As referências foram inventadas, principalmente para suportar uma forma de parâmetros de entrada/saída chamada **passagem de argumentos por referência**. Veja o seguinte exemplo:

```
void add(int &r, int val) {
    r += val ;
}

int main() {
    int v = 5 ;
    add(v, 10) ;
    // Now the value of v is 15
    return 0 ;
}
```

Na ausência de referências, o exemplo anterior teria de ser programado usando apontadores, o que seria menos elegante. Seria assim:

```
void add(int *r, int val) {
    *r += val ;
}

int main() {
    int v = 5 ;
    add(&v, 10) ;
    // Now the value of v is 15
    return 0 ;
}
```

Comparação entre apontadores e referências

O compilador implementa as referências usando apontadores. Na verdade as referências são uns apontadores especiais com propriedades especiais, bastante restritivas como se pode ver:

1. São obrigatoriamente inicializadas no ponto da declaração, ou no momento da passagem de parâmetros no caso dos argumentos por referência.
2. Não podem ser inicializadas com *null*.
3. Não podem ser alteradas depois de inicializadas.
4. Não se pode declarar uma referência para uma referência, ou seja o tipo `T&&` não existe.
5. Só admitem um nível de desreferenciação e essa desreferenciação é aplicada automaticamente sempre que são usadas (sem usar explicitamente o operador "*").

Em resumo, as referências constituem um mecanismo que dá menor liberdade ao programador, mas que é mais seguro e também envolve uma sintaxe mais simples.

Exercício traiçoeiro: Explique o que fazem as duas últimas instruções abaixo.

```
int v ;
int *pv = &v ;
int &rv = v ;
pv = 0 ;
rv = 0 ;
```

Referências em Java

Em Java, todos os objetos são manipulados através de referências. Mas as referências do Java são mais poderosas do que as do C++, pois só obedecem às restrições 4 e 5 da lista anterior. Além disso, o operador `new` do Java produz

referencias. Na verdade a flexibilidade das referências do Java está a meio caminho entre a flexibilidade das referências no C++ e dos apontadores do C++.

Métodos de passagem de argumentos recomendados

Existem demasiadas formas de passagens de argumentos em C++. Na prática recomenda-se que sejam usadas só as seguintes, nas situações indicadas.

Passagem por valor

Quando uma função recebe um argumento por valor, ela na realidade trabalha com uma cópia local do valor passado. A função pode manipular o valor livremente, pois nunca altera o valor original que foi passado.

É a forma natural de implementar argumentos de entrada, mas é ineficiente se o argumento for muito grande.

Usa-se com argumentos de entrada de tipos simples.

```
void F(int i, double d)    // Dois argumentos de entrada passados por valor
```

Passagem por referência

Quando uma função recebe um argumento por referência, ela trabalha diretamente com a variável que foi passada. Ao alterar o argumento, está-se a a alterar diretamente a variável original. Mas na chamada da função, o argumento passado tem mesmo de ser uma variável; não pode ser uma expressão qualquer.

É a forma natural de implementar argumentos de saída e argumentos de entrada/saída. É eficiente, mesmo que o argumento seja muito grande.

Usa-se com argumentos de saída e argumentos de entrada/saída, em todas as circunstâncias.

```
void ReadInt(std::istream &input, int &i) ;    // Argumento de entrada/saída e argumento
de saída passados por referência
```

Passagem por referência constante

É igual ao caso anterior, com a diferença que não podemos modificar o valor da variável; só podemos ler. Um pormenor excelente, é que se o argumento atual for uma expressão qualquer, o compilador gera automaticamente uma variável com o valor da expressão e passa essa variável.

É uma forma natural de implementar argumentos de entrada e é eficiente, mesmo que o argumento seja muito grande.

Usa-se com argumentos de entrada de tipos objeto.

```
void Process(const std::string &str) ;    // Argumento de entrada passado por referência
constante
```

Passagem por apontador + passagem por apontador constante

Estas duas formas de passagens de parâmetros são perecidas com as duas anteriores, mas têm um uso mais especializado. Só são usadas quando se está a lidar com estruturas de dados dinâmicas, ou seja com estruturas de dados cujos elementos são

objeto criados dinamicamente (usando a primitiva `new`). objeto criados dinamicamente são manipulados através de apontadores, e por isso faz todo o sentido escrever funções que aceitam apontadores como argumento.

```
void AddToTree(Tree *tree, Node *node) ;                 // Dois argumentos de
entrada/saída passados por apontador
void PrintNode(std::ostream &output, const Node *node) ; // O Segundo argumento é de
entrada e passado por apontador constante
```

Comentário - A identidade dos objeto

Repare que, seguindo estas recomendações, a manutenção da identidade dos objeto fica respeitada. Repare que os objeto são sempre passados por referência ou por apontador.

Algumas classes da biblioteca padrão

Classe std::string

O C++ oferece a classe std::string como alternativa ao tipo char * do C. A classe std::string é muito rica em funcionalidade que permite ao programador tornar-se bastante produtivo.

A funcionalidade da classe std::string lembra um pouco, e de forma geral, a funcionalidade da classe StringBuffer do Java. Veja o exercício 35 da aula prática 9 para aprender mais sobre as strings do C++.

Classe std::vector

A linguagem C++ suporta os mesmos arrays da linguagem. Mas os arrays são um pouco limitados pois não crescem dinamicamente e suportam diretamente apenas a operação de indexação. Se precisarmos de algo mais temos de ser nós a programar tudo. O seguinte exemplo preenche um array de tamanho fixo com os primeiros valores da sucessão de Fibonacci:

```
#include <iostream>

int main() {
    const int maxFib = 20 ;
    int fib[maxFib] = { 1, 1} ;
    for( int i = 2 ; i < maxFib ; i++ )
        fib[i] = fib[i-1] + fib[i-2] ;
    for( int i = 0 ; i < maxFib ; i++ )
        std::cout << "fib(" << i << ") = "
                  << fib[i] << std::endl ;
    return 0 ;
}
```

A biblioteca padrão do C++ oferece a classe genérica std::vector<T>, que faz lembrar um pouco a classe Vector do Java, embora operações tenham nomes bastante diferentes. Eis o mesmo exemplo, mas agora programado usando um vector que cresce dinamicamente:

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

int main() {
    const int maxFib = 20 ;
    std::vector<int> fib ;
    fib.push_back(1) ;
    fib.push_back(1) ;
    for( int i = 2 ; i < maxFib ; i++ )
        fib.push_back(fib[i-1] + fib[i-2]) ;
    for( int i = 0 ; i < maxFib ; i++ )
```

```
        std::cout << "fib(" << i << ") = "
                  << fib[i] << std::endl ;
    return 0 ;
}
```

Além de crescerem dinamicamente, os vetores suportam diversas funções, como por exemplo a função de ordenação. Para ordenar um vetor `v`, faz-se assim

```
std::sort(v.begin(), v.end()) ;
```

onde os argumentos da função são iteradores (explicados mais à frente). Para mais informação sobre esta classe siga o link STL da coluna da esquerda da página da cadeira.

Classes std::ifstream e std::ofstream

Estas classes permitem abrir canais de entrada e de saída sobre ficheiros. Ambas são subclasses da classe std::stream.

Segue-se o tradicional exemplo de cópia de ficheiros que temos apresentado para todas as linguagens, até agora.

```
#include <iostream>
#include <fstream>

bool Copy(const std::string& ni, const std::string& no) {
    std::ifstream si(ni.c_str()) ;  // Em C++ os "canais" do ML chamam-se "fstreams"
    std::ofstream so(no.c_str()) ;
    int c ;                // Tem de ser int porque get retornar 257 valores diferentes
    if( !si || !so )       // Testa se os dois ficheiros foram abertos com sucesso
        return false ;
    while( (c = si.get()) != EOF )
        so.put(c) ;
    si.close() ;     // Desnecessário porque o destrutor da classe stream
    so.close() ;     // já fecha automaticamente as streams quando a função termina
    return true ;
}

int main() {
    std::string ni, no ;
    std::cout << "IN> " ;
    std::cin >> ni ;
    std::cout << "OUT> " ;
    std::cin >> no ;
    if( !Copy(ni, no) )
        std::cerr << "Copy failed!\n" ;
    return 0 ;
}
```

Cópia, linha a linha.

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

bool Copy(const std::string& ni, const std::string& no) {
    std::ifstream si(ni.c_str()) ;
    std::ofstream so(no.c_str()) ;
    std::string line ;
    if( !si || !so )
        return false ;
    while( getline(si, line) )
        so << line << std::endl ;
    si.close() ;
    so.close() ;
    return true ;
}

int main() {
    std::string ni, no ;
```

```
    std::cout << "IN> " ;
    std::cin >> ni ;
    std::cout << "OUT> " ;
    std::cin >> no ;
    if( !Copy(ni, no) )
        std::cerr << "Copy failed!\n" ;
    return 0 ;
}
```

Classe std::sstream

Uma std::stringstream não é mais do que uma stream especial que armazena internamente (numa string) tudo o que nela se escreve. É possível escrever nela o que se quiser usando o operador <<, e depois ler a partir dela usando o operador >>.

A função IntToString, abaixo definida, converte números inteiros em strings duma forma trivial. Estude o código.

```
#include <sstream>

std::string IntToString(int i)
{
   std::stringstream ss ;
   std::string s ;
   ss << i ;   // Escreva na stringstream
   ss >> s ;   // Lê da stringstream
   return s ;
}
```

Belo truque não é? É bom saber que a operação de escrita em streams também pode também ser indiretamente usada para escrever em strings e não só em ficheiros e na consola.

Operadores e sobrecarga (overloading) de operadores

O C++ disponibiliza cerca de 30 operadores com aridades e associatividades específicas. A maioria desses operadores podem ser usados para dar nome a funções. esta possibilidade é útil para definir linguagens orientadas para domínios específicos dentro da própria linguagem C++: imagine por exemplo uma linguagem de manipulação de matrizes que aproveite os operadores aritméticos.

Para definir uma função chamada "+", por exemplo, basta escrever uma função com o seguinte cabeçalho, onde T1, T2 e T3 representam tipos quaisquer:

```
T1 operator+(T2 a, T3 b) ;
```

É possível definir simultaneamente diversas funções chamadas "+", desde que elas difiram nos tipos dos argumentos.

Eis um exemplo simples onde se define uma operação de soma aplicável a vetores de inteiros. Aproveite para aprender uma útil técnica de inicialização de vetores:

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

std::vector<int> operator+(std::vector<int> a, std::vector<int> b)
{
    std::vector<int> res ;
    int m = std::min(a.size(), b.size()) ;
    for( int i = 0 ; i < m ; i++ )
        res.push_back(a[i] + b[i]) ;
    for( int i = m ; i < a.size() ; i++ )
        res.push_back(a[i]) ;
    for( int i = m ; i < b.size() ; i++ )
```

```
        res.push_back(b[i]) ;
    return res ;
}

int main() {
    int av[] = { 1, 2, 3, 4, 5} ;
    int bv[] = { 1, 2, 3, 4, 5, 6, 7} ;
    std::vector<int> a(av, av+5) ;
    std::vector<int> b(bv, bv+7) ;
    std::vector<int> res = a + b ;   // <- repare na elegância da chamada da função +
    for( int i = 0 ; i < res.size() ; i++ )
        std::cout << res[i] << std::endl ;
    return 0 ;
}
```

Templates

Através do mecanismo dos **templates**, a linguagem C++ suporta a definição de funções e de classes completamente gerais, parametrizadas usando tipos genéricos e valores constantes.

Sempre que um template é usado, o compilador usa-o para gerar código concreto adequado aos argumentos que foram passados para o template. Chama-se a este processo **instanciação**. A instanciação dum template faz lembrar um pouco a expansão duma macro. No entanto o mecanismo de instanciação dos templates é mais sofisticado porque suporta expansão recursiva e também manipulações do código com algum entendimento da sintaxe e da semântica da linguagem. Apesar desta sofisticação toda, o C++ tem a necessidade de validar o código após cada instanciação dum template, e nesse momento podem ser descobertos erros, muitas vezes difíceis de interpretar pelo programador. Em contraste com isto, o Java valida completamente as suas entidades genéricas antes de serem instanciadas. Há muitas classes da biblioteca padrão que foram implementadas como instanciações particulares de templates mais gerais. É o caso das classes std::string e std::stream, por exemplo.

O programador de C++ deve estar sempre à procura de boas oportunidades para definir templates, por forma que o código escrito seja genérico e reutilizável.

O seguinte template implementa um função `swap` genérica e pode ser aplicado a quaisquer tipos não-const. Repare o o tipo T usado na instanciação do template pode ser *inferido*, não sendo obrigatório passar o tipo concreto como parâmetro.

```
#include <iostream>

template <typename T>
void swap(T &a, T &b) {
    T aux = a ;
    a = b ;
    b = aux ;
}

int main() {
    int x = 5, y = 6 ;
    std::cout << x << " " << y << std::endl ;
    swap<int>(x, y) ;     // Instanciação explícita
    std::cout << x << " " << y << std::endl ;
    swap(x, y) ;          // Instanciação implícita
    std::cout << x << " " << y << std::endl ;
    return 0 ;
}
```

O seguinte template só funciona se for aplicado a tipos que suportam `operator+()` (e ainda um construtor de cópia, mas isto só poderá ser compreendido nas próximas aulas).

```
template <typename T>
T add(const T &a, const T &b) {
    return a + b ;
}
```

No exemplo anterior, repare que usámos passagem de argumentos por referência constante. Em geral, recomenda-se a utilização, nos templates, desse mecanismo de passagem da argumentos porque ele é eficiente para passar argumentos de grandes dimensões, mas também funciona bem com os tipos primitivos pequenos.

O seguinte template tem uma constante Size como argumento. O argumento Type representa um array com Size elementos de tipo Type. O conteúdo do array não pode ser modificado.

```
#include <iostream>

template <typename Type, int Size>
Type sum(const Type (&array)[Size]) {
    Type t = Type() ;         // Chama o construtor por omissão para obter o valor por
omissão de Type
    for( int idx = 0; idx < Size; idx++ )
        t += array[idx] ;
    return t ;
}

int main() {
    int values[] = { 1, 1, 1, 1, 1 } ;    // Size == 5
    int *ip = values ;
    std::cout << sum(values) << std::endl ;    // Compila sem problemas
    std::cout << sum(ip) << std::endl ;        // DÁ ERRO DE COMPILAÇÃO. Porquê?
}
```

STL - Contentores, Iteradores e muito mais

A Standard Template Library (STL) é uma parte essencial da biblioteca padrão do C++, orientada para a programação genérica. É constituída por contentores (e.g. vetores), algoritmos genéricos (e.g. sort), iteradores, funções-objeto, estruturas de dados genéricas, etc.

Um contentor é um objeto que guarda outros objeto (chamados *os seus elementos*) e fornece mecanismos para aceder aos seus elementos. Em particular, cada tipo-contentor tem associado um **iterador** que pode ser usado para iterar através dos elementos.

Os **iteradores** são objeto que funcionam de forma parecida com os apontadores. Um iterador tem as seguintes características:

- Dois iteradores podem ser comparados usando == ou !=.
- Dado um iterador it, *it representa o objeto corrente.
- ++it ou it++ avança o iterador para o próximo elemento.
- Pode ser usada aritmética de apontadores, por exemplo escrevendo *(it+2) para aceder ao objeto guardado duas à posições frente.

O seguinte exemplo mostra um iterador a ser usado para escrever todos os elementos dum vector pela ordem direta no intervalo [begin(), end()[, e depois pela ordem inversa no intervalo [rbegin(), rend()[.

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

using namespace std;

int main(int argc, char **argv) {
    vector<string> args(argv, argv + argc) ;
```

```
    for(
        vector<string>::iterator iter = args.begin();
            iter != args.end() ;
                ++iter
    )
        cout << *iter << " " ;
    cout << endl ;

    for (
        vector<string>::reverse_iterator iter = args.rbegin();
            iter != args.rend() ;
                ++iter
    )
        cout << *iter << " " ;
    cout << endl ;

    return 0 ;
}
```

A STL também suporta *iteratores-const* que permitem visitar a sequência de valores guardados, mas sem permitir a alteração dos elementos visitados.

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

using namespace std;

int main(int argc, char **argv) {
    vector<string> args(argv, argv + argc) ;

    for(
        vector<string>::const_iterator iter = args.begin();
            iter != args.end() ;
                ++iter
    )
        cout << *iter << " " ;
    cout << endl ;

    for (
        vector<string>::const_reverse_iterator iter = args.rbegin();
            iter != args.rend() ;
                ++iter
    )
        cout << *iter << " " ;
    cout << endl ;

    return 0 ;
}
```

A biblioteca STL é um mundo! Os exemplos anteriores constituem uma introdução muito básica a aspetos da STL. Mas há muito mais de útil para aprender. A parte dos algoritmos genéricos é especialmente interessante. Infelizmente o tempo é escasso e tudo isto sai fora do âmbito da cadeira de LAP.

Introdução às classes em C++

Uma **classe** descreve um conjunto de objetos, todos com a mesma estrutura. Uma classe contém:

- **Membros de dados**, geralmente privados;
- **Membros funcionais**, geralmente públicos.

Há quatro variedades de funções membro que geralmente se definem:

- **Construtores -** Inicializam os objetos, logo a seguir à sua criação (era melhor que se chamassem "initializadores").
- **Destrutor -** Cada classe deve definir um destrutor. O destrutor é ativado logo antes do objeto ser destruído. O destrutor nunca é chamado diretamente pelo programador. Declaramos sempre os destrutores como virtuais.
- **Seletores -** Consultam o estado de this, o objeto da função, sem nunca o modificar. Os seletores são declarados virtual e const.
- **Modificadores -** Os modificadores alteram o estado de this, o objeto da função. São declarados virtuais e retornam sempre void.

Os seletores e os modificadores são introduzidos por uma questão metodológica. Programar usando praticamente apenas seletores e modificadores permite criar um programa mais fácil de entender e de modificar.

O nome da classe é considerado um tipo de dados e que pode ser usado para declarar variáveis, argumentos de funções, etc.

Repare: ao contrário do Java a declaração duma classe acaba sempre com um ";".

Em C++ uma classe é normalmente declarada num ficheiro ".h" e implementada num ficheiro ".cpp". Veja no exemplo a seguir.

Programação com classes - Stack de inteiros

O seguinte exemplo ilustra uma classe típica em C++.

Nesta classe o construtor reserva dinamicamente memória para a representação interna dos stacks e o destrutor liberta essa memória quando o objeto é eliminado. O C++ não tem gestão de memória automática (o Java tem), mas os destrutores ajudam a libertar memória nas alturas apropriadas.

No cabeçalho da implementação do construtor, repare na **lista de inicializadores**.

Este exemplo ilustra também a utilização de exceções em C++. Em C++ qualquer valor, independentemente do seu tipo, pode ser usado para representar uma exceção.

Estude tudo atentamente.

Ficheiro IntStack.h

```
#ifndef _IntStack_
#define _IntStack_

#include <string>

class IntStack {
  private:
    int *elems ;
    int top ;
    const int capacity ;
    static const int defaultCapacity ;
    void ErrorMesg(std::string s) const ;

  public:
    IntStack(int capacity = defaultCapacity) ;
    virtual ~IntStack() ;
    virtual int Capacity() const ;
    virtual int Size() const ;
    virtual bool IsEmpty() const ;
    virtual bool IsFull() const ;
```

```
    virtual int Top() const ;
    virtual void Push(int i) ;
    virtual int Pop() ;
} ;

#endif
```

Ficheiro IntStack.cpp

```
#include <iostream>
#include <string>
#include "IntStack.h"

const int IntStack::defaultCapacity = 100 ;

IntStack::IntStack(int capacity):
            elems(new int[capacity]), top(0), capacity(capacity) {
    std::cout << "CONSTRUCTOR activated\n" ;
}

IntStack::~IntStack() {
    std::cout << "DESTRUCTOR activated\n" ;
    delete elems ;
}

void IntStack::ErrorMesg(std::string s) const {
    throw s ;
}

int IntStack::Capacity() const {
    return capacity ;
}

int IntStack::Size() const {
    return top ;
}

bool IntStack::IsEmpty() const {
    return Size() == 0 ;
}

bool IntStack::IsFull() const {
    return Size() == Capacity() ;
}

int IntStack::Top() const {
    if( IsEmpty() )
        ErrorMesg("Pop: Stack is empty") ;
    else
        return elems[top-1] ;
}

void IntStack::Push(int v) {
    if( IsFull() )
        ErrorMesg("Push: Stack is full") ;
    else
        elems[top++] = v ;
}

int IntStack::Pop() {
    if( IsEmpty() )
        ErrorMesg("Pop: Stack is empty") ;
    else
        return elems[--top] ;
}
```

Ficheiro Main.cpp

```
#include <iostream>
#include "IntStack.h"

int main() {
    try {
        std::cout << "Testing the IntStack class!!\n" ;
        IntStack s ;
        std::cout << s.Capacity() << std::endl ;
        s.Push(123) ;
        std::cout << s.Pop() << std::endl ;
        std::cout << s.Pop() << std::endl ;
    } catch( std::string str ) {
            std::cout << "Caught exception - " << str << std::endl ;
    } catch( int i ) {
            std::cout << "Caught exception and rethrow it - " << i << std::endl ;
            throw ;
    } catch( ... ) {
            std::cout << "Caught any other exception\n" ;
    }
    return 0 ;
}
```

O output deste programa é o seguinte:

```
Testing the IntStack class!!
CONSTRUCTOR activated
100
123
DESTRUCTOR activated
Caught exception - Pop: Stack is empty
```

No ficheiro Main.cpp, o stack foi definido usando `IntStack s` e não `IntStack *s`. O que se passa é que a variável `s` é guarda sempre o mesmo stack. Se pudesse conter diferentes stacks ao longo do tempo, e se a identidade destes fosse importante (normalmente é!), então `s` teria de ter o tipo `IntStack *s`. Uma segunda razão: se desejássemos criar o stack dinamicamente, o operador `new`, então `s` também teria de ter o tipo `IntStack *s`

Programação com classes - Expressões algébricas

Considere o problema de representar em C++ expressões algébricas constituídas pelas seguintes entidades:

- operador binário "+";
- operador binário "*";
- operador unário "-";
- valores literais reais;
- variável "x".

Dois exemplos de expressões:

```
2.5*x+5.1
-(x-5.9)
```

Para capturar a estrutura destas expressões e facilitar a sua manipulação, o ideal é usar uma representação baseada em árvores. Os nós dessas árvores irão ter tipos muito diversificados.

As árvores, em geral, são estruturas de dados dinâmicas, e por isso os nós da árvore vão ter de ser objetos criados dinamicamente, usando o operador `new`.

No programa abaixo, há duas razões para usar apontadores:

- Porque os nós das árvores serão criados dinamicamente (repare que o operador `new` retorna um apontador).
- Porque a hierarquia de classes que usaremos exige a utilização de polimorfismo (as variáveis e os argumentos das funções devem ser polimórficos).

Para representar os nós das árvores, vamos definir uma hierarquia de classes com três níveis:

- **Nível 1:** A classe abstrata ExpNode implementa Big(), a única operação que é definida da mesma maneira para todos os nós da árvore.
- **Nível 2:** As classes abstratas BinNode, UnaryNode e ZeroNode implementam as operações Size() e Height(), ou seja as operações que dependem apenas da aridade dos nós.
- **Nível 3:** As classes concretas AddNode, MultNode, SimNode, ConstNode e VarNode implementam as operações Eval() e Deriv(), ou seja as operações que dependem do significado dos nós.

Uma **classes abstrata** é uma classe que corresponde a um conceito tão geral (e.g. Animal, Veículo) que não faz sentido criar instâncias diretas dessas classes, nem concretizar código para alguns dos seus métodos. Tal como o Java, o C++ não permite criar instâncias duma classe abstrata. Os métodos sem corpo chamam-se **métodos puros** (métodos abstratos, em Java), e são declarados usando a sintaxe "**= 0**" (... bem podiam ter arranjado uma sintaxe melhor).

Uma **classes concreta** é uma classe não abstrata, ou seja uma classe que representa um conceito bem concreto (e.g. Tigre, Automóvel) e não tem métodos puros.

Uma subclasse concreta tem a responsabilidade de definir todos os métodos puros que herda. Já uma subclasse abstrata já não tem essa responsabilidade.

Além das classes do nível 1, 2, e 3, este programa coloca ainda mais acima na hierarquia uma classe chamada Exp. A classe Exp só contém **funções puras** (i.e. funções sem corpo, equivalentes aos métodos abstratos do Java). Em C++ não existem as interfaces do Java, mas é fácil imitar uma interface do Java usando uma classe abstrata contendo apenas funções puras.

No futuro, será fácil adicionar novas operações aritméticas à representação das expressões. Basta criar uma nova classe por cada novo operador aritmético, e a maior parte do código da nova classe será herdado; e, acima de tudo, não é necessário alterar código existente. Todas estas coisas boas só são possíveis porque o programa foi planeado, organizado e escrito a pensar nas questões da **factorização** e da **extensibilidade**:

- Um **sistema fatorizado** é um sistema sem código repetido, ou seja, sem redundância. Num sistema fatorizado, código que em princípio apareceria repetido em várias classes, fica concentrado (ou seja, fatorizado) numa superclasse, ficando disponível por herança.
- Um **sistema extensível** é um sistema que se pode fazer crescer, sem que se tenha de se alterar o que já foi escrito. A extensibilidade obtém-se pela via da abstração: Código que seja escrito com base em conceitos abstratos consegue lidar com as entidades iniciais descritas no enunciado do problema, mas também com entidades a criar no futuro (desde que se enquadrarem nas abstrações inicialmente consideradas.)

Ficheiro Exp.h

```
#ifndef _Exp_
#define _Exp_

class Exp {  /* "Interface" */
  public:
    virtual bool Big() const = 0 ;
    virtual int Size() const = 0 ;
    virtual int Height() const = 0 ;
    virtual double Eval(double v) const = 0 ;
    virtual Exp *Deriv() const = 0 ;
} ;

class ExpNode: public Exp {
  public:
    ExpNode() ;
```

```
    virtual ~ExpNode() ;
    bool Big() const ;
} ;

class BinNode: public ExpNode {
  protected:
    Exp *l, *r ;
  public:
    BinNode(Exp *l, Exp *r) ;
    virtual ~BinNode() ;
    int Size() const ;
    int Height() const ;
} ;

class UnaryNode: public ExpNode {
  protected:
    Exp *e ;
  public:
    UnaryNode(Exp *e) ;
    virtual ~UnaryNode() ;
    int Size() const ;
    int Height() const ;
} ;

class ZeroNode: public ExpNode {
  public:
    ZeroNode() ;
    virtual ~ZeroNode() ;
    int Size() const ;
    int Height() const ;
} ;

class AddNode: public BinNode {
  public:
    AddNode(Exp *l, Exp *r) ;
    virtual ~AddNode() ;
    double Eval(double v) const ;
    Exp *Deriv() const ;
} ;

class MultNode: public BinNode {
  public:
    MultNode(Exp *l, Exp *r) ;
    virtual ~MultNode() ;
    double Eval(double v) const ;
    Exp *Deriv() const ;
} ;

class SimNode: public UnaryNode {
  public:
    SimNode(Exp *e) ;
    virtual ~SimNode() ;
    double Eval(double v) const ;
    Exp *Deriv() const ;
} ;

class ConstNode: public ZeroNode {
  private:
    double c ;
  public:
    ConstNode(double c) ;
    virtual ~ConstNode() ;
    double Eval(double v) const ;
    Exp *Deriv() const ;
} ;

class VarNode: public ZeroNode {
  public:
    VarNode() ;
```

```
    virtual ~VarNode() ;
    double Eval(double v) const ;
    Exp *Deriv() const ;
} ;

#endif
```

Ficheiro Exp.cpp

```
#include <algorithm>   // Makes std::max available
#include "Exp.h"

ExpNode::ExpNode() {}
ExpNode::~ExpNode() {}
bool ExpNode::Big() const { return Size() > 1000 ; }

BinNode::BinNode(Exp *l, Exp *r) : ExpNode(), l(l), r(r) {}
BinNode::~BinNode() {}
int BinNode::Size() const { return 1 + l->Size() + r->Size() ; }
int BinNode::Height() const { return 1 + std::max(l->Height(),r->Height()) ; }

UnaryNode::UnaryNode(Exp *e) : ExpNode(), e(e) {}
UnaryNode::~UnaryNode() {}
int UnaryNode::Size() const { return 1 + e->Size() ; }
int UnaryNode::Height() const { return 1 + e->Height() ; }

ZeroNode::ZeroNode() : ExpNode() {}
ZeroNode::~ZeroNode() {}
int ZeroNode::Size() const { return 1 ; }
int ZeroNode::Height() const { return 1 ; }

AddNode::AddNode(Exp *l, Exp *r) : BinNode(l, r) {}
AddNode::~AddNode() {}
double AddNode::Eval(double v) const { return l->Eval(v) + r->Eval(v) ; }
Exp *AddNode::Deriv() const { return new AddNode(l->Deriv(), r->Deriv()) ; }

MultNode::MultNode(Exp *l, Exp *r) : BinNode(l, r) {}
MultNode::~MultNode() {}
double MultNode::Eval(double v) const { return l->Eval(v) * r->Eval(v) ; }
Exp *MultNode::Deriv() const { return new AddNode(new MultNode(l, r->Deriv()),
                                          new MultNode(l->Deriv(), r)) ; }
SimNode::SimNode(Exp *e) : UnaryNode(e) {}
SimNode::~SimNode() {}
double SimNode::Eval(double v) const { return -e->Eval(v) ; }
Exp *SimNode::Deriv() const { return new SimNode(e->Deriv()) ; }

ConstNode::ConstNode(double c) : ZeroNode(), c(c) {}
ConstNode::~ConstNode() {}
double ConstNode::Eval(double v) const { return c ; }
Exp *ConstNode::Deriv() const { return new ConstNode(0) ; }

VarNode::VarNode() : ZeroNode() {}
VarNode::~VarNode() {}
double VarNode::Eval(double v) const { return v ; }
Exp *VarNode::Deriv() const { return new ConstNode(1) ; }
```

Ficheiro Main.cpp

```
#include <iostream>
#include "Exp.h"

int main() {
    Exp *e = new MultNode(
                    new AddNode(new ConstNode(4), new VarNode()),
                    new ConstNode(6.5)) ;
```

```
    std::cout << "((4 + x) * 6.5)' (3) = " << e->Deriv()->Eval(3) << std::endl ;
    return 0 ;
}
```

Extensibilidade

Um **sistema extensível** é um sistema que se pode fazer crescer, sem que se tenha de se alterar o que já foi escrito.

A extensibilidade obtém-se pela via da **abstracção**:

- Código que seja escrito com base em conceitos abstractos consegue lidar com as entidades iniciais descritas no enunciado do problema, mas também (e aqui é que surge a extensibilidade) com entidades a criar no futuro. Claro que isto só funciona se as entidades futuras se enquadrarem nas abstracções inicialmente consideradas.

Há todo o interesse em software extensível. As principais razões são as seguintes:

- Os programas ficam organizados em torno de ideias gerais e claras.
- Os programas ficam mais fáceis de actualizar.
- Os programas ficam mais fiáveis.
- O tempo de vida útil dos programas aumenta.
- Poupa-se tempo e dinheiro.
- Os programas ficam mais elegantes e tornam-se fonte de prazer estético (pelo menos para quem os escreve).

Extensibilidade através da fatorização

A fatorização das classes é um **requisito** para obter extensibilidade, pois essa fatorização faz parte do processo de desenho que ajuda a identificar as abstrações naturais dos programas.

Mas a fatorização, só por si, **não é suficiente** para obter extensibilidade. Para obter extensibilidade, falta ainda, pelo menos, um segundo ingrediente: garantir que o programa não usa testes explícitos de classe.

A evitar: Testes explícitos de classe

Existe uma questão que, de forma dramática, dá origem a **código não extensível**. Trata-se do uso de **testes explícitos para determinar a classe concreta dum objeto** (feita usando `dynamic_cast` ou de outra forma).

Tal código nunca pode ser extensível, pois a sua lógica está comprometida com as classes existentes: ou seja, esse código não conseguirá lidar com objetos de classes a criar no futuro. Para estender a lógica a esses novos objetos, seria necessário reescrever o código...

Técnicas para evitar os testes explícitos de classe

Por vezes surge a tentação de testar directamente a classe concreta a que pertence um objeto. Isso tem de ser evitado a todo o custo, se tivermos como objectivo a escrita de código extensível.

Vejamos algumas técnicas que permitem evitar os testes explícitos de classe.

Técnica do envio de mensagem

Em vez de testar directamente a classe concreta dum objeto, podemos enviar-lhe uma mensagem perguntando algo. Tal código já é extensível pois funciona com quaisquer objetos que suportem um dado método, mesmo com objetos de classes a criar futuramente.

Por exemplo, num jogo baseado numa matriz bidimensional, em que vários monstros perseguem um herói humano, o que é que um monstro deverá fazer quando se cruza com outra personagem?

- **Versão errada** (visão dum mundo fechado) -- O monstro determina, usando a expressão `dynamic_cast<HeroClass>(vizinho) != 0`, se o vizinho é o herói. Se for o herói, então o monstro almoça o herói. Se não for o herói, então não faz nada. Este código não é nada extensível pois a sua lógica está dependente das classes concretas iniciais.
- **Versão correcta** (visão dum mundo aberto) -- O monstro envia ao vizinho a mensagem `vizinho->Comestível()` e depois actua em conformidade com a resposta. Este código já é extensível, pois funciona com qualquer personagem, mesmo com uma personagem a criar futuramente, desde que esta saiba responder à mensagem `Comestível()`.

Em conclusão, conseguimos escrever código extensível **introduzindo o conceito abstracto** *comestível*.

Técnica das interfaces auxiliares

O conceito *comestível*, referido no ponto anterior, é realmente abstracto. Suponha que introduzimos esse conceito usando uma interface `Comestível`. Suponha ainda que todas as classes que definem personagens comestíveis seguem a regra de implementar a interface `Comestível`.

Nestas condições é fácil testar de uma dada personagem vizinha é, ou não, comestível. Basta escrever:

```
dynamic_cast<Comestível *>(vizinho) != 0
```

Este código é extensível porque é abstracto.

A interface `Comestível` pode ficar vazia; a sua simples existência é quanto basta. (O C++ não tem interfaces, mas estas podem ser simuladas usando uma classe abstrata contendo apenas métodos puros, como já se viu.)

Técnica dos níveis

Suponha que no jogo existem muitas classes diferentes de personagens, e que entre essas classes de personagens se estabelecem complexas regras de alimentação, do tipo *cadeia alimentar*.

Nesse caso, convém associar um *nível alimentar* a cada tipo de personagem e estabelecer a seguinte regra: uma personagem pode comer outra personagem caso o nível alimentar da primeira seja superior ao da segunda. Concretamente, um objeto pode comer o seu vizinho se:

```
NivelAlimentar() > vizinho->NivelAlimentar()
```

Técnica da *mesma classe*

Para efeitos de reprodução, por exemplo, um objeto pode precisar de saber se um outro objeto, seu vizinho, é da mesma classe. Como fazer isso de forma genérica?

Faz-se assim:

```
#include <typeinfo>

typeid(*this) == typeid(*vizinho)
```

Construtores

Coerções controladas pelo programador

Se na classe T2 se define um construtor com um argumento de tipo T1, então o C++ define uma regra de coerção de T1 para T2. Uma **coerção** é uma conversão automática de tipo.

O segundo construtor define uma regra de coerção de double para Complex:

```
class Complex {
  public:
    Complex(double re, double re) ;
    Complex(double d) ;
    ...
}

Complex::Complex(double re, double re) : re(re), im(im) {}

Complex::Complex(double d) : re(d), im(0.0) {}

int main() {
    Complex a(2.5, 6.6) ;
    Complex b(34.5) ;    // chamada direta do construtor de coerção
    a = 5.6 ;            // chamada indireta do construtor de coerção
    a = 54 ;             // dupla coerção: int -> double -> Complex
}
```

A palavra reservada `explicit`

Por vezes queremos um construtor com um único argumento, mas não pretendemos a regra de coerção. Para resolver o problema, usa-se a palavra reservada **explicit**. Exemplo no contexto da classe IntStack da aula teórica anterior:

```
class IntStack {
  public:
    explicit IntStack(int capacity = defaultCapacity) ;
} ;
```

Sem usar `explicit` o seguinte código absurdo seria válido:

```
IntStack s ;
s = 5 ;
```

Coerções para tipos pré-definidos

Para definir uma regra de coerção para um tipo pré-definido T, define-se dentro da classe uma função pública com nome "operator T" (não é um construtor), como se exemplifica:

```
class Complex {
  public:
    virtual operator double() ;
    ...
}

Complex::operator double() {
    return re ;
}

int main() {
    Complex c(2.5, 6.6) ;
    double d = c ;
}
```

Excesso de coerção

As regras de coerção combinam-se automaticamente, ou seja se existir uma coerção T1->T2 e uma coerção T2->T3, então também existe uma coerção composta T1->T3. É preciso ter o cuidado de não definir demasiadas regras de coerção. Quando

são usadas pelo compilador, cada coerção têm de poder ser determinadas de forma não ambígua; caso contrário gera um erro de compilação.

Construtor por omissão

É um construtor que pode ser chamado sem argumentos. Na sua declaração, ou não tem argumentos, ou então com todos os argumentos aceitam valores por omissão. O C++ define automaticamente um construtor por omissão nas classes que não contêm qualquer construtor.

A classe IntStack da aula teórica anterior tem um construtor por omissão explicitamente definido. Exemplo de chamada de construtor por omissão:

```
IntStack s ;
```

Para se criar um array de objetos da classe IntStack, esta classe precisa de ter um construtor por omissão. Exemplos que geram erros:

```
IntStack arr[5] ;
IntStack *parr = new IntStack[5] ;
```

Mas se inicializarmos o array de objetos explicitamente, o construtor por omissão já não é exigido:

```
IntStack arr[5] = { IntStack(10), IntStack(10), IntStack(10), IntStack(10), IntStack(10)
} ;
```

Para se criar um array de apontadores, também não é preciso haver construtor por omissão:

```
IntStack *arr[5] ;
```

Construtor de cópia

O construtor de cópia é usado para **inicializar** novos objetos a partir de objetos existentes do mesmo tipo. Por exemplo o construtor de cópia é chamado na segunda e terceira linhas:

```
IntStack s ;
IntStack t(s) ;
IntStack u = s ;
```

O C++ define automaticamente um construtor de cópia em cada classe, mas por vezes esse construtor não serve os interesses do programador e precisa de ser redefinido. É o que acontece na classe IntStack da aula anterior. Usando a classe original, as linhas de código anteriores fazem com que os três objetos fiquem a partilhar o mesmo array interno de elementos e depois isso causa um erro de execução quando o destrutor é chamado pela segunda vez.

Eis como se deve definir o construtor de cópia na classe IntStack:

```
class IntStack {
  public:
    IntStack(const IntStack &other) ;
    ...
}

IntStack::IntStack(const IntStack &other) {   // copy constructor
    elems = new int[other.capacity] ;
    for( int i = 0 ; i < other.top ; i++ )
        elems[i] = other.elems[i] ;
    top = other.top ;
    capacity = other.capacity ;
}
```

É útil saber que o construtor de cópia também é chamado na passagem de argumentos por valor e no retorno de objetos.

Em geral, as classes que têm apontadores entre os seus membros de dados devem redefinir o constructor de cópia.

Operação de atribuição

A operação de atribuição não é implementada num construtor, mas tem uma forte relação com o constructor de cópia e por isso é discutida aqui.

A operação de atribuição está predefinida para todos os tipos do C++. A segunda linha exemplifica uma atribuição:

```
IntStack s, t ;
s = t ;
```

Por vezes a implementação por omissão da atribuição para um dado tipo não serve os interesses do programador e precisa de ser redefinida. É o que acontece na classe IntStack da aula anterior. Usando a classe original, a atribuição anterior faz com que os dois objetos fiquem a partilhar o mesmo array interno de elementos e depois isso causa um erro de execução quando o destrutor é chamado pela segunda vez.

Para redefinir a operação de atribuição, faz-se overloading do operador "=", dentro da classe IntStack:

```
class IntStack {
  public:
    IntStack& operator=(const IntStack&);
    ...
}

IntStack& IntStack::operator=(const IntStack &other) {    // assignement
    if( this == &other )    // preocupa-se com o caso s = s
        return *this ;
    free(elems) ;        // tem de se eliminar primeiro o array antigo
    elems = new int[other.capacity] ;
    for( int i = 0 ; i < top ; i++ )
        elems[i] = other.elems[i] ;
    top = other.top ;
    capacity = other.capacity ;
    return *this ;
}
```

É possível desativar a operação de atribuição para um dada classe. Basta definir essa operação na zona *privada* dessa classe! A biblioteca padrão do C++ faz isso para as streams.

Em geral, as classes que têm apontadores entre os seus membros de dados devem redefinir a operação de atribuição.

Relação com a STL

Para as classes definidas pelo programador funcionarem bem com os contentores da STL (e.g. std::vector), essas classes devem providenciar uma certa funcionalidade básica nomeadamente:

- Construtor por omissão;
- Construtor de cópia;
- Operação de atribuição;
- Destrutor virtual.

Os contentores da STL que capazes de fazer ordenações exigem um pouco mais:

- Igualdade definida usando o operador "==";
- Predicado menos, definido usando o operador "<".

O apontador `this`

Dentro duma classe C++, o objeto genérico que recebe as mensagens chama-se **objeto da função** e é denotado pela palavra reservada **`this`**. Este nome está implicitamente declarado dentro de cada membro funcional da classe.

Convém saber que `this` é um apontador constante. Nas funções da classe IntStack, a declaração implícita é a seguinte:

```
extern IntStack *const this ;
```

Funções **friend** e classes **friend**

Membros de dados declarados como privados não podem ser acedidos a partir do exterior da classe. Mas usando a palavra reservada **friend**, o programador pode abrir exceções e autorizar o acesso a algumas funções globais e a algumas classes não locais. Portanto o C++ permite que uma classe proporcione diferentes visões a diferentes entidades exteriores, consoante as necessidades da classe.

Exemplos de funções `friend`. As primeiras duas funções estendem os operadores de escrita e leitura das streams aos números complexos. A terceira função define uma soma na qual os dois argumentos estão em pé de igualdade para efeitos da aplicação de coerções (por exemplo, permite a expressão `2.3 + Complex(2.2, 5.6)`).

```
class Complex {
  private:
    double re, im ;
  public:
    ...
    Complex& operator +=(const Complex &other) ;
// friend:
    friend std::ostream& operator << (std::ostream& output, const Complex &c) ;
    friend std::istream& operator >> (std::istream& input, Complex &c) ;
    friend Complex operator +(const Complex &a, const Complex &b) ;
}

std::ostream& operator << (std::ostream& output, const Complex &c) {
    output << c.re << "+" << c.im << "i" ;
    return output ;
}

std::istream& operator >> (std::istream& input, Complex &c) {
    input >> c.re >> c.im ;
    return input ;
}

Complex operator +(const Complex &a, const Complex &b) {
    return Complex(a.re + b.re, a.im + b.im) ;
}

Complex& Complex::operator +=(const Complex &other) {
    re += other.re ;
    im += other.im ;
    return *this ;
}
```

Exemplo de classe `friend`. A classe Node, que implementa os nós duma árvore binária, abre uma exceção para a classe BinaryTree e permite-lhe aceder à representação interna desses nós.

```
class Node  {
  private:
    int data ;
    int key ;
  public:
    ...
// friend:
    friend class BinaryTree ;
} ;
```

Overloading dos operadores `[]` e `()`

Para ilustrar mais possibilidades relativas ao overloading de operadores em C++, vamos adicionar à nossa classe IntStack da aula anterior a possibilidade de acesso aos elementos dum stack usando os operadores "[]" e "()".

```
class IntStack {
  public:
    int &operator[](unsigned int index) ;
    int &operator()(unsigned int index) ;
    ...
}

int &IntStack::operator[](unsigned int index) {
    if( index >= top )
        ErrorMesg("[]: No such element") ;
    else
        return elems[index] ;
}

int &IntStack::operator()(unsigned int index) {
    if( index >= top )
        ErrorMesg("(): No such element") ;
    else
        return elems[index] ;
}
```

Repare que com estas definições, as expressões da forma `s[i]` e `s(i)` tanto produzem valores como aceitam valores, consoante o local em que são usadas nas expressões (como *rvalues* ou como *lvalues*). Exemplos de utilização:

```
IntStack s ;
...
int i = s[1] ;
int j = s(2) ;
s[3] = i ;
s(4) = j ;
```

Overloading dos operadores `++` prefixo e `++` pós-fixo

Para distinguir a versão prefixa da versão pós-fixa dos operadores ++, em C++ a versão pós-fixa é declarada com um argumento inteiro a mais, argumento esse que depois é ignorado na implementação (e também não aparece na chamada).

```
class Complex {
  private:
    double re, im ;
  public:
    ...
    Complex& operator ++() ;      // prefixo
    Complex operator ++(int) ;    // pós-fixo, o argumento não usado
}

Complex& Complex::operator ++() {
    re++ ;
    return *this ;
}

Complex Complex::operator ++(int) {
    Complex tmp(*this) ;
    re++ ;
    return tmp ;
}
```

Exemplos de utilização:

```
Complex c(12.3, 56.3) ;
Complex e = ++c ;     // prefixo
```

```
Complex d = c++ ;     // pós-fixo, argumento não usado
```

Operadores que podem ser overloaded

A seguinte tabela mostra todos os operadores que podem ser overloaded em C++:

```
+       -       *       /       %       ^       &       |
~       !       ,       =       <       >       <=      >=
++      --      <<      >>      ==      !=      &&      ||
+=      -=      *=      /=      %=      ^=      &=      |=
<<=     >>=     []      ()      ->      ->*     new     new[]
delete  delete[]
```

A seguinte tabela mostra todos os operadores que não podem ser overloaded em C++:

```
.       .*      ::      ?:      sizeof  type
```

Operators in C and C++ na Wikipedia

Sobre os operadores `->*` e `.*` ver aqui. Em resumo, o C++ permite obter um apontador `A` para uma variável de instância duma classe `C::v` (mesmo sem se saber qual o valor de `this`!) e, na altura de desreferenciar, especifica-se o valor de `this`, assim `object.*A` ou assim `ptobject->*A`. Estes operadores também funcionam com apontadores para funções de instância de classes.

Membros de dados estáticos e membros funcionais estáticos

Numa classe em C++, os membros de dados estáticos e os membros funcionais estáticos são declarados como **static**.

Tal como em Java, os membros estáticos duma classe pertencem à classe e não a nenhuma sua instância particular. Os membros estáticos podem ser acedidos por todas as instâncias da classe.

Herança em C++

Tal como em Java, uma característica essencial das classes em C++ é o facto de elas serem incrementalmente modificáveis. Usando o *mecanismo de herança*, o programador consegue criar de forma expedita uma nova classe (**classe derivada**) a partir de outra (**classe base**), definindo apenas as componentes da nova classe que são *adicionadas* ou *modificadas*, relativamente à classe original. As componentes da classe base que não forem redefinidas na classe derivada são automaticamente **herdadas**, ou seja, são reaproveitadas na classe derivada.

Em Java cada classe só pode ter uma classe base imediata, pelo que a herança se diz **simples**. Já no C++, cada classe pode diversas classes base imediatas, dizendo-se por isso que a herança é **múltipla**.

O seguinte pequeno exemplo ilustra a flexibilidade do mecanismo de herança, quando se usam funções virtuais. Na nossa cadeira, declaramos todas as funções como virtuais, exceto os construtores e as funções estáticas, por ser proibido. Repare que em Java todas as funções são virtuais, sem ser preciso declarar isso explicitamente.

```
class B {
  public:
    virtual void f() {
```

```
            std::cout << g() << std::endl ;
        }
        virtual int g() {
            return 1 ;
        }
    } ;

    class D : public B {
      public:
        int g() {
            return B::g() + 1 ;
        }
    } ;
```

O seguinte exemplo produz o output "1 2".

```
    int main() {
        B b ;
        D d ;
        b.f() ;
        d.f() ;
        return 0 ;
    }
```

No exemplo anterior convém chamar a atenção para os seguintes dois pontos, todos importantíssimos:

- Na classe derivada, a função redefinida g tem acesso à definição original por meio do operador de escopo "::". (Em Java seria usada a palavra super para obter o mesmo efeito.)
- Na classe derivada, a função herdada f adapta-se ao novo contexto e chama a versão de g da classe derivada. Isto funciona porque a função g foi declarada como virtual. (O Java funciona da mesma maneira.)

Membros funcionais virtuais

A utilização de funções virtuais em C++ é o ponto fulcral dum mecanismo de herança flexível, que permite que os métodos herdados sejam reinterpretados no contexto da classe derivada. Por outras palavras, permite que o código herdade funcione bem com os objetos da classe derivada.

A diferença entre uma função ser não-virtual ou ser virtual é a seguinte:

- A chamada duma função não-virtual é resolvida em tempo de compilação (ligação estática);
- A chamada duma função virtual é resolvida parcialmente em tempo de compilação e parcialmente em tempo de execução (ligação semi-dinâmica).

No exemplo anterior, se a função g não tivesse sido declarada como virtual, o output do programa teria sido "1 1". Medite e tente perceber que isso é resultado de ser estática a ligação do nome da função invocada à definição dessa função.

Herança pública, protected e privada

A forma de herança que se usa mais em C++ é, de longe, a herança pública.

Quando se usa **herança pública**, através duma declaração da forma que se exemplifica abaixo, os direitos de acesso dos membros da classe base passam sem alteração para a classe derivada.

```
    class Derived: public Base
```

Quando se usa **herança protected**, através duma declaração da forma que se exemplifica abaixo, os membros públicos da classe base passam a membros protected na classe derivada. Os direitos de acesso dos restantes membros não sofre alteração.

```
    class Derived: protected Base
```

Quando se usa **herança privada**, através duma declaração da forma que se exemplifica abaixo, todos os membros da classe base passam a membros privados na classe derivada.

```
class Derived: private Base
class Derived: Base
```

Herança e subtipos

A herança pública é a única forma de herança que faz com que a classe derivada constitua um **subtipo** da classe base.

As outras duas formas de herança servem para reutilizar o código da classe base, mas sem definir subtipo. Neste aspeto o C++ é mais flexível do que o Java. Em Java, herança implica sempre a definição dum subtipo.

Implementação das funções virtuais em C++

É assim que se implementam as funções virtuais em C++:

- A cada classe associa-se uma tabela de apontadores para as funções virtuais da classe. Essa tabela costuma ser chamada de **vtable**.
- A tabela de funções virtuais de uma classe derivada D é construída com base na tabela de funções virtuais da classe base B. Podem ser introduzidas no entanto adições (se na classe D aparecerem funções virtuais novas) ou alterações (no caso de algumas funções virtuais serem redefinidas).
- Todos os objetos contêm um apontador escondido para a tabela de funções virtuais da sua classe-mãe.
- Quando se envia uma mensagem f para o object referido pelo apontador p, assim `p->f(args)`, a determinação da função a ser executada é feita com base no object apontado por p e num valor inteiro i constante (determinado pelo compilador) que funciona como índice numa tabela de funções virtuais. [No exemplo do início deste capítulo sobre herança, repare que a chamada `g()` a partir do interior de `B::f()` corresponde realmente à chamada `this->g()`.]

Curiosidade: a razão da sintaxe "= 0" para as funções puras virtuais (faladas na aula anterior) é a seguinte: A cada função pura também corresponde uma célula da tabela de funções da respetiva classe. Mas como uma função pura não tem código executável associado, então coloca-se um zero nessa célula de memória.

Classes abstratas e funções virtuais puras

Estas questões já foram discutidas na aula anterior, a propósito do 2º exemplo.

Subtipos e polimorfismo em C++

O C++ suporta uma forma particular de polimorfismo:

> Uma variável ou argumento de tipo T* pode apontar para objetos do tipo T* assim como para objetos de qualquer subtipo de T.

Considere o seguinte pedaço de código:

```
class B {...} ;
class D : public B {...} ;
B b, *bp ;
D d, *dp ;
```

As atribuições seguintes são corretas:

```
bp = &b ;
bp = &d ;
dp = &d ;
```

```
bp = dp ;
```
As atribuições seguintes são incorretas:

```
dp = &b ;
dp = bp ;
```
Na chamada

```
bp->f() ;
```

- Se f() for uma função membro não-virtual da classe B, então a ligação de f é determinada estaticamente com base no tipo do apontador bp.
- Mas se f() for uma função membro virtual da classe B, então a ligação de f é determinada dinamicamente com base no tipo do object para o qual bp correntemente aponta.

Como se vê, para escrever código polimórfico em C++ é necessário manipular os objetos através de apontadores. Também se podem usar referências, mas as referências têm uma utilização limitada pois não podem ser alteradas depois de inicializadas.

Para reforçar esta ideia, repare que o C++ inclusivamente não permite declarar variáveis cujo tipo seja uma classe abstrata. Só é possível declarar apontadores ou referências para classes abstratas.

Apesar de tudo, repare que nem todo o código que se escreve em C++ ambiciona ser polimórfico. Por isso não é obrigatório estar sempre a usar apontadores em C++.

Herança múltipla em C++

A existência de herança apenas simples, pode forçar o uso de um estilo de programação pouco natural em que a integridade conceptual do desenho do programa é quebrada.

Por exemplo, em Smalltalk-80, uma linguagem que suporta apenas herança simples, a class Transcript suporta a funcionalidade das classes Window e WriteStream. Como não pode ser declarada como subclasse destas duas classes ao mesmo tempo, ela é declarada apenas como subclasse de Window, tendo os métodos de WriteStream de ser duplicados dentro de Transcript.

É mais **natural** permitir que uma classe possa ter múltiplas superclasses. Assim a **herança múltipla** surge como algo de útil de desejável. Imagine por exemplo o problema da representação dum *canivete suíço*: trata-se dum canivete, mas também duma tesoura, duma chave de fendas, dum abre-latas, dum saca-rolhas, etc.

Mas a herança múltipla também traz complicações:

- Se o mesmo membro m for herdado várias vezes por via de classes derivadas distintas, o que fazer?
- Se membros distintos mas com o mesmo nome m forem herdados de diferentes classes base, o que fazer?

Algumas das soluções possíveis:

- Proibir estas situações (i.e. dar erro). Não é realmente solução... O C++ não usa isto.
- Resolver os conflitos por seleção explícita, ou seja, na classe derivada o programador tem de escolher explicitamente qual dos membros quer herdar. O C++ permite que isto seja feito através da redefinição do membro no caso de se tratar duma função, mas não é obrigatório fazer isto.
- Formar a união disjunta de todos os membros herdados, associado um escopo particular a cada classe base interveniente. Desta forma são herdados todos os membros ao mesmo tempo. Depois é usado um operador de resolução de escopo "::" para distinguir entre membros diferentes que, porventura tenham nomes repetidos. O C++ funciona muito desta maneira.

Em C++, uma classe herda todo o conteúdo das suas classes base. Inclusivamente, se existirem classes base repetidas então a herança é repetida, como no seguinte exemplo, onde a classe C herda todos os membros das classes A e B, e onde todos os membros de L aparecem duplicados.

```
class L { public: void f() ; ...} ;
class A : public L {...} ;
class B : public L {...} ;
class C : public A, public B {...} ;

C c ;
```

Se quisermos chamar a função f, podemos chamar `c.A::L::f()` ou `c.B::L::f()` e temos dois fs à nossa disposição. A chamada simples `c.f()` é ambígua e gera um erro.

No entanto é possível evitar essa repetição usando **classes virtuais**, como neste exemplo onde a classe C também herda todos os membros das classes A e B, mas onde agora os elementos de L são herdados apenas uma vez pois a classe L é virtual.

```
class L { public: void f() ; ...} ;
class A : virtual public L {...} ;
class B : virtual public L {...} ;
class C : public A, public B {...};
```

Agora a chamada `c.f()` já não é ambígua.

Polimorfismo

Os programadores gostam de escrever código geral que possa ser aplicado a vários tipos. É penoso, e causador de erros, ter de reescrever um algoritmo com ligeiras variações só porque surgiu a necessidade de o aplicar a um tipo de dados diferente.

Uma **função polimórfica** é uma função que pode ser aplicada a argumentos de vários tipos. A nossa conhecida função `len` em ML é polimórfica pois aplica-se a listas de qualquer tipo:

```
len : 'a list -> int
```

Um **tipo polimórfico** é uma tipo cujas operações se aplicam a valores de mais do que um tipo. Em ML o tipo da listas `'a list` é polimórfico. Em Java o tipo `Vector<E>` também é.

Uma **variável polimórfica** é uma variável mutável que pode conter valores de tipos diferentes. Em Java, uma variável de tipo `Animal` pode referir qualquer objeto cujo tipo seja subtipo de `Animal`. Em C uma variável de tipo `void *` pode guardar qualquer apontador.

Entidades que não sejam polimórficas dizem-se **monomórficas**.

Muitas das linguagens com tipificação estática modernas suportam polimorfismo. Todas as linguagens com tipificação dinâmica suportam polimorfismo de forma inerente.

Variedades de polimorfismo

O seguinte diagrama identifica as variedades e sub-variedades de polimorfismo de funções em linguagens com tipificação estática, de acordo com Cardelli:

- Polimorfismo
  - Universal
    - Paramétrico
    - Inclusão (ou subtipo)

- Ad hoc
  - Overloading
  - Coerção

Polimorfismo universal - A função trabalha uniformemente sobre uma diversidade de tipos. A implementação é única e o mesmo código consegue lidar com todos os tipos considerados. O número de tipos considerados é infinito e todos eles partilham a mesma estrutura.

Polimorfismo ad hoc - A função trabalha sobre uma diversidade de tipos, mas não de forma uniforme visto que para cada tipo o comportamento pode ser diferente. Na realidade são escritas múltiplas implementações, uma para cada tipo considerado. O número de tipos considerados é finito e geralmente eles não partilham a mesma estrutura.

Polimorfismo paramétrico - É uma forma de polimorfismo universal onde a função polimórfica tem um parâmetro de tipo implícito ou explicito. Na chamada da função o parâmetro de tipo pode ser ou não inferido. A função `len` em ML é polimórfica paramétrica, sendo `'a` o nome do parâmetro de tipo:

```
len : 'a list -> int

let rec len l =
    match l with
        [] -> 0
      | x::xs -> 1 + len xs
;;
```

A seguinte função em Java é polimórfica paramétrica, sendo `T` o nome do parâmetro de tipo:

```
<T> void fromArrayToCollection(T[] a, Collection<T> c) {
  for (T o : a) {
   c.add(o) ;
}}
```

Polimorfismo de inclusão - É uma forma de polimorfismo universal que resulta imediatamente da noção de subtipo. Uma função que declare aceita argumentos dum dado tipo, digamos `Animal`, também aceita argumentos de subtipos desse tipo, digamos `Cat`. Qualquer linguagens com subtipos suporta polimorfismo de inclusão.

A seguinte função em Java é polimórfica de inclusão:

```
int Weight(Animal a) { ... }
```

Overloading - O mesmo nome de função é usado para denotar diferentes implementações monomórficas. No ponto da chamada usa-se o contexto para descobrir qual das implementações deve ser usada. Portanto esta forma de polimorfismo não é mais do que uma conveniência sintática. Um exemplo: o operador "+" em Java denota três operações distintas, não relacionadas entre si: (1) soma de inteiros; (2) soma de reais; (3) concatenação de strings.

Coerção - Uma coerção é uma conversão automática de tipo. As coerções fazem com que funções essencialmente monomórficas se tornem polimórficas, pois passam a poder ser chamadas com argumentos de diferentes tipos A seguinte função em C foi escrita para ser monomórfica

```
double inc(double d) { return d + 1 ; }
```

mas devido ao facto de em C existir uma coerção de inteiro para double, a função passa a poder ser aplicada tanto a reais como inteiros.

Este tipo de polimorfismo é especialmente importante em C++ devido à capacidade que o programador tem de definir novas coerções na linguagem.

Interoperabilidade de linguagens

Em certos projetos de software, por vezes interessa escrever o código usando mais do que uma linguagem de programação. Isso só é possível se as linguagens suportarem mecanismos de **interoperabilidade**. Para haver interoperabilidade entre as linguagens A e B são necessárias pelo menos duas coisas:

- Cada linguagem têm de permitir a chamada de funções externas escritas na outra linguagem.
- Tem de haver a possibilidade de troca de dados entre as duas linguagens usando formatos convencionados.

Eis alguns cenários simples de perceber:

- Um programa está a ser desenvolvido em C, mas entretanto surge o interesse em usar a partir do C alguma da funcionalidade da imensa plataforma Java.
- Um programa está a ser desenvolvido em Java, mas surge a necessidade de chamar a partir de Java código C que implementa operações de acesso ao sistema.
- Um programa que usa técnicas de inteligência artificial implementa a parte do "raciocínio" em Prolog e implementa a interface gráfica com o utilizador em C++ usando a biblioteca wxWidgets.

Um detalhe curioso é o facto da maioria das linguagens de programação existentes terem uma especial consideração pela linguagem C. Geralmente suportam a chamada de funções externas escritas em C, e também permitem que o seu código seja chamado a partir do C.

Interoperabilidade entre C e C++

Possibilidade 1: Usar o compilador de C++

Se quisermos incorporar módulos escritos em C e módulos em C++ num mesmo programa, uma boa solução consiste em tentar compilar todo o código usando o compilador de C++.

Mesmo assim isso pode obrigar a alterar um pouco o código C, por exemplo a inserir de alguns casts de apontadores e a adicionar alguns protótipos (cabeçalhos de funções). Recorde-se que, como foi dito na aula 18, existem cerca de 20 situações em que código C válido é considerado código C++ inválido, ou se comporta de maneira diferente.

Possibilidade 2: Usar a construção `extern "C"`

Outra hipótese é compilar os módulos escritos em cada linguagem usando o compilador dessa mesma linguagem e no fim ligar os ficheiros objeto obtidos. Mas vimos na aula 19 que o C++ implementa overloading de funções usando *name mangling* o que significa que vão haver problemas de interoperabilidade ao nível do ligador.

O C++ introduziu a construção **`extern "C"`** para resolver o problema. O C++ permite ao programador informar o compilador de C++ que uma ou mais funções devem ser compiladas usando as regras dos compiladores de C, portanto sem usar *name mangling*. Esta solução só se aplica a funções que não sejam overloaded. Eis como se declara uma função:

```
extern "C" void myfun(int x, int y) ;
```

Também é possível declarar múltiplas funções simultaneamente:

```
extern "C" {
  void myfun1(int x, int y) ;
  void myfun2(double x, int y) ;
}
```

e mesmo fazer um include:

```
extern "C" {
  #include "fich.h"
}
```

A construção `extern "C"` pode ser usada de duas formas diferentes:

1. No caso de tratar de C++ a chamar C, usa-se do lado do C++ a declaração `extern "C"` para se ganhar acesso às funções escritas em C. Esta solução é elegante pois o código C não precisa de ser alterado, sendo apenas o código cliente que usa a declaração `extern "C"`.
2. No caso de tratar C a chama C++, é também do lado do C++ que se tem de atuar. Do lado do C++ é preciso declarar como especiais as funções C++ que se destinam a ser chamadas a partir do lado do C. Esta solução quebra a modularidade pois é o código do "fornecedor" que tem de se preocupar em estar na forma certa para ser usado por um cliente especial. Felizmente as funções C++ declaradas `extern "C"` podem continuar a ser chamadas do lado do C++.

Exemplo

No seguinte exemplo temos um módulo escrito em C e outro em C++. A função `main` está escrita em C++ e chama a função C `cfun` que, por sua vez, chama a função C++ `cppfun`.

Ficheiro a.h

```
#ifndef _A_
#define _A_

void cfun(void) ;

#endif
```

Ficheiro a.c

```
#include <stdio.h>
#include "a.h"
#include "b.h"

void cfun(void) {
    printf("cfun here\n") ;
    cppfun() ;     // C calling C++
}
```

Ficheiro b.h

O símbolo `__cplusplus` só está definido nos compiladores de C++. Assim é possível saber se o ficheiro ".h" está a ser incluído do lado do C++ ou do lado do C. Só no caso de estar a ser incluído do lado do C++ é que se adiciona a declaração `extern "C"`.

Repare que o quando há funções C++ a serem chamadas do lado do C, é o lado do C++ que tem a responsabilidade de se preparar para isso.

```
#ifndef _B_
#define _B_

#ifdef __cplusplus
 extern "C" {
#endif

void cppfun() ;

#ifdef __cplusplus
 }
#endif

#endif
```

Ficheiro b.cpp

```
#include <iostream>
#include "b.h"

void cppfun() {
    std::cout << "CPPFUN HERE" << std::endl ;
}
```

Ficheiro main.cpp

A função `main` tem de ser compilada pelo compilador de C++ para que a inicialização dos membros de dados estáticos do C++ aconteça. Assim um programa misto C/C++ começa sempre a correr do lado do C++, embora a partir daí não haja quaisquer restrições quanto a quem chama quem.

```
extern "C" {
    #include "a.h"
}
#include "b.h"

int main() {
    cfun() ;       // C++ calling C
    return 0 ;
}
```

Compilação e ligação

O processo de ligação tem de ser controlado pelo C++ para que fiquem disponíveis as bibliotecas de que o código escrito em C++ precisa.

```
gcc -c a.c
g++ -c b.cpp main.cpp
g++ -o main a.o b.o main.o
```

Mais detalhes

Para mais detalhes, por exemplo sobre como aceder a objetos do C++ a partir do lado do C, veja o seguinte documento.

Interoperabilidade entre C/C++ e Java

Há duas técnicas que costumam ser usadas para fazer código Java comunicar com com código C/C++:

- **Sockets**: Usam-se os mecanismos de comunicação suportados pelo sistema operativo para permitir um processo escrito em Java comunicar com um processo escrito em C/C++.
- **Java Native Interface** - Trata-se duma biblioteca e dum conjunto de regras que permitem que código Java compilado a a correr na JVM invoque código nativo escrito em C/C++ e vice-versa.

Java Native Interface (JNI)

Nesta cadeira vamos estudar apenas a técnica da **JNI**. Trata-se duma boa solução para o problema da integração de código Java com código C/C++.

A JNI é complexa e exige-se algum esforço para dominar todos os detalhes. No entanto o seu estudo é um bom investimento de tempo.

Uma das melhores fontes de documentação sobre a JNI é o seguinte livro aqui disponível em html para consulta imediata.

A JNI prevê que um programa misto Java-C/C++ pode começar a correr tanto do lado do Java como do lado do C/C++. Por limitações de tempo vamos só estudar o caso em que **o programa começa a correr do lado do C/C++**. Em todo o caso, em muitos casos o que é relevante é que a partir do momento em que o programa começa a correr, qualquer um dos lados pode chamar o outro lado sem limitações. Atenção que a maior parte dos tutoriais ensinam a usar a JNI com o programa a começar a correr do lado do Java; aqui fazemos ao contrário.

Repare que fazer o programa começar a correr do lado do C/C++, implica embeber uma instância da máquina virtual do Java dentro do programa em C/C++. Para criar a máquina virtual chama-se a função de biblioteca `JNI_CreateJavaVM`. Esta é apenas uma das muitas funções que se encontram disponíveis na biblioteca da JNI (que no Linux está guardada no ficheiro "jre/lib/i386/client/libjvm.so"). As funções da JNI permitem ao C/C++ aceder a todos os aspetos do ambiente de execução do Java, por exemplo:

- Criar uma nova classe;
- Procurar uma classe por nome;
- Numa classe procurar variáveis estáticas por nome e ler ou modificar o seu conteúdo;
- Numa classe procurar métodos estáticos por assinatura e chamar esses métodos;
- Criar um objeto a partir duma classe;
- Determinar qual a classe dum objeto;
- Numa classe procurar por nome uma variável de instância e ler ou modificar o seu conteúdo para um objeto particular;
- Numa classe procurar por nome e tipo um método de instância e chamar esse método para um objeto particular;
- Intercetar as exceções que ocorrem do lado do Java e tratá-las do lado do C;
- Gerar exceções para serem tratadas do lado do Java;
- Avisar o gestor de memória do Java que determinados objetos estão em uso do lado do C, e que portanto não devem ser apagados;
- Registar uma função C como sendo a implementação dum determinado método nativo numa classe;
- Manipular os arrays do Java.

Quando se procura uma variável num objeto ou numa classe é preciso indicar o nome e o tipo da variável. Chama-se **tipo JNI** a um tipo Java escrito no formato compacto da JNI: A sintaxe dos tipos JNI é dada pela seguinte gramática:

```
<type> ::=
          Z                    // boolean
        | B                    // byte
        | C                    // char
        | S                    // short
        | I                    // int
        | J                    // long
        | F                    // float
        | D                    // double
        | L<classname>;        // object of class "classname"
        | [<type>              // array of type "type"
        | (<typeseq>)<type>    // method returning "type"following table
        | (<typeseq>)V         // void method
    <classname> ::=
          <name>
        | <name>/<classname>
    <typeseq> ::=
          // can be empty
        | <typeseq>
```

Por exemplo, uma variável inteira tem o tipo JNI '**I**', um array simples de inteiros tem o tipo JNI '**[I**', um arrays a duas dimensões de strings tem o tipo JNI '**[[Ljava/lang/String;**'.

Quando se procura um método num object ou numa classe também é preciso indicar o nome e o tipo JNI do método. Por exemplo, o tipo JNI do método newInstance da classes java.lang.reflect.Array é '**(Ljava/lang/Class;I)Ljava/lang/Object;**': como pode ver este método tem dois argumentos, de tipo Class e de tipo int, e o resultado é de tipo Object.

Exemplo

O seguinte exemplo constitui um programa misto escrito parcialmente em C e parcialmente em Java. Poderá observar que o esforço desenvolvido do lado do C é enorme.

Ficheiro JniTest.java

Este é o lado Java da aplicação. É constituído por uma pequena classe apenas.

Eis uma descrição breve dos métodos:

- **Método javaMethod1** - Este método será chamada a partir do lado C, só para exemplificar;
- **Método javaMethod2** - Este método será também chamada a partir do lado C, mas depois ele mesmo chamará o C a partir do lado do Java através dum método nativo;
- **Método NativePrint** - Método nativo, implementado do lado do C.

```
public class JniTest {
    public static void javaMethod1(String str) {
        System.out.println(str) ;
    }
    public static void javaMethod2(String str) {
        NativePrint(str) ;          // Java calls C
    }
    public native static void NativePrint(String str) ;
}
```

Ficheiro JniTest.c

Aqui encontra-se o lado C da aplicação.

Eis uma descrição breve das principais funções:

- **Função JNIStart** - Criação da máquina virtual do Java (dentro do C) e inicialização de mais algumas variáveis.
- **Função JNITest1** - Chamada dum método de instância dum object de biblioteca: método `System.out.println(String)`.
- **Função JNITest2** - Chamada dum método estático duma classe do utilizador: método `JniTest.javaMethod1(String)`.
- **Função JNIInstallNativeMethod** - Instalação dum método nativo numa classe do utilizador: método `JniTest.NativePrint(String)`.
- **Função JNITest3** - Chamada dum método `JniTest.javaMethod2(String)`, que por sua vez chama o lado do C usando o método nativo `JniTest.NativePrint(String)`.
- **Função JNIStop** - Destruição da máquina virtual do Java.

O programa é apresentado partido em blocos para possibilitar a introdução de algumas explicações pelo meio.

Vamos começar. A parte inicial do ficheiro faz as inclusões necessárias (repare no include do ficheiro "jni.h"), define diversas variáveis globais, macros e funções auxiliares.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include "jni.h"

#define USER_CLASSPATH "." /* where Prog.class is */
```

```
/* Some global variables */

JNIEnv *env ;
JavaVM *jvm = NULL ;
jclass systemClass, classClass, stringClass ;
jclass booleanObjClass, byteObjClass, charObjClass, shortObjClass ;
jclass intObjClass, longObjClass, floatObjClass, doubleObjClass ;

/* My JNI macros */

#define JIsInstanceOf(o,c)          (*env)->IsInstanceOf(env, o, c)
#define JFindClass(n)               (*env)->FindClass(env, n)
#define JGetObjectClass(o)          (*env)->GetObjectClass(env, o)
#define JIsSameObject(o1,o2)        (*env)->IsSameObject(env, o1, o2)
#define JNewObjectA(c,id,args)      (*env)->NewObjectA(env, c, id, args)

#define JExceptionOccurred()        (*env)->ExceptionOccurred(env)
#define JExceptionClear()           (*env)->ExceptionClear(env)
#define JExceptionDescribe()        (*env)->ExceptionDescribe(env)

#define JGetMethodID(c,n,s)         (*env)->GetMethodID(env, c, n, s)
#define JGetStaticMethodID(c,n,s)   (*env)->GetStaticMethodID(env, c, n, s)
#define JCallMethod(k,c,id)         (*env)->Call##k##Method(env, c, id)
#define JCallMethodA(k,w,o,id,args) ((w)?(*env)->CallStatic##k##MethodA(env, o, id,
args) \
                                        :(*env)->Call##k##MethodA(env, o, id, args))

#define JGetFieldID(c,n,s)          (*env)->GetFieldID(env, c, n, s)
#define JGetStaticFieldID(c,n,s)    (*env)->GetStaticFieldID(env, c, n, s)
#define JGetField(k,w,o,id)         ((w)?(*env)->GetStatic##k##Field(env, o, id) \
                                        :(*env)->Get##k##Field(env, o, id))
#define JSetField(k,w,o,id,args)    ((w)?(*env)->SetStatic##k##Field(env, o, id, args) \
                                        :(*env)->Set##k##Field(env, o, id, args))

#define JGetStringUTFChars(o)       (*env)->GetStringUTFChars(env, o, NULL)
#define JReleaseStringUTFChars(o,str) (*env)->ReleaseStringUTFChars(env, o, str)
#define JNewStringUTF(n)            (*env)->NewStringUTF(env, n)

#define JNewArray(k,size)           (*env)->New##k##Array(env, size)
#define JNewObjectArray(size,c)     (*env)->NewObjectArray(env, size, c, NULL)
#define JArrayGet(k,a,i,b)          (*env)->Get##k##ArrayRegion(env, a, i, 1,
cVoidPt(b))
#define JObjectArrayGet(a,i,b)      ((*b) = (*env)->GetObjectArrayElement(env, a, i))
#define JArraySet(k,a,i,vo,vn)      if( vo == NULL || *(vn) != *(vo) ) \
                                        (*env)->Set##k##ArrayRegion(env, a, i, 1,
cVoidPt(vn))
#define JObjectArraySet(a,i,vo,vn)  if( vo == NULL || !JIsSameObject(*(vn), *(vo)) ) \
                                        (*env)->SetObjectArrayElement(env, a, i, *(vn))
#define JGetArrayLength(a)          (*env)->GetArrayLength(env, a)

#define DeleteLocalRef(lref)        (*env)->DeleteLocalRef(env, lref)

void Error(char *mesg)
{
    fprintf(stderr, "%s\n", mesg) ;
    exit(1) ;
}

void CheckException(void)
{
    if( JExceptionOccurred() ) {
        JExceptionDescribe() ;
        exit(1) ;
    }
}

jclass FindClass(char *name)
{
    jclass cls ;
```

```
    if( (cls = JFindClass(name)) != NULL ) {
        return cls ;
    }
    Error("Cannot access java class") ;
}

jstring NewJString(char *name)
{
    jstring jstr ;
    if( (jstr = JNewStringUTF(name)) == NULL )
        Error("Could not create java string") ;
    return jstr ;
}
```

A função JNIStart cria a máquina virtual do Java e pede à JNI as referências de algumas classes muito usadas no Java, apenas para facilitar a utilização posterior dessas classes do lado do C.

```
void JNIStart(void)
{
    JavaVMInitArgs vm_args ;
    JavaVMOption options[1] ;
    JavaVM *j ;
    int res ;

    if( jvm != NULL )
        Error("Java interface already running") ;

     options[0].optionString =                // Options passed to the JVM
         "-Djava.class.path=" USER_CLASSPATH ;
    vm_args.version = JNI_VERSION_1_2 ;
    vm_args.options = options ;
    vm_args.nOptions = 1 ;
    vm_args.ignoreUnrecognized = JNI_TRUE ;   // Ignore unknown options
    res = JNI_CreateJavaVM(&j, (void**)&env, &vm_args) ;
    if( res != 0 )
        Error("Could not create Java Virtual Machine") ;

    systemClass = FindClass("java/lang/System") ;
    classClass = FindClass("java/lang/Class") ;
    stringClass = FindClass("java/lang/String") ;
    booleanObjClass = FindClass("java/lang/Boolean") ;
    byteObjClass = FindClass("java/lang/Byte") ;
    charObjClass = FindClass("java/lang/Character") ;
    shortObjClass = FindClass("java/lang/Short") ;
    intObjClass = FindClass("java/lang/Integer") ;
    longObjClass = FindClass("java/lang/Long") ;
    floatObjClass = FindClass("java/lang/Float") ;
    doubleObjClass = FindClass("java/lang/Double") ;

    jvm = j ;   /* Now is oficial: the jvm is active */

    if( JExceptionOccurred() )
        JExceptionDescribe() ;
}
```

A função JNIStop destrói a máquina virtual do Java para reaproveitar a memória RAM ocupada por ela. Sempre são algumas largas dezenas de MB reaproveitadas.

```
void JNIStop()
{
    if( jvm == NULL )
            Error("Java interface is not running") ;
    (*jvm)->DestroyJavaVM(jvm) ;
    jvm = NULL ;
}
```

A partir do lado do C, a função JNITest1 manda o Java executar `System.out.println("Hello world!")`. Leia o código devagar. Verá que percebe.

```
void JNITest1(void)
{
    jfieldID fid = JGetStaticFieldID(systemClass, "out", "Ljava/io/PrintStream;") ;
```

```
    jobject obj = JGetField(Object, true, systemClass, fid) ;
    jclass cls = JGetObjectClass(obj) ;
    jmethodID mid = JGetMethodID(cls, "println", "(Ljava/lang/String;)V") ;
    jstring str = NewJString("Hello world!") ;
    jvalue args[10] ;
    args[0].l = str ;
    JCallMethodA(Void, false, obj, mid, args) ;
    CheckException() ;
}
```

A partir do lado do C, a função JNITest2 manda o Java executar `JniTest.javaMethod1("Hello world again!")`. O interesse deste caso é o facto da classe `JniTest` ter sido escrita pelo programador.

```
void JNITest2(void)
{
    jclass cls = FindClass("JniTest") ;
    jmethodID mid = JGetStaticMethodID(cls, "javaMethod1", "(Ljava/lang/String;)V") ;
    jstring str = NewJString("Hello world again!") ;
    jvalue args[10] ;
    args[0].l = str ;
    JCallMethodA(Void, true, cls, mid, args) ;
    CheckException() ;
}
```

Veja como se define e instala um método nativo em Java. A implementação em C do método nativo NativePrint obedece a determinadas regras, como pode observar. A instalação da implementação do método na classe apropriada é feita usando a operação `RegisterNatives` da JNI.

```
void JNICALL NativePrint(JNIEnv *env, jobject self, jstring obj)
{
    if( JIsInstanceOf(obj, stringClass) ) {
        char *jstr ;
        if( (jstr = (char *)JGetStringUTFChars(obj)) == NULL )
            Error("Couldn't access the contents of a java string") ;
        printf("%s\n", jstr) ;
        JReleaseStringUTFChars(obj, jstr) ;
    }
}

void JNIInstallNativeMethod(void)
{
    jclass cls = FindClass("JniTest") ;
    JNINativeMethod nm ;
    nm.name = "NativePrint" ;
    nm.signature = "(Ljava/lang/String;)V" ;
    nm.fnPtr = NativePrint ;
    (*env)->RegisterNatives(env, cls, &nm, 1) ;
}
```

A partir do lado do C, a função JNITest3 manda o Java executar `JniTest.javaMethod2("Hello world again and again!")`. Se olhar para o código da classe `JniTest` poderá observar que, por sua vez, este método chama o C a partir do lado do Java, via método nativo.

```
void JNITest3(void)
{
    jclass cls = FindClass("JniTest") ;
    jmethodID mid = JGetStaticMethodID(cls, "javaMethod2", "(Ljava/lang/String;)V") ;
    jstring str = NewJString("Hello world again and again!") ;
    jvalue args[10] ;
    args[0].l = str ;
    JCallMethodA(Void, true, cls, mid, args) ;
    CheckException() ;
}
```

A função `main`.

```
int main(void) {
    JNIStart() ;
    JNITest1() ;
    JNITest2() ;
    JNIInstallNativeMethod() ;
    JNITest3() ;
```

```
    JNIStop() ;
    return 0 ;
}
```

Compilar e executar

Para compilar o ficheiro C é necessário indicar onde estão os ficheiros de include da JNI (usa-se a opção -I do compilador) e também que é preciso indicar a biblioteca jvmlib.so (usa-se -ljvm):

```
gcc -I/usr/local/lib/jdk1.6.0_02/include -I/usr/local/lib/jdk1.6.0_02/include/linux -
L/usr/local/lib/jdk1.6.0_02/jre/lib/i386/client -ljvm -o JniTest JniTest.c
```

Para compilar a classe em Java, faz-se como de costume:

```
javac JniTest.java
```

Assumindo que o sistema não está devidamente configurado, para correr o programa é preciso indicar onde se encontram as bibliotecas dinâmicas da JVM:

```
LD_LIBRARY_PATH=/usr/local/lib/jdk1.6.0_02/jre/lib/i386/client ./JniTest
```

Assumimos que tudo se passa do contexto do Linux. No Windows não seria muito diferente.
