Linguagens e Ambientes de Programação (2010/2011)

Teórica 01 (14/Fev/2011)

Apresentação da disciplina.

Discussão introdutória sobre alguns aspetos das Linguagens de Programação.

Nota: A primeira aula teórica de LAP foi dada com base nos documentos "Teórica 01" e "Teórica 02".

Sequência da apresentação da matéria de LAP, ao longo do semestre

- Introdução à linguagem ML (dialeto Caml) e à sua utilização.
- Plataformas computacionais. Implementação de linguagens de programação. Ligações, ambientes, âmbitos.
- Introdução à linguagem C e à sua utilização.
- Modelos de execução para diversos tipos de linguagens de programação.
- Introdução à linguagem C++ e à sua utilização, comparando com Java.
- Sistemas de tipos. Polimorfismo.
- Interoperabilidade entre C, C++ e Java.
- Uma linguagem de scripting: JavaScript.
- Ambientes de programação.

Dimensões do estudo das Linguagens de Programação

- **Sintaxe** - Estuda as construções válidas da linguagem, independentemente do significado dessas construções. Os tópicos mais importantes são a especificação formal da sintaxe e o desenvolvimento de técnicas de reconhecimento eficientes. Por uma questão prática, a sintaxe costuma ser tratada a dois níveis:
    - **Nível independente de contexto** - Considera apenas a estrutura dos termos da linguagem. Especifica-se usando gramáticas independentes do contexto ou outras técnicas.
    - **Nível contextual** - Considera também as restrições de contexto, como por exemplo "um identificador só pode ser usado depois de declarado". Especifica-se usando gramáticas de atributos ou outras técnicas.
- **Semântica** - Estuda o significado das construções duma linguagem ou seja os efeitos da execução dos programas. Por vezes descreve-se a semântica informalmente, mas existem muitas técnicas formais em uso: gramáticas de atributos, semântica operacional estruturada, semântica matemática, semântica axiomática, etc.
- **Pragmática** - Estuda tudo o que se relaciona com o uso prático das linguagens incluindo: como usar bem uma linguagem; análise critica das vantagens e desvantagens práticas de cada mecanismo; como escolher a melhor linguagem para resolver um dado problema?

A cadeira LPA concentra-se na pragmática e, de forma subsidiária, na semântica informal.

Porque há tantas Linguagens de Programação?

Realmente existem milhares de linguagens de programação diferentes. Eis algumas razões:

- **Juventude da Informática** - A Informática é uma disciplina recente, e estão constantemente a ser descobertas novas e melhores maneiras de fazer as coisas.
- **Domínios de aplicação** - A maioria das linguagens são de uso geral, mas mesmo assim cada uma delas tende a estar mais bem adaptada à resolução de certos problemas.
    - O Lisp, ML e Prolog são ótimos para manipulação simbólica e para aplicações de IA.
    - O C é ótimo para programação de de sistemas.
    - O Java e C# são ótimos para desenvolver aplicações com interface gráfico sobre a WEB.
    - O C++ é excelente para programar jogos de vídeo sofisticados e rápidos.
    - O CDuce é excelente para processar documentos XML.
- **Preferência pessoal** - A diversidade de preferências pessoais contribui para a diversidade de linguagens.
    - Há quem prefira pensar recursivamente mas outros preferem usar iteração.
    - Há quem esteja habituado aos apontadores do C e C++ e goste deles mas outros preferem a desreferenciação implícita do Java e do ML.
    - Há quem aprecie a simplicidade, compacidade e enorme flexibilidade do C, mas outros preferem usar linguagens com sistemas de tipos mais seguros como o Java ou ML.
    - Há quem considere que o C++ é o melhor compromisso entre a possibilidade de programar com classes e criar programas muito eficientes, mas outros preferem a maior simplicidade e consistência da linguagem Java, não dispensando também a sua extensa biblioteca de classes.

Qualidade duma Linguagem de Programação

Eis alguns critérios de avaliação de qualidade:

- **Legibilidade** - Facilidade em, através do exame de um programa, seguir a sua lógica e descobrir a presença de erros. Muito importante para a manutenção dos programas.
  Para isso são importantes os seguintes fatores:
    - **Simplicidade** - Ajuda a conhecer bem a linguagem, em todos os seus detalhes.
    - **Ortogonalidade** - Este termo descreve o fenómeno dum conjunto de primitivas pode ser combinado dum número de formas conhecido, sendo legítimas todas as combinações imagináveis. Pode ser uma forma habilidosa de combinar simplicidade com expressividade, desde que o número de primitivas e de combinações sejam pequenos.
    - **Estruturas de Controlo** - Os seus efeitos devem ser claros e fáceis de descrever. Cuidado com o goto.
    - **Estruturas de dados** - Devem ser suficientemente claras para ajudar a perceber as intenções contidas nos programas.
    - **Sintaxe** - Para diferentes significados usar formas sintáticas diferentes.
- **Redigibilidade** - Possibilidade de expressar os problemas de uma forma natural, sem que a atenção do programador seja desviada por detalhes ou "truques" da linguagem.
  Para isso são importantes os seguintes fatores:
    - **Simplicidade** - Assim há menos hipóteses de cometer erros.
    - **Expressividade** - Convém que haja suporte natural (direto) para o estilo de programação usado.
    - **Ortogonalidade** - Ajuda a que não se perca tempo a pensar em exceções às regras gerais da linguagem.
    - **Estruturas de Controlo** - Devem ser suficientemente diversas para permitir expressividade.
    - **Estruturas de dados** - Devem ser suficientemente diversas para facilitar a expressividade. Mas cuidado com determinados mecanismos, como os apontadores.
    - **Suporte para abstração** - Ajuda a dominar a complexidade dos problemas pois permite esquecer os detalhes que não são importantes em cada contexto. Os humanos não conseguem abarcar ao mesmo tempo todos os detalhes duma entidade complexa.
    - **Sintaxe** - Não deve haver restrições desnecessárias, e.g. comprimento dos identificadores.

- **Segurança** - Possibilidade de escrever programas que deem garantias de que atingem o efeito desejado, ou seja que obedecem à sua especificação (em todas as situações).
  Para isso são importantes os seguintes fatores:
  - **Verificação de tipos** - Estática ou dinâmica, há vantagem e desvantagens em cada um delas. Sistemas de tipos estático detetam todas as incompatibilidades de tipo em tempo de compilação, o que é bom, mas também excluem algumas situações legítimas. Sistemas de tipos dinâmicos descobrem os erros de tipo só em tempo de execução, mas podem funcionar bem se os programas forem desenvolvidos seguindo uma metodologia de unit tests.
  - **Exceções** - Tratamento de erros e de situações excecionais ajuda a que o programa funcione de acordo com o esperado em TODAS as situações.
  - **Aliasing (sinonímia)** - Fator negativo. Permite flexibilidade mas pode ser perigoso deixar uma mesma entidade ser conhecida por dois nomes diferentes.
- **Eficiência** - Atualmente a eficiência já não é mais medida apenas com base velocidade de execução dos programas e na economia no uso da memória. Considera-se também o esforço necessário para produzir os programas e o esforço necessário para os manter.
- **Rigor** - A definição da linguagem deve ser rigorosa, para não levantar dúvidas aos implementadores e aos utilizadores.

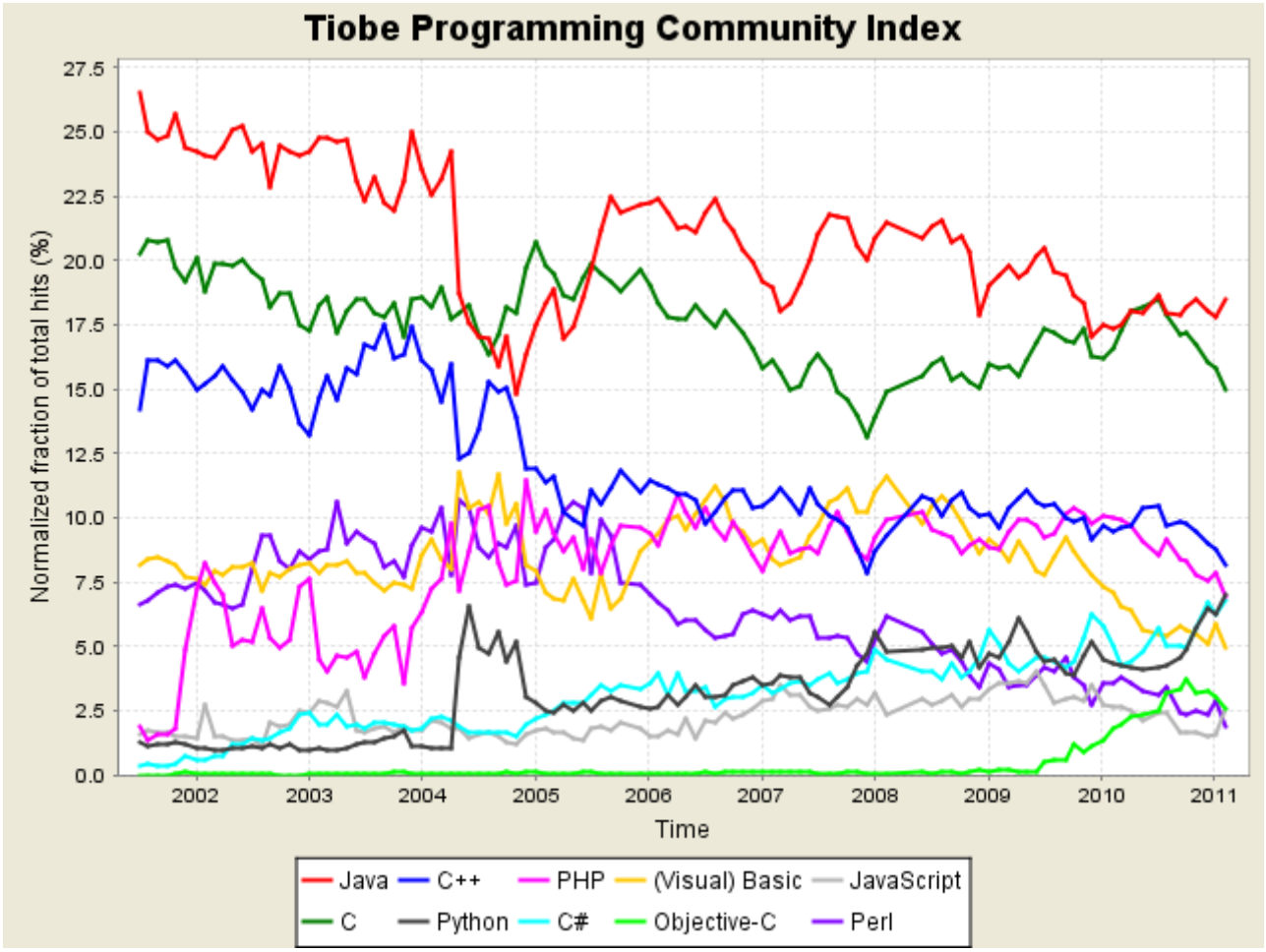
Popularidade das Linguagens de Programação atuais

O site Tiobe calcula mensalmente um índice de popularidade de Linguagens de Programação.Citação: *"The ratings are based on the world-wide availability of skilled engineers, courses and third party vendors. The popular search engines Google, MSN, Yahoo!, and YouTube are used to calculate the ratings."*

Popularidade em Fev/2011

Position Feb 2011	Position Feb 2010	Delta in Position	Programming Language	Ratings Feb 2011	Delta Feb 2010	Status
1	1	=	Java	18.482%	+1.13%	A
2	2	=	C	14.986%	-1.62%	A
3	4	↑	C++	8.187%	-1.26%	A
4	7	↑↑↑	Python	7.038%	+2.72%	A
5	3	↓↓	PHP	6.973%	-3.03%	A
6	6	=	C#	6.809%	+1.79%	A
7	5	↓↓	(Visual) Basic	4.924%	-2.13%	A
8	12	↑↑↑↑	Objective-C	2.571%	+0.79%	A
9	10	↑	JavaScript	2.558%	-0.08%	A
10	8	↓↓	Perl	1.907%	-1.69%	A
11	11	=	Ruby	1.615%	-0.82%	A
12	-	=	Assembly*	1.269%	-	A-
13	9	↓↓↓↓	Delphi	1.060%	-1.60%	A
14	19	↑↑↑↑↑	Lisp	0.956%	+0.39%	A
15	37	↑↑↑↑↑↑↑↑↑↑	NXT-G	0.849%	+0.58%	A--
16	30	↑↑↑↑↑↑↑↑↑↑	Ada	0.805%	+0.44%	A--
17	17	=	Pascal	0.735%	+0.13%	A
18	21	↑↑↑	Lua	0.714%	+0.21%	A--
19	13	↓↓↓↓↓↓	Go	0.707%	-1.07%	A--
20	32	↑↑↑↑↑↑↑↑↑↑	RPG (OS/400)	0.626%	+0.27%	A--

Evolução da popularidade ao longo dos anos



História: Linguagem máquina

No final da década de 40, não existiam alternativas ao uso de linguagem máquina.

Características

- Instruções especificadas por meio de códigos numéricos, em binário.
- Utilização direta de endereços absolutos nos programas.

Discussão

- Programas difíceis de escrever e quase impossíveis de ler.
- Grande facilidade em cometer erros.
- Os programas só funcionam no tipo específico de hardware para que foram escritos.
- Dificuldade em inserir ou remover instruções nos programas, por causa dos endereços absolutos (a instrução **nop** ajuda a minorar um pouco este problema).

História: Assemblers

Começaram a surgir no início dos anos 50.

Características

- Começaram a surgir no início dos anos 50.

- Permitem atribuir nomes a códigos de operação (mnemónicas), a localizações de memória (etiquetas) e a constantes.
- Um tradutor - chamado **assembler** - traduz instruções assembler em instruções máquina.
- Um programa pode conter diretivas que não dão origem a código: declaração de constantes, por exemplo.

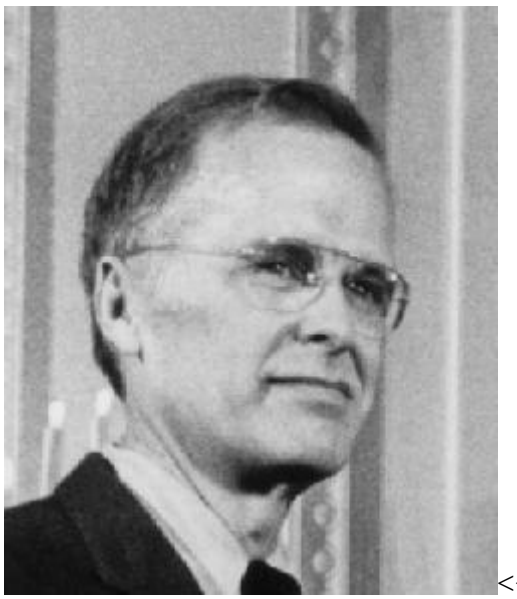
Discussão

- Os aspetos negativos mais dramáticos da linguagem máquina ficam minorados.
- No entanto os programas continuam a ser difíceis de escrever e de ler.
- Os programas também continuam a ter de ser escritos para arquiteturas particulares.
- Mas já se observam algumas "sementes" das futuras linguagens de programação.

Exemplo: Programa "fatorial" escrito para o Pentium

```
.file   "fact.c"
        .section    .rodata
.LC0:   .string "> "
.LC1:   .string "%d"
.LC2:   .string "fact(%d) = %d\n"
.text
.globl main
.type main, @function
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        andl    $-16, %esp
        movl    $0, %eax
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        subl    %eax, %esp
        movl    $.LC0, (%esp)
        call    printf
        leal    -12(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC1, (%esp)
        call    scanf
        movl    $1, -4(%ebp)
        movl    $1, -8(%ebp)
        jmp     .L2
.L3:    movl    -4(%ebp), %eax
        imull   -8(%ebp), %eax
        movl    %eax, -4(%ebp)
        leal    -8(%ebp), %eax
        addl    $1, (%eax)
.L2:    movl    -12(%ebp), %eax
        cmpl    %eax, -8(%ebp)
        jle     .L3
        movl    -12(%ebp), %edx
        movl    -4(%ebp), %eax
        movl    %eax, 8(%esp)
        movl    %edx, 4(%esp)
        movl    $.LC2, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
.size main, .-main
```

História: Fortran


<< John Backus

O Fortran foi anunciado por John Backus da IBM em 1954. A implementação inicial, designada Fortran I, ficou disponível em 1957, com compiladores para IBM 704 e IBM 709.

Prometia

- Eficiência dos programas escritos em assembler. [Quase! O Fortran sempre foi conhecido pela excelente qualidade do código gerado pelos seus compiladores.]
- Simplificar a escrita de programas mediante o uso de notação matemática (Fortran = "FORmula TRANslator"). [Confirmou-se!]
- Eliminar praticamente todos os erros de programação. [Mentira!]

Características

- Instruções de controlo pobres e influenciadas pelas instruções da máquina IBM 704. Necessário recorrer muito à instrução goto.
- Suporte para inteiros, reais e arrays, mas não para registos (records).
- Suporte para variáveis estáticas. Impossível criar novas variáveis em tempo de execução.
- Suporte para sub-rotinas não recursivas.

Discussão

- Depois do Fortran I foram surgindo versões melhoradas: Fortran II em 1958, Fortran IV em 1962, Fortran 77 em 1978 e Fortran 90 em 1990.
- Tornou-se na primeira linguagem de programação popular e, a partir do início dos anos 60, mudou de forma revolucionária a forma como os computadores eram usados.
- Ainda é bastante usada atualmente em aplicações de cálculo numérico!
- Passou a ser possível escrever programas portáveis!

Exemplo: Programa "fatorial" escrito em Fortran

```
      READ 10,I
10    FORMAT(I3)
      J=1
      DO 20 K=1,I
      J=J*K
20    CONTINUE
      PRINT 10,J
      STOP
```

História: Outras linguagem pioneiras


<<

Comité do Algol 60

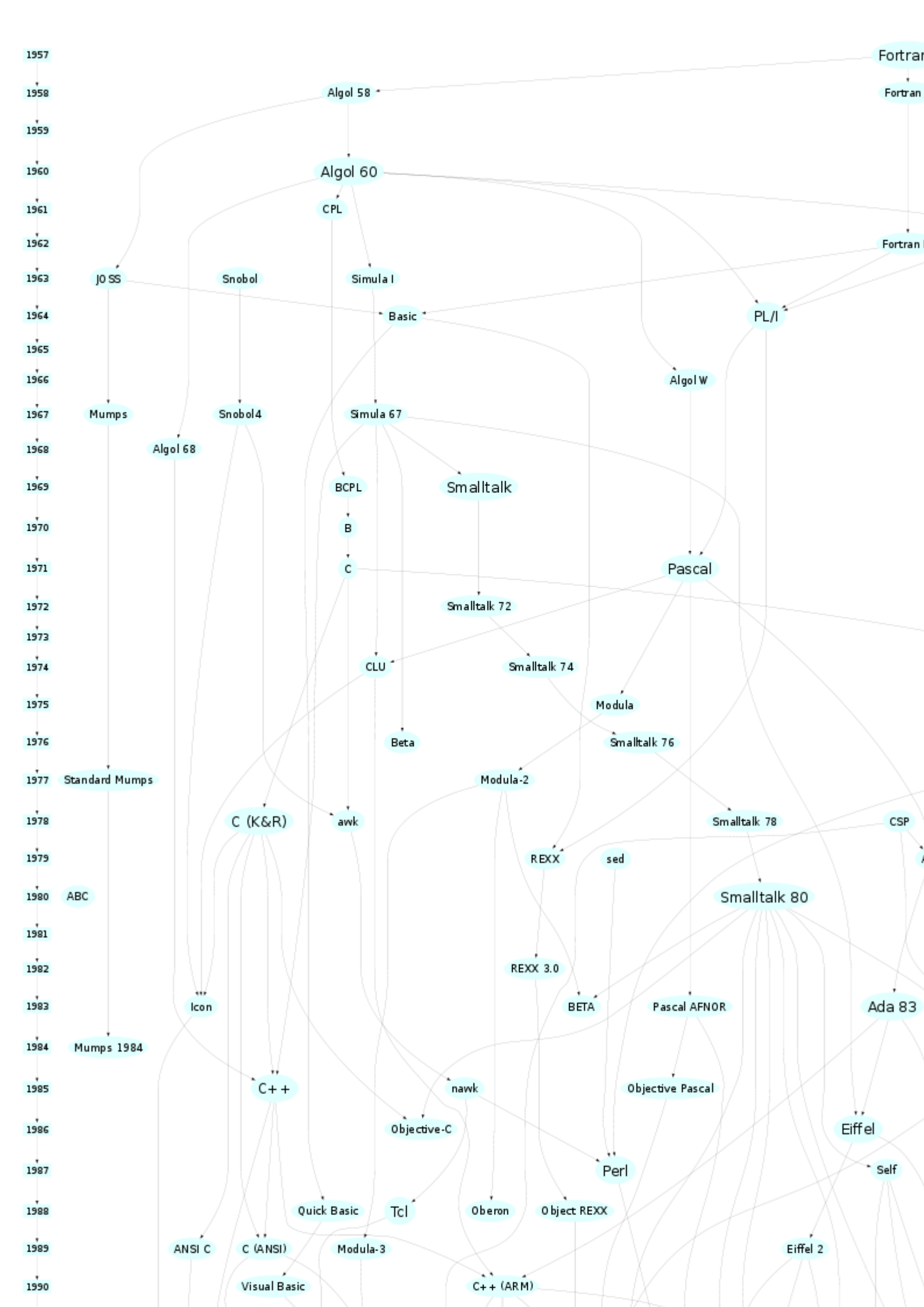
Início dos anos 60.

- **Algol 60** - Primeiro passo em direção à sofisticação das linguagens modernas.
- **Lisp** - Dominou as aplicações de IA durante 25 anos.
- **Cobol** - Linguagem das empresas e do DoD.
- **Basic** - Muito simples de aprender, destinada a não especialistas.

Mais um pouco de história

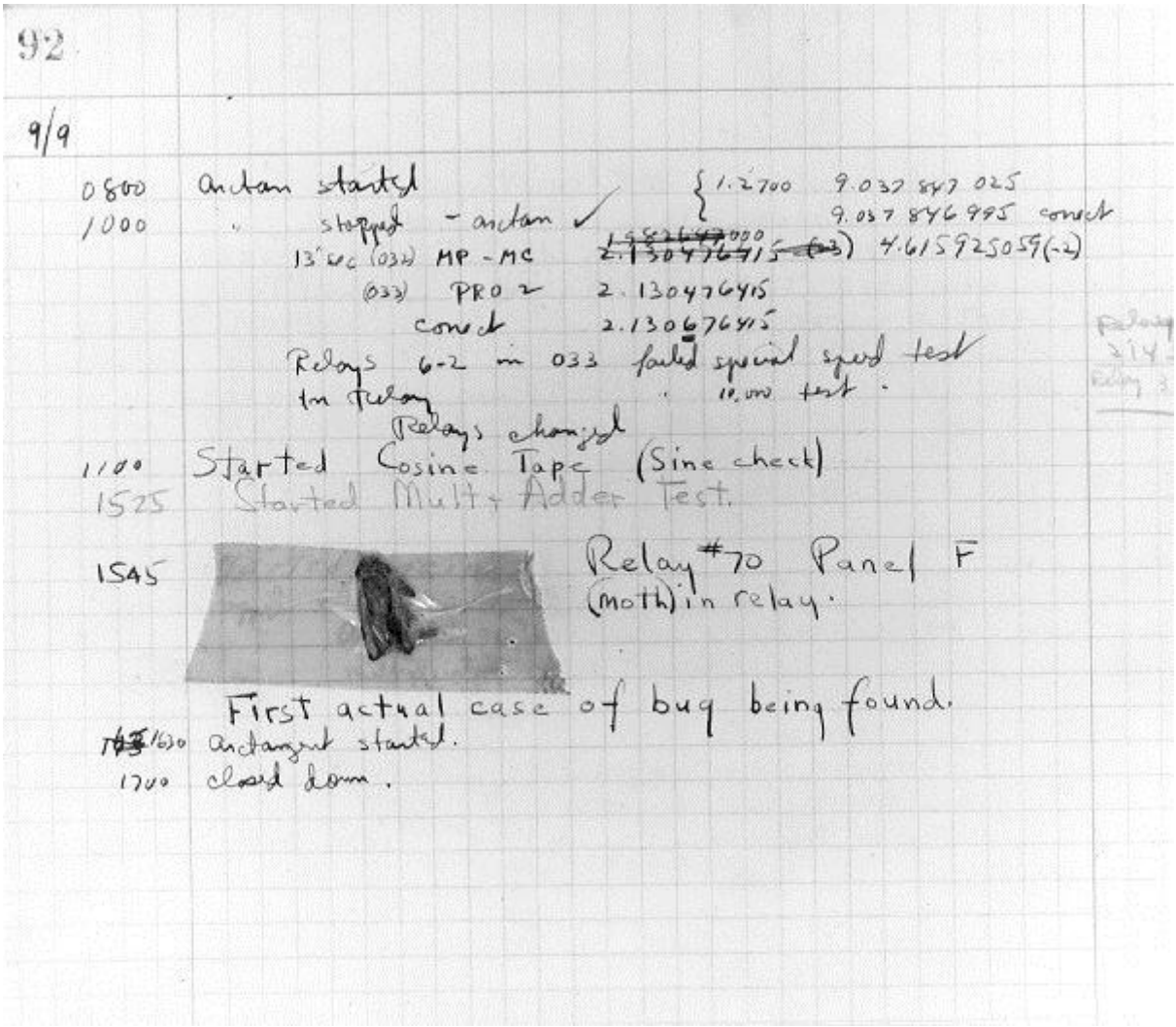
Ver: Key Events in the History of Computing

Algumas linguagens de 1957 até 2007

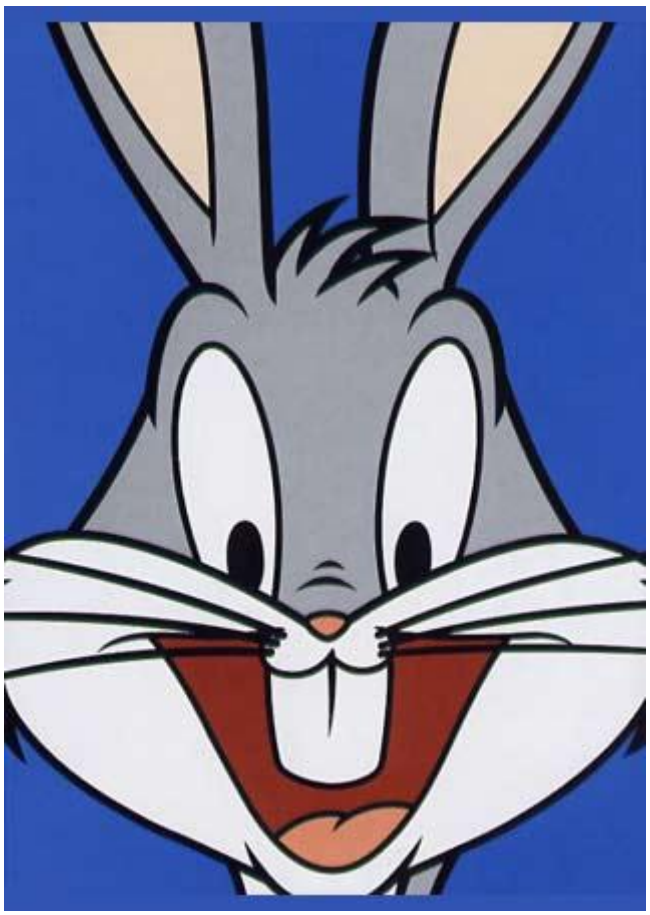

1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
Fortran
Algol 58
Algol 60
CPL
JOSS
Snobol
Simula I
Basic
PL/I
Algol W
Mumps
Snobol4
Simula 67
Algol 68
BCPL
Smalltalk
B
C
Pascal
Smalltalk 72
CLU
Smalltalk 74
Modula
Beta
Smalltalk 76
Standard Mumps
Modula-2
C (K&R)
awk
Smalltalk 78
CSP
REXX
sed
ABC
Smalltalk 80
REXX 3.0
Icon
BETA
Pascal AFNOR
Ada 83
Mumps 1984
C++
nawk
Objective Pascal
Objective-C
Eiffel
Perl
Self
Quick Basic
Tcl
Oberon
Object REXX
ANSI C
C (ANSI)
Modula-3
Eiffel 2
Visual Basic
C++ (ARM)

O primeiro bug :)

Provavelmente foi em 1947 que o termo **bug** foi pela primeira vez associado (com alguma ironia) aos computadores "modernos". Um computador Mark II parou por causa duma traça que ficou presa num comutador mecânico.



Mas a palavra **bug** sempre foi associada a mecanismos defeituosos. Pense no Bugs Bunny, cujo nome significa coelho maluco, coelho "com bugs" ou coelho avariado.



#120

Linguagens e Ambientes de Programação (2010/2011)

Teórica 02 (14/Fev/2011)

Computação, Algoritmos, Programas e Linguagens de Programação. Paradigmas de Programação

Paradigma imperativo versus paradigma funcional. Elementos essenciais da linguagem OCaml.

Avaliação de expressões em OCaml.

Tipos em OCaml. Tipos básicos e tipos estruturados. Inferência de tipos. Funções monomórficas e funções polimórficas. Não há sobreposição.

Atenção: A primeira aula teórica de LAP foi dada com base nos documentos "Teórica 01" e "Teórica 02".

Computação

Computação

- Computação significa: processamento automático de informação.
- É a atividade realizada pelos computadores.
- O objetivo da informática é estudar a computação e formas úteis de tirar partido dela para resolver problemas importantes.

Facetas da computação

- **Faceta interna:** a informação é codificada usando símbolos (e.g. bytes), sendo a computação uma atividade de manipulação e transformação automática de símbolos.
- **Faceta externa:** ... mas, geralmente, as computações também estabelecem um diálogo interativo com o ambiente exterior (constituído por humanos e por outras máquinas).

Automatismos e sua especificação

- Qualquer computação é realizada de acordo com regras estabelecidas antes desse processamento se iniciar. É isso que significa o processamento ser automático.
- Daí a necessidade que especificar as computações de forma rigorosa e exaustiva, prevendo todas as eventualidades.

Algoritmos, Programas e Linguagens de Programação

Algoritmo

- Um algoritmo é um conjunto de regras abstratas que determinam, passo a passo, como uma computação vai decorrer.
- Em geral um problema pode ser resolvido usando diversos algoritmos.
- Um algoritmo é independente de qualquer linguagem de programação particular. Podemos imaginá-lo como se fosse para ser executado num computador ideal com memória infinita.

A palavra "algoritmo" deriva da palavra **Algoritmi** que por sua vez corresponde à Latinização do nome do matemático Persa Muhammad ibn Musa al-Khwarizmi, nascido por volta de 780DC. Este cientista, que trabalhou quase toda a sua vida em Bagdad, deu importantes contribuições para a Álgebra, Trigonometria e Aritmética.

Muhammad ibn Musa al-Khwarizmi

Programa

- Um programa é a expressão concreta dum algoritmo.
- Um programa implementa um algoritmo numa linguagem de programação concreta.

Linguagem de programação

- É uma notação para escrever programas.

Exemplo de algoritmo

Algoritmo de Euclides - mdc(m,n)

- Usar dois contadores x e y e inicializar x com m e y com n.
- Se x > y então x recebe x - y
- Se x < y então y recebe y - x
- Repetir os dois passos anteriores até que os valores de x e y fiquem iguais. Quando isso acontecer, esse é o resultado final.

Implementação em C do algoritmo anterior

```
#include <stdio.h>

int main() /* Implementação do algoritmo de Euclides */
{
    int m, n, x, y ;
    printf(">> ") ;
    scanf("%d %d", &m, &n) ;
    x = m ;
    y = n ;
    do {
        if( x > y ) x = x - y ;
        else if( x < y ) y = y - x ;
    } while( x != y ) ;
    printf("mdc(%d,%d) = %d\n", m, n, x) ;
    return 0 ;
}
```

Antiguidade dos Algoritmos

- Os algoritmos mais antigos que se conhecem devem-se aos Babilónios (3000AC-1500AC). Os seus livros de Matemática eram acima de tudo receitas sobre como efetuar os cálculos para resolver determinados problemas. Esses algoritmos eram para executar "à mão".
- A principal motivação para se usarem máquinas para executar algoritmos é a grande velocidade de execução que elas permitem.

Duas questões importantes sobre computação e linguagens

- Existem limites para a computação?
  - Sim, realmente existem problemas com solução para os quais não se consegue inventar qualquer algoritmo que descubra essa solução.
- Existe alguma linguagem de programação que permita implementar todos os algoritmos que se possam imaginar?
  - Sim, qualquer linguagem "normal" permite isso. Do ponto de vista do poder computacional, todas as linguagens normais são equivalentes.

Paradigmas de Programação

Já sabemos que computação significa processamento automático de informação, mas esta noção é demasiado vaga. Podemos interrogar-nos sobre quais os mecanismos concretos através dos quais esse processamento se efetua.

Na verdade, uma linguagem não pode deixar de se comprometer com algum conjunto mecanismos primitivos para processar informação. Ao efetuar essa escolha, a linguagem adere a um paradigma de programação particular.

É surpreendente a grande variedade de paradigma de programação que têm sido inventados.

Exemplos de paradigmas de programação e respetivos conceitos de base

- **Paradigma imperativo**
  - Conceitos: estado, atribuição, sequenciação
  - Linguagens: Basic, Pascal, C, Assembler.
- **Paradigma funcional**
  - Conceitos: função, aplicação, avaliação
  - Linguagens: Lisp, ML, OCaml, Haskell.
- **Paradigma lógico**
  - Conceitos: relação, dedução
  - Linguagens: Prolog.
- **Paradigma orientado pelos objetos**
  - Conceitos: objetos, mensagem
  - Linguagens: C++, Java, Eiffel.
- **Paradigma concorrente**
  - Conceitos: processo, comunicação (síncrona ou assíncrona)
  - Linguagens: Occam, Ada, Java.

Aspetos práticos

- Cada paradigma de programação determina uma forma particular de abordar os problemas e de formular as respetivas soluções. De facto, diferentes paradigmas de programação representam muitas vezes visões irreconciliáveis do processo de resolução de problemas.
- O grau de sucesso dum programador depende em parte da coleção de paradigmas que domina e da sua arte em escolher o modelo conceptual (paradigma) mais indicado para analisar e resolver cada problema.

Uma linguagem de programação pode combinar mais do que um paradigma

- C++ --- Paradigma imperativo + paradigma orientado pelos objetos.
- Java --- Paradigma imperativo + paradigma orientado pelos objetos + paradigma concorrente.
- OCaml --- Paradigma funcional + Paradigma imperativo + paradigma orientado pelos objetos.
- Ada --- Paradigma imperativo + paradigma concorrente.

Paradigma imperativo *versus* paradigma funcional

Paradigma imperativo

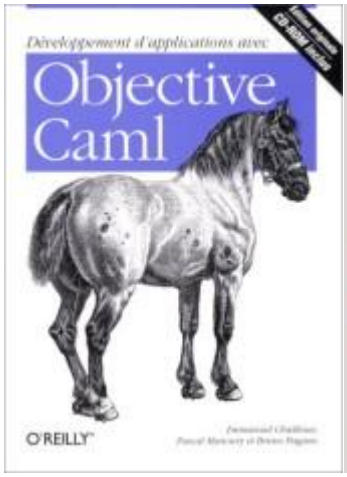
- A computação baseia-se na existência de um "estado" (conjunto de variáveis mutáveis) que vai evoluindo ao longo do tempo.
- O estado é modificado usando um "comando de atribuição" e existe um "operador de sequenciação".
- Linguagens: Basic, Pascal, C, etc., mas o Java também tem uma base imperativa, visto suportar variáveis mutáveis.

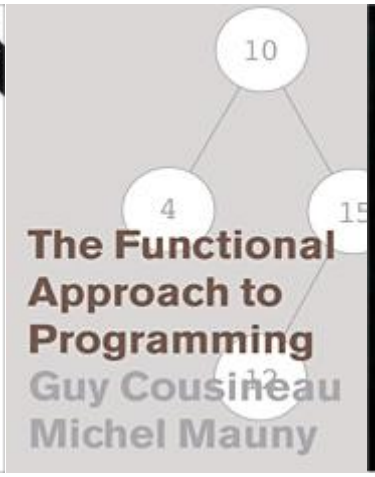
Paradigma funcional

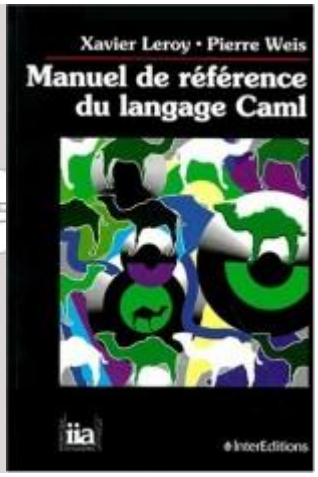
- A computação consiste na avaliação de "expressões" nas quais podem ocorrer chamadas de funções.
- O objetivo da avaliação duma expressão é a produção dum "resultado" ou "resposta".
- Não há "atribuição" nem variáveis cujo valor possa ser alterado.

- Cada função limita-se a produzir um resultado a partir dos seus argumentos.
- Linguagens: Lisp, ML, Haskell, Scheme.

Linguagem OCaml








Xavier Leroy


Didier Rémy


Damien Doligez


Jérôme Vouillon

Algumas características

- O OCaml foi desenvolvido no INRIA, a partir de 1991, por Xavier Leroy e Damien Doligez, a que se juntaram Jérôme Vouillon, Didier Rémy e outros em 1996.
- A linguagem OCaml é uma das muitas linguagens que derivam da linguagem ML. [O ML começou por ser a meta-linguagem do demonstrador de teoremas LCF, desenvolvido no final dos anos 70 por Robin Milner e outros na University of Edinburgh. Mas o ML evoluiu para passar a ser usado como linguagem de programação"normal". Eis alguns dialetos da linguagem ML: Standard ML, Caml, OCaml, Alice, F#.]
- O OCaml é uma linguagem onde se unificam os paradigmas funcional, imperativo e orientado pelos objetos. Na nossa cadeira estamos interessados em estudar apenas a parte funcional pura da linguagem.
- Tem um sistema de tipos estático com suporte para polimorfismo e inferência de tipos.
- As implementações de OCaml colocam um grande ênfase na velocidade de execução. A velocidade é superior à do C++, para programas orientados pelos objetos.
- Tipos básicos: caracteres, inteiros, reais, booleanos.
- Tipos derivados: funções, listas, tuplos, registos, tipos soma.
- Tem gestão automática de memória.
- Modularidade e compilação separada.
- Biblioteca.

- A linguagem não tem nenhum padrão reconhecido por um organismo oficial normativo. A implementação do INRIA funciona como padrão *de facto*.

Elementos essenciais da linguagem OCaml

- O OCaml é uma linguagem multi-paradigma, mas nesta cadeira vamos usar apenas o sua componente funcional, para aprender o estilo de programação funcional, sem distrações.
- Um programa é uma sequência de funções. As funções podem ser recursivas.
- São dois os principais mecanismos que podem ser usados na escrita do corpo das funções em OCaml:
  - APLICAÇÃO (aplicação duma função a argumentos). Ex: `sqrt 9`
  - IF-THEN-ELSE. Ex: `if prime x then x else x/2`

Exemplo de função que usa os dois mecanismos:

```
let rec fact x =
    if x = 0 then 1 else x * fact (x-1)
;;
```

Avaliação de expressões em OCaml

A ideia de **avaliação** está no centro do processo de execução de programas funcionais. Merece pois a nossa melhor atenção.

Em OCaml as expressões são avaliadas usando uma estratégia chamada *call-by-value*: uma funções só é aplicada aos seus argumentos depois de eles terem sido avaliados (ou, mais simplesmente, uma função só pode ser aplicada a valores). Esta é a estratégia usada na maioria das linguagens incluindo: Java, C, C++, Pascal, etc.

Exemplo de avaliação:

```
(fun x -> x+1)(2+3)
= (fun x -> x+1)5
= 5+1
= 6
```

Exemplo de avaliação que não termina. Considere a função `loop`, assim definida:

```
let rec loop x = loop x ;;
```

Avaliação:

```
(fun x -> 5)(loop 3)
= (fun x -> 5)(loop 3)
= (fun x -> 5)(loop 3)
= (fun x -> 5)(loop 3)
= ...  não termina
```

Mais uma avaliação. Considere a função `fact`, assim definida:

```
let rec fact x =
    if x = 0 then 1 else x * fact (x-1)
;;
```

Avaliação:

```
fact 3 =
= 3 * fact 2
= 3 * (2 * fact 1)
= 3 * (2 * (1 * fact 0))
= 3 * (2 * (1 * 1))
= 3 * (2 * 1)
= 3 * 2
```

```
= 6
```

Tipos em OCaml

Um **tipo** representa uma coleção de valores e tem associados um conjunto de **literais** e um conjunto de **operações**. (Um literal é uma expressão que não precisa de ser avaliada e denota um valor particular.)

Tipos básicos

TIPO	LITERAIS	OPERAÇÕES MAIS USADAS
int	5 -456	+ - * / mod min_int max_int int_of_float
float	3.14e-21	+. -. *. /. sqrt exp log sin ceil floor float_of_int
string	"" "ola"	^ String.length String.sub String.get
bool	false true	not || &&
char	'a' '$'	int_of_char char_of_int
unit	()	ignore

Tipos compostos

TIPO	LITERAIS	OPERAÇÕES MAIS USADAS
'a->'b	(fun x -> x+1)	aplicação
'a*'b	(5, 5.6)	fst snd *emparelhamento de padrões*
'a list	[] [3;5;7]	*emparelhamento de padrões*
tipos produto	*diversos*	*. emparelhamento de padrões*
tipos soma	*diversos*	*emparelhamento de padrões*

Inferência de tipos

O tipo dos argumentos e do resultado das funções não se declaram em OCaml. A implementação faz **inferência de tipos**: ela infere para cada função o tipo mais geral que é possível atribuir a essa função.

Qual o tipo das seguintes funções anónimas?

```
fun x -> x+1                    : int -> int
fun x -> x +. 1.0               : float -> float
fun x -> x ^ x                  : string -> string
fun (x,y) -> x + y              : (int * int) -> int
fun (x,y) -> (y,x)              : ('a*'b) -> ('b*'a)
fun x y -> (y,x)                : 'a -> 'b -> ('b*'a)
```

As quatro primeiras funções dizem-se **monomórficas** porque só aceitam argumentos de tipos fixos.
A duas últimas funções dizem-se **polimórficas** pois aceitam argumentos de tipos diversos.
De todas estas funções, a última é a única que aceita mais do que um argumento, neste caso dois. A razão pela qual o seu tipo tem duas setas será estudada mais tarde.

Não há sobreposição (overloading)

A linguagem OCaml não suporta sobreposição (overloading) de nomes ou operadores. Por exemplo, em OCaml o operador "+" denota apenas a soma inteira (a soma real denota-se "+.").

Para contrastar, as linguagens C, C++ e Java suportam sobreposição. Por exemplo, em Java o operador "+" é usado para denotar três operações: soma de inteiros, soma de reais, concatenação de strings.

Operadores

Para alguns dos operadores mais usados em OCaml, eis uma tabela de correspondência entre o OCaml e o Java:

OCaml	Java
&&	&&
||	||
not	!
mod	%
=	==
<>	!=
+	+
+.	+
^	+
-	-
-.	-

Comentários

```
(* Assim *)
```

#120

Linguagens e Ambientes de Programação (2010/2011)

Teórica 03 (16/Fev/2011)

Funções com múltiplos parâmetros. Formas curried e não-curried. Aplicação parcial de funções curried.

Associatividade do operador de aplicação. Associatividade do construtor de tipos funcionais "->". Formas equivalentes de escrita de funções.

Funções como valores de primeira classe. Exemplos.

Listas. Construtores de listas. Emparelhamento de padrões. Exemplos.

Funções com múltiplos argumentos em OCaml

Em OCaml há duas formas de escrever funções com mais do que um argumento: a **forma não-curried** e a **forma curried**. A segunda forma é mais elaborada e a que tem mais vantagem técnicas.

Forma não-curried

Os vários argumento agrupam-se num único tuplo ordenado. Exemplos:

```
let nAdd (x,y) = x+y ;;
nAdd: (int * int) -> int

let nAdd3 (x,y,z) = x+y+z ;;
nAdd3: (int * int * int) -> int

nAdd (3,4) = 7
```

```
nAdd3 (2,8,1) = 11
```
Tecnicamente, a função `nAdd` tem apenas um parâmetro, de tipo `int*int`. Mas claro, através desse único parâmetro conseguimos passar duas unidades de informação.

Forma curried

Os argumentos ficam separados. Exemplos:
```
let cAdd x y = x+y ;;
cAdd: int -> int -> int

let cAdd x y z = x+y+z ;;
cAdd: int -> int -> int -> int

cAdd 3 4 = 7
cAdd 2 8 1 = 11
```
Escrever *funções curried* não tem nada que saber: basta usar espaços em branco a separar os parâmetros, na cabeça de cada função. Contudo o tipo destas funções afigura-se surpreendente para quem o observa pela primeira vez. Isso é resultado da representação interna das funções em OCaml. Segue-se a explicação...

Representação interna das funções em OCaml

Por uma questão de simplicidade e regularidade de implementação, o OCaml só usa internamente funções anónimas com um único argumento. Uma função com múltiplos argumentos é convertida para um formato interno especial - chamado **forma interna** - que envolve apenas funções anónimas com um único argumento.

Por exemplo, a função

```
let cAdd x y = x+y ;;
```
é internamente convertida em:
```
let cAdd = fun x -> (fun y -> x+y) ;;
```
Repare na engenhosa a ideia que está por detrás do esquema de tradução.

Vejamos mais alguns exemplos de tradução, lado a lado:

```
let cAdd x y z = x+y+z ;;      let cAdd = fun x -> (fun y -> (fun z -> x+y+z)) ;;

let f x = x+1 ;;               let f = fun x -> x+1 ;;

let nAdd (x,y) = x+y ;;        let nAdd = fun (x,y) -> x+y ;;
```

Avaliação de expressões envolvendo funções com múltiplos argumentos

Compare a avaliação das duas seguintes expressões:
```
nAdd (2,3)
= (fun (x,y) -> x+y) (2,3)
= 2 + 3
= 5

cAdd 2 3
= (fun x -> (fun y -> x+y)) 2 3
= (fun y -> 2+y) 3
= 2 + 3
= 5
```

Aplicação parcial

As funções curried têm a vantagem de poderem ser **aplicadas parcialmente** ou seja, poderem ser invocadas omitindo alguns dos argumentos do final. Eis um exemplo de aplicação parcial:

```
let succ = cAdd 1 ;;
succ: int -> int
```

Associatividades

- O **operador de aplicação** é associativo à esquerda. (Note que este operador é *invisível*, pois nunca se escreve).
  - Portanto, a expressão `f a b` deve ser interpretada como `(f a) b`.
- O **construtor de tipos funcionais ->** é associativo à direita.
  - Portanto, o tipo `int -> int -> int` deve ser interpretado como `int -> (int -> int)`.

Formas equivalentes de escrever a mesma função

As seguintes quatro declarações são equivalentes, no sentido em que declaram exatamente a **mesma** função `f:int->int->int`.

```
let f x y = x + y ;;                    (* formato externo preferido *)
let f x = (fun y -> x+y) ;;             (* formato externo *)
let f = (fun x y -> x+y) ;;             (* formato externo *)
let f = (fun x -> (fun y -> x+y)) ;;    (* formato interno *)
```

Todas são convertidas para a mesma forma interna.

Funções como valores de primeira classe

Nas linguagens funcionais as funções têm o estatuto de **valores de primeira classe**. Isso significa que as funções têm um estatuto tão importante como o dos inteiros, reais, e outros tipos predefinidos.

Concretamente, numa linguagem funcional as funções podem ...

- Ser passadas como argumento para outras funções;
- Podem ser retornadas por outras funções;
- Podem ser usadas como elementos constituintes de estruturas de dados;
- Têm uma representação literal própria. Exemplo: `(fun x->x+1)`

Exemplos

Exemplo 1. Composição de funções.

```
let compose f g = fun x -> f (g x) ;;
compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

Também se pode escrever:

```
let compose f g x = f (g x) ;;
```

Exemplo 2. Função para converter do formato não-curried para o formato curried.

```
let curry f = fun x -> fun y -> f (x,y) ;;
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

Também se pode escrever:

```
let curry f x y = f (x,y) ;;
```

Exercício: Escrever a função inversa `uncurry`.

Exercício: Que funções são as seguintes e quais os seus tipos:

- `uncurry compose`
- `compose curry uncurry`
- `compose curry uncurry`

Exemplo 3. Como representar conjuntos usando apenas funções?

Um conjunto é uma entidade cuja principal característica é a possibilidade de se poder saber se um valor lhe pertence ou não lhe pertence. Assim vamos representar cada conjunto por uma função booleana que aplicada a um valor produz `true` se esse valor pertence ao conjunto e `false` se não pertence. Esta é a chamada **função característica** do conjunto.

- Conjunto vazio:

```
let set0 = fun x -> false ;;
```

- Conjunto universal:

```
let setu = fun x -> true ;;
```

- Criação de conjunto singular:

```
let set1 x = fun y -> y = x ;;
```

Exercício: Usando esta representação, escrever as funções `belongs`, `union` e `intersection`.

Exemplo 4. Estrutura de dados com funções: lista de funções.

```
let mylist = [(fun x -> x+1); (fun x -> x*x)] ;;
```

Uma limitação das funções

No caso geral, saber se duas funções dadas são iguais (i.e. saber se produzem sempre os mesmos resultados para os mesmos argumentos) é um problema que não pode ser resolvido por computador. Numa tal situação, costuma dizer-se que o problema é indecidível.

```
# (fun x -> x+1) = (fun x -> x+1) ;;
Exception: Invalid_argument "equal: functional value".
```

Listas homogéneas em OCaml

Apresentam-se aqui os três aspetos essenciais relativos as listas em OCaml: como se escrevem listas literais; como se constroem novas listas a partir de listas mais simples; como se analisam listas e se extraem os seus elementos constituintes.

Listas literais

Exemplos de listas literais:

```
[]                                  : 'a list
[2;4;8;5;0;9]                       : int list
["ola"; "ole"]                       : string list
[[1;2]; [4;5]]                      : int list list
[(fun x -> x+1); (fun x -> x*x)]    : (int->int) list
```

Construtores de listas

Estão disponíveis dois construtores de listas que, como o nome indica, servem para construir listas novas:

```
[] : 'a list
:: : 'a -> 'a list -> 'a list
```

- O construtor `[]` chama-se "lista vazia" e representa a lista vazia.
- O construtor `::` chama-se "cons" e serve para construir listas não vazias.

O operador :: é associativo à direita.

Exemplos de utilização de `cons`:

```
2::[3;4;5] = [2;3;4;5]
1::2::3::[] = [1;2;3]
[]::[] = [[]]
[1;2]::[3;4] = ERRO
4::5 = ERRO
[1;2]::[[3;4;5]] = [[1;2];[3;4;5]]
```

Processamento de listas usando emparelhamento de padrões

O processamento de listas efetuar-se por *análise de casos*, usando a construção **match** e *padrões*. Exemplo:

```
(* len : 'a list -> int *)

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

A função anterior trata dois casos, cada um dos quais tem um padrão diferente associado. Os vários casos são analisados sequencialmente, de cima para baixo.

Mais um exemplo. A seguinte função aplica-se a uma lista de pares ordenados e troca entre si as componentes de cada par:

```
(* swapPairs : ('a * 'b) list -> ('b * 'a) list *)

let rec swapPairs l =
    match l with
      [] -> []
    | (x,y)::xs -> (y,x)::swapPairs xs
;;
```

#120

Linguagens e Ambientes de Programação (2010/2011)

Teórica 04 (21/Fev/2011)

O método indutivo (redução de problemas a problemas mais simples).

Muitas funções sobre listas programadas usando o método indutivo.

Nomes locais. Funções mutuamente recursivas.

Método indutivo

Redução de problemas a problemas mais simples

Considere as seguintes duas funções recursivas, escritas em ML:

```
let rec fact n =
    if n = 0 then 1
    else n * fact (n-1)
;;

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

Analisando estas duas funções recursivas verificamos que quando lidam com o caso não-trivial - o chamado **caso geral** - ambas efetuam **reduções de problemas a problemas mais simples**. Concretamente, a função `len` reduz o problema `len (x::xs)` ao problema mais simples `len xs`, e a função `fact` reduz o problema `fact n` ao problema mais simples `fact (n-1)`.

Além de efetuarem *redução de problemas*, as duas funções lidam também, separadamente, com o caso trivial de cada problema - o chamado **caso base**. Concretamente, a função `len` trata separadamente o caso base `len []` e a função `fact` trata separadamente o caso base `fact 0`.

Repare que quando uma destas duas funções é aplicada a um argumento, inicia-se uma sucessão de *reduções a problemas mais simples* que só termina quando o caso base é alcançado. Por exemplo, a sequência de *reduções* produzida pela aplicação `len [7;6;5;4]` é a seguinte:

```
len [7;6;5;4]              <-- problema inicial
= 1 + len [6;5;4]          <-- problema um pouco mais simples
= 1 + 1 + len [5;4]        <-- problema ainda mais simples
= 1 + 1 + 1 + len [4]      <-- problema ainda mais simples
= 1 + 1 + 1 + 1 + len []   <-- caso base
= 1 + 1 + 1 + 1 + 0
= 4
```

As funções recursivas `len` e `fact` são representativas do que é programar usando o núcleo funcional do ML. Em ML programa-se essencialmente *reduzindo problemas a problemas mais simples*.

Técnica do método indutivo

O método indutivo (também conhecido como técnica da divisão e conquista) serve para ajudar a programar funções recursivas que efetuam *reduções de problemas a problemas mais simples*.

O método indutivo consiste na seguinte técnica:

- Para programar o caso geral G duma função recursiva, assume-se como PONTO DE PARTIDA (para começar a raciocinar) que o resultado S de aplicar a função a argumentos *mais simples do que os iniciais* (por exemplo a cauda da lista-argumento) já se encontra disponível.
- O problema que é então realmente preciso então resolver é o seguinte: COMO SE PASSA DE S PARA G?

Exemplo de utilização do método indutivo

Problema: Programar uma função `len` que determine o comprimento de listas.

Resolução: Para começar, a função `len` recebe uma lista, a qual terá de ser processada usando emparelhamento de padrões. Aplicando o método indutivo ao caso geral `G=len (x::xs)`, vamos começar por assumir que já temos o problema resolvido para o caso, mais simples, `S=len xs`.

Desde já podemos ir escrevendo a seguinte estrutura incompleta:

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

O problema que temos para resolver é o seguinte: *Como é que se obtém o valor de `G=len (x::xs)` a partir do valor de `S=len xs`?* Mas este problema é simples! De facto, basta adicionar uma unidade a S para se obter G!

Preenchendo o que falta no esquema anterior, surge a solução final:

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

NOTA1: Neste exemplo, reduzimos o problema `G=len (x::xs)` ao problema `S=len xs`. Mas esta não é a única redução possível... Existem muitas outras... Voltaremos a este assunto noutra aula.

NOTA2: A função `len` é tão simples que faz o método indutivo parecer óbvio e desinteressante. No entanto, este método tem imensas virtualidades:

- Ajuda-nos a organizar o pensamento quando queremos resolver problemas mais complicados, e.g. funções `power` e `sort`.
- Simplifica a resolução de problemas. [Repare que já não temos de resolver "o problema completo": passamos a ter de resolver "apenas" o problema da passagem de S para G, o que costuma ser "muito mais fácil".]
- Ajuda-nos a descobrir soluções simples e compactas.

Mais funções sobre listas programadas usando o método indutivo

Estude-as todas de forma muito atenta. Depois tente programá-las todas "sem espreitar".

Comprimento de lista:

```
len : 'a list -> int

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

Soma dos elementos de lista:

```
sum : int list -> int

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

Concatenação de listas:

```
append : 'a list -> 'a list -> 'a list

let rec append l1 l2 =
    match l1 with
        [] -> l2
      | x::xs -> x::append xs l2
;;
```

Acrescentar elemento no final de lista:

```
putAtEnd : 'a -> 'a list -> 'a list

let rec putAtEnd v l =
    match l with
```

```
        [] -> [v]
      | x::xs -> x::putAtEnd v xs
;;
```

Inversão de lista:

```
rev : 'a list -> 'a list

let rec rev l =
    match l with
        [] -> []
      | x::xs -> append (rev xs) [x]
;;
```

Máximo duma lista (tratamento implícito do erro):

```
maxList : 'a list -> 'a

let rec maxList l = (* pre: l <> [] *)
    match l with
        [x] -> x
      | x::xs -> max x (maxList xs)
;;
```

Máximo duma lista (tratamento explícito do erro):

```
maxList : 'a list -> 'a

let rec maxList l =  (* pre: l <> [] *)
    match l with
        [] -> raise (Arg.Bad "maxList: lista vazia")
      | [x] -> x
      | x::xs -> max x (maxList xs)
;;
```

Aplicação de função a todos os elementos de lista:

```
map : ('a -> 'b) -> 'a list -> 'b list

let rec map f l =
    match l with
        [] -> []
      | x::xs -> (f x)::map f xs
;;
```

Seleção dos elementos duma lista que verificam uma dada propriedade:

```
filter : ('a -> bool) -> 'a list -> 'a list

let rec filter b l =
    match l with
        [] -> []
      | x::xs ->
            if b x then x::filter b xs
            else filter b xs
;;
```

Concatenação de todas as listas contidas numa lista de listas:

```
flatten : 'a list list -> 'a list

let rec flatten ll =
    match ll with
        [] -> []
      | l::ls ->
            l @ flatten  ls
;;
```

Concatenação de todos os resultados gerados por uma função de mapping que produz listas:

```
flatMap : ('a -> 'b list) -> 'a list -> 'b list

let rec flatMap f l =
    match l with
        [] -> []
      | x::xs ->
            (f x) @ flatMap f xs
;;
```

Ordenação de listas:

```
sortList : 'a list -> 'a list

let rec insOrd v l =
```

```
    match l with
        [] -> [v]
      | x::xs ->
          if v <= x then v::x::xs
          else x::insOrd v xs
;;

let rec sortList l =
    match l with
        [] -> []
      | x::xs ->
          insOrd x (sortList xs)
;;
```

Fusão de listas ordenadas sem repetições (aqui dá jeito usar matching duplo):

```
fusion : 'a list -> 'a list -> 'a list

let rec fusion l1 l2 =
    match l1, l2 with
        _, [] -> l1
      | [], _ -> l2
      | x::xs, y::ys ->
          if x < y then x::fusion xs l2
          else if y < x then y::fusion l1 ys
          else x::fusion xs ys
;;
```

Exemplos de avaliações

```
sum [1;2;3]
= 1 + sum [2;3]
= 1 + 2 + sum [3]
= 1 + 2 + 3 + sum []
= 1 + 2 + 3 + 0
= 6

append [1;2;0] [3;4]
= 1::append [2;0] [3;4]
= 1::2::append [0] [3;4]
= 1::2::0::append [] [3;4]
= 1::2::0::[3;4]
= [1;2;0;3;4]

rev [1;2;3]
= (rev [2;3]) @ [1]
= (rev [3]) @ [2] @ [1]
= (rev []) @ [3] @ [2] @ [1]
= [] @ [3] @ [2] @ [1]
= [3;2;1]
```

A função @

A função `append` está predefinida em ML, sendo denotada pelo operador binário `@`. Assim, podemos escrever:

```
[1;2;3] @ [4;5;6] = append [1;2;3] [4;5;6] = [1;2;3;4;5;6]
```

No entanto, a função `@` é pouco eficiente e deve ser usada com critério. Por exemplo, para acrescentar um zero à cabeça duma lista `list` é má ideia escrever `[0]@list`; escreve-se sempre `0::list`. No entanto, para acrescentar um zero ao final duma lista `list` não temos alternativa e escreve-se mesmo `list@[0]`.

Nomes locais

Construção let-in

A construção **let-in**:

```
let n = exp in
    exp1
```

associa o nome `n` ao valor da expressão `exp` no contexto da expressão `exp1`. O nome `n` passa a definir uma constante local à expressão `exp1`.

O nome `n` também pode ser parametrizado, assim

```
let n x = exp in
    exp1
```

mas repare que esta forma não introduz nada de realmente novo por ser equivalente a

```
let n = (fun x -> exp) in
    exp1
```

A palavra reservada **and** pode ser usada para definir simultaneamente vários nomes no mesmo `let-in`. Por exemplo:

```
let a = 1 and b = 2 in
    a + b ;;
```

ou

```
let f x = x + 1 and g x = x + 2 in
    f (g x) ;;
```

Exercício: Procure no manual da linguagem a forma sintática geral da construção **let-in**.

Vamos ver de seguida que a construção **let-in** permite:

1. Aumentar a legibilidade de certas função;
2. Aumentar a eficiência de certas função;
3. Definir funções mutuamente recursivas.

1. Legibilidade

Considere o seguinte problema: `Escreva em ML uma função chamada` **`parque`** `que calcule o preço a pagar no parque de estacionamento subterrâneo dos Restauradores, em Lisboa. A função parque recebe 2 pares ordenados de inteiros como argumentos: hora e minuto de entrada, hora e minuto de saída. Os tempos de permanência são arredondados para horas completas e sempre para cima. Assuma que o carro nunca está no parque mais do que 24:00. Em Março de 1999, a tabela de preços era a seguinte:`

```
1ª hora        120 cêntimos
2ª hora        140 cêntimos
3ª hora        150 cêntimos
seguintes      155 cêntimos
```

Apresentam-se a seguir duas versões equivalentes da função `parque`, a segunda programada usando **let-in**. A escolha entre elas é, em grande medida, uma questão de gosto, mas pode argumentar-se que a segunda versão, apesar de mais longa, é mais fácil de perceber porque torna explícita a ordem de avaliação das sub-expressões dentro duma expressão que é demasiado grande e complexa.

```
let parque (he, me) (hs, ms) =
    pagar (convHoras (permanencia (convMinutos he me) (convMinutos hs ms)))
;;
let parque (he, me) (hs, ms) =
    let te = convMinutos he me in
        let ts = convMinutos hs ms in
            let perm = permanencia te ts in
                let h = convHoras perm in
                    pagar h
;;
```

Exercício: As funções auxiliares não foram ainda programadas. Tente escrevê-las.

2. Eficiência

A construção **let-in** também pode ser usada para aumentar a eficiência de certas funções. Por exemplo a segunda função é mais eficiente do que a primeira (porquê?):

```
let f g x =
    g x + g x * g x
;;

let f g x =
    let r = g x in
        r + r * r
;;
```

3. Funções mutuamente recursivas

A forma geral da construção **let-in** inclui a possibilidade de utilização simultânea de **rec** e **and**. Isso permite definir 2 ou mais **funções mutuamente recursivas**. Eis um exemplo curioso com duas funções:

```
        let rec even n = if n = 0 then true else odd (n-1)
             and odd n = if n = 0 then false else even (n-1) in
                 exp
;;
```

O que fazem estas funções?

Tradução do let-in

A construção **let-in**

```
let n = exp in
    exp1
```

é internamente traduzida para a seguinte representação:

```
(fun n -> exp1) exp
```

As duas formas são equivalentes (medite um pouco nisso!), mas a primeira é mais fácil de perceber.

Tradução do let simples

A já nossa conhecida construção **let-simples**, que permite definir nomes globais, é internamente traduzida para a construção **let-in** - os nomes globais são habilidosamente traduzidos para nomes locais.

Por exemplo, a seguinte sequência de definições e expressões:

```
# let f x = x + 1 ;;
# let g x = x + 2 ;;
# f (g x) ;;
```

é internamente traduzida para um conjunto de **let** aninhados:

```
let f x = x + 1 in
    let g x = x + 2 in
        f (g x) ;;
```

#120

Linguagens e Ambientes de Programação (2010/2011)

Teórica 5 (23/Fev/2011)

Tipos soma (uniões). Os tipos somas árvore binária e árvore n-ária.

Padrões. Padrões disjuntos. Emparelhamento de padrões.

Tipos soma (uniões)

Muitas linguagens de programação incluem uma construção específica para a definição de tipos cujos valores podem assumir **diferentes variantes**, disjuntas entre si. Essa construção designa-se genericamente por **tipo soma**.

Por exemplo, numa linguagem com suporte para tipos soma é possível definir um tipo de dados *cshape*, para representar formas geométricas coloridas cujos valores podem assumir as três seguintes variantes: *Line*, *Circle* e *Rect*.

Um tipo soma permite conciliar o **geral** com o **particular**. Por um lado os elementos das variantes *Line*, *Circle* e *Rect* são formas geométricas coloridas com algumas propriedades comuns: no mínimo todas têm um cor! Por outro lado são variantes, cada uma delas com dados específicos associados: um círculo tem um centro e um raio, uma linha tem dois pontos extremos, etc.

Os tipos soma do Pascal chamam-se *registos com variante*.
Os tipos soma do C são as *uniões*.
Os tipos soma do Java, Smalltalk e C++ são as *classes abstratas* (imagine uma classe abstrata *cshape* com três subclasses concretas *Line*, *Circle* e *Rect*). [Em rigor, as classes abstratas são um bocadinho diferentes dos tipos soma: os tipos soma são entidades fechadas enquanto que as classes abstratas são extensíveis.]

Como é em ML?

Tipos soma em ML

Para exemplificar, o tipo soma cshape atrás referido pode ser definido em ML da seguinte forma, usando os tipos auxiliares color e point:

```
type color = int ;;
type point = float*float ;;

type cshape = Line of color*point*point
            | Circle of color*point*float
            | Rect of color*point*point ;;
```

Repare que o nome dos tipos definidos pelo utilizador começa sempre por letra minúscula e o nome de cada variante começa sempre por letra maiúscula. Chamam-se **etiquetas**, aos nomes das variantes.

Continuando a usar o mesmo exemplo, vejamos agora quais são os mecanismos essenciais para escrever e manipular valores de tipos soma.

Literais: Eis dois literais de tipo cshape. Repare como o nome de cada variante é usado para marcar os respetivos literais.

```
let a = Line (34658, (2.5, 7.8), (-24.005, 1000.0001)) ;;
let b = Circle (11111, (-24.005, 1000.0003), 3.1233333) ;;
```

Construção: Por exemplo, a seguinte função cria círculos centrados no ponto zero:

```
let zeroCircle c r = Circle (c, (0.0, 0.0), r) ;;
```

Processamento: Eis uma função que calcula a área duma forma colorida. Os elementos de qualquer tipo soma são processados usando emparelhamento de padrões.

```
let area cs =
    match cs with
        Line   (_, _, _) -> 0.0
      | Circle (_, _, r) -> 3.14159 *. r *. r
      | Rect   (_, (tx,ty), (bx,by)) -> abs_float ((bx -. tx) *. (by -. ty))
;;
```

Eis uma função que determina o raio duma forma. Só está definida para círculos.

```
let radius (Circle (_, _, r)) = r ;;
```

Alguns tipos soma predefinidos em ML

1. O tipo **'a list** é um tipo soma. Internamente a sua definição assemelha-se a:
   ```
   2. type 'a list = Nil | Node of 'a * 'a list ;;
   ```
3. O tipo **bool** também é um tipo soma, internamente é definido da seguinte forma:
   ```
   4. type bool = false | true ;;
   ```

   O ML abre uma exceção neste caso, e permite que o nome das variantes comece por letra minúscula.

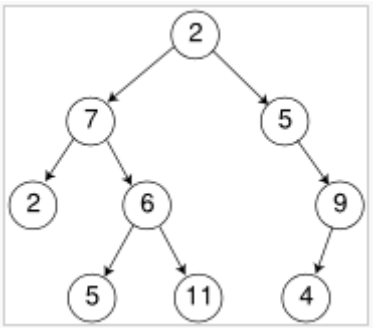
5. O tipo **'a option** é um tipo soma que permite representar o conceito de **ausência de valor**, em situações que tal possa ser útil. Internamente, é definido assim:
   ```
   6. type 'a option = None | Some of 'a ;;
   ```

Os três papéis das etiquetas dum tipo soma

As etiquetas dum tipo soma tem três papéis:

- Na definição do tipo, Identificam os vários ramos.
- Denotam os *construtores* que o sistema cria automaticamente para o tipo em causa. Um *construtor* é uma função especial que gera valores dum tipo soma. Quando escrevemos Circle(111, (0.0, 0.0), 12.4) estamos a chamar um construtor das nossas formas.
- São elementos constituintes de novos padrões que a linguagem passa a reconhecer automaticamente.

Árvores binárias



Em programação, as árvores binárias permitem exprimir informação hierarquizada e permitem organizar dados por forma a aumentar a velocidade de acesso a eles.

O tipo soma **árvore binária** não está predefinido em ML, mas é fácil de definir usando um tipo soma:

```
type 'a tree = Nil | Node of 'a * 'a tree * 'a tree ;;
```

Literais: Eis uma constante do tipo `int tree`:

```
Node(1, Node(2,Nil,Nil), Node(3,Nil,Nil)) ;;
```

Construção: Por exemplo, a seguinte função cria folhas da árvore:

```
let makeLeaf v = Node(v, Nil, Nil) ;;
```

Processamento: Eis uma função que determina o número total de nós duma árvore binária:

```
let rec size t =
    match t with
        Nil -> 0
      | Node(x,l,r) ->
            1 + size l + size r
;;
```

Eis um exemplo de avaliação duma expressão envolvendo uma chamada da função "size":

```
  size (Node(1, Node(2,Nil,Nil), Node(3,Nil,Nil)))
= 1 + size (Node(2,Nil,Nil)) + size (Node(3,Nil,Nil))
= 1 + (1 + size Nil + size Nil) + (1 + size Nil + size Nil)
= 1 + (1 + 0 + 0) + (1 + 0 + 0)
= 1 + 1 + 1
= 3
```

Vocabulário relativo a árvores

- Raiz - nó sem ascendentes
- Nó interno - nó diferente da raiz com pelo menos um filho
- Folha - nó sem filhos

Método indutivo aplicado a árvores binárias

No caso das árvores binárias, o método indutivo consiste na seguinte técnica:

- Para programar o caso geral G duma função recursiva, assume-se como PONTO DE PARTIDA PARA COMEÇAR A RACIOCINAR que os resultados L, R de aplicar a própria função a argumentos *mais simples do que os iniciais* (por exemplo às duas sub-árvores) já se encontram disponíveis. O problema que é então preciso resolver é o seguinte: COMO É QUE A PARTIR DE L E R SE CHEGA A G?

Exemplo de utilização do método indutivo

Problema: Programar uma função "size" que determine o número total de nós duma árvore binária:

Resolução: A função "size" aplica-se a uma árvore, a qual será, naturalmente, processada usando emparelhamento de padrões. Aplicando o método indutivo ao caso geral "G=Node(x,l,r)", vamos começar por assumir que já temos o problema resolvido para os casos "L=size l" e "R=size r". Obtemos assim o seguinte PONTO DE PARTIDA PARA COMEÇAR A RACIOCINAR:

```
 let rec size t =
    match t with
         Nil ->  ...
       | Node(x,l,r) ->
            ... size l ... size r ...
;;
```

O problema que temos para resolver é o seguinte: *Como é que se obtém o valor de "G=size (Node(x,l,r))" a partir dos valores "L=size l" e "R=size r"?* Mas este problema é simples. De facto, basta adicionar uma unidade a L+R para se obter G!

Preenchendo o que falta no esquema anterior, surge a solução final:

```
let rec size t =
    match t with
        Nil -> 0
      | Node(x,l,r) ->
           1 + (size l) + (size r)
;;
```

Mais duas funções sobre árvores binárias

Altura duma árvore:

```
(* height: 'a tree -> int *)

let rec height t =
    match t with
       Nil -> 0
     | Node(x,l,r) ->
          1 + max (height l) (height r)
;;
```

Árvore ao espelho:

```
(* mirror: 'a tree -> 'a tree *)
```

```
let rec mirror t =
    match t with
      Nil -> Nil
    | Node (x,l,r) ->
          Node (x,mirror r,mirror l) (* o r e l trocam de posição *)
;;
```

Árvores n-árias

Numa árvore n-ária, cada nó pode ter um número qualquer (ilimitado) de filhos.

O tipo soma **árvore n-ária** pode definir-se assim em ML:

```
type 'a ntree = NNil | NNode of 'a * 'a ntree list ;;
```

Literais: Eis uma constante de tipo "int ntree":

```
NNode(1, [NNode(2,[]); NNode(3,[]); NNode(4,[])]) ;;
```

Construção: Por exemplo, a seguinte função cria folhas da árvore:

```
let makeLeaf v = NNode(v, []) ;;
```

Processamento: A função `size` determina o número de nós duma árvore n-ária. Note que se usa uma função auxiliar que processa uma lista de árvores.

```
(* size: 'a ntree -> int *)

let rec lsize ts =
    match ts with
        [] -> 0
      | NNil::ts -> lsize ts
      | NNode(x,cs)::ts -> 1 + lsize cs + lsize ts
;;

let size t =
    lsize [t]
;;
```

No caso geral da função anterior, repare que há duas chamadas recursivas: uma para os filhos dum nó, e outra para os irmãos à direita desse nó.

Esquema geral da utilização do método indutivo no tratamento de árvores n-árias:

```
let rec lf ts =
    match ts with
        [] -> ...
      | NNil::ts -> ... lf ts
      | NNode(x,cs)::ts -> ... lf cs ... lf ts ...
;;

let f t =
    lf [t]
;;
```

Árvore ao espelho:

```
(* mirror: 'a ntree -> 'a ntree *)

let rec lmirror ts =
    match ts with
        [] -> []
      | NNil::ts -> lmirror ts @ [NNil]
      | NNode(x,cs)::ts -> lmirror ts @ [NNode(x,lmirror cs)]
;;

let mirror t =
    List.hd (lmirror [t])
```

```
;;
```

Padrões

Um **padrão** é uma expressão especial que representa um **conjunto de valores**.

A utilização de padrões torna as funções mais fáceis de escrever e de entender. Os padrões são, portanto, bons amigos do/a programador/a. As linguagens funcionais antigas (e.g. Lisp) não usavam padrões, mas as linguagens funcionais modernas (e.g. ML, Haskell) não os dispensam.

Exemplos de padrões:

```
Padrão          Conjunto de valores representados
~~~~~~          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[]              lista vazia
[x]             listas com um elemento
[x;y]           listas com dois elementos
x::xs           listas não vazias
x::y::xs        listas com pelo menos dois elementos
5::xs           listas cujo primeiro elemento é 5
x               todos os valores (padrão universal)
_               padrão universal anónimo
(x,y)           todos os pares ordenados
(0,y)           todos os pares ordenados cuja 1ª componente é 0
8               inteiro 8
(x,y)::xs       lista não vazia de pares ordenados
'a'..'z'        letras de 'a' a 'z'
```

Numa expressão **match** podem ocorrer padrões em número variável (ver exemplos 1 e 2). Durante a avaliação, esses padrões são explorados sequencialmente, de cima para baixo.

Exemplos

Nos exemplos que se seguem, os padrões aparecem assinalados a negrito.

Exemplo1:

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

Exemplo2:

```
let rec count5 l =
    match l with
        [] -> 0
      | 5::xs -> 1 + count5 xs
      | _::xs -> count5 xs
;;
```

Exemplo 3:

```
fun (x,y) -> (y,x)
```

Exemplo 4:

```
let f (x,y) =
    (y,x)
;;
```

Exemplo 5:

```
let f g =
    let (x,y) = g 0 in
        x + y
```

Regra da seta ->

- No contexto que antecede a seta -> (portanto nas expressões **match** e nos argumentos das funções anónimas), as expressões são interpretadas como padrões. Por exemplo, nesse contexto, x::xs representa o conjunto de todas as listas não vazias.
- Fora do contexto que antecede a seta ->, as expressões são interpretadas da forma habitual. Por exemplo, fora desse contexto, x::xs representa a aplicação do construtor "cons" aos nomes x e xs).

Note que devido à forma como a construção **let-in** se define por tradução, toda a expressão que aparece imediatamente a seguir ao `let` também é interpretada como padrão. Também devido à forma como as funções com nome são traduzidas para para funções anónimas, toda a expressão que aparece no cabeçalho imediatamente antes do sinal de igual é interpretada como padrão.

Padrões disjuntos

Dois padrões dizem-se **disjuntos** se os conjuntos que eles representam são disjuntos. Numa expressão `match` a ordem dos casos só é irrelevante se os padrões forem disjuntos entre si.

Exemplos:

- Os dois padrões que ocorrem na função `len` **são disjuntos**:
  - `let rec len l =`
  - `match l with`
  - `[] -> 0`
  - `| x::xs -> 1 + len xs`
  - `;;`
- Os dois padrões que ocorrem na função `fact` **não são disjuntos**:
  - `let rec fact n =`
  - `match n with`
  - `0 -> 1`
  - `| n -> n * fact (n-1)`
  - `;;`

Emparelhamento

Como resultado do **emparelhamento dum valor com um padrão**, há duas consequências possíveis:

1. Falhanço;
2. Sucesso, e os nomes que ocorrem no padrão são ligados a valores.

Restrições

- Num padrão, não se permite a repetição de nomes. Por exemplo o padrão `(x,x)::xs` é inválido.
- Todos os nomes que ocorrem num padrão são considerados nomes novos, mesmo que esses nomes ocorram no ambiente envolvente.

A seguinte função aplica-se a uma lista de pares ordenados, e conta o número de pares com as duas componentes iguais:

```
let rec eqPairs l =
```

```
    match l with
      [] -> 0
    | (x,y)::xs ->    (* o padrão (x,x)::xs parece melhor, mas seria inválido *)
        if x=y then 1 + eqPairs xs
        else eqPairs xs
;;
```

A seguinte função aplica-se a uma lista de pares ordenados, e conta o número de pares em que a segunda componente é igual a um valor dado:

```
let rec countPairs v l =
    match l with
      [] -> 0
    | (x,y)::xs ->    (* o padrão (x,v)::xs parece melhor, mas não interessa pois contém
um v "novo" *)
        if y=v then 1 + countPairs v xs
        else countPairs v xs
;;
```

#110

Linguagens e Ambientes de Programação (2010/2011)

Teórica 6 (28/Fev/2011)

Tipos produto (tuplos e registos).
Tratamento de exceções.
Funções parciais

Tipos produto (tuplos e registos)

A maioria das linguagens de programação incluem nos seus sistemas de tipos uma construção específica que permite representar agrupamentos de dados heterogéneos. O nome genérico dessa construção, independente da linguagem particular, é **tipo produto**.

Numa linguagem com tipos produto é possível definir, por exemplo, um tipo de dados *pessoa* constituído por um nome (de tipo string), um ano de nascimento (de tipo int), uma morada (de tipo string), etc.

Os tipos produto do Pascal são os *registos*.
Os tipos produto do C são as *estruturas*.
Os tipos produto do Java, Smalltalk e C++ são as *classes*.
No Fortran, os tipos produto apareceram na versão Fortran 90 com o nome de *tipos derivados*.

Como é em ML?

Tipos produto em ML

Em ML há duas variedades de tipos produto: **tipos produto não etiquetados** e **tipos produto etiquetados**.

Tipos produto não etiquetados (tuplos)

Os produtos cartesianos do ML são exemplos de tipos produto, neste caso ditos **não-etiquetados**, e também conhecidos por **tuplos**. Por exemplo, para representar *pessoas*, pode usar-se em ML o seguinte tipos produto não etiquetado:

```
string * int * string
```

Literais: Para exemplificar, eis um valor do tipo anterior:

```
("João ", 1970, "Lisboa")
```

Construção: Para exemplificar vejamos uma função que muda a morada duma pessoa, criando um tuplo novo:

```
let moveTo (x,y,_) city = (x, y, city) ;;
```

Processamento: Como se processam tuplos? De duas formas:

- Usando emparelhamento de padrões:
  - `let getName p =`
  - `    match p with`
  - `      (x, _, _) -> x`
  - `;;`
- Ou usando as operações de acesso **fst** e **snd** predefinidas, se o registo tiver duas componentes:
  - `não aplicável ao nosso exemplo`

Tipos produto etiquetados (registos)

Mas o ML, também suporta **tipos produto etiquetados**, também conhecidos como **registos**, os quais requerem definição explícita. Eis um exemplo de tipo produto etiquetado:

```
type pessoa = { nome:string ; anoNasc:int ; morada:string } ;;
```

Literais: Eis um literal deste tipo:

```
{ nome = "João" ; anoNasc = 1970 ; morada = "Lisboa" }
```

Construção: Para exemplificar vejamos uma função que muda a morada duma pessoa, criando um registo novo:

```
let moveTo p city =
    { nome = p.nome ; anoNasc = p.anoNasc ; morada = city }
;;
```

Processamento: Como se processam registos? De duas formas:

- Usando emparelhamento de padrões:
  - `let getNome p =`
  - `    match p with`
  - `      { nome = x ; anoNasc = _ ; morada = _ } -> x`
  - `;;`
- Ou usando a operação de acesso "." e que funciona em ML exatamente como em Pascal ou C:
  - `let getNome p = p.nome ;;`

Tratamento de exceções em ML

Durante a execução de um programa, por vezes verificam-se determinadas condições (geralmente anómalas, mas nem sempre) às quais é necessário reagir alterando o fluxo de execução normal. Tais condições chamam-se **exceções**.

As exceções podem ser geradas:

- Ao nível do hardware. Por exemplo, em virtude duma divisão por zero.
- Ao nível do sistema operativo. Por exemplo, devido à impossibilidade de abrir um ficheiro inexistente, ou devido à tentativa de continuar a ler dum ficheiro que já chegou ao fim.
- Deliberadamente pelos próprios programas, usando a construção **raise**. Por exemplo, para lidar explicitamente com **argumentos proibidos**:
  - `let rec fact x =`

```
    if x = 0 then 1
    else if x>0 then x * fact(x-1)
    else raise (Arg.Bad "fact")
;;
```

Captura e tratamento de exceções

Quando uma exceção é gerada, o controlo da execução do programa é transferido para o **tratador de exceções** (*exception handler*) mais recentemente cativado. É abortada a avaliação de todas as funções chamadas depois da ativação desse tratador de exceções.

Em ML, um tratador de exceções escreve-se usando uma expressão **try-with**, como se exemplifica de seguida. A expressão `exp`, no seu interior, diz-se uma **expressão protegida**.

```
try
    exp
with Sys_error _ -> exp1
   | Division_by_zero -> exp2
   | End_of_file -> exp3
   ...
;;
```

Como é avaliada uma expressão `try-with`?

- Começa-se por avaliar a expressão protegida `exp`.
- Caso nenhuma exceção seja gerada por `exp`, então o `try-with` não tem qualquer efeito e a expressão global `try-with` tem o mesmo valor da expressão `exp`.
- Mas se for gerada alguma exceção por `exp`, então o `try-with` recebe o controlo da execução e usa emparelhamento de padrões para descobrir qual das expressões `exp1`, `exp2`, `exp3`, etc., deve ser avaliada em substituição de `exp`.
- Finalmente, se o emparelhamento de padrões anterior falhar, então a exceção é propagada para o `try-with` cronologicamente anterior.

Existe um tratador de exceções de sistema que apanha as exceções não tratadas e aborta a execução do programa com uma mensagem de erro apropriada a cada caso. Exemplos:

```
# 4/0;;
Exception: Division_by_zero.

# open_in "fdsg" ;;
Exception: Sys_error "fdsg: No such file or directory".
```

Definição de novas exceções

A lista de exceções predefinidas na linguagem encontra-se aqui: Index of exceptions.

O programador pode definir novas exceções. A sintaxe da definição duma nova exceção é igual à sintaxe da definição duma variante dum tipos soma.

Exemplos. O primeiro exemplo define uma exceção com argumento; o segundo define uma exceção sem argumento.

```
exception Stack_overflow of string * int ;;
exception I_m_so_out_of_here ;;
```

Funções parciais

Uma **função parcial** é uma função que só está definida em parte do seu domínio. (Não confundir com *aplicação parcial*, que é outra coisa.)

Por exemplo, a função `fact` é parcial porque só está definida para valores não-negativos:

```
let rec fact n = (* pre: n >= 0 *)
    if n = 0 then 1
    else n * fact (n-1)
;;
```

Outro exemplo: A função `maxList` é parcial porque só está definida para listas não-vazias:

```
let rec maxList l = (* pre: l <> [] *)
    match l with
        [x] -> x
      | x::xs -> max x (maxList xs)
;;
```

Quando se escreve uma função parcial, espera-se que essa função seja sempre aplicada a argumentos válidos. Mas os programas podem ter erros e é importante que os programas não disfarcem esses erros - é muito melhor a execução dum programar abortar do que terminar produzindo resultados errados. Por isso, convém garantir que a função não produz qualquer resultado quando aplicada a argumentos inválidos (por outras palavras, temos de obrigar o resultado a ser realmente *indefinido*).

A melhor forma de impedir uma função de produzir resultado, ao mesmo tempo gerando uma mensagem de erro clara, é gerar uma exceção. Assim:

```
let rec fact n =  (* pre: n >= 0 *)
    if n = 0 then 1
    else if n > 0 then n * fact (n-1)
    else raise (Arg.Bad "fact: numero negativo")
;;

let rec maxList l =  (* pre: l <> [] *)
    match l with
         [] -> raise (Arg.Bad "maxList: lista vazia")
       | [x] -> x
       | x::xs -> max x (maxList xs)
;;
```

Mas, repare, mesmo sem lançar exceções explícitas, as versões originais destas duas funções já garantem a não produção de resultados: a primeira aborta com "Stack overflow" e a segunda gera a exceção `Match_failure`.

#110

Linguagens e Ambientes de Programação (2010/2011)

Teórica 7 (02/Mar/2011)

Efeitos laterais em ML. Motivação do tipo "unit" e do operador ";".

Input/Output em ML. Utilização do método indutivo na programação de funções sobre ficheiros.

Introdução aos módulos em ML.

Efeitos laterais

O ML usa um modelo de input/output imperativo, no qual as operações de input/output são tratadas como *efeitos laterais*.

O que é um **efeito lateral**? Um *efeito lateral* é qualquer atividade que uma função desenvolva para além de calcular um resultado a partir dos argumentos. Por exemplo:

- Escrever num ficheiro.
- Escrever no ecrã do computador.
- Ler dum ficheiro.

Como sabemos, no paradigma funcional é suposto uma função limitar-se a calcular um resultado a partir dos seus argumentos. O facto é que a linguagem ML ultrapassa os limites estritos do paradigma funcional, já que admite efeitos laterais nas funções.

Operador de sequenciação

Para sequenciar os efeitos laterais, o ML dispõe dum *operador de sequenciação* que se escreve ";" (como em Pascal!). Assim, por exemplo, para escrever no ecrã a string "ola", e logo a seguir a string "ole" usamos a seguinte função:

```
let hello () =
    print_string "ola" ; print_string "ole"
;;
```

Tecnicamente, ";" é o nome duma função com o seguinte tipo:

```
";": unit -> 'b -> 'b
```

e a seguinte definição interna:

```
";" x y = y ;;
```

Tipo unit e valor ()

Por vezes gostaríamos de definir funções sem argumentos ou sem resultados. Isso faz sentido para funções só com efeitos laterais. Mas o ML não o permite, pois obriga todas as funções a ter argumentos e um resultado. O tipo **unit** foi inventado para ser usado nessas situações.

O tipo **unit** é um tipo básico com apenas um valor, que se escreve **()**. Para comparação, note que o tipo **bool** é um tipo com apenas dois valores, que se escrevem **true** e **false**. Por seu lado, o tipo **void** do C é um tipo sem valores e por isso não resolveria o nosso problema em ML.

Exemplos de utilização de unit e ().

```
print_string: string -> unit
read_int: unit -> int
print_newline: unit -> unit

let x = read_int () in
    print_int (x+1)
```

Exemplo de função para escrever uma lista de inteiros no ecrã:

```
  let rec printList l =
      match l with
         [] -> ()
       | x::xs -> print_int x ; print_newline () ; printList xs
  ;;
```

Input-Output em ML

Canais

As operações sobre ficheiros são efetuadas através de **canais**. Os canais são valores dos tipos predefinidos *in_channel* e *out_channel*.

Os "canais" do ML correspondem às "streams" do Java.

Em ML existem três canais predefinidos que são automaticamente abertos quando o programa começa a correr:

- `stdin : in_channel`
- `stdout : out_channel`
- `stderr : out_channel`

Primitivas de input/output

- **Primitivas para escrita em stdout**
  - `print_char: char -> unit`
  - `print_string: string -> unit`
  - `print_int: int -> unit`
  - `print_float: float -> unit`
  - `print_newline: unit -> unit`
- **Primitivas para leitura de stdin**
  - `read_line: unit -> string`
  - `read_int: unit -> int`
  - `read_float: unit -> float`
- **Primitivas para abertura e fecho de canais**
  - `open_in: string -> in_channel`
  - `open_out: string -> out_channel`
  - `close_in: in_channel -> unit`
  - `close_out: out_channel -> unit`
- **Primitivas para escrita em canais**
  - `output_char: out_channel -> char -> unit`
  - `output_string: out_channel -> string -> unit`
  - `flush: out_channel -> unit`
- **Primitivas para leitura de canais**
  - `input_char: in_channel -> char`
  - `input_line: in_channel -> string`

Esta lista de primitivas não é exaustiva. Há mais primitivas listadas no manual de referência (ver Basic Operations).

Exemplo de função sobre ficheiros programada usando o método indutivo

Cópia de ficheiros:

```
(* copyChannel: copia o canal de input ci para o canal de output co *)

let rec copyChannel ci co =
    try
        let s = input_line ci in
            output_string co s ;
            output_string co "\n" ;
            copyChannel ci co
    with End_of_file -> ()
;;

(* copyFile: abre os ficheiros ni e depois usa a função copyChannel *)

let copyFile ni no =
    let ci = open_in ni in
        let co = open_out no in
            copyChannel ci co ;
```

```
            close_in ci ;
            close_out co
    ;;
```

Esquema geral de utilização do método indutivo no tratamento de ficheiros linha a linha:

```
    let rec f ci =
        try
            let s = input_line ci in
                ... f ci ...
        with End_of_file -> ...
    ;;
```

Introdução aos módulos em ML

Nenhuma linguagem moderna dispensa um sistema de módulos. Um sistema de módulos permite:

- Agrupar definições relacionadas: por exemplo permite agrupar um tipo de dados com as suas operações associadas.
- Forçar uma forma coerente de nomear essas definições para evitar conflitos de nomes.
- Ocultar a representação interna dos dados e ocultar as operações auxiliares.
- Está na base duma conceção modular de software.

O sistema de módulos do OCaml é muito rico. Para não dedicarmos excessivo tempo a este assunto, vamos ignorar quase todos os aspetos técnicos e vocabulário especializado associados ao sistema. Focamos a nossa atenção no estudo de alguns cenários de utilização que, afinal, cobrem a maioria das necessidades práticas.

Utilização de módulos existentes

Dentro do interpretador `ocaml` estão sempre disponíveis todos os módulos de biblioteca do ML (Index of modules).

Mas para aceder a módulos criados pelo utilizador dentro do interpretador `ocaml`, usa-se a diretiva `#load` assim:

```
    # #load "MySet.cmo" ;;
```

Para referenciar elementos dum módulo, podem usar-se referências qualificadas, como se exemplifica:

```
    # Sys.os_type ;;
    - : string = "Unix"

    # Sys.command "ls -l" ;;
    total 24
    -rw-r--r-- 1 amd amd  259 2008-03-11 10:26 main.ml
    -rw-r--r-- 1 amd amd  440 2008-03-11 10:28 MySet.cmi
    -rw-r--r-- 1 amd amd  654 2008-03-11 10:28 MySet.cmo
    -rw-r--r-- 1 amd amd  394 2008-03-11 10:11 MySet.ml
    -rw-r--r-- 1 amd amd  164 2008-03-11 10:11 MySet.mli
    - : int = 0

    # List.rev [1;2;3] ;;
    - : int list = [3; 2; 1]

    #
```

Mas é bastante mais prático abrir (importar) primeiro o módulo, para depois não ter de usar prefixos qualificadores:

```
    # open Sys ;;

    # os_type ;;
    - : string = "Unix"

    # command "ls -l" ;;
    total 24
    -rw-r--r-- 1 amd amd  259 2008-03-11 10:26 main.ml
    -rw-r--r-- 1 amd amd  440 2008-03-11 10:28 MySet.cmi
    -rw-r--r-- 1 amd amd  654 2008-03-11 10:28 MySet.cmo
```

```
-rw-r--r-- 1 amd amd  394 2008-03-11 10:11 MySet.ml
-rw-r--r-- 1 amd amd  164 2008-03-11 10:11 MySet.mli
- : int = 0
#
```

Criação dum módulo aberto

Para criar um módulo aberto, no qual todas as definições são públicas, basta criar um ficheiro com o nome pretendido e extensão "ml", digamos "MySet.ml". Depois é necessário compilar o módulo usando o comando

```
ocamlc -c MySet.ml
```

sendo gerados os ficheiros "MySet.cmi" e "MySet.cmo".

Exemplo de utilização do módulo aberto MySet:

```
$ ocaml
        Objective Caml version 3.09.2

# #load "MySet.cmo" ;;
# open MySet ;;
# empty ;;
- : 'a list = []
# make [1;2;3] ;;
- : int list = [1; 2; 3]
#
```

Eis o conteúdo do ficheiro "MySet.ml":

```
(* Module body MySet *)

type 'a set = 'a list ;;

let empty = [] ;;

let rec belongs v l =
    match l with
      [] -> false
    | x::xs -> x = v || belongs v xs
;;

let rec union l1 l2 =
    match l1 with
      [] -> l2
    | x::xs -> (if belongs x l2 then [] else [x])@union xs l2
;;

let rec clear l =
    match l with
      [] -> []
    | x::xs -> let cl = clear xs in
                 (if belongs x cl then [] else [x])@cl
;;

let make l =
    clear l
;;
```

Criação dum módulo fechado

Para ocultar a representação interna dos dados e as operações auxiliares é necessário criar adicionalmente um **ficheiro interface** "MySet.mli" onde se declaram todas as entidades públicas da forma que se pretende que sejam vistas do exterior. Depois compila-se o módulo assim

```
ocamlc -c MySet.mli MySet.ml
```

sendo gerados os ficheiros "MySet.cmi" e "MySet.cmo".

Exemplo de utilização do módulo fechado MySet:

```
$ ocaml
        Objective Caml version 3.09.2

# #load "MySet.cmo" ;;
# open MySet ;;
# empty ;;
- : 'a MySet.set = <abstr>
# make [1;2;3] ;;
- : int MySet.set = <abstr>
# clear [1;2;3] ;;
Unbound value clear
#
```

Eis o conteúdo do ficheiro "MySet.mli":

```
(* Module interface MySet *)

type 'a set                 (* abstract *)
val empty : 'a set
val belongs : 'a -> 'a set -> bool
val union : 'a set -> 'a set -> 'a set
val make : 'a list -> 'a set
```

Repare que neste ficheiro interface omitimos a representação interna do tipo `'a set` assim como a declaração da função auxiliar `clear`. Repare ainda na subtileza do tipo da função `make` (recebe uma lista mas retorna um set).

Quando se esconde a representação interna dum tipo de dados, diz-se que esse tipo é **abstracto** ou **opaco**.

#100

Linguagens e Ambientes de Programação (2010/2011)

Teórica 8 (14/Mar/2011)

Funções recursivas. Usando o método indutivo.

O método indutivo aplicado à resolução de problemas sobre: inteiros, listas, árvores binárias, árvores n-árias, ficheiros, strings.

Categorias de funções recursivas: funções de tipo 1, 2, 3 e 4.

Funções recursivas

Uma função recursiva bem definida faz sempre uma *análise de casos* sobre os seus parâmetros. Por exemplo, a função recursiva "fib" considera três casos relativamente ao seu parâmetro "n":

```
let rec fib n =
     if n = 0 then 1
     else if n = 1 then 1
     else fib (n-1) + fib (n-2)
;;
```

Existem sempre dois géneros de casos que uma função recursiva tem sempre de tratar: *casos base* e *casos gerais*. Os **casos base** são aqueles que não conduzem, nem directa nem indirectamente, a chamadas recursivas da função; os **casos gerais** são aqueles que conduzem, directa ou indirectamente, a chamadas recursivas da função. A função "fib" trata dois casos base, n=0 e n=1, e trata um caso geral, n>=2.

Toda a função recursiva deve tratar pelo menos um caso base e um caso geral. Além disso todas as chamadas recursivas que estão associadas aos casos gerais devem corresponder **problemas mais simples** do que o problema que a função, no seu todo, resolve (*problema mais simples* significa *problema mais próximo de algum dos casos base*). Apenas este conjunto de condições garante terminação.

Usando o método indutivo

O **método indutivo**, introduzido na aula teórica 6, ajuda a programar os casos gerais das funções recursivas. Concretamente, dá alguma orientação sobre como fazer a *redução de problemas a problemas mais simples*.

Quando se usa o método indutivo, **devem-se ter em mente** apenas as propriedades lógicas do problema a resolver. **Não se devem ter em mente** quaisquer preocupações relacionadas com as propriedades operacionais da função que está a ser programada. Misturar do uso do método indutivo com preocupações de natureza operacional é uma receita para o desastre.

De qualquer forma, depois de programada a função, já não há problema em tentar compreendê-la operacionalmente. Pode mesmo valer a pena escrever uma avaliação da função para se ganhar confiança nela. Exemplo: avaliação de (len [1;5;3;8]):

```
len [1;5;3;8] =
    1 + len [5;3;8] =
    1 + 1 + len [3;8] =
    1 + 1 + 1 + len [8] =
    1 + 1 + 1 + 1 + len [] =
    1 + 1 + 1 + 1 + 0 =
    4
```

Categorias de funções recursivas

Vamos agora estudar 4 categorias de funções recursivas, todas programadas usando o método indutivo: funções de tipo 1, de tipo 2, de tipo 3 e de tipo 4.

Na avaliação da qualidade duma função, um dos critérios que surge em primeiro lugar é o da **facilidade em compreender a função**. As funções de tipo 1 e 2 tendem a ser as mais simples de perceber, mas nem todos os problemas podem ser resolvidos directamente usando apenas funções de tipo 1 e 2.

Funções de tipo 1

As **funções de tipo 1** são programadas usando o método indutivo e caracterizam-se pela seguinte propriedade:

- A estrutura da função baseia-se num dos esquemas rígidos predefinidos, indicados a seguir. Repare que em todos esses esquemas, os argumentos das chamadas recursivas são subpadrões dos padrões que representam os argumentos da função.

Nesses esquemas, o PONTO DE PARTIDA PARA COMEÇAR A RACIOCINAR aparece sublinhado.

Orientação prática: Quando se tenta resolver um problema, convém começar por procurar construir uma função de tipo 1 já que estas são as mais fáceis de escrever e compreender, na maioria dos casos.

Inteiros

```
let rec f n =
    if n = 0 then ...
    else ... f (n-1) ...
;;
```

Listas

```
let rec f l =
    match l with
        [] -> ...
      | x::xs -> ... f xs ...
;;
```

Árvores binárias

```
let rec f t =
    match t with
        Nil -> ...
      | Node(x,l,r) -> ... f l ... f r ...
;;
```

Árvores n-árias

```
  let rec lf ts =
      match ts with
          [] -> ...
        | NNil::ts -> ... lf ts
        | NNode(x,cs)::ts -> ... lf cs ... lf ts ...
  ;;

  let f t =
      lf [t]
  ;;

  let f t =
      let r = lf [t] in
          if r = [] then NNil
          else List.hd r
  ;;
```

Ficheiros: leitura de sequência

```
let rec f ci =
   try
       let s = input_line ci in
           ... f ci ...
   with End_of_file -> ...
;;
```

Strings

```
let cut s = (String.get s 0, String.sub s 1 ((String.length s)-1)) ;;
let rec f s =
    if s = "" then ...
    else
        let (x,xs) = cut s in
            ... f xs ...
;;
```

Exemplos de funções de tipo 1: fact, len, append, rev, belongs, union, power, height, size, zeros, treeToList, balanced, subTrees, etc. (em suma, a maioria das funções estudadas até ao momento).

Mais exemplos de funções de tipo 1:

```
stringAsList: converte string em lista de caracteres
    let cut s = (String.get s 0, String.sub s 1 ((String.length s)-1)) ;;
    let rec stringAsList s =
        if s = "" then []
        else
            let (x,xs) = cut s in
                x::stringAsList xs
    ;;

sortList: ordena lista
    let rec insOrd v l =
        match l with
            [] -> [v]
          | x::xs ->
              if v <= x then v::x::xs
              else x::insOrd v xs
    ;;
    let rec sortList l =
        match l with
            [] -> []
          | x::xs ->
              insOrd x (sortList xs)
    ;;
```

Funções de tipo 2

As **funções de tipo 2** são programadas usando o método indutivo e caracterizam-se pelas duas seguintes propriedades:

- Os argumentos das chamadas recursivas são subpadrões dos padrões que representam os argumentos da função.
- Fogem aos esquemas rígidos das funções de tipo 1.

Orientação prática: Quando se tenta escrever uma função de tipo 1, por vezes descobre-se que é necessário ou conveniente fazer alguns pequenos ajustamentos ao esquema rígido de base. Desta forma somos levados a inventar uma função de tipo 2. O PONTO DE PARTIDA PARA COMEÇAR A RACIOCINAR envolve um pouco de descoberta mas, em geral, é fácil de descobrir esse ponto de partida pois os argumentos das chamadas recursivas são subpadrões dos padrões que representam os argumentos da função.

Exemplos de funções de tipo 2: maxList, fall, fib.

Mais exemplos de funções de tipo 2:

```
halfHalf: reparte os elementos duma lista por duas listas, alternadamente
     (esta versão é programada com base na ideia de processar os elementos da lista dois a
dois)
    let rec halfHalf l =
        match l with
            [] -> ([],[])
          | [x] -> ([x],[])
          | x::y::xs ->
                let (us,vs) = halfHalf xs in
                    (x::us, y::vs)
    ;;

addEvenPos: soma todos os elementos duma lista que estão em posições de índice par (0, 2, 4,
...)
```

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

Funções de tipo 3

As **funções de tipo 3** são programadas usando o método indutivo e caracterizam-se pelas seguinte propriedade:

- A chamada recursiva envolve expressões que *não são* subpadrões dos padrões que representam os argumentos da função. As expressões que constituem os argumentos da chamada recursiva são *calculados*.

Orientação prática: As funções de tipo 3 surgem geralmente quando se tenta resolver um problema aplicando uma estratégia previamente pensada, em vez de se tentar encontrar uma solução baseada nos esquemas predefinidos que caracterizam as funções de tipo 1.

Precaução: Programar uma função de tipo 3 em vez duma de tipo 1 ou 2 significa, quase sempre, **complicar desnecessariamente**. Além disso, as funções de tipo 3 são mais difíceis de perceber do que as funções de tipo 1 ou 2. Em todo o caso, ocasionalmente surge uma boa razão para se escrever uma função de tipo 3: por exemplo, o aumento da eficiência (se for caso disso, porque por vezes a função fica menos eficiente).

Exemplo de função de tipo 3:

```
quickSort: ordena lista eficientemente (reduz-se o tratamento de
              uma lista ao tratamento de duas secções dessa lista)
    let rec partition p l =
        match l with
            [] -> ([],[])
          | x::xs ->
                let (a,b) = partition p xs in
                    if x <= p then (x::a,b) else (a, x::b)
    ;;
    let rec quickSort l =
        match l with
            [] -> []
          | x::xs ->
                let (us,vs) = partition x xs in
                    (quickSort us) @ [x] @ (quickSort vs)
    ;;

minSort: ordena lista usando o algoritmo de selecção directa
    let rec removeFromList v l =
        match l with
            [] -> []
          | x::xs ->
                if x = v then xs
                else x::removeFromList v xs
    ;;
    let rec minList l =
        match l with
            [x] -> x
          | x::xs -> min x (minList xs)
    ;;
    let rec minSort l =
        match l with
            [] -> []
          | list ->
              let m = minList list in
                  m::minSort (removeFromList m list)
    ;;
```

A função `quickSort` permite ganhar eficiência. Já a função `minSort` é muito mais complicada do que a função de tipo 1 `sortList` e não é mais eficiente.

Funções de tipo 4

Uma **função de tipo 4** é uma função não recursiva que se define à custa de funções auxiliares de tipo 1, 2, ou 3.

Orientação prática: Existem problemas que não podem ser resolvidos directamente usando o método indutivo. Nestes casos surge a necessidade de escrever uma função de tipo 4. Quase sempre é preciso inventar um novo problema que já se consiga resolver usando o método indutivo e que ajude a resolver o problema original.

Exemplos de funções de tipo 4:

```
prime: determina de um inteiro é ou não primo

    let rec hasDiv n a z =
        if a > z  then false
        else (n mod a = 0) or (hasDiv n (a+1) z)
    ;;
    let prime n =
       not (hasDiv n 2 (n-1))
    ;;

width: determina a largura duma árvore binária

    type 'a tree = Nil | Node of 'a * 'a tree * 'a tree ;;

    let rec maxList l =
        match l with
            [x] -> x
          | x::xs -> max x (maxList xs)
    ;;
    let rec sumLists l1 l2 =
        match l1,l2 with
            [],l -> l
          | l,[] -> l
          | x::xs, y::ys -> (x+y)::sumLists xs ys
    ;;
    let rec levels t =
        match t with
            Nil -> []
          | Node(x,l,r) -> 1::sumLists (levels l) (levels r)
    ;;
    let width t =
         if t = Nil then 0
         else maxList (levels t) ;;
```

Os dois problemas anteriores não são indutivos. Porquê?

#90

Linguagens e Ambientes de Programação (2010/2011)

Teórica 09 (16/Mar/2011)

Plataformas computacionais

Implementação de linguagens de programação: Compilação e Interpretação

Técnicas de implementação mistas. Máquinas virtuais.

Níveis de interpretação.

Detalhes da implementação de interpretadores.

Detalhes da implementação de compiladores.

Ligação.

Carregamento.

Plataformas computacionais

É demasiado simplista dizer que os programas correm sobre hardware. É mais rigoroso dizer que os programas correm sobre uma **plataforma computacional**.

Uma plataforma é constituída pelos seguintes elementos:

- **Hardware** - Inclui: processador, memória, canais de entrada e saída;
- **Sistema operativo** - Software que faz a gestão dos recursos do computador e que fornece aos programas uma interface para aceder a esses recursos.
- **Bibliotecas** - Sã componentes de software opcionais que fornecem aos programas interfaces especializadas e de mais alto-nível para o sistema operativo.

Exemplos de plataformas:

- Intel-32/Microsoft Windows
- Intel-32/Mac OS
- PowerPC/Mac OS
- Intel-32/Linux
- Mips/Linux
- Sparc/Linux
- Java (é independente do hardware)
- .Net (é independente do hardware)

Implementação de linguagens de programação: Compilação e Interpretação

A generalidade das linguagens de programação suporta conceitos e abstrações mais sofisticados do que os mecanismos suportados pelas plataformas computacionais usuais. Assim, para conseguir executar numa plataforma computacional programas escritos numa linguagem de alto-nível é preciso recorrer a uma das duas seguintes técnicas, ou a uma mistura das duas:

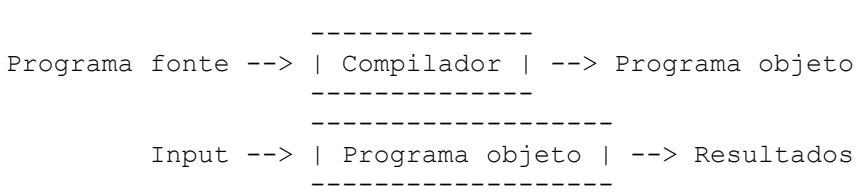
- Compilação
- Interpretação

Compilação

Um **compilador** é um programa tradutor com as seguintes características:

- Converte programas duma linguagem de programação de alto-nível para programas equivalentes escritos numa linguagem de mais baixo nível.
- A linguagem de mais baixo nível é geralmente linguagem-máquina, o que já permite que cada programa possa ser executado diretamente na plataforma computacional.
- O programa executável, gerado pelo compilador, pode ser corrido as vezes que se quiser, sem ser preciso voltar a usar o compilador.

O seguinte diagrama descreve a situação:



Alguns exemplos. Na plataforma Intel/Linux usada nas aulas práticas estão disponíveis os seguintes compiladores:

- Compilador de OCaml chamado **ocamlopt**.
- Compilador de C/C++ chamado **gcc**.
- Compilador de Java chamado **gcj**.

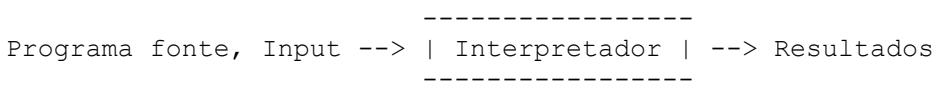
Todos estes compiladores geram código nativo.

Interpretação

Um **interpretador** é um programa "executor" com as seguintes características:

- Executa diretamente o programa fonte.
- Para correr o programa novamente é necessário voltar a usar o interpretador.

O seguinte diagrama descreve a situação:



Alguns exemplos. Na plataforma usada nas aulas práticas está pelo menos disponível um interpretador:

- Um interpretador de OCaml chamado **ocaml**.

Vantagens/Desvantagens

Vantagens de usar um compilador:

- Grande velocidade de execução (depois da compilação ter terminado).
- Os programas podem ser distribuídos sob a forma executável, sem ser necessário fornecer uma cópia do compilador.

Os ganhos de eficiência dos compiladores devem-se à seguinte razão:

- O compilador toma muitas decisões ao processar o programa fonte (considere por exemplo o acesso a variáveis). O ganho está no facto dessas decisões já não terem de ser tomadas novamente durante a execução de programa. Além disso, se o programa contiver ciclos ou funções recursivas, esta questão torna-se ainda mais importante - um interpretador estaria sempre, repetidamente, a tomar as mesmas decisões.

Vantagens de usar um interpretador:

- Maior rapidez na escrita e teste de programas, pois podem correr-se os programas diretamente, sem esperar pela compilação.
- Melhores diagnósticos de erros.
- Maior potencial para escrever *source-level debuggers* de qualidade.
- Maior facilidade em suportar linguagem reflexivas, ou seja linguagens em que os programas se podem observar a eles mesmos e auto-modificarem-se durante a execução. Exemplos de linguagens destas: Prolog, Lisp.

Técnicas de implementação mistas

São bem claras as diferenças entre os conceitos de compilação e interpretação. Contudo a implementação de muitas linguagens de programação acaba por baseada numa mistura de compilação e de interpretação.

Vejamos três situações diferentes:

Coexistência de código compilado e de código interpretado

Algumas implementações da linguagem Prolog permitem a coexistência de código compilado com código interpretado.

Predicados declarados como estáticos (i.e. que não podem ser alterados em tempo de execução) são normalmente compilados; predicados declarados como dinâmicos (i.e. que podem ser modificados em tempo de execução) são obrigatoriamente interpretados.

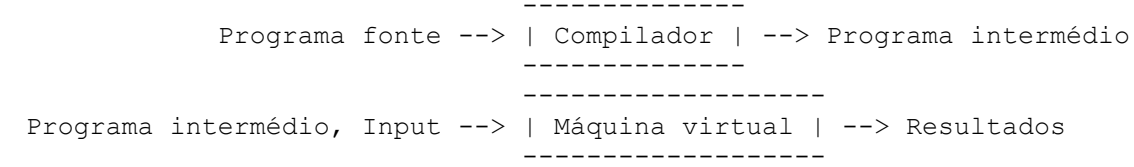
A vantagem desta técnica é que permite obter velocidade de execução nas partes estáticas do programa, e a flexibilidade necessária nas partes dinamicamente modificáveis do programa.

Máquinas virtuais

Outra forma de implementação mista de linguagens de programação envolve a invenção duma linguagem intermédia para a qual se escreve um interpretador. O termo **máquina virtual** é muitas vezes usado, tanto para designar a linguagem intermédia, como o seu interpretador.

Agora basta escrever um compilador para traduzir programas da linguagem de alto-nível para código intermédio. O resultado da tradução pode depois ser executado na máquina virtual.

O seguinte diagrama descreve a situação:



É muito importante que a máquina virtual seja criada com três objetivos em mente:

- As suas particularidades devem facilitar a tradução dos programas fonte para código intermédio.

- Deve ser simples escrever um interpretador razoavelmente eficiente para ela.
- Deve favorecer a compacidade do código intermédio.

Vantagens da técnica da máquina virtual:

- O compilador torna-se mais simples de escrever.
- A velocidade de execução é mediana: melhor do que num interpretador puro mas pior do que num compilador nativo.
- Os programas intermédios podem ser executados em qualquer plataforma onde a máquina virtual esteja disponível.
- Os programas intermédios são geralmente bastante compactos pelo que podem ser transferidos através da WEB mais rapidamente.

Eis alguns exemplos de implementações baseadas em máquinas virtuais:

- Compilador de OCaml chamado **ocamlc** que gera código para uma máquina virtual chamada **CAML** (Categorical Abstract Machine Language).
- Compilador de Java chamado **javac** que gera código para uma máquina virtual chamada **JVM** (Java Virtual Machine).
- Compilador de C# chamado **mcs** que gera código para uma máquina virtual chamada **CLR** (Common Language Runtime).

Máquinas virtuais just-in-time

Quando é muito importante tornar a implementação duma máquina virtual tão rápida quanto possível, torna-se necessário implementar o respetivo interpretador usando a técnica **just-in-time** (JIT), também conhecida por **tradução dinâmica**.

A ideia da técnica é simples, embora a implementação seja complicada de fazer. Durante a execução do programa intermédio, este vai sendo dinamicamente traduzido em código-máquina que é imediatamente executado. A tradução dinâmica é aplicada a unidades a unidades de código, tais como métodos ou função, e a implementação gere uma cache de unidades de código já processadas. Ao fim de algum tempo de execução, quando a maior parte do código já foi traduzido, consegue-se atingir uma velocidade de execução que ronda os 70% da velocidade de execução dum programa compilado nativamente. Alguns exemplos de máquinas virtuais para as quais existem versões JIT:

- **JVM** - Java Virtual Machine.
- **CLR** - Common Language Runtime.

Ponto de vista externo e interno

As técnicas de implementação mistas, fazem esbater a fronteira entre as noções de compilação e de interpretação. Por vezes, para caracterizar a implementação duma linguagem com rigor, temos de considerar dois pontos de vista:

- Um **ponto de vista externo** que considera a implementação dos compiladores e interpretadores como caixas negras.
- Um **ponto de vista interno** que se interessa pela implementação interna dos compiladores e interpretadores.

Para discutir os dois pontos de vista, vamos considerar uma máquina virtual just-in-time, por exemplo a JVM:

- Do ponto de vista externo, uma máquina virtual just-in-time é um simples interpretador, pois constitui um programa que permite executar diretamente programas (neste caso escritos na chamada linguagem intermédia).
- Do ponto de vista interno, uma máquina virtual just-in-time é um complexo sistema que intercala a execução dum compilador nativo com a execução de porções de código máquina diretamente pelo hardware.

Consideremos agora o interpretador de OCaml **ocaml**, muito usado nas nossas aulas práticas.

- Do ponto de vista externo, trata-se dum interpretador, pois é visível que ele permite executar diretamente programas escritos em OCaml.
- No entanto, consultando a documentação, aprende-se que, do ponto de vista interno, é um **compilador/interpretador em duas fases**, pois ele compila internamente os programas fonte para código da máquina virtual CAML, sendo o código gerado internamente imediatamente executado pela máquina virtual.

Consideremos agora a arquitetura de hardware Intel-32. Claramente um processador da Intel implementa uma linguagem que se chama linguagem máquina. Será que deve ser encarado como um compilador ou um interpretador.

- Do ponto de vista externo, trata-se dum interpretador, pois é notório que ele permite executar diretamente programas escritos em linguagem máquina.
- No entanto, o processador compila internamente a linguagem máquina para uma linguagem de mais baixo nível chamada de micro-código. Depois, é o micro-código que efetua o trabalho útil do programa, ao ser executado por um interpretador implementado em hardware. São as instruções de micro-código que atuam sobre os circuitos físicos do processador.

Níveis de interpretação

Vejamos os três níveis de interpretação envolvidos na execução dum programa em Java sobre uma JVM implementada em hardware da Intel:

1. Interpretador de micro-código implementado dentro do processador físico.
2. Interpretador de código-máquina - é o processador físico.
3. Interpretador da JVM - implementado por software.

Podem existir níveis de interpretação ainda menos elevados:

- A arquitetura da Motorola MC68000 inclui nano-código, a mais baixo nível do que o micro-código.

E podem existir níveis de interpretação ainda mais elevados:

- Uma forma popular de implementar a linguagem Prolog é através da escrita dum interpretador em Java (porque isso facilita a interoperabilidade das duas linguagens).
- O nosso CLIP está implementado numa linguagem chamada Compass. Esta linguagem está implementada como um interpretador sobre a linguagem em Prolog.

Detalhes da implementação de interpretadores

Já sabemos que um interpretador é um programa que executa diretamente programas fonte.

Código interpretado é geralmente mais lento (por vezes, 10 vezes mais lento!) do que código compilado. Algumas razões:

- O interpretador tem de analisar cada parte do programa antes de a executar, e tem de a voltar a analisar sempre que a quiser voltar a executar (por exemplo, num ciclo). Por seu lado o compilador executa o código diretamente, sem perdas de tempo, pois a analise do programa já foi efetuada anteriormente, em tempo de compilação.

- Outra razão pela qual um interpretador é mais lento prende-se com o acesso às variáveis do programa. Um interpretador gasta imenso tempo a mapear dinamicamente os nomes das variáveis do programa em células de memória (geralmente com a ajuda duma hash table). Por seu lado um compilador associa endereços fixos às variáveis do programa, pelo que o acesso às variáveis é praticamente instantâneo em tempo de execução.

Um aspeto importante da interpretação é a forma como o interpretador lida com o texto do programa fonte. Há varias técnicas que têm sido usadas historicamente.

Interpretação direta

Nos interpretadores mais rudimentares, o texto é carregado em memória e é usado diretamente como matéria prima para a execução dos programas.

Os primeiros interpretadores de Basic, em meados dos anos 60, foram feitos desta forma. O manual da linguagem recomendava que não se escrevessem muitos comentários para os programas correrem mais rapidamente. Com efeito, os mesmos comentários eram lidos (e ignorados) sempre que uma dada parte do programa era executada (por exemplo, num ciclo).

Também convinha escolher variáveis com nomes curtos, para acelerar a execução dos programas.

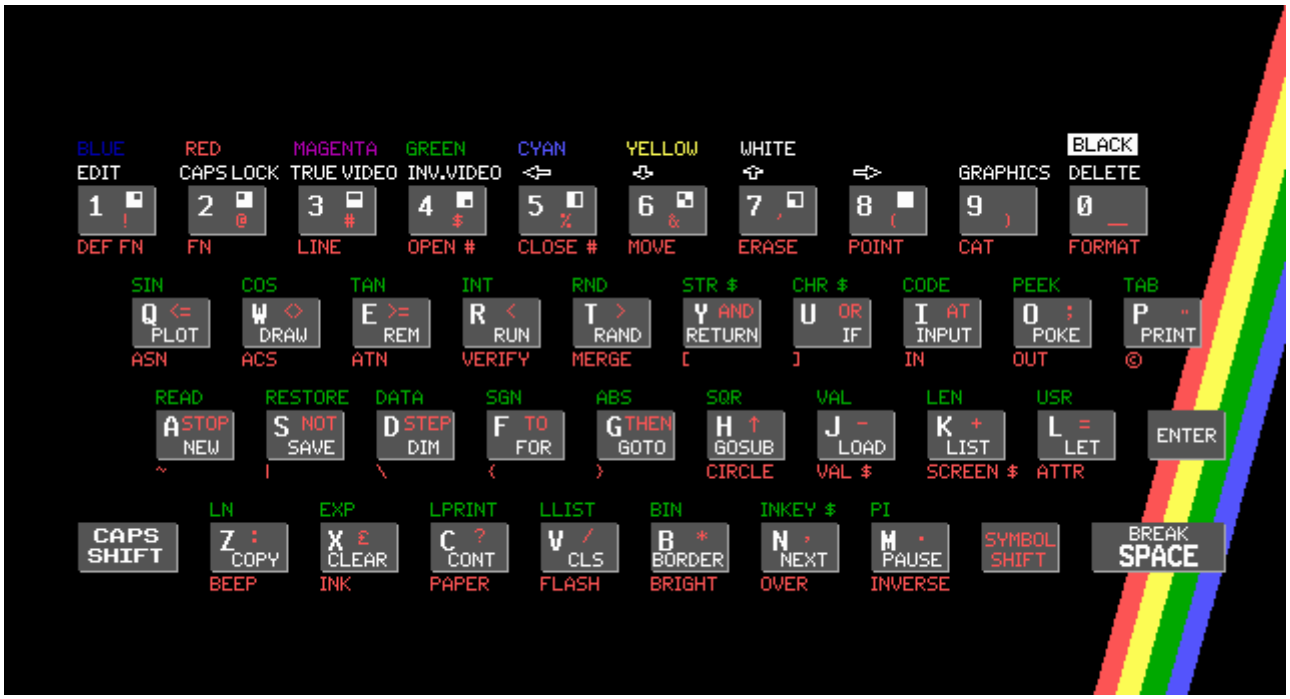
Preprocessamento com identificação de tokens

A maioria dos interpretadores faz algum processamento do ficheiro fonte no momento do carregamento. Tipicamente, os comentários e espaços em branco são removidos, e todas as sequências de caracteres significativas são agrupadas em **tokens**, ou seja em símbolos que representam palavras reservadas, números, nomes de variáveis, etc.

Durante a interpretação, a matéria prima para a execução dos programas já não são sequências de caracteres, mas sim sequências de tokens. Consegue-se assim obter maior eficiência

O Basic do ZX Spectrum

Uma variante curiosa desta técnica foi usada na implementação do Basic do ZX Spectrum. O ZX Spectrum foi um computador pessoal muito popular na Europa durante os anos 80. Os programas podiam ser carregados a partir duma cassete áudio ou ser metidos à mão. Quando os programas eram metidos à mão, o editor de texto obrigava o utilizador a introduzir as palavras reservadas como tokens, o que era possível devido ao curioso teclado.



Preprocessamento com identificação de árvore sintática

Muitos interpretadores modernos efetuam um processamento bastante sofisticado do ficheiro fonte, no momento do carregamento. Além de identificarem todos os tokens, também identificam a estrutura sintática do programa de entrada e constroem em memória a correspondente **árvore sintática** (*parse tree*).

Durante a interpretação, a matéria prima para a execução dos programas é a árvore sintática. Consegue-se assim obter ainda maior eficiência.

Interpretador ou compilador?

Note que a identificação dos tokens e a construção da árvore sintática correspondem às duas primeiras fases de processamento dos compiladores tradicionais. Realmente, para aumentar a eficiência dum interpretador é preciso integrar nele alguma funcionalidade típica dos compiladores. Prosseguindo nesta linha de integrar mais e mais funcionalidade dos compiladores num interpretador, em breve se chega à técnica mista das máquinas virtuais, estudada da aula anterior. Do ponto de vista interno, já deixámos de ter um interpretador, e passámos a ter um sistema misto constituído por um compilador que gera código intermédio e um interpretador duma máquina virtual.

Detalhes da implementação de compiladores

Num compilador típico, a compilação evolui ao longo duma série de fases. Cada fase descobre informação que é necessária nas fases seguintes, ou então transforma o programa numa forma que é requerida pela fase seguinte.

1. Leitura de caracteres do ficheiro fonte

   sequência de caracteres ->

2. Análise lexical (scanner)

   -> sequência de tokens ->

3. Análise sintática (parser)

   -> árvore de parsing ->

4. Análise semântica

   -> árvore Abstract ->

5. Melhoramento e completação da árvore Abstract

   -> árvore Abstract melhorada ->

6. Geração de código intermédio

   -> código intermédio ->

7. Otimização de código intermédio

   -> código intermédio otimizado ->

8. Geração de código máquina

-> código máquina executável

Ligação

Para simplificar as discussões anteriores relativas a compiladores, temos vindo a omitir qualquer referência à questão da **ligação de ficheiros objeto**. Vamos tratar agora dessa questão que não pode ser ignorada quando se discute o tema da implementação de linguagens de programação.

Código fonte e código objeto

Um programa é muitas vezes constituído por diversos **ficheiros fonte**, sendo o conjunto habitualmente designado por **código fonte** do programa. Exemplos:

```
a.c b.c    --- código fonte dum programa em C
a.ml b.ml  --- código fonte dum programa em ML
```

Quando o compilador processa o código fonte dum programa, o compilador gera um **ficheiro objeto** distinto por cada ficheiro fonte. Exemplos:

```
gcc -c a.c b.c      --> a.o b.o
ocamlc -c a.ml b.ml --> a.cmo b.cmo
```

O conjunto dos ficheiros objeto chama-se **código objeto**.

Ligador

Depois de compilado o código fonte, para se obter o programa executável é preciso usar um programa **ligador** que junta os diversos ficheiros objeto num único programa executável. Na altura de ligar um programa é preciso também indicar quais são as bibliotecas (arquivos de ficheiros objetos predefinidos) que interessa juntar ao programa.

Em Linux, o ligador chama-se `ld` e pode ser invocado diretamente pelo utilizador, se tal for desejado. Mas geralmente é mais prático invocar indiretamente o ligador através do comando de compilação. Em todo o caso, veja o que diz o início do manual do comando `ld`:

```
$ man ld

LD(1)                       GNU Development Tools                        LD(1)

NAME
       ld - The GNU linker

SYNOPSIS
       ld [options] objfile ...

DESCRIPTION
       ld  combines a number of object and archive files, relocates their data
       and ties up symbol references. Usually the last  step  in  compiling  a
       program is to run ld.
```

No Linux usa-se a opção "-o" na linha de comando dos compiladores serve para invocar indiretamente o ligador como último passo da compilação. Exemplos:

```
ocaml -o prog a.cmo b.cmo  --> prog
gcc -o prog a.o b.o        --> prog
```

Outra missão do ligador é resolver as referências cruzadas de nomes globais que podem ocorrem nos diversos ficheiros object. Para perceber o que está em causa vamos observar a tabela de símbolos e informação de relocação que é guardada dentro do ficheiro object correspondente ao seguinte ficheiro fonte "a.c":

```
/* File a.c */
/* This is a C source file */
```

```
#include <stdio.h>

extern int g(int) ;
extern int global_e ;

int global_x, global_y, global_z ;
int global_a = 123, global_b = 123, global_c = 124 ;

int f(int x)
{
    return x + global_a ;
}

int main()
{
    printf("%d", f(2) + g(3)) ;
    return 0 ;
}
```

O comando `nm` do Linux mostra os símbolos e informação de relocação de ficheiros-objeto individuais. Aplicando `nm` ao ficheiro object "a.o", obtém-se o seguinte:

```
amd@sunra2:z$ nm a.o

00000000 T f
         U g
00000000 D global_a
00000004 D global_b
00000008 D global_c
00000004 C global_x
00000004 C global_y
00000004 C global_z
0000000d T main
         U printf
```

Carregamento

O **carregador** é a componente do núcleo do sistema operativo que trata de carregar em memória os programas para depois os executar.

No Linux, `execve` é o nome da chamada ao sistema que permite usar o carregador. Leia o que diz o início do manual do comando `execve`:

```
$ man execve

EXECVE(2)                  Linux Programmer's Manual                 EXECVE(2)

NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int execve(const char *filename, char *const argv[],
                  char *const envp[]);

DESCRIPTION
       execve() executes the program pointed to by filename.  filename must be
       either a binary executable, or a script starting with  a  line  of  the
       form:

            #! interpreter [optional-arg]

       For details of the latter case, see "Interpreter scripts" below.

       argv  is  an array of argument strings passed to the new program.  envp
```

```
       is an array of strings, conventionally of the form key=value, which are
       passed  as  environment to the new program.  Both argv and envp must be
       terminated by a null pointer.  The argument vector and environment  can
       be  accessed  by the called program's main function, when it is defined
       as:

            int main(int argc, char *argv[], char *envp[]).

       execve() does not return on success, and the text, data, bss, and stack
       of the calling process are overwritten by that of the program loaded.
```

Em algumas plataformas, e.g. System/360 da IBM, o carregador tem a tarefa de efetuar relocação de endereços, porque o hardware só suporta endereçamento absoluto.

Nas plataformas modernas está geralmente disponível um segundo carregador - um **carregador dinâmico** - que permite carregar bibliotecas dinâmicas a meio da execução dum programa. Em Windows as bibliotecas dinâmicas são guardadas em ficheiros com extensão "dll". No Linux a extensão das bibliotecas dinâmicas é "so".

#80

Linguagens e Ambientes de Programação (2010/2011)

Teórica 10 (21/Mar/2011)

Eficiência - otimização da última chamada.

Funções de ordem superior sobre listas.

Eficiência - otimização da última chamada (tail call optimization)

Iteração e recursão em Java

Esqueçamos agora por alguns momentos a linguagem OCaml. A discussão que se segue é feita no contexto da linguagem Java.

Pode obter-se **repetição** em Java de duas formas: usando **iteração** ou usando **recursão**.

O seguinte método calcula o somatório `1 + 1 + 1 + ...` n vezes usando iteração:

```
int count(int n) {
    int a = 0 ;
    for( int i = 0 ; i < n ; i++ )
        a++ ;
    return a ;
}
```

O seguinte método calcula o mesmo resultado, mas agora usando recursão:

```
int countR(int n) {
    if( n == 0 ) return 0 ;
    else return 1 + countR(n-1) ;
}
```

Vamos comparar a eficiência das duas funções:

- Do ponto de vista da **velocidade de execução**, ambas as funções têm complexidade linear O(n). Contudo, há um fator constante muito grande a separar a eficiência das funções. Na verdade, em Java, a versão recursiva tende a ser 10 vezes mais lenta do que a segunda por causa das complicações envolvidas na implementação da recursividade.
- Do ponto de vista do **uso da memória**, a versão recursiva perde de forma ainda mais evidente: enquanto a versão iterativa tem complexidade constante O(1), a versão recursiva tem complexidade linear O(n). Realmente, cada chamada recursiva implica sempre algum gasto de memória (criação dum *registo de ativação* na chamada *pilha de execução*) e acontece até que, para argumentos grandes, a execução da função recursiva aborta por falta de memória.

Em Java nunca há qualquer hesitação: se um problema pode ser resolvido de forma simples usando iteração, não há qualquer razão para usar recursão.

Recursão em OCaml e outras linguagens funcionais

Nas linguagens funcionais, a questão anteriormente discutida é muito relevante, porque a filosofia dessas linguagens só admite a utilização de recursão.

Por exemplo, reescrevendo a função anterior em OCaml e testando, é fácil verificar que a sua execução também aborta para argumentos muito grandes.

```
let rec countR n =
    if n = 0 then 0
    else 1 + countR (n-1)
;;
```

Otimização da última chamada

Para minorar o preço a pagar pela recursão, os implementadores de linguagens funcionais descobriram uma técnica de implementação que dá bons resultados, embora só possa ser aplicada numa situação muito particular.

Trata-se da técnica da **otimização da última chamada**. Como o nome indica, esta técnica pode ser aplicada na última chamada que é feita dentro duma função, antes de retornar. A última chamada é tratada assim:

- Em vez de se criar um novo registo de ativação para tratar a última chamada, o que faz é reaproveitar o registo de ativação da própria função e simplesmente saltar para a função chamada (depois de se passarem os argumentos, claro). Repare que nesta situação não é necessário guardar qualquer *endereço de retorno*, e por isso este esquema funciona.

Portanto, a última chamada pode ser implementada sem gastar memória e poupando algum tempo de execução.

O OCaml e a generalidade das linguagens funcionais implementam esta técnica. O Java não a implementa porque não precisa - já dispõe de iteração.

Mas repare que a função OCaml anterior beneficia muito pouco desta técnica de implementação. A última chamada efetuada é uma soma. O que convinha era que a última chamada fosse a chamada recursiva, para se poder executar a função sem gastar memória.

Bem, isso pode fazer-se. Basta reescrever a função como se mostra abaixo, usando uma função auxiliar com um argumento suplementar que serve para acumular o resultado final:

```
let rec countX n a =
    if n = 0 then a
    else countX (n-1) (a+1)
;;
let count n =
    countX n 0
;;
```

Experimente! Por maior que seja o argumento passado para a função `count`, a sua execução nunca aborta por falta de memória.

Preço a pagar

Infelizmente a função `count` é muito menos legível do que a original. A parte pior é ter um sabor imperativo, no sentido em que para perceber o que a função faz é preciso executá-la mentalmente.

Será que, nas linguagens funcionais, somos forçados a escrever este tipo de código em programas reais, que lidam com valores grandes, listas grandes, etc.?

A resposta é "sim", mas só se os argumentos forem mesmo muito grandes. Para ver as magnitudes envolvidas, façamos a experiência de compilar o seguinte programa usando os compiladores `ocamlc` e `ocamlopt`:

```
let rec countR n =
    if n = 0 then 0
    else 1 + countR (n-1)
;;

let rec test n =
    print_int (countR  n) ;
    print_string "\n" ;
    flush stdout ;
    test (2 * n)
;;

test 100000 ;;
```

Correndo o programa gerado com o `ocamlc`, obtém-se:

```
100000
200000
Fatal error: exception Stack_overflow
```

Correndo o programa gerado com o `ocamlopt`, obtém-se:

```
100000
200000
400000
800000
1600000
Fatal error: exception Stack_overflow
```

Algumas funções de ordem superior sobre listas

Num programa que manipule listas, muitas vezes há vantagem algumas destas funções.

Programar bem numa linguagem funcional envolve duas coisas:

- Saber usar bem o método indutivo;
- Sabem reconhecer oportunidades de utilização das funções de biblioteca mais importantes.

Função `map`

Aplica uma função `f: 'a -> 'b` a todos os elementos duma lista, para produzir a lista das imagens.

A função `f` define uma relação de um-para-um, pelo que a lista dos resultados tem o mesmo comprimento da lista de entrada.

```
(* map : ('a -> 'b) -> 'a list -> 'b list *)
let rec map f l =
    match l with
        [] -> []
      | x::xs -> (f x) :: map f xs
;;
```

Esta função está disponível na biblioteca do OCaml como **`List.map`**.

Função `flatMap`

Aplica uma função `f: 'a -> 'b list` a todos os elementos duma lista, para produzir a lista das imagens.

A função `f` define uma relação de um-para-n, pelo que a lista dos resultados tem um comprimento sem qualquer relação com comprimento da lista de entrada.

```
(* flatMap : ('a -> 'b list) -> 'a list -> 'b list *)
let rec flatMap f l =
    match l with
        [] -> []
      | x::xs -> (f x) @flatMap f xs
;;
```

Esta função não está diretamente disponível na biblioteca do OCaml, pode ser obtida como a combinação de **`List.flatten`** com **`List.map`**. Concretamente, a seguinte definição também é válida.

```
(* flatMap : ('a -> 'b list) -> 'a list -> 'b list *)
let rec flatMap f l =
    List.flatten (List.map f l)
;;
```

Função `for_all`

Testa se todos os elementos duma lista satisfazem um dado predicado `p: 'a -> bool`.

```
(* for_all : ('a -> bool) -> 'a list -> bool *)
let rec for_all f l =
    match l with
        [] -> true
      | x::xs -> (f x) && for_all f xs
;;
```

Esta função está disponível na biblioteca do OCaml como **`List.for_all`**.

Função `count_all`

Conta todos os elementos duma lista satisfazem um dado predicado `p: 'a -> bool`.

```
(* count_all : ('a -> bool) -> 'a list -> int *)
let rec count_all f l =
    match l with
        [] -> 0
      | x::xs -> (if f x then 1 else 0) + count_all f xs
;;
```

Esta função não está disponível na biblioteca do OCaml.

Mais funções de ordem superior sobre listas

`List.exists` : `('a -> bool) -> 'a list -> bool` (* Testa se pelo menos um dos valores duma lista satisfaz um dado predicado. *)

`List.filter` : `('a -> bool) -> 'a list -> 'a list` (* Seleciona os elementos duma lista que satisfazem um dado predicado. *)

Função de teste de pertença a uma lista ("membership")

`List.mem` : `'a -> 'a list -> bool` (* Testa se um valor pertence a uma lista. *)

Problemas envolvendo quantificação universal

Verificar se todos os elementos duma lista são pares:

```
let allEven l =
    List.for_all (fun x -> x mod 2 = 0) l
;;
```

Testar se todos os elementos duma lista são menores do que os elementos de outra lista:

```
let allLess l1 l2 =
    List.for_all (fun x ->
          List.for_all (fun y -> x < y) l2) l1
;;
```

E se forem três listas?

```
let allLess3 l1 l2 l3 =
    List.for_all (fun x ->
          List.for_all (fun y ->
             List.for_all (fun z -> x < y && y < z) l3) l2) l1
;;
```

Problemas sobre tabuleiros do jogo das damas

Algumas funções básicas

```
let boardSize = 8 ;;
let maxPos = boardSize - 1 ;;

let direction player =
    if player = black then 1
    else if player = white then -1
    else raise (Arg.Bad "playerToDelta")
;;

let inside (li,co) =
     0 <= li && li <= maxPos && 0 <= co && co  <= maxPos
;;

let nextPos direction (li,co) =
    List.filter inside [(li+direction,co-1); (li+direction,co+1)]
;;
```

Atenção: na função anterior foi corrigida.

Gerar todas as posições dum tabuleiro

```
let rec range a b =
    if a > b then []
    else a::range (a+1) b
;;
let allIdx =
    range 0 maxPos
```

```
;;
let allPos =
    List.flatten (List.map (fun li -> List.map (fun co -> (li,co)) allIdx) allIdx)
;;
```

Selecionar todas as posições dum tabuleiro que verificam uma determinada propriedade

```
let filterBoard f board  =
    List.flatten (List.map (fun p -> if f board p then [p] else []) allPos)
;;
```

Exemplo de utilização: todas as posições com uma peça da cor `player`:

```
let playerPos player board  =
    filterBoard (fun b (li,co) -> get b li co = player) board
;;
```

#90

Linguagens e Ambientes de Programação (2010/2011)

Teórica 11 (23/Mar/2011)

Ligações, ambientes, âmbitos. Regras de escopo.

Ligações (bindings)

Uma ligação consiste numa associação entre duas entidades. As **entidades** podem ser nomes, localizações de memória, tipos, objetos, etc.

Uma ligação é sempre unidirecional e associa uma entidade mais simples a outra entidade mais complexa. O caso das duas entidades terem complexidade semelhante também ocorre por vezes. As ligações facilitam o trabalho do programador pois ele pode usar a entidade mais simples para representar a entidade mais complexa.

Por exemplo:

- Ligação dum nome a um valor fixo, que pode ver um inteiro, uma função, etc. (constante).
- Ligação dum nome a uma localização de memória (variável mutável).
- Ligação duma localização de memória a outra localização de memória (apontador).
- Ligação dinâmica dum nome a um valor (variável mutável).

Diferentes ligações duma entidade

Para uma mesma entidade, digamos um nome, podem estar definidas diferentes ligações. Por exemplo, em C o nome duma variável global tem as seguintes ligações:

- Ligação a um tipo fixo (estabelecida em tempo de compilação).
- Ligação a uma localização de memória fixa (estabelecida em tempo de carregamento).

- Ligação a um valor (estabelecida em tempo de execução).

Semântica e ligações

A semântica de qualquer linguagem de programação é determinada de forma essencial a partir:

- O conjunto de formas de ligação que se podem estabelecer;
- Do momento em que essas ligações se estabelecem.

Momento da ligação

Eis diversos momentos em que uma ligação pode ser estabelecida:

- Tempo de conceção da linguagem. Exemplo em OCaml: operador "+".
- Tempo de implementação da linguagem Exemplo em OCaml: constante "max_int".
- Tempo de compilação. Exemplo em C: constante "#define A 2"
- Tempo de ligação. Exemplo em C: função externa "extern int f(void) ;".
- Tempo de carregamento. Exemplo em C: variável global "int x ;".
- Tempo de execução. Exemplo em OCaml: constante "let a = 2 + x in ...".

Em geral, quanto mais tarde se estabelecem as ligações mais flexível é a linguagem.

Em geral, quanto mais cedo se estabelecem as ligações, mais rápida é a linguagem.

Por exemplo, é mais eficiente invocar um procedimento em C (ligação em tempo de ligação) do que enviar uma mensagem para um objeto em C++ (ligação em tempo de execução). Em todo o caso, justifica-se o preço a pagar em C++: é o facto da ligação entre a mensagem e o método ser estabelecida muito tarde (late binding) que permite o paradigma orientado pelos objetos.

Classificação das ligações

Classificação das ligações em função do momento de ligação:

- **Estáticas**: efetuadas antes da execução do programa.
- **Semi-dinâmicas**: efetuadas em tempo de execução mas determinadas em grande parte antes de o programa começar a correr.
- **Dinâmicas**: efetuadas completamente em tempo de execução.

Exemplos de ligações estáticas:

- nome->valor: constantes "#define" em C;
- nome->tipo: tipos, variáveis e constantes em ML e C; (ML->inferência de tipos)
- nome->localização: variáveis globais em C.

Exemplos de ligações semi-dinâmicas:

- nome->localização: variáveis locais em C e constantes locais em ML;
- nome->método: mensagens em C++ e Java.

Discussão do caso das variáveis locais em C, um caso de ligação semi-dinâmica: Essas variáveis residem na chamada *pilha de execução* do C, e as respetivas localizações de memória são determinadas em tempo de execução. No entanto, é em tempo de compilação que se determina o *offset* dentro do registo de cativação da função onde a variável reside. As constantes locais do ML "let a = 2 + x in ..." também são implementadas da mesma forma.

Exemplos de ligações dinâmicas:

- localização->valor: variáveis mutáveis em C;
- localização->localização: apontadores em C;
- nome->valor: variáveis mutáveis em qualquer linguagem;
- nome->tipo: variáveis em Smalltalk e Ruby.

Discussão do caso das variáveis em Smalltalk e Ruby: Estas variáveis não têm tipos estáticos associados. Aceitam valores de qualquer tipo e portanto, sempre que há uma atribuição muda o valor e o tipo dessas variáveis.

Tempo de vida duma ligação

O tempo de vida duma ligação é o período de tempo da execução dum programa durante a qual essa ligação persiste. As ligações estáticas persistem durante a execução de topo o programa. As ligações semi-dinâmicas e dinâmicas persistem geralmente apenas durante parte da execução do programa.

Ambiente (conjunto de ligações para nomes)

O conceito de ambiente tem a ver com um tipo particular de ligações: as ligações de nomes.

Chama-se **ambiente** a um conjunto de ligações que associam nomes (identificadores) a entidades. Matematicamente um ambiente é uma função de nomes para entidades, ou seja uma função com o seguinte tipo:

```
Nomes -> Entidades
```

O seguinte pequeno programa em ML define quatro ambientes diferentes, consoante o ponto do programa que for considerado:

```
let f x = x + 1 ;;
let rec g x = f x + 1 ;;
```

1. O ambiente antes da função f inclui apenas as ligações dos nomes predefinidos na linguagem ML. São exemplos desses nomes: max, "^", "+", int, float.
2. O ambiente no interior da função f inclui as ligações dos nomes predefinidos, mais a a ligação do nome "x" que representa o argumento de f. Repare que o nome "f" não tem ligação dentro da função f, porque esta função não é recursiva.
3. O ambiente no interior da função g inclui as ligações dos nomes predefinidos, mais a ligação do nome "x" que representa o argumento de g, mais os nomes "f" e "g" que representam funções.
4. O ambiente após a função g inclui as ligações dos nomes predefinidos, mais os nomes "f" e "g" que representam funções.

Âmbito (escopo) duma ligação

Âmbito (escopo) duma ligação é a região do programa na qual esse nome tem os atributos estabelecidos pela declaração que introduz a ligação.

Na maior parte das linguagens de programação, o âmbito duma ligação é determinado pela estrutura sintática do programa (ver "Escopo estático", mais abaixo).

Há exemplos de âmbitos na secção "Blocos", um pouco mais abaixo.

Construções definidoras de âmbitos

A generalidade das linguagens de programação possuem construções que têm implicações nos âmbitos das ligações que se estabelecem nos programas. Eis alguns exemplos dessas construções:

- Blocos (C, C++, Java, Pascal, Ada)
- Let-in (OCaml)

- Módulos (Módula-2, OCaml)
- Classes (C++, Java, C#)
- Packages (Java)
- Namespaces (C++, C#)
- Espaço global (duma forma ou de outra, todas as linguagens dispõem dum espaço global de nomes)

Blocos

As linguagens que descendem do antigo Algol-60 possuem uma construção sintética chamada **bloco**. Um bloco tem duas utilidades:

- Serve para introduzir um novo ambiente no qual todas as novas ligações têm aproximadamente o mesmo âmbito. (Esta é a parte que nos interessa aqui.)
- Serve para agregar uma sequência de comandos num comando naquilo que, tecnicamente, é um comando único, dito composto.

O seguinte bloco, em C, determina ligações para os nomes `i`, `j` e `k`. Todas essas ligações têm como âmbito aproximadamente todo o interior do bloco. "Aproximadamente", porque realmente o âmbito de `j` é ligeiramente mais pequeno do que o âmbito de `i`, e o âmbito de `k` é ligeiramente mais pequeno do que o âmbito de `j`. Onde é que começa exatamente o âmbito de cada uma das três ligações?

```
{
    int i = 0
    int j = i + 2
    int k = i + j ;
    printf("%d %d %d\n", i, j, k) ;
}
```

O seguinte exemplo, também em C, é mais interessante e ilustra um **bloco aninhado** dentro de outro bloco.

```
{
    int i ;
    int b = 5 ;
    i = a + b ;
    {
        int i = 0
        int j = i + 2
        int k = i + j ;
        printf("%d %d %d\n", i, j, k) ;
    }
    printf("%d %d\n", i, b) ;
}
```

Repare que o âmbito da variável `b`, introduzida no bloco exterior, abrange aproximadamente todo o bloco externo, o que inclui o bloco interno. No entanto o âmbito da variável `i` introduzida no bloco exterior abrange o bloco exterior *menos* o bloco interior, porque a variável `i` é redefinida no bloco interior.

Este exemplo mostra que numa linguagem onde as construções definidores de âmbitos podem ser aninhadas, o âmbito duma ligação pode não corresponder a uma zona contígua de programa. Por outras palavras, pode ter "buracos" âmbito duma ligação!

Resolução de nomes

Chama-se **resolução de nomes** ao processo de descoberta do significado (ou seja, da ligação) de alguns nomes num ponto do programa onde esses nomes são usados.

Escopo estático

Escopo estático é o nome da regra de resolução de nomes usada na maioria das linguagem modernas, incluindo o OCaml, C, C++ e Java.

A regra é muito simples e diz apenas o seguinte:

- Um uso dum nome refere-se sempre à ligação sintaticamente envolvente, mais próxima.

Portanto, para saber o que o significado dum nome num dado ponto do programa, basta olhar para o código "à volta" (de acordo com as regras da sintaxe da linguagem), e procurar aí a declaração mais próxima desse nome. Um caso particular: se o nome estiver definido localmente, então é a declaração local que vale para o nome em causa.

Quem aprendeu a programar numa linguagem moderna, está tão habituado a esta regra que geralmente nem se apercebe dele. Bem, nos exemplos da secção "Blocos", atrás, nós já usámos esta regra "sem dar por isso"...

Mais um exemplo, agora em ML:

```
let z = 5 in
    let f x = x + z in
        let z = 6 in
            f 0
```

Neste exemplo aparece uma utilização duma variável `z`, dentro da função `f`, que pode criar dúvidas. Será que uso do nome `z` se refere à declaração de `z` exterior, ou à declaração de `z` interior?

Como a linguagens ML usa a regra de escopo estático, a resposta correta é: o uso do nome `z` refere-se à declaração de `z` exterior (ou seja, envolvente).

Pergunta: Quando o valor da expressão do exemplo, 5 ou 6?
Resposta: 5.

Escopo dinâmico

Escopo dinâmico é o nome da regra de resolução de nomes atualmente em desuso, mas que importa conhecer. É usado em algumas versões da linguagem Lisp e também na linguagem APL, por exemplo.

A regra também é muito simples e diz apenas o seguinte:

- Um uso dum nome refere-se sempre à ligação mais recentemente estabelecida para esse nome, durante a execução do programa.

Regressemos ao exemplo:

```
let z = 5 in
    let f x = x + z in
        let z = 6 in
            f 0
```

Pergunta: Usando a regra de escopo dinâmico, qual o valor da expressão do exemplo, 5 ou 6?
Resposta: 6.

A resposta é 6 porque, repare, quando a função `f` é chamada, a ligação mais recente para `z` é que foi estabelecida na declaração de `z` interior.

Efeitos das regras

Em muitas situações, como por exemplo quando estão em causa nomes declarados localmente, as duas regras de escopo acabam por dar os mesmos resultados, ou seja, resolvem os nomes da mesma forma. Só perante situações semelhantes à do exemplo anterior é que os efeitos são diferentes.

Em rigor, os efeitos das regras só diferem quando estão em causa acessos a nomes não-locais a partir do interior de funções. Por isso as duas regras de escopo podem ser apresentadas da seguinte forma alternativa:

- **Escopo estático** - As funções são chamadas no ambiente da sua definição.
- **Escopo dinâmico** - As funções são chamadas no ambiente de quem os invoca.

Comparação

O escopo estático é usado em praticamente em todas as linguagens modernas pois faz com que a estrutura estática de um programa se aproxime do seu comportamento dinâmico. Isto simplifica imenso a compreensão dos programas.

A regra de escopo dinâmico tem ainda mais estas desvantagens:

- Não permite fazer verificação de tipos estática;
- Os programas ficam sensíveis aos nomes escolhidos para as variáveis e procedimentos.

Estado (conjunto de ligações para localizações)

O conceito de estado tem a ver com um tipo particular de ligações: as ligações de localizações.

Chama-se **estado** a um conjunto de ligações que associam localizações de memória a entidades. Matematicamente um estado é uma função de localizações para entidades, ou seja uma função com o seguinte tipo:

```
Localizações -> Entidades
```

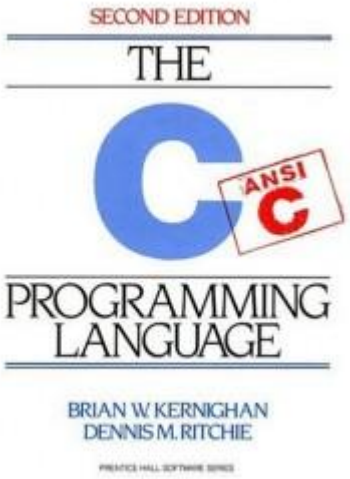
As linguagens funcionais puras não possuem estado. Este facto tem a desvantagem de reduzir a variedade de ligações que se podem estabelecer. Mas tem a vantagem de simplificar a linguagem; outra vantagem é o facto da linguagem ficar mais segura pois sabe-se que a maioria dos dos bugs dos programas estão relacionados com variáveis mutáveis ou com apontadores.

Linguagens e Ambientes de Programação (2010/2011)

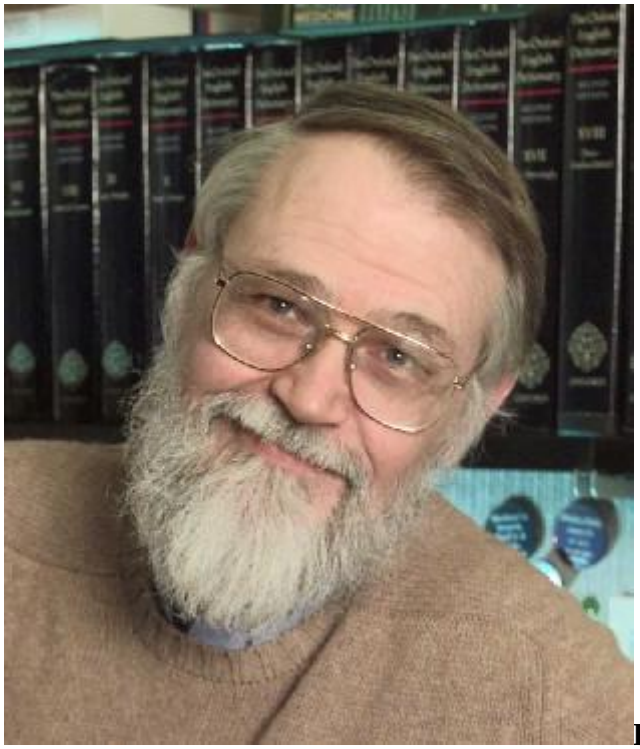
Teórica 12 (28/Mar/2011)

Introdução à linguagem C. Características e história. Padronizações do C.
Elementos Básicos. Tipos básicos.
Variáveis.
Estruturas de controlo

Introdução à linguagem C




Dennis Ritchie


Brian Kernighan

Nota prévia

Esta é a segunda cadeira em que os alunos lidam com a linguagem C. Nas próximas aulas faremos um percurso pela maioria dos mecanismos da linguagem C, discutido-os com base nos conceitos gerais apresentados na cadeira.

Mas a ênfase será colocada nas seguintes partes, onde se espera que os alunos ganhem novas competências:

- Estruturas de dados dinâmicas e manipulação de apontadores.
- Polimorfismo implementado usando apontadores e macros.
- Módulos.
- Discussão das inseguranças da linguagem C.

Nesta linha:

- Toda a aula prática 2 será sobre estruturas de dados dinâmicas e manipulação de apontadores.
- O projeto de programação também será principalmente sobre estruturas de dados dinâmicas e manipulação de apontadores.
- No exame, quase sempre os problemas sobre C são sobre estruturas de dados dinâmicas e manipulação de apontadores.

Algumas características do C

- Concebida e implementada por Dennis Ritchie entre 1969 e 1973 nos Bell Labs da AT&T. A primeira versão do Unix foi escrita em assembler, mas em 1973 o Unix foi reescrito em C.
- Uma das linguagens atualmente mais populares tem sido utilizada para escrever todo o tipo de aplicações tais como compiladores, bases de dados, sistemas operativos (Unix por exemplo), editores gráficos e de texto, etc.
- É uma linguagem padronizada com standards ANSI e ISO.
- Linguagem de alto nível, mas que oferece facilidades para manipulações de baixo nível, nomeadamente acesso direto à memória. Originalmente concebida para programação de sistemas, o C permite ao programador escrever código muito eficiente e com acesso direto aos recursos da máquina.
- Apesar dos mecanismos de baixo nível, a linguagem encoraja independência da máquina. Este aspeto é extremamente importante pois cumpre um dos objetivos das linguagens de programação que é a possibilidade de escrever programas independentes de cada máquina particular.
- A maioria das implementações são muito eficientes.
- Necessidade de uma boa disciplina na programação para produção de programas legíveis --> WOP - 'write only programming' (observação irónica).
- Tipos básicos: caracteres, inteiros, reais, booleanos, enumerados.
- Tipos derivados: vetores, registos, uniões, apontadores.
- Tipificação fraca: por exemplo, os caracteres e os valores enumerados são tratados como inteiros.
- Não têm gestão automática de memória (ao contrário do Java e do OCaml). É necessário efetuar gerir a memória manualmente (usando as funções "malloc" e "free"). A gestão manual de erros é uma conhecida fonte de erros no software.
- Estruturas de controlo: condicionais, iterativas, seletivas.
- Funções. Recursividade é suportada. Não suporta aninhamento de funções.
- Aritmética de apontadores.
- Modularidade e compilação separada.
- Biblioteca padrão.
- Oferece um grande controlo ao programador:
    - a nível sintático devido ao sistema de macros implementado no pré-processador;
    - a nível de dados por causa dos apontadores, uniões e campos de bits;
    - a nível de abstrações de execução por causa dos apontadores para função;
    - a nível de modularidade devido a um sistema de módulos simples e flexível que suporta compilação separada e ocultação de informação.

Exemplo 1

```
#include <stdio.h>

int fact(int i)
{
    if( i==0 ) return 1 ;
    else return i * fact(i-1) ;
}

int main(void)
{
    int n ;

    for( n = 0 ; n < 10 ; n++)
        printf("fact(%d)=%d\n", n, fact(n)) ;
    return 0 ;
}
```

Exemplo 2

```
int main(int argc, char *argv[])
{
    int i ;
    for( i = 0 ; i < argc ; i++ )
        printf("%s\n", argv[i]) ;
    return 0 ;
}
```

Padronizações do C

- K&R C - Padrão informal estabelecido em 1978 com a publicação da 1ª edição do livro "The C Programming Language". Os argumentos das funções não eram validados e as funções retornavam inteiros por omissão.
- Ansi C89 - Padrão criado em 1983 mas só ratificado em 1989. Foram introduzidos **protótipos** que, quando presentes, permitem validar os argumentos e resultado das funções.
- ISO C90 - O padrão anterior foi adotado com muito ligeiras alterações pela ISO em 1990.
- ISO C99 - Padrão de 1999. Introduziu diversas características úteis tais como flexibilidade quanto ao ponto onde se definem as variáveis, vetores de tamanho variável e booleanos.
- ANSI C99 - Em 2000, a ANSI adotou o padrão anterior sem alterações.

O GCC é uma das implementações de C atualmente mais usadas. Suporta a maioria do C99 e mais algumas extensões, das quais a mais notável é a possibilidade de definir funções locais a outras funções, algo que sempre foi proibido no padrão do C.

Como o suporte para C99 no GCC ainda não é completo, por omissão o GCC ainda se baseia no C89. Para forçar o GCC a reconhecer o C99, tanto quanto possível, é necessário invocar o GCC assim:

```
gcc -std=c99
```

Documentação sobre o compilador de C

Olhamos o que diz o início do manual do comando `cc`, no Linux:

```
$ man cc
GCC(1)                                          GNU
GCC(1)

NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [-c|-S|-E] [-std=standard]
           [-g] [-pg] [-Olevel]
           [-Wwarn...] [-pedantic]
           [-Idir...] [-Ldir...]
           [-Dmacro[=defn]...] [-Umacro]
           [-foption...] [-mmachine-option...]
           [-o outfile] infile...

       Only the most useful options are listed here; see below for the remainder.  g++
accepts
       mostly the same options as gcc.

DESCRIPTION
       When you invoke GCC, it normally does preprocessing, compilation, assembly and
linking.
       The "overall options" allow you to stop this process at an intermediate stage.
For exam-
       ple, the -c option says not to run the linker.  Then the output consists of
object files
       output by the assembler.
```

```
       Other options are passed on to one stage of processing.  Some options control the
prepro-
       cessor and others the compiler itself.  Yet other options control the assembler
and
       linker; most of these are not documented here, since you rarely need to use any
of them.
```

Elementos Básicos

Programa em C

Sequência de constantes variáveis, definições de tipos e funções, possivelmente distribuídas por vários ficheiros.

Identificadores

Começam por uma letra podendo conter letras, algarismos e ainda o carácter sublinhado '_'. É feita distinção entre maiúsculas e minúsculas.

Delimitadores de comentário

```
/* comentário  */

// os comentários de linha foram introduzidos no padrão C99
```

Terminador de instrução

Todas as declarações e todas as instruções são terminadas por um ponto e vírgula ';'. Exceção: as funções são terminadas por uma chaveta a fechar '}'.

Literais

Há literais dos tipos carácter, inteiro, real, string e booleano. Exemplos:

- carácter: 'a' '\n' '\t' '\r' '\0' '\123'
- inteiro: 1 5 21056 -56
- real: 5.6 4e7 -5E-5
- string: "" "supercalifragilisticoexpialidoso"
- bool: false true

Definição de constantes

```
#define PI   3.1415962
#define a6   "aaaaaa"
#define um   1
```

Inicialização de variáveis

```
int i = 100 ;
double d = 12.3e56 ;
```

As variáveis estáticas são inicializadas a zero por omissão.

Atribuição a variáveis

```
v = 14 ;
x = 5 + 7 + v ;
```

Tipos básicos

Tipos numéricos

A linguagem suporta os seguintes cinco tipos básicos numéricos:

- char
- short
- int
- float
- double

Um char ocupa 1 byte, um short ocupa pelo menos 2 bytes, um int ocupa pelo menos 2 bytes (normalmente tem 4 bytes em máquinas de 32 bits).

O ficheiro <limits.h> contém informação sobre o tamanho exato de cada tipo, na implementação de C que estiver a ser usada.

O operador `sizeof` também pode ser usado para saber qual o número de bytes ocupados por um valor de qualquer tipo.

Qualificadores dos tipos numéricos

Alguns dos tipos numéricos podem ter a sua semântica modificada por meio dos seguintes qualificadores:

- long (int, double)
- long long (int)
- unsigned (char, short, int, long int)
- signed (char, short, int, long int)

Um `long int` ocupa pelo menos 4 bytes; um `long long int` ocupa pelo menos 8 bytes.

Exemplos de tipos numéricos qualificados:

```
unsigned int
long double
unsigned long int
unsigned char
```

Por omissão todos os tipos são signed exceto o tipo char cujas características dependem da implementação.

O nome do tipo int pode ser omitido, quando qualificado. Portanto os seguintes são tipos válidos:

```
long
unsigned
signed
```

Tipo booleano

Tradicionalmente, o C não costumava ter um tipo booleano explícito, embora o conceito sempre tenha existido na linguagem. O valor de verdade é representado por qualquer valor diferente de zero, e o valor de falsidade é representado por zero. Existem três operadores lógicos que interpretam os argumentos como "booleanos": &&, ||, !.

Foi introduzido no padrão C99 um tipo chamado `bool` com os literais `false` e `true`, mas o uso desse tipo requer a inclusão do ficheiro <stdbool.h>. O `false` é simplesmente representado usando o inteiro 0 e o `true` é representado usando o inteiro 1.

Tipos enumerados

Os tipos enumerados foram introduzidos no padrão C89. São úteis para especificar um número de opções para um atributo. Exemplos de possíveis atributos: cor, mês do ano, dia da semana, etc.

Exemplo:

```
typedef enum {
    JANUARY, FEBRUARY, MARCH,
    APRIL, MAY, JUNE,
    JULY, AUGUST, SEPTEMBER,
    OCTOBER, NOVEMBER, DECEMBER
} Month ;
```

Os valores dos tipos enumerados são representados usando inteiros e a representação não é oculta. Por defeito os valores começam em zero e são incrementados sucessivamente - portanto JANUARY vale 0, FEBRUARY vale 1, etc. Veja a seguinte função:

```
Month NextMonth(Month m) {
    return m == DECEMBER ? JANUARY : m + 1 ;
}
```

O C permite mesmo ao programador especifique a representação inteira de cada valor enumerado. Exemplo:

```
typedef enum {
    RED = 1, GREEN = 2, BLUE = 4, YELLOW = 8
} Color ;
```

Tipos derivados

Os tipos derivados serão discutidos mais tarde:

- Arrays
- Registos (structures)
- Uniões
- Apontadores

Variáveis

Definição

As variáveis podem ser definidas globalmente, fora de qualquer função, ou definidas localmente, dentro duma função, logo no primeiro nível ou então dentro dum bloco interno.

```
int count, n ;
double stdvar, media ;
char car, key ;
```

Inicialização

As variáveis podem ser inicializadas no momento da sua definição.

```
int x = 6 ;
Complex z = { 2.0, 5.7  };
int v[5] = { 1, 2, 3, 4, 5 } ;
```

Na ausência de qualquer expressão de inicialização, as variáveis globais e as variáveis locais estáticas são inicializadas a zero.

As variáveis locais não têm qualquer inicialização por omissão, ficando indefinidas (com um valor aleatório) enquanto não se fizer a primeira atribuição.

Atributos das variáveis locais

Eis os atributos disponíveis para as variáveis locais, e o seu significado:

- **static** - A variável mantém valor entre ativações da função.
- **auto** - (atributo por omissão) variável automática ou seja não estática.
- **register** - Se possível a variável é guardada num registo do CPU.
- **volatile** - Indica que o valor da variável pode mudar fora do controlo do compilador. Exemplo: posição de memória cujo valor pode ser alterado pelo hardware quando da ocorrência de um interrupt.
- **const** - A variável é inicializada no ponto da sua definição e o seu valor não pode ser alterado depois. Também se aplica a argumentos de funções.

Atributos das variáveis globais

Eis os atributos disponíveis para as variáveis globais, e o seu significado:

- **static** - A variável é privada no ficheiro, não podendo ser usada a partir de outros ficheiros fonte.
- **volatile** - Indica que o valor da variável pode mudar fora do controlo do compilador.
- **const** - A variável é inicializada no ponto da sua definição e o seu valor não pode ser alterado depois.

Relativamente aos diversos tipos de variáveis, classifique as respetivas ligações em função do momento de ligação e diga qual é o tempo de vida de cada uma delas.

Estruturas de controlo

Observe as posições onde se escreve o terminador ';'.

```
if(exp) stat

if(exp) stat else stat

while(exp) stat

do stat while(exp);

for(init; test; advance) stat

switch(exp){
    case const: stats
    case const: stats
    default: stats
}

{ // bloco
    decls e stats
}

exp;

break;

continue;

return;

return exp;

goto label;
```

```
label: stat

; // instrução nula
```

Quais são as diferenças relativamente ao Java?

#80

Linguagens e Ambientes de Programação (2010/2011)

Teórica 13 (30/Mar/2011)

Operadores.
Avaliação de expressões.
Tipos derivados.
Apontadores e sua utilidade.
Apontadores: Parâmetros de saída de funções.
Apontadores: Manipulação de vetores e relação entre vectores e apontadores.

Operadores

Eis a lista completa dos operadores do C que podem ser usados em expressões:

```
aritméticos:    +  -  *  /  %
lógicos:        !  &&  ||
relacionais:    <  >  <=  >=  ==  !=
bits:           >> << & ^ |~
atribuição:     =  +=  -=  *=  /=  %=  &=  |=  ^=  <<=  >>=
condicional     ?:
cast:           (type)
inc/dec:        expr++  ++expr  expr--  --expr
sequenciação:   ,
sizeof:         sizeof
apontadores:    *  &  ->  []
field:          .               (para acesso a campos de registos e uniões)
agrupamento:    (expr)
```

Note que em C (tal como em Java), a atribuição produz um resultado e por isso é considerada uma expressão, não um comando. O valor da expressão `v = exp` é o valor que fica na variável `v` depois da expressão `exp` ter sido avaliada e depois da atribuição ter sido concretizada.

O C faz sobrecarga (overloading) de alguns operadores. Por exemplo, o operador + é usado para denotar três operações diferentes: a soma inteira; a soma real; a soma entre um apontador e um inteiro.

Precedências e associatividades

Precedências dos operadores por ordem decrescente de prioridade:

```
()
[] -> . expr++ expr--                          esq
! ~ ++ -- - (type) * & sizeof ++expr --expr     dir
```

```
* / %                                                    esq
+ -                                                     esq
<< >>                                                   esq
< <= > >=                                               esq
== !=                                                   esq
&                                                       esq
^                                                       esq
|                                                       esq
&&                                                      esq
||                                                      esq
?:                                                      esq
= += -= etc.                                            dir
,                                                       esq
```

Exemplo:

```
if( year % 4 == 0 && year % 100 != 0 || year % 400 == 0 )
    printf("Ano bissexto\n") ;
else
    printf("Ano comum\n") ;
```

Avaliação de expressões

Ordem de avaliação

O compilador tem a liberdade de rearranjar as expressões por forma a optimizar a eficiência da sua avaliação mesmo que esta envolva efeitos laterais. Os parêntesis podem mesmo não ser respeitados (!) se a sua mudança de posição não violar nenhuma das regras da álgebra. O compilador também ter a liberdade de avaliar os argumentos nas chamadas das funções por qualquer ordem.

Portanto a ordem de avaliação não está geralmente definida e devemos evitar escrever expressões cujos efeitos ou resultados dependam da ordem de avaliação. Exemplos:

```
i = i++ ;           /* o valor final de i não está definido */
f(i++, i++) ;       /* o valor dos argumentos não está definido, mas o valor final de i
não tem problema */
f(*p1++, *p2++) ;  /* o valor dos argumentos não está definido no caso dos dois
apontadores referirem a mesma posição de memória */
f() + g()           /* qualquer das funções pode ser executada em primeiro lugar */
```

O último exemplo só é problemático se as duas funções produzirem efeitos laterais que sejam dependentes da ordem de avaliação.

Pontos de sequenciação

Mas nem tudo está indefinido na ordem de avaliação de expressões. Temos os **pontos de sequenciação** para nos ajudar. São pontos dentro das expressões que garantem que os efeitos laterais das expressões anteriores já foram completamente concretizados.

Os pontos de sequenciação do C estão associados aos seguintes operadores:

```
,  &&  ||  ?:
```

Hierarquia dos tipos numéricos

Os tipos numéricos podem ser livremente misturados em expressões. Quando isso acontece, são efectuadas promoções automáticas de tipo de acordo com a seguinte hierarquia:

```
char short
int
unsigned int
long int
unsigned long int
```

```
long long int
unsigned long long int
double
long double
```

Fora desta hierarquia encontra-se o tipo `float`, o que é promovido para `double` nos contextos onde se fazem contas com doubles.

Conversões automáticas de tipos numéricos

Aplicam se sempre sucessivamente as seguintes regras na avaliação de uma expressão:

1. char, short, enum -> int
2. Para cada operando binário com operandos de tipo diferente, o menos importante é convertido no tipo do mais importante, antes de se efectuar a operação.
3. Nas atribuições (v = exp) o tipo do valor da direita é convertido num valor do tipo da esquerda antes de se fazer a atribuição. Muitas vezes isso implica uma despromoção de tipo e uma truncagem de valor.

Exemplos:

```
5 + 2.0 * 'a'
(5.3 + 5) + 7
int i = 200 * 'a'
```

Tipos derivados

Há 4 variedades de tipos derivados:

- Vectores (arrays)
- Registos (structures)
- Uniões
- Apontadores

Podem ser usados directamente, como na seguinte declaração de variável

```
struct
{
    double re, im ;
} v ;
```

mas muitas vezes usa-se a construção **typedef** para lhes associar um nome. Exemplos:

```
typedef struct
{
    double re, im ;
} Complex;

typedef union
{
    int x;
    char c;
} IntChar;

typedef int Vector[5];

typedef int Matriz[2][3];

typedef char String[256];

typedef void *Pointer;

typedef int *IntPointer;

typedef IntPointer *PointerToIntPointer;

typedef int **PointerToIntPointer;
```

```
typedef (*IntFunction)(void);

typedef IntFunction IntFunctionArray[100];
```

Eis algumas declarações de variáveis usando os tipos declarados anteriormente:

```
Complex z;
IntChar u;
Vector vector;
Matriz matriz;
String str;
Pointer v;
IntPointer pt;
IntFunctionArray ops ;
```

Apontadores

Introdução: Variáveis e apontadores

Os programas processam dados armazenados na memória do computador. O mecanismo mais simples que permite aceder a essa memória são as **variáveis**. Cada variável tem um nome e um tipo e representa um pedaço da memória do computador onde podem ser guardados valores desse tipo.

Através da utilização de variáveis, é possível ir muito longe na escrita de programas em C. Mas há algumas situações em que o uso de variáveis não é suficiente e é necessário usar mecanismo mais flexível de manipulação da memória:

- Trata-se do mecanismo dos **apontadores**!

Conceitos básicos sobre apontadores

Portanto os apontadores constituem uma segundo mecanismo de acesso à memória:

- Em vez de se usar um nome para identificar um pedaço da memória, usa-se directamente o endereço dessa zona de memória para a identificar e chama-se a esse endereço um **apontador**. Diz-se que o apontador **aponta** para uma determinada zona de memória.

Normalmente os apontadores são guardados em **variáveis de tipo apontador**.

Em C há tipos específicos para representar apontadores. O tipo dos apontadores que apontam para valores de tipo T escreve-se:

```
T *
```

Para exemplificar, eis a definição duma variável de tipo apontador para inteiro:

```
int *pt ;
```

Em C, há duas operações principais para manipular apontadores: o operador **`&`** permite obter um apontador para uma variável qualquer, assim

```
&Variável
```

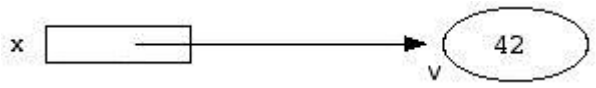
e o operador **`*`** permite aceder ao valor apontado por um apontador, assim:

```
*Apontador
```

Vejamos um exemplo. Abaixo, define-se uma variável inteira normal `v`. A seguir define-se uma variável de tipo apontador para inteiros `x` e fazemo-la apontar para a variável v. Depois, usamos o apontador para colocar o valor 42 na zona de memória apontada por `x`.

```
int v = 0 ;
int *x = &v ;
*x = 42
```

A seguinte figura, ilustra a situação após a atribuição do valor 42 a `*x`.

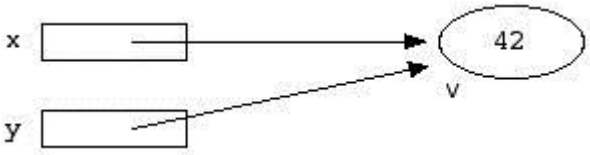


Repare que a variável `v` pode ser acedida de duas formas: (1) usando o nome `v`; (2) usando a expressão `*x`.

Para perceber melhor as possibilidades dos apontadores vamos definir agora um segundo apontador, `y` a fazê-lo também apontar para a variável `v`:

```
int *y = x ;
```

Obtém-se a seguinte situação:



Agora o conteúdo da variável `v` pode ser acedido de três formas diferentes: (1) usando o nome `v`; (2) usando a expressão `*x`; (3) usando a expressão `*y`.

O operador `&` chama-se **operador endereço**. O operador `*` chama-se **operador de desreferenciação**.

Repare na seguinte curiosa equivalência, que é válida em C para qualquer variável `v`:

```
*&v == v
```

Falta ainda uma referência à constante predefinida de tipo apontador, **`NULL`**. Garante-se que este apontador constante não aponta para sítio nenhum. Pode ser atribuído a uma variável de tipo apontador, por exemplo assim:

```
int *z = NULL ;
```

Neste caso serve para assinalar que a variável `z` não está a apontar para sítio nenhum, de momento.

O apontador NULL também pode ser usado em testes, assim:

```
if (z == NULL) ...
```

O apontador NULL não pode ser desreferenciado, pois não aponta para sítio nenhum.

Apontadores para registos

O seguinte tipo registo permite representar datas:

```
typedef struct {
    int day, month, year ;
} Date ;
```

Vamos definir agora uma variável de tipo `Date` e coloquemos um apontador de tipo `Date *` a apontar para a primeira:

```
Date d = {25, 12, 2008};
Date *p = &d;
```

Para aceder, através do apontador, ao ano da data `d`, podemos escrever:

```
(*p).year
```

Mas a utilização de apontadores para registos em C é tão frequente, que foi criada uma notação mais compacta e sugestiva para fazer isso: o operador **`->`**. A seguinte expressão é equivalente à anterior:

```
p->year
```

Em geral, a seguinte notação geral permite aceder a campos de registos através de apontadores:

```
Apontador->Etiqueta
```

Compatibilidade entre apontadores

Em C, tipos de apontadores que apontem para valores de tipos diferentes são incompatíveis entre si. Com uma excepção: o tipo especial `void *` - o tipo `void *` é compatível com todos os tipos de operadores.

```
int *pti;
double *ptd ;
void *v ;

pti = ptd ;             /* Errado */
pti = (int *)ptd ;      /* Válido por causa do cast */

v = ptd ;               /* Válido porque se trata de void * */
pti = v ;               /* Válido porque se trata de void * */
```

Utilidade dos apontadores

Os exemplos anteriores são interessantes, mas não mostram ainda as situações práticas em que a utilização de apontadores é essencial na linguagem C. As situações práticas em que se usam apontadores em C são as seguintes:

1. **Implementação de parâmetros de saída nas funções.**
2. **Manipulação de vectores.**
3. **Manipulação de variáveis criadas dinamicamente usando a função de biblioteca `malloc`.**
4. **Programação genérica através de apontadores de tipo `void *`.**

Apontadores: Parâmetros de saída de funções

Vamos tentar programar uma função para trocar o valor de duas variáveis inteiras. Este é um exemplo clássico que ilustra a necessidade de suportar **parâmetros de saída** na linguagem C.

Esta primeira tentativa não funciona:

```
void Swap(int a, int b)   /* Não funciona!!!! */
{
    int temp = a;
    a = b;
    b = temp;
}
```

A chamada de `Swap(x,y)` não muda nada, porque os parâmetros das funções são implementados usando variáveis locais, inicializadas com cópias dos valores que aparecem na chamada. A função `Swap` faz a troca das cópias locais, mas não troca o conteúdo das variáveis originais. Diz-se que os parâmetros `a` e `b` são **parâmetros de entrada**, porque eles permitem apenas transferir dados de fora para dentro da função.

Para resolver este problema, temos de usar apontadores. A função `Swap` precisa de aceder às variáveis originais através dos apontadores para efectuar a troca. Fica assim:

```
void Swap(int *a, int *b)   /* Funciona!!!! */
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Agora a chamada escreve-se `Swap(&x,&y)` e a troca é realmente efectuada. Diz-se que os parâmetros `a` e `b` são **parâmetros de saída**, porque eles permitem passar dados de dentro para fora da função.

Repare que agora ficámos a conhecer duas maneiras de fazer uma função produzir dados para o seu exterior:

1. Através do seu resultado.
2. Através de parâmetros de saída (usando apontadores).

O seguinte exemplo mostra uma função com dois *resultados*, sendo os *resultados* implementados usando parâmetros de saída. A função faz o seguinte: dado um vector de reais e o respectivo comprimento, a função calcula o máximo e o mínimo do vector, ao mesmo tempo:

```
void MaxMin(double v[], int n, double *max, double *min) /* precondição: n > 0 */
{
    double lmax = v[0];
    double lmin = v[0];
    int i;
    for( i = 1 ; i < n ; i++ ) {
        if (v[i] > lmax) lmax = v[i] ;
        if (v[i] < lmin) lmin = v[i] ;
    }
    *max = lmax;
    *min = lmin;
}
```

Exemplo de chamada:

```
double vect[] = {1.0, 2.9, 34.6, 44.2,  0.01};
double max, min;
MaxMin(vect, 5, &max, &min);
```

Algumas funções de biblioteca também usam parâmetros de saída. Por exemplo, é o caso da função de biblioteca `scanf`. Veja um exemplo de utilização:

```
scanf("%d %lf %c", &i, &r, &c);
```

Apontadores: Manipulação de vetores e relação entre vectores e apontadores

Provavelmente você vai ficar surpreendido(a), mas em C o nome duma variável de tipo vector representa um apontador constante para a primeira componente do vector guardado na variável.

Considere o seguinte vector

```
int vector[100];
```

Para aceder ao primeiro elemento do vector, normalmente nós escrevemos:

```
vector[0]
```

Mas também podemos escrever o que está a seguir, pois o resultado é exactamente o mesmo.

```
*vector
```

A linguagem C também permite a seguinte atribuição:

```
int *pt = vector;
```

e, inclusivamente, permite-se a aplicação de operações sobre vectores a argumentos de tipo apontador. Por exemplo as seguintes expressões são legítimas:

```
pt[0]
pt[5]
pt[99]
```

Note que, quando de passa um vector como parâmetro para uma função, o que realmente se passa é um apontador para a primeira componente do vector. Portanto um parâmetro de tipo vector é sempre um parâmetro de saída, apesar de não ser explicitamente declarado como apontador.

Aritmética de apontadores

Os operadores + e - podem ser aplicados a apontadores e inteiros nos seguintes casos:

- **pt + i** - o resultado é um apontador que referência um valor i posições depois de pt.
- **pt - i** - o resultado é um apontador que referência um valor i posições antes de pt.
- **pt1 - pt2** - o resultado é um inteiro que indica o numero de objectos entre pt1 e pt2. Estes dois apontadores têm de ser do mesmo tipo.

Para o vector abaixo são verdadeiras as equivalências indicadas:

```
Vector v ;

v[0] == *v
v[1] == *(v+1)
v[-1] == *(v-1)
&v[0] == v
&v[1] == v + 1
```

Fazer `v = v + 1` é proibido pois v é um apontador constante.

Exemplo - Duas formas diferentes de copiar vectores (neste caso bidimensionais)

```
#define SIZE    20

int i1[SIZE][SIZE], i2[SIZE][SIZE] ;
int *pt1, *pt2, *ptEnd ;
int i, j;

/* Forma 1 */
for( i = 0 ; i < SIZE ; i++ )
    for( j = 0 ; j < SIZE ; j++ )
        i2[i][j] = i1[i][j] ;

/* Forma 2 */
for( pt1 =i1 , pt2 = i2, ptEnd = pt2 + SIZE * SIZE ;
            pt2 < ptEnd ;
                *pt1++ = *pt2++ ) ;
```

Repare que a primeira forma envolve um ciclo embutido noutro e que, durante a execução, é necessário fazer muitas contas para repetidamente determinar quais as localizações correspondentes às expressões `i2[i][j]` e `i1[i][j]`.

Quanto à segunda forma, envolve um único ciclo e evita a necessidade de se fazerem as contas atrás referidas.

A segunda forma é mais difícil de perceber, mas é um pouco mais eficiente.

#90

Linguagens e Ambientes de Programação (2010/2011)

Teórica 14 (04/Abr/2011)

Apontadores: Criação dinâmica de variáveis.
Compilação separada e modularidade.
Módulos abertos em C.
Módulos fechados em C.
Listas em C.

Apontadores: Criação dinâmica de variáveis

A linguagem C não dispõe dum gestor de memória automático. A criação e a eliminação de variáveis dinâmicas são da responsabilidade do programador.

Usa-se a função de biblioteca `malloc` para criar e eliminar variáveis dinâmicas. A função `malloc` recebe o número de bytes do bloco de memória a criar e retorna um apontador (de tipo `void *`) para o bloco que foi criado. Eis o cabeçalho desta função, tal como está definido no módulo `stdlib`:

```
void *malloc(size_t size) ;
```

Exemplo de utilização:

```
#include <stdlib.h> /* declara a função malloc, entre muitas outras coisas */

int *a = malloc(10 * sizeof(int)) ;   /* cria var. dinâmica; o respetivo apontador é
guardado na var. a */
a[3] = 10 ;                           /* altera uma célula do bloco de memória apontado
(usando notação de vetor) */
```

A função `malloc` retorna a constante `NULL` se o gestor de memória não tiver mais memória disponível.

Para libertar a memória ocupada por uma variável dinâmica que já não irá mais ser usada, usa-se a operação:

```
void free(void *ptr) ;
```

Interessa criar variáveis dinâmicas principalmente nas duas seguintes situações:

- Quando é preciso criar um vetor, cujo tamanho só é conhecido depois do programa começar a correr.
- Para criar estruturas de dados dinâmicas, e.g. listas e árvores.

Mais abaixo, mostra-se como se manipulam com listas em C. Uma lista é uma estrutura de dados dinâmica que só pode ser manipulada através de apontadores. Para organizar bem o nosso exemplo, vamos apresentá-lo sob a forma de módulo. Mas primeiro temos de aprender a lidar módulos em C...

Compilação separada e modularidade

A linguagem C suporta modularidade e compilação separada, existindo mecanismos de controlo de visibilidade de símbolos ao nível do ficheiro.

Controlo de visibilidade

Por omissão, todas as variáveis globais e funções definidas num ficheiro são públicas, isto é podem ser acedidas a partir de outros ficheiros. Consegue-se impedir esse acesso precedendo a definição dessas entidades pela palavra `static`.

```
static Vetor privado ;
static int f(int x) { return x + 1 ; }
```

Antes de se poder aceder a uma entidade definida noutro ficheiro é preciso declarar essa entidade para que o compilador a possa conhecer. Isso faz-se usando a palavra reservada `extern`. No caso da declaração das funções o atributo `extern` pode ser omitido, pois o compilador vê que a declaração não tem corpo e assume que se trata duma entidade externa.

```
extern char key ;
extern int f(int x) ;    /* neste caso a palavra extern pode ser omitida */
```

Módulo

Jogando de forma disciplinada com os mecanismos de visibilidade atrás descritos, é possível implementar **módulos** em C.

Um módulo "fich" é tipicamente definido usando dois ficheiros:

- Um **ficheiro de interface** "fich.h";
- um **ficheiro de implementação** "fich.c".

No ficheiro de interface definem-se todas as entidades exportadas pelo módulo, o que pode incluir: funções, variáveis globais, tipos e constantes.

O cliente do módulo só precisa fazer

```
#include "fich.h"
```

para ter acesso à definição dos símbolos exportados.

Módulos abertos em C

Apresenta-se uma implementação de listas ligadas, feita num módulo aberto em C. Estas listas são homogéneas e podem conter valores de tipo `double`.

Este módulo diz-se **aberto** porque a representação das listas é pública. Veremos na secção seguinte como ocultar essa representação.

Definimos um pequeno conjunto de operações, só para exemplificar.

Ficheiro LinkedList.h

A definição do símbolo `_LinkedList_`, protege o ficheiro de interface relativamente ao problema da inclusão múltipla do mesmo ficheiro.

Uma **lista ligada** consiste numa sequência de nós. Cada nó contém um valor dum dado tipo Data e um apontador indicando qual o nó seguinte. O valor NULL serve para indicar que não existe nó seguinte.

É interessante comparar a utilização de listas face à utilização de vetores, para guardar sequências de valores:

- A vantagem duma lista relativamente um vetor é o facto da sucessão de nós não ter de estar em posições contíguas de memória. Isso permite a inserção e remoção de valores em tempo constante.
- A desvantagem duma lista relativamente um vetor é não ser possível acesso indexado em tempo constante.

```
#ifndef _LinkedList_
#define _LinkedList_

#include <stdbool.h>

typedef double Data ;
typedef struct Node {
    Data data ;
    struct Node *next ;
} Node, *List ;  /* Uma lista e' um apontador para um no' */

List ListMakeRange(Data a, Data b) ;
int ListLength(List l) ;
bool ListGet(List l, int idx, Data *res) ;
List ListPutAtHead(List l, Data val) ;
List ListPutAtEnd(List l, Data val) ;
void ListPrint(List l) ;

#endif
```

Ficheiro LinkedList.c

Repare que a função auxiliar `NewNode` é declarada como estática para ficar privada ao módulo. A palavra reservada `static`, quando aplicada a uma função ou a uma variável global, torna essas entidades privadas: portanto o **âmbito** da sua definição é o ficheiro onde essa definição ocorre.

Repare que todas as funções deste módulo estão programadas com base em raciocínios imperativos e sem usar recursividade. Essa é a forma normal de trabalhar em C.

Algumas destas funções são complicadas de escrever, especialmente as que requerem a utilização de diversos apontadores auxiliares. **Geralmente, a melhor forma de programar estas funções é começar por escrever o ciclo principal, e só se pensa em qual deve ser a inicialização desse ciclo - realmente, antes de escrever o ciclo principal, é difícil adivinhar quais são as suas necessidades.**

A função `ListMakeRange` ilustra uma técnica importante. É sempre mais fácil fazer crescer as listas acrescentando novos nós à cabeça. Mas usando a técnica apresentada, consegue-se criar uma nova lista acrescentando os novos nós no final. Usa-se de forma habilidosa um apontador auxiliar `p` que aponta sempre para o último nó da lista.

Repara ainda que a função `ListPutAtEnd` define uma operação "destrutiva", que altera a lista original.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "LinkedList.h"

static List NewNode(Data val, List next)
{
    List n = malloc(sizeof(Node)) ;
    if( n == NULL )
        return NULL ;
    n->data = val ;
    n->next = next ;
    return n ;
}

List ListMakeRange(Data a, Data b)
{
    if( a > b )
        return NULL ;
    double i ;
    List l = NewNode(a, NULL), p = l ;
    for( i = a + 1 ; i <= b ; i++ )
        p = p->next = NewNode(i, NULL) ;
    return l ;
}

/* Outra maneira, mais palavrosa, de escrever a funcao anterior:

List ListMakeRange(Data a, Data b)
{
    if( a > b )
        return NULL ;
    double i ;
    List l = NewNode(a, NULL) ;
    List p = l ;
    for( i = a + 1 ; i <= b ; i++ ) {
        List q = NewNode(i, NULL) ;
        p->next = q ;
        p = q ;
    }
    return l ;
}
*/
```

```
int ListLength(List l) {
    int count ;
    for( count = 0 ; l != NULL ; l = l->next, count++ ) ;
    return count ;
}

bool ListGet(List l, int idx, Data *res)
{
    int i ;
    for( i = 0 ; i < idx && l != NULL ; i++, l = l->next ) ;
    if( l == NULL )
        return false ;
    else {
        *res = l->data ;
        return true ;
    }
}

List ListPutAtHead(List l, Data val)
{
    return NewNode(val, l) ;
}

List ListPutAtEnd(List l, Data val)
{
    if( l == NULL )
        return NewNode(val, NULL) ;
    else {
        List p ;
        for( p = l ; p->next != NULL ; p = p->next ) ;
        p->next = NewNode(val, NULL) ;
        return l ;
    }
}

void ListPrint(List l)
{
    for( ; l != NULL ; l = l->next )
        printf("%lf\n", l->data) ;
}
```

Módulos fechados em C

Os módulos fechados em C baseiam-se na possibilidade de definir um tipo apontador para um tipo estrutura que ainda não foi definido, assim:

```
typedef struct Node *List ;
```

Onde é que, finalmente, se define o tipo `struct Node`? Somente no interior ficheiro LinkedList.c.

Ficheiro LinkedList.h

Deste ficheiro apaga-se a maior parte da definição do tipo, ficando apenas a linha que se vê. O resto do ficheiro fica igual.

```
#ifndef _LinkedList_
#define _LinkedList_

#include <stdbool.h>

typedef double Data ;
typedef struct Node *List ;

...
```

Ficheiro LinkedList.c

A definição do tipo passa para dentro do ficheiro ".c". Mas é removida a parte "*List" pois isso já está presente no ficheiro ".h". O resto do ficheiro fica igual.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "LinkedList.h"

typedef struct Node {
    Data data ;
    List next ;
} Node ;

...
```

Recursividade e método indutivo usados no processamento de estruturas com apontadores

Eis uma nova implementação do nosso módulo, agora usando recursividade e raciocínios indutivos.

A técnica indutiva tem a vantagem de dar origem a código mais legível, mas tem a desvantagem de poder fazer crescer muito a pilha de execução a ponto de isso causar "stack overflows". Note que, ao contrário do OCaml, a linguagem C não está equipada de forma especial para suportar recursividade de forma económica.

Portanto a técnica recursiva é de evitar quando se processam lista. No processamento de listas, a complexidade espacial (tamanho da pilha de execução) dos algoritmos fica linear, como se viu numa das aulas anteriores.

Contudo, para processar árvores e outras estruturas não-lineares, já faz sentido usar recursividade em muitas das operações. Sem usar recursividade essas operações ficariam demasiado complicados. Além disso, a complexidade espacial dos algoritmos recursivos sobre árvores é muitas vezes `log N`.

```
/* ESTE É UM EXEMPLO NEGATIVO. É má ideia programar listas
   ligadas em C desta forma, apesar de funcionar */

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "LinkedList.h"

typedef struct Node {
    Data data ;
    List next ;
} Node ;

static List NewNode(Data val, List next)
{
    List n = malloc(sizeof(Node)) ;
    if( n == NULL )
        return NULL ;
    n->data = val ;
    n->next = next ;
    return n ;
}

List ListMakeRange(Data a, Data b)
{
    if( a > b )
```

```
        return NULL ;
    else
        return NewNode(a, ListMakeRange(a+1, b)) ;
}

int ListLength(List l)
{
    if( l == NULL )
        return 0 ;
    else
        return 1 + ListLength(l->next) ;
}

bool ListGet(List l, int idx, Data *res)
{
    if( l == NULL )
        return false ;
    else if( idx == 0 ) {
        *res = l->data ;
        return true ;
    }
    else
        return ListGet(l->next, idx-1, res) ;
}

List ListPutAtHead(List l, Data val)
{
    return NewNode(val, l) ;
}

List ListPutAtEnd(List l, Data val)
{
    if( l == NULL )
        return NewNode(val, NULL) ;
    else {
        l->next = ListPutAtEnd(l->next, val)
        return l ;
    }
}

void ListPrint(List l)
{
    if( l != NULL ) {
        printf("%lf\n", l->data) ;
        ListPrint(l->next) ;
    }
}
```

#90

Linguagens e Ambientes de Programação (2010/2011)

Teórica 15 (06/Abr/2011)

Pré-processador.
Polimorfismo implementado usando mecanismos de baixo nível.
Módulos genéricos (polimórficos) em C.

Pré-processador

Programa que corre sempre antes do compilador de C e que faz substituição de macros, inclusão de ficheiros, e possibilita compilação condicional.

Em Linux, o pré-processador chama-se `cpp` e pode ser invocado diretamente pelo utilizador. Mas é mais prático deixar que seja o compilador de C a invocar o pré-processador automaticamente. Em todo o caso, vejamos o que diz o início do manual do comando `cpp`:

```
$ man cpp

CPP(1)                                                GNU
CPP(1)

NAME
       cpp - The C Preprocessor

SYNOPSIS
       cpp [-Dmacro[=defn]...] [-Umacro]
           [-Idir...] [-iquotedir...]
           [-Wwarn...]
           [-M|-MM] [-MG] [-MF filename]
           [-MP] [-MQ target...]
           [-MT target...]
           [-P] [-fno-working-directory]
           [-x language] [-std=standard]
           infile outfile

       Only the most useful options are listed here; see below for the remainder.

DESCRIPTION
       The C preprocessor, often known as cpp, is a macro processor that is used
automatically by
       the C compiler to transform your program before compilation.  It is called a
macro proces-
       sor because it allows you to define macros, which are brief abbreviations for
longer con-
       structs.

       The C preprocessor is intended to be used only with C, C++, and Objective-C
source code.
```

Exemplos de macros:

```
#define max(a,b)      ((a) > (b) ? (a) : (b))
#define new(type)     ((type *)malloc(sizeof(type)))
#define not           !
#define MAX_STACK     1000

#undef max
#undef new
```

Exemplos de inclusão de ficheiros

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include "MyQueue.h"
```

Exemplos de compilação condicional

```
#define DEBUGLEVEL   1

#ifdef DEBUG
    ...
#  if DEBUGLEVEL > 2
    ...
#  else
    ...
#  endif
    ...
#else
    ...
#endif


#ifndef DEBUG
    ...
#else
    ...
#endif
```

Polimorfismo implementado usando mecanismos de baixo nível

Função monomórfica

A seguinte função permite trocar os valores de duas variáveis inteiras. É uma função monomórfica pois os seus argumentos são de tipo fixo.

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

Esta função pode ser testada assim:

```
int main(void) {
    int x = 5, y = 6 ;
    printf("%d %d\n", x, y) ;
    swap(&x, &y) ;
    printf("%d %d\n", x, y) ;
    return 0 ;
}
```

A função `swap` é segura pois o compilador valida todas as suas utilizações.

Polimorfismo implementado usando manipulação direta de memória

A seguinte função permite trocar os valores de duas variáveis de qualquer tipo. É uma função polimórfica pois os seus argumentos podem ser aplicados a argumentos de tipos diversos.

```
#include <string.h>

void swap(void *a, void *b, int n)
{
    char aux[n] ;     /* array criado com tamanho variável - novidade do C99 */
    memcpy(aux, a, n) ;
    memcpy(a, b, n) ;
    memcpy(b, aux, n) ;
}
```

Esta função ser testada assim:

```
int main(void) {
    int x = 5, y = 6 ;
```

```
    printf("%d %d\n", x, y) ;
    swap(&x, &y, sizeof(int)) ;
    printf("%d %d\n", x, y) ;
    return 0 ;
}
```

A função `swap` não é segura pois o compilador não tem a possibilidade de validar as chamadas.

Polimorfismo implementado usando macros

A seguinte macro também permite trocar os valores de duas variáveis de qualquer tipo.

```
#define swap(a,b,T)         \
    do {                    \
        T __aux = (a)  ;    \
        (a) = (b) ;         \
        (b) = __aux ;       \
    } while( 0 )
```

Esta macro ser testada assim:

```
int main(void) {
    int x = 5, y = 6 ;
    printf("%d %d\n", x, y) ;
    swap(x, y, int) ;
    printf("%d %d\n", x, y) ;
    return 0 ;
}
```

A macro `swap` é segura pois o compilador consegue detetar qualquer erro de tipo na sua utilização. Repare que o tipo dos valores a trocar é passado como parâmetro.

Repare nos cuidados que é preciso ter para escrever uma macro que não dê problemas. Os argumentos devem ser envolvidos em parêntesis, e qualquer nova variável que seja introduzido deve ter um nome que não entre em conflito com possíveis nomes existentes. Porque todos estes cuidados? E porque razão a macro foi definida usando um `do-while`? (Elimine o `do-while`, teste, e logo descobrirá o problema.)

Polimorfismo implementado usando manipulação direta de memória e macros

A seguinte implementação da operação `swap` é a mais agradável de usar do ponto de vista sintático. Até parece que está em causa uma operação primitiva da linguagem. Contido este implementação não é segura, pelo que o utilizador tem de ter algumas cautelas.

```
#include <string.h>

void __swap(void *a, void *b, int n)
{
    char aux[n] ;
    memcpy(aux, a, n) ;
    memcpy(a, b, n) ;
    memcpy(b, aux, n) ;
}

#define swap(a,b)     __swap(&(a), &(b), sizeof(a))
```

Esta função ser testada assim:

```
int main(void) {
    int x = 5, y = 6 ;
    printf("%d %d\n", x, y) ;
    swap(x, y) ;
    printf("%d %d\n", x, y) ;
    return 0 ;
}
```

A macro `swap` não é segura pois o compilador não tem a possibilidade de validar as chamadas.

Em GCC

O pré-processador do GCC possui uma construção `typeof` que permite escrever macros ainda mais sofisticadas do que as anteriores e seguras. Por exemplo, a expressão `typeof(a)` representa o tipo da variável `a` e pode ser usada como um tipo normal, inclusivamente para definir outras variáveis com o mesmo tipo de `a`.

É pena o C padrão não suportar a construção `typeof`.

Exemplo de polimorfismo na biblioteca padrão do C - função qsort

A função **qsort** permite ordenar vetores de qualquer tipo.

```
$ man qsort

QSORT(3)                                    Linux Programmer's Manual
QSORT(3)

NAME
       qsort - sorts an array

SYNOPSIS
       #include <stdlib.h>

       void qsort(void *base, size_t nmemb, size_t size,
                  int(*compar)(const void *, const void *));

DESCRIPTION
       The qsort() function sorts an array with nmemb elements of size size.  The base
argument points to the
       start of the array.

       The contents of the array are sorted in ascending order according to a comparison
function pointed  to
       by compar, which is called with two arguments that point to the objects being
compared.

       The  comparison function must return an integer less than, equal to, or greater
than zero if the first
       argument is considered to be respectively less than, equal to, or greater than
the  second.   If  two
       members compare as equal, their order in the sorted array is undefined.

RETURN VALUE
       The qsort() function returns no value.

CONFORMING TO
       SVr4, 4.3BSD, C89, C99.
```

Exercício: Como faria para ordenar um vetor com componentes do seguinte tipo:

```
typedef struct
{
    int day, month, year ;
} Date;
```

Módulos genéricos (polimórficos) em C

É possível escrever módulos genéricos em C usando macros com várias linhas e a operação de concatenação de tokens que se escreve ##.

Os módulos genéricos são muito difíceis de escrever e de ler, mas são fáceis de usar.

No seguinte módulo, o tipo `Data` passa a ser argumento de duas macros.

Ficheiro Generic_LinkedList.h

```
#include <stdbool.h>

#define Generic_LinkedListHeader(Data)                                  \
                                                                        \
typedef struct Data##Node {                                             \
    Data data ;                                                         \
    struct Data##Node *next;                                            \
} Data##Node, *Data##List ;                                             \
                                                                        \
int Data##ListSize(Data##List l) ;                                      \
bool Data##ListGetByIndex(Data##List l, int idx, Data *res) ;           \
Data##List Data##ListPutAtHead(Data##List l, Data val) ;                \
Data##List Data##ListPutAtEnd(Data##List l, Data val) ;                 \
void Data##ListPrint(Data##List l) ;
```

Ficheiro Generic_LinkedList.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include "Generic_LinkedList.h"

#define Generic_LinkedListImplementation(Data, Print)                   \
                                                                        \
Generic_LinkedListHeader(Data)                                          \
                                                                        \
static Data##List Data##NewNode(Data val, Data##List next) {            \
    Data##List n = malloc(sizeof(Data##Node)) ;                         \
    if( n == NULL )                                                     \
        return NULL ;                                                   \
    n->data = val ;                                                     \
    n->next = next ;                                                    \
    return n ;                                                          \
}                                                                       \
                                                                        \
int Data##ListSize(Data##List l) {                                      \
    int count ;                                                         \
    for( count = 0 ; l != NULL ; l = l->next, count++ ) ;               \
    return count ;                                                      \
}                                                                       \
                                                                        \
bool Data##ListGetByIndex(Data##List l, int idx, Data *res) {           \
    int i ;                                                             \
    for( i = 0 ; i < idx && l != NULL ; i++, l = l->next ) ;            \
    if( l == NULL )                                                     \
        return false ;                                                  \
    else {                                                              \
        *res = l->data ;                                                \
        return true ;                                                   \
    }                                                                   \
}                                                                       \
                                                                        \
Data##List Data##ListPutAtHead(Data##List l, Data val) {                \
    return Data##NewNode(val, l) ;                                      \
}                                                                       \
                                                                        \
Data##List Data##ListPutAtEnd(Data##List l, Data val) {                 \
    Data##List n = Data##NewNode(val, NULL) ;                           \
    if( n == NULL )                                                     \
        return NULL ;                                                   \
    if( l == NULL )                                                     \
        return n ;                                                      \
    else {                                                              \
```

```
        Data##List p ;                                                  \
        for( p = l ; p->next != NULL ; p = p->next ) ;                  \
        p->next = n ;                                                   \
        return l ;                                                      \
    }                                                                   \
}                                                                       \
                                                                        \
void Data##ListPrint(Data##List l) {                                    \
    for(  ; l != NULL ; l = l->next )                                   \
        Print(l->data) ;                                                \
}
```

Exemplo de instanciação do módulo genérico

Ficheiro LinkedList_double.h

```
#include "Generic_LinkedList.h"

Generic_LinkedListHeader(double)
```

Ficheiro LinkedList_double.c

```
#include "Generic_LinkedList.c"

static doublePrint(double d) {
    printf("%lf\n", d) ;
}

Generic_LinkedListImplementation(double, doublePrint)
```

#90

Linguagens e Ambientes de Programação (2010/2011)

Teórica 16 (11/Abr/2011)

Bibliotecas. Biblioteca padrão do C.
Strings.
Funções com número variável de argumentos
Tratamento de erros.
Output formatado
Input formatado
Funções em C
Parâmetros funcionais
Inseguranças da Linguagem C

Bibliotecas

Há linguagens de programação onde as funcionalidades específicas para manipular strings, ficheiros, etc. estão disponíveis ao nível da própria linguagem e não em bibliotecas externas. São os casos das linguagens: Fortran, Cobol e Pascal, por exemplo.

Pelo contrário, noutra linguagens essas funcionalidades estão disponíveis em bibliotecas externas e não ao nível da linguagem. São o caso das linguagens: Java, OCaml, C, C++, etc.

Biblioteca padrão

No caso do C, existe uma pequena biblioteca padronizada, conhecida por **libc**.

A funcionalidade da biblioteca padrão está disponível através de 24 ficheiros de interface, dos quais destacamos os seguintes: <ctype.h>, <limits.h>, <locale.h>, <math.h>, <setjmp.h>. <signal.h>, <stdarg.h>, <stdbool.h>, <stdio.h>, <stdlib.h>, <string.h>, <time.h>.

A aula de hoje envolve temas que permitem a exploração de parte das funcionalidades que estão disponíveis na biblioteca padrão do C.

Outras bibliotecas

A biblioteca padrão do C é muito útil, mas é relativamente limitada (mais limitada do que a do Java, por exemplo). Por isso alguns programas em C costumam recorrer a outras bibliotecas (tipicamente bibliotecas de código *livre*), tais como:

- C Posix library - Extensão da **libc** que permite aceder ao sistema operativo, especialmente no contexto do Unix e variantes, mas não só. O `gcc`, mesmo na versão para Windows, vem com esta biblioteca incorporada, com o nome `glibc` (GNU libc).
- GLib - Para escrever programas com interface gráfico que correm no ambiente Gnome.
- crypt - Para escrever programas que usam criptografia.
- pthread - Para escrever programas que usam threads.

Num sistema Linux observe o conteúdo das diretorias `/lib` e especialmente `/usr/lib` para ver o enorme número de bibliotecas disponível. Os ficheiros de interface correspondentes a todas essas bibliotecas são geralmente guardados na diretoria `/usr/include`.

Usar bibliotecas

Para usar uma biblioteca é preciso primeiro incluir um ou mais ficheiros de interface no código fonte. Por exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <crypt.h>
#include <pthread.h>
#include <glib-2.0/glib.h>
```

Depois de escrito o programa, na altura de compilar é preciso indicar quais as bibliotecas a ligar. Usa-se para isso a opção `-l` do compilador. Exemplo:

```
cc -o myprog myprog.c mym1.c mym2.c -lm -lcrypt -lpthread -lglib-2.0
```

Algumas funções de biblioteca

Apresentamos algumas das funções da biblioteca padrão do C. No Linux, o comando `man` permite obter a documentação sobre qualquer uma destas funções (desde que o pacote "manpages-dev" esteja instalado).

Input/Output

```
#include <stdio.h>

#define EOF    (-1)

typedef struct{...} FILE ;

FILE *stdin, *stdout, *stderr ;

FILE *fopen(char *, char *) ;
FILE *fclose(FILE *) ;
int getc(FILE *) ;
int putc(int, FILE *) ;
int fprintf(FILE *, char *, ...) ;
int fscanf(FILE *, char *, ...) ;
int getchar(void)
int putchar(int)
int printf(char *, ...) ;
int scanf(char *, ...) ;
```

Matemática

```
#include <math.h>

double sin(double) ;
double asin(double) ;
double acos(double) ;
double sqrt(double) ;
```

Strings

```
#include <string.h>

int strlen(char *) ;
char *strcpy(char *, char *) ;
char *strcat(char *, char *) ;
int strcmp(char *, char *) ;
```

Memória

```
#include <stdlib.h>

void *malloc(int) ;
void free(void *) ;
void *realloc(void *, int) ;
```

Exemplo - Cópia de ficheiros

Exemplo que usa funções de biblioteca e ilustra a manipulação de ficheiros.

```
#include <stdio.h>
#include <stdbool.h>

bool Copy(char* dest, char* orig) { /* Copia ficheiro, carácter a carácter. */
    FILE *f, *g ; /* Em C os "canais" do ML chamam-se "ficheiros internos". */
    int c ;    /* Tem de ser int porque fgetc retorna 257 valores diferentes. */

    if( (f = fopen(dest, "w")) == NULL )
        return false ;
    if( (g = fopen(orig, "r")) == NULL )
        return false ;
    while( (c = fgetc(g)) != EOF )
        fputc(c, f) ;
    fclose(f) ;
    fclose(g) ;
    return true ;
```

```
}

int main(void) {
    char in[256], out[256] ;

    printf("IN> ") ; scanf("%s", in) ;
    printf("OUT> ") ; scanf("%s", out) ;
    if( !Copy(out, in) )
        fprintf(stderr, "Copy failed!\n") ;
    return 0 ;
}
```

Strings

Na linguagem C, as strings são simples vetores de caracteres e cada string é terminada pelo carácter nulo, que se escreve `'\0'`. Na seguinte inicialização de variável, a string tem um comprimento nominal de 3, mas internamente ocupa 4 bytes, por causa do terminador.

```
char *str = "ola" ;
```

O facto das strings terem todas uma marca de fim, faz com elas sejam muito flexíveis e práticas de usar. A sua flexibilidade é equivalente à dos vetores acompanhados, pois o programa pode controlar o comprimento duma string.

Funções sobre strings

Para exemplificar a manipulação de strings, eis uma função que conta o número de ocorrências dum carácter numa string. Examine com atenção o ciclo for, porque este é típico das funções que manipulam strings.

```
int Count(char str[], char c)
{
    int i, soma = 0 ;
    for( i = 0 ; str[i] != '\0' ; i++ )
        if (str[i] == c) soma++ ;
    return soma ;
}
```

A chamada `Count("hello",'l')` produz o resultado 2.

A seguinte função acrescente um carácter no final duma string, assumindo que há espaço na string:

```
int Append(char str[], char c)
{
    int i, soma = 0 ;
    for( i = 0 ; str[i] != '\0' ; i++ ) /* procura o final da string. */
        ;
    str[i] = c ;                        /* escreve c por cima de '\0' */
    str[i+1] = '\0' ;                   /* acrescenta '\0' no final da string */
}
```

Eis outra forma de programar as mesmas duas funções, desta vez usando apontadores em vez de indexação:

```
int Count(char str[], char c)
{
    int soma = 0 ;
    char *pt ;
    for( pt = str ; *pt != '\0' ; pt++ )
        if( *pt == c ) soma++;
    return soma;
}

int Append(char str[], char c)
{
    int i, soma = 0;
    char *pt ;
    for( pt = str ; *pt != '\0' ; pt++ ) /* procura o final da string. */
        ;
    pt[0] = c;                          /* escreve c por cima de '\0' */
    pt[1] = '\0';                      /* acrescenta '\0' no final da string */
```

```
}
```

A biblioteca padrão 'string'

A seguinte diretiva no início do nosso programa

```
#include <string.h>
```

permite ganhar acesso a numerosas funções predefinidas de manipulação de strings. Eis as funções de biblioteca mais usadas:

```
size_t strlen(const char *s) ;

char *strcat(char *dest, const char *src) ;
char *strncat(char *dest, const char *src, size_t n) ;

int strcmp(const char *s1, const char *s2) ;
int strncmp(const char *s1, const char *s2, size_t n) ;

char *strcpy(char *dest, const char *src) ;
char *strncpy(char *dest, const char *src, size_t n) ;
```

O seguinte código copia uma string:

```
char a[10], *aa = a, *b = "ola" ;
while( *aa++ = *b++ ) ;
```

Este código tem o mesmo efeito:

```
char a[10], *b = "ola" ;
strcpy(a, b) ;
```

Funções com número variável de argumentos

Vamos adicionar ao módulo LinkedList uma função com número variável de argumentos para criar listas novas. A função tem de ter pelo menos um argumento com nome (neste caso `n`) com informação sobre os argumentos que se seguem (neste exemplo, `n` indica o número de argumentos); os argumentos anónimos são representados por `...`.

```
#include <stdarg.h>

List ListNew(int n, ...)
{
    va_list va ;
    List l = NULL ;
    va_start(va, n) ;
    while( n-- )
        l = ListPutAtEnd(l, va_arg(va, double)) ;
    va_end(va) ;
    return l ;
}
```

Pode ser testada usando:

```
int main(void) {
    List l = ListNew(5, 1.2, 2.6, 3.7, 4.1, 5.0) ;
    ListPrint(l) ;
    return 0 ;
}
```

Os argumentos anónimos não podem ser validados no ponto da chamada. Se, na chamada, forem passados argumentos a mais, os argumentos em excesso são ignorados pela função. Se forem passados argumentos a menos, ou argumentos de tipo errado alguns argumentos da função conterão valores indefinidos.

Para aprender mais sobre funções com número variável de argumentos, usar o comando `man stdarg`.

Tratamento de erros

Tratamento de erros gerados ao nível do sistema operativo ou das bibliotecas

Na linguagem C, muitas das chamadas a funções do sistema operativo ou a chamadas de funções de biblioteca retornam um valor especial a informar que um erro ocorreu. Esse valor deve ser testado e medidas apropriadas devem ser tomadas.

Alguns exemplos:

- A função `fopen` retorna NULL para informar que não conseguiu abrir o ficheiro.
- A função `fputc` retorna EOF para informar que não conseguiu escrever no ficheiro (provavelmente porque o disco está cheio).
- A função `malloc` retorna NULL para informar que não há mais memória disponível para criar variáveis dinâmicas.

Depois de detetado o erro é possível obter mais informação sobre este usando o módulo da biblioteca padrão `errno`. Ao incluirmos o ficheiro de interface <errno.h> ganhamos acesso a uma variável inteira chamada `errno`. Após um erro de sistema, essa variável contém sempre um código que indica qual a razão exata do erro. O valor `0` significa que não há erro. Exemplo:

```
#include <stdio.h>
#include <errno.h>

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

FILE *OpenFile(char *name) {
    FILE *f ;
    if( (f = fopen(name, "w")) == NULL ) {
        switch( errno ) {
            case EMFILE: Error("Too many open files") ;
            case ENAMETOOLONG: Error("Filename too long") ;
            case ENFILE: Error("Too many open files in system") ;
            case ENOENT: Error("No such file") ;
            case EROFS: Error("Read-only file system") ;
            default: Error("File error") ;
        }
    }
    return f ;
}

int main(void) {
    FILE *f = OpenFile("ola") ;
    /* mais código ... */
    fclose(f) ;
    return 0 ;
}
```

Este esquema de tratamento de erros é bastante mau pois obriga a misturar o tratamento de erros com a lógica normal do programa.

Saltos não-locais

A linguagem C não dispõe de qualquer mecanismo de alto-nível para tratamento exceções. Contudo a biblioteca padrão contém um módulo chamado `setjmp` que fornece um mecanismo de saltos não locais que permite tratar exceções a baixo nível.

O módulo `setjmp` disponibiliza as funções `setjmp` e `longjmp`:

- A função `setjmp` guarda numa variável de tipo `jmp_buf` parte do estado da execução do programa, incluindo a indicação de qual é o registo de ativação corrente e quais os valores dos diversos registos do processador, incluindo o program counter. Quando a função `setjmp` é chamada, ela retorna o valor `0`.
- A função `longjmp` repõe o estado da máquina, o que implica um "regresso ao passado" e força `setjmp` a retornar outra vez, desta vez com o valor passado em `longjmp`, um valor que deve ser diferente de `0`. Diz-se que isto é um "salto não-local", porque de certa maneira, a função `setjmp` causa um salto para dentro da função `jmp_buf`.

```
#include <stdio.h>
#include <errno.h>
#include <setjmp.h>

static jmp_buf jbuf ;

FILE *OpenFile(char *name) {
    FILE *f ;
    if( (f = fopen(name, "w")) == NULL )
        longjmp(jbuf, errno) ;
    return f ;
}

int main(void) {
    switch( setjmp(jbuf) ) {
        case 0: {   /* código protegido */
            FILE *f = OpenFile("ola") ;
            /* mais código ... */
            fclose(f) ;
            break ;
        }
        case EMFILE: Error("Too many open files") ;
        case ENAMETOOLONG: Error("Filename too long") ;
        case ENFILE: Error("Too many open files in system") ;
        case ENOENT: Error("No such file") ;
        case EROFS: Error("Read-only file system") ;
        default: Error("File error") ;
    }
    return 0 ;
}
```

Comparando com o OCaml, repare que em C a função `longjmp` desempenha o mesmo papel da expressão `raise`. A função `setjmp`, juntamente com o switch envolvente, desempenha o mesmo papel da expressão `try-with`.

Tratamento de erros gerados ao nível do hardware

Para apanhar erros do género `divisão-por-zero`, a aplicação tem de instalar rotinas de serviço para detetar determinadas interrupções geradas pelo hardware. Para isso é necessário usar os serviços do módulo `signal` da biblioteca padrão.

O seguinte exemplo ilustra a utilização da função `signal` e apanha todas as interrupções geradas a partir do teclado usando CTRL-C.

```
#include <stdio.h>
#include <signal.h>

static void InterruptHandler(int sig)
{
    if( sig == SIGINT ) {
        signal(SIGINT, SIG_IGN) ;
        printf("Interrupt\n") ;
        signal(SIGINT, InterruptHandler) ;
    }
}

int main(void) {
```

```
    int i ;
    signal(SIGINT, InterruptHandler) ;
    for( i = 0 ; i < 1000000000 ; i++ )
        if( i % 100000000 == 0 )
            printf("%d\n", i) ;
   return 0 ;
}
```

É possível fazer um `longjmp` para fora duma rotina de serviço de interrupções? O padrão não dá garantias sobre o assunto e geralmente não funciona mesmo.

Em muitas implementações de C (e.g. GCC) o módulo `setjmp` inclui funções extra (`sigsetjmp` e `siglongjmp`) que permitem fazer isso. Mas esta funcionalidade faz parte do padrão da biblioteca C. Faz sim parte do padrão POSIX para bibliotecas de acesso aos serviços do sistema operativo.

Output formatado

O output formatado em C é gerado usando funções da família `printf`. A função original escreve o output no canal de saída padrão, mas há uma variante que escreve num ficheiro de saída qualquer e outra variante que escreve numa string. Estas funções aceitam um número variável de argumentos.

```
#include <stdio.h>
int printf(const char *format, ...) ;
int fprintf(FILE *out_stream, const char *format, ...) ;
int sprintf(char *out_str, const char *format, ...) ;

#include <stdarg.h>
int vprintf(const char *format, va_list ap) ;
int vfprintf(FILE *stream, const char *format, va_list ap) ;
int vsprintf(char *str, const char *format, va_list ap) ;
```

A **string de formatação** é quase toda copiada literalmente para o output. A exceção são os **especificadores de formato**, começados pelo carácter especial '%', que provocam a escrita dos parâmetros que estiverem colocados após a strings de formatação. Eis um exemplo, seguido do respetivo output:

```
#include <stdio.h>

int main(void) {
   printf("Characters: %c %c\n", 'a', 67) ;
   printf("Decimals: %d %ld\n", 1977, 650000) ;
   printf("Preceding with blanks: %10d\n", 9999) ;
   printf("Preceding with zeros: %010d\n", 9999) ;
   printf("Some different radixes: %d %x %o %#x %#o\n", 100, 100, 100, 100, 100);
   printf("floats: %4.2f %+.0e%E\n", 3.14159, 3.14159, 3.14159) ;
   printf("Width trick: %*d\n", 5, 10) ;
   printf("% \n", "Hello world!") ;
   return 0 ;
}

Characters: a C
Decimals: 1977 650000
Preceding with blanks:       9999
Preceding with zeros: 0000009999
Some different radixes: 100 64 144 0x64 0144
floats: 3.14 +3e+000 3.141590E+000
Width trick:    10
Hello world!
```

A função `printf` retorna o número de caracteres escritos. Em caso de erro retorna um valor negativo.

printf em C++

Em C++ existem outras primitivas de output baseadas no operador "<<", mas a operação `printf` também está disponível.

printf em Java

A operação `printf` é tão popular, que foi incorporada na biblioteca do Java. Em Java, os especificadores de formato são ainda em maior número do no C.

```
System.out.printf("%d\n", 123) ;

123
```

printf em OCaml

A operação `printf` também está disponível na biblioteca do OCaml:

```
# Printf.printf "%d\n" 123 ;;
123
- : unit = ()
```

Input formatado

O input formatado em C é lido usando funções da família `scanf`. A função original lê o input no canal de entrada padrão, mas há uma variante que lê num ficheiro de entrada qualquer e outra variante que lê duma string. Estas funções aceitam um número variável de argumentos.

```
#include <stdio.h>
int scanf(const char *format, ...) ;
int fscanf(FILE *stream, const char *format, ...) ;
int sscanf(const char *str, const char *format, ...) ;

#include <stdarg.h>
int vscanf(const char *format, va_list ap) ;
int vsscanf(const char *str, const char *format, va_list ap) ;
int vfscanf(FILE *stream, const char *format, va_list ap) ;
```

A **string de formatação** é usada de forma literal para fazer emparelhamento com o input, mas há duas exceções: (1) os caracteres brancos (' ', '\t', '\n') emparelham com qualquer sequência de brancos exceção; (2) os **especificadores de formato**, começados pelo carácter especial '%', que causam a leitura de valores para os parâmetros que estiverem colocados após a strings de formatação.

Note que todos os parâmetros após a strings de formatação são parâmetros de saída, portanto são implementados usando apontadores. Eis um exemplo:

```
#include <stdio.h>

int main (void) {
  char str[100];
  int i;

  printf("Enter name: ") ;
  scanf("%s", str) ;
  printf("Enter age: ") ;
  scanf("%d", &i);
  printf("%s is %d years old.\n", str,i) ;
  printf("Enter hexadecimal number: ") ;
  scanf("%x", &i);
  printf("Value %#x (%d).\n", i, i) ;
  return 0;
}
```

A função `scanf` retorna o número de parâmetros lidos com sucesso; em caso de falhanço de emparelhamento, esse valor pode ser inferior ao esperado. Antes do primeiro argumento ter sido lido, caso o input termine subitamente durante emparelhamento que esteja a ser bem sucedido, então é retornado o valor EOF.

scanf em C++

Em C++ existem outras primitivas de input baseadas no operador ">>", mas a operação `scanf` também está disponível.

scanf não existe em Java

A operação `scanf` não existe na biblioteca do Java, mas as classes `Scanner` e `Pattern` oferecem uma solução ainda mais complete e sofisticada.

scanf em OCaml

A operação `scanf` também está disponível na biblioteca do OCaml:

```
# Scanf.scanf "%d" (fun x->x) ;;
123
- : int = 123
```

Funções em C

Quando se define uma função é preciso indicar o seu nome, argumentos e tipo do resultado. Um exemplo:

```
int Sum(int a, int b)
{
    return a + b ;
}
```

As funções podem produzir efeitos laterais. Se o tipo do resultado for **void** isso significa que a função não produz resultado e que essa função só produz efeitos laterais. Exemplo:

```
void OutN(int n)
{
    while( n--)
        printf("%d\n", n) ;
}
```

Parâmetros

Nas funções, a linguagem C suporta apenas passagem de parâmetros por valor. Se quisermos definir funções com parâmetros de saída é necessário passar apontadores como argumentos (isto já foi estudado na secção sobre apontadores da aula passada). Eis um exemplo de função void com um parâmetro de saída, em que esse parâmetro é usado de forma sofisticada em diversos pontos do corpo da função - estude tente perceber o exemplo completamente:

```
void factorial(int i, int *r)
{
    if( i == 0 )
        *r = 1 ;
    else {
        factorial(i-1, r) ;
        *r *= i ;
    }
}

int main(void)
{
    int arg = 10, res ;
    factorial(arg, &res) ;  /* passa uma referencia para a variável x */
    printf("fact(%d)=%d\n", arg, res) ;
    return 0 ;
}
```

Os parâmetros vetor são um caso especial pois são tratados como apontadores constantes.

```
int setArray(int array[], int index, int value)
{
    array[index] = value;
}
```

```
int main(void)
{
    int  a[1] = {1} ;
    setArray(a, 0, 2) ;
    printf ("a[0]=%d\n", a[0]) ;
    return 0;
}
```

Antes da norma C89 os registos e as uniões não podiam ser passadas como parâmetro; só se permitia passar apontadores para eles.

Validação dos parâmetros

Antes da norma C89 não era feita verificação do número e tipo dos argumentos no ponto da chamada das funções. O programa compilava e tudo parecia bem até o programa começar a correr!?

Atualmente, a verificação é devidamente feita e o programador deve garantir que o cabeçalho (protótipo) de cada função é conhecido no ponto da chamada, senão o programa torna-se inválido. No seguinte exemplo, se retirarmos o protótipo, o programa deixa de poder ser compilado:

```
#include <stdio.h>

void g(double d) ;    /* protótipo */

void f(void)
{
    g(1) ;
}

void g(double d)
{
    printf("%5.5lf\n", d) ;
}

int main(void)
{
    f() ;
    return 0 ;
}
```

As funções com zero argumentos devem ser definidas com a palavra void na posição dos argumentos, tal como na função f anterior. Quando a lista de argumentos é vazia isso significa que a função pode ser chamada com qualquer número de argumentos, ou seja que a chamada da função não é validada.

Função main

A função main, onde o programa começa a correr, pode ser escrita usando qualquer dos três cabeçalhos seguintes:

```
int main(void)

int main()

int main(int argc, char *argv[])
```

Parâmetros funcionais

Usando apontadores para funções, em C é possível escrever funções com parâmetros funcionais. Para adicionar ao módulo LinkedList uma função `ListSearch` para procurar o primeiro valor da lista com uma propriedade dada, faz-se assim. A propriedade é especificada usando uma função booleana.

```
bool ListSearch(List l, bool (*f)(Data), Data *res)
{
    for( ; l != NULL ; l = l->next )
        if( (*f)(l->data) ) {
            *res = l->data ;
```

```
            return true ;
        }
    return false ;
}
```

Pode ser testada assim:

```
bool MyF(Data d) {
    return d < 0 ;
}

int main(void) {
    Data d ;
    List l = ListNew(5, 1.2, 2.6, 3.7, -47.1, 5.0) ;
    if( ListSearch(l, &MyF, &d) )
        printf("---> %lf\n", d) ;
    else
        printf("No found\n") ;
    return 0 ;
}
```

Inseguranças da Linguagem C

A linguagem C contém diversas inseguranças:

- Não validação dos limites nos acessos a arrays;
- Não validação dos argumentos em chamadas de funções com um número variável de argumentos;
- Permite manipulação de memória a baixo nível através de apontadores;
- As regras de conversão automática de tipo fazem com que todas quase todas as expressões estruturalmente válidas sejam aceites;
- Nas atribuições pode haver perda de precisão dos valores.

Ferramentas que ajudam a reduzir a insegurança

Para detetar problemas, chamar o compilador de C com todos os warnings ligados ajuda (cc -Wall), mas em geral recomenda-se a utilização duma ferramenta especializada para detetar código duvidoso, antes de o submeter ao compilador de C.

Em 1979 foi criado um verificador chamado **lint** que passou a ser distribuído com todas as versões do Unix.

Uma versão melhorada, atualmente em uso, chama-se **splint**:

```
$ man splint
splint(1)
splint(1)

NAME
       splint - A tool for statically checking C programs

SYNOPSIS
       splint [options]

DESCRIPTION
       Splint  is a tool for statically checking C programs for security vulnerabilities
and com-
       mon programming mistakes. With minimal effort, Splint can be used as a  better
lint(1).If
       additional  effort is invested adding annotations to programs, Splint can perform
stronger
       checks than can be done  by  any  standard  lint.   For  full  documentation,
please  see
```

```
       http://www.splint.org.  This man page only covers a few of the available options.
```

Uma situação dramática em C e C++ é quando, numa fase adiantada de desenvolvimento, um programa grande começa a rebentar com problemas de acesso à memória. Quando não há a mínima ideia sobre qual possa ser a origem do problema, a ferramenta **valgrind** pode ser a salvação. Permite executar o programa executável dentro duma máquina virtual que valida todos os acessos à memória, conseguindo detetar:

- Acessos a variáveis não inicializadas,
- Acessos a blocos de memória fora dos seus limites,
- Acessos a blocos de memória que já não estão em uso por se ter feito o `free` deles,
- Outros tipos de usos errados das funções do módulo `malloc`.

```
VALGRIND(1)                                 Release 3.4.0
VALGRIND(1)

NAME
       valgrind - a suite of tools for debugging and profiling programs

SYNOPSIS
       valgrind [[valgrind] [options]] [your-program] [[your-program-options]]

DESCRIPTION
       Valgrind is a flexible program for debugging and profiling Linux executables. It
consists
       of a core, which provides a synthetic CPU in software, and a series of "tools",
each of
       which is a debugging or profiling tool. The architecture is modular, so that new
tools can
       be created easily and without disturbing the existing structure.

       This manual page covers only basic usage and options. For more comprehensive
information,
       please see the HTML documentation on your system:
/usr/share/doc/valgrind/html/index.html,
       or online: http://www.valgrind.org/docs/manual/index.html.

INVOCATION
       Valgrind is typically invoked as follows:

           valgrind program args
```

Para usar o Valgring, faça assim: Compile o programa da seguinte forma:

```
gcc -g -O0 -o prog prog.c
```

e depois corra o programa assim:

```
valgrind ./prog
```

#90

Linguagens e Ambientes de Programação (2010/2011)

Teórica 17 (13/Abr/2011)

Modelos de execução para diversos tipos de linguagens de programação.
Modelo de execução detalhado para linguagens com escopo estático e aninhamento de funções.

Modelo de execução para diversos tipos de linguagens de programação com escopo estático

Na implementação da maioria das linguagens de programação, é necessário considerar diversos aspetos para garantir que os efeitos da execução dos programas são os desejados. Algumas das questões mais importantes a considerar são as seguintes:

- Avaliação de expressões
- Acesso a variáveis
- Chamada e retorno de funções
- Passagem de funções como argumento
- Funções retornadas por outras funções

Vamos estudar estas questões com a ajuda dum modelo de implementação. Note que o objectivo de qualquer modelo é simplificar o estudo uma realidade, omitindo deliberadamente pormenores não essenciais e outras complicações.

Avaliação de expressões

No nosso modelo usamos uma pilha para avaliar expressões. Para cada operação a executar, primeiro os seus argumentos são empilhados e depois a operação é activada. A operação retira os argumentos do topo da pilha, calcula o resultado e deixa-o no topo da pilha.

Acesso a variáveis

1 - Numa linguagem sem recursividade nem aninhamento de funções ou procedimentos, como é o caso do Fortran 77, cada procedimento é dono duma zona de memória própria onde as suas variáveis locais são guardadas. Todas as variáveis são estáticas, ou seja têm localizações de memória fixas. Diversas chamadas do mesmo procedimento reutilizam a mesma zona de memória... não há problema porque, como foi dito, a recursividade não é permitida.

2 - Numa linguagem com recursividade mas sem aninhamento de funções, como é o caso do C, há dois casos a considerar. As variáveis globais, ou seja as variáveis declaradas fora das funções, são consideradas variáveis estáticas e têm localizações de memória fixas. Já as variáveis locais são criadas dinamicamente no topo duma pilha sempre que a sua função-mãe é activada, e são removidas do topo da pilha quando a execução da função-mãe termina. A partir do interior duma função em C, as variáveis acessíveis são as seguintes: as variáveis locais da activação mais recente dessa função e as variáveis globais que não tenham sido redefinidas dentro da função.

3 - Numa linguagem com recursividade e com aninhamento de funções, como são os casos do Pascal, do OCaml e do GCC, a implementação fica mais complicada porque surgem variáveis declaradas nas funções envolventes, as chamadas **variáveis intermédias**. Para aceder às variáveis intermédias é preciso guardar na activação de cada função o respectivo ambiente de definição ou seja um apontador para a activação mais recente da função envolvente (**static link**). Para aceder a uma variável intermédia é preciso percorrer parte duma lista ligada constituída por static links. Foi Dijkstra quem, em 1960, inventou a técnica dos static links. [Nunca linguagem com escopo dinâmico a implementação seria mais simples por não ser necessario usar static links.]

Nota: GCC é um nome dum compilador que suporta uma linguagem um pouco mais vasta do que o C de base. Mas o termo GCC também pode ser usado para designar a linguagem suportada por esse compilador.

Considere o seguinte exemplo, em GCC. Do ponto de vista do interior da função h, as duas variáveis locais da função f são consideradas variáveis intermédias.

```
#include <stdio.h>
```

```
int i = 1, j = 1, k = 1, l = 1 ;    // variáveis globais

void f(void) {
    int i = 2, j = 2 ;
    void h(void) {        // função aninhada
        int i = 3 ;
        l = i + j + k ;
    }
    void g(void) {        // outra função aninhada
        int i = 7, j = 7, k = 7, l = 7 ;
        h() ;
    }
    g() ;
}

int main(void)
{
    f() ;
    printf("%d\n", l) ;
    return 0 ;
}
```

Chamada e retorno de funções

Nas linguagem que suportam recursividade, o estado da invocação duma função é capturado numa estrutura chamada **registo de activação (frame)**. Num registo de activação guarda-se a seguinte informação: os argumentos e as variáveis locais da função, o endereço de retorno da função, e um apontador para o registo de activação da função que fez a chamada (**dinamic link**).

Se a linguagem também permitir aninhamento de funções, ainda é preciso guardar no registo de activação o ambiente de definição da função ou seja um apontador para o registo de activação mais recente da função envolvente (**static link**). Repare que a função envolvente pode não ser a mesma função que fez a chamada.

Os registos de activação organizam-se de forma natural numa pilha, a chamada **pilha de execução**. A chamada duma função causa o empilhamento dum novo registo de activação; o retorno duma função causa o desempilhar do registo de activação mais recente.

Passagem de funções como argumento - closures

No caso duma linguagem sem aninhamento de funções, como o C, a passagem duma função como argumento é trivial: basta passar o endereço da função.

No caso duma linguagem com aninhamento de funções, a passagem duma função como argumento é mais complicada. A função passada pode ter necessidade de aceder a variáveis intermédias, e por isso, juntamente com o endereço da função tem também de ser passado o respectivo ambiente de definição (static link).

Portanto a implementação da passagem duma função como argumento envolve tecnicamente a passagem dum par ordenado a que se chama **closure** (ou **fecho** em Português):

closure = (endereço da função, static link)

A razão de ser do nome closure é a seguinte: Em geral, uma função é uma entidade "aberta" (incompleta) pois o seu corpo pode referir entidades desconhecidas localmente. Mas o par *função + respectivo ambiente de definição* já é uma entidade fechada pois o ambiente de definição "fecha" (completa) o significado da função.

No seguinte exemplo, em GCC, ocorre uma passagem de função como argumento. Medite no código para confirmar a necessidade das closures.

```
#include <stdio.h>

int i = 1, j = 1, k = 1, l = 1 ;
```

```
void f(void) {
    int i = 2, j = 2 ;
    void h(void) {        // função aninhada
        int i = 3 ;
        l = i + j + k ;
    }
    void g(void (*p)(void)) {     // outra função aninhada com par. funcional
        int i = 7, j = 7, k = 7, l = 7 ;
        (*p)() ;
    }
    g(&h) ;
}

int main(void)
{
    f() ;
    printf("%d\n", l) ;
    return 0 ;
}
```

Funções retornadas por outras funções

No caso duma linguagem sem aninhamento de funções, como o C, implementar o retorno duma função a partir de outra função é trivial: basta retornar o endereço da função-resultado.

No caso duma linguagem com aninhamento de funções, como o OCaml, a implementação desse mecanismo é mais complicada. Quando se retorna uma função, o que tecnicamente precisa de ser retornado é uma closure.

Mas aqui pode haver problema. A closure retornada pode depender de entidades locais à função que produz o resultado. Assim quando a função retorna a closure e termina, o seu registo de activação não pode ser destruído. Uma solução é transferir o registo de activação para o heap, ou seja para a zona das variáveis dinâmicas. Esta é a solução adoptada na linguagem funcional Lua. Outra solução, mais radical, consiste em abandonar completamente o modelo da pilha de execução e passar a criar todos os registos de activação no heap (deixando que seja o gestor de memória a descobrir o momento em que eles podem ser removidos com segurança).

Esta complicação explica o facto do retorno de funções ser um mecanismo não suportado pela maioria das linguagens de programação. Praticamente só as linguagens funcionais é que suportam este mecanismo.

No seguinte exemplo, em OCaml, a função f retorna uma closure que depende do x local ao f. Quando a closure retornada é depois chamada, o x tem de continuar disponível, apesar da execução de f já ter terminado.

```
let f x =
    let g y = x + y in
        g
;;

(f 5) 4 ;;
```

Modelo de execução detalhado para linguagem com escopo estático e aninhamento de funções

Para concretizar melhor as ideias anteriores, vamos desenvolver um modelo de execução ainda mais detalhado para o caso das linguagens com escopo estático e aninhamento de funções.

Usaremos a linguagem C como veículo de formalização, para concretizar melhor as ideias. O código C que vai aparecer tem grandes semelhanças com o código duma máquina virtual típica.

Áreas de memória e registos

O nosso modelo tem três áreas de memória:

- **stack** - Guarda os registos de activação e também como memória de trabalho para avaliar expressões.
- **code** - Guarda o código do programa que está a correr.
- **heap** - Guarda as variáveis dinâmicas do programa.

Pouco será dito sobre a zona de código e sobre o heap. O stack vai receber quase toda a nossa atenção.

Os registos do modelo são os seguintes:

- **SP** - Indica o topo da pilha de execução.
- **FP** - Indica o registo de execução correntemente activo, ou seja para o registo de execução que se encontra no topo da pilha de execução.
- **PC** - Indica qual a instrução corrente, na zona de código.

Eis uma concretização em C destas ideias:

```
#define STACK_SIZE   20000
#define CODE_SIZE    20000
#define HEAP_SIZE    20000

typedef unsigned Word ;
typedef Word *Pt ;

#define Push(v)      (*SP++ = (Word)(v))
#define Pop()        (*--SP)

Word stack[STACK_SIZE] ;
Word code[CODE_SIZE] ;
Word heap[HEAP_SIZE] ;

Pt SP = stack;       // Stack pointer
Pt FP = stack ;      // Frame pointer
Pt PC = code ;       // Program counter
```

Avaliação de expressões

Como já foi dito, usamos a pilha de execução como auxiliar da avaliação de expressões. Os argumentos das operações são empilhados e depois a operação encontra os seus argumentos no topo da pilha.

Eis uma concretização em C destas ideias:

```
void PushConst(Word c) {
    Push(c) ;              // push the constant
}

void Add(void) {
    Word arg1 = Pop() ;    // pop the first argument
    Word arg2 = Pop() ;    // pop the second argument
    Push(arg1 + arg2) ;    // push the result
}

...
```

Acesso a variáveis

Em tempo de execução os nomes das funções não precisam de estar disponíveis. Para referenciar uma variável em tempo de execução basta usar as chamadas **coordenadas da variável**, constituídas por dois valores:

- **Diferença de nível (nesting)** - Diferença de nível de aninhamento entre o ponto da referência e o ponto da declaração.
- **Deslocamento (offset)** - Deslocamento da variável dentro do registo de activação a que pertence.

A diferença de nível representa o número de vezes que é necessário descer pelos static links para se atingir o registo de activação onde se encontra a variável pretendida. No caso do acesso a uma variável local, a diferença de nível é zero.

O compilador da linguagem consegue determinar com muita facilidade as coordenadas que permitem referenciar cada variável. Um ser humano também o faz facilmente. Por exemplo, no primeiro exemplo desta aula, qual é a diferença de nível das utilizações das variáveis i, j, k e l dentro da função f.

Concretização em C:

```
void LoadVarAddr(int nesting, int offset) {
    for( Pt pt = FP ; nesting-- ; pt = SL(pt) ) ;  // find the frame of the var
    Push(pt + offset) ;            // push the var address
}

void Value() {
    Pt pt = Pop() ;         // pop the var address
    Push(*pt) ;             // push the value of the variable
}

void Assign() {
    Pt pt = Pop() ;         // pop the var address
    Pt value = Pop() ;      // pop the value to store
    *pt = value ;           // do the assignement
}
```

Chamada e retorno de funções

No nosso modelo, o formato dum registo de activação é o que se apresenta abaixo. Convenciona-se que registo FP aponta para a célula onde é guardado o dynamic link do registo de activação mais recente. Nos acessos aos argumentos e variáveis da função, usa-se o registo FP como ponto de referência. Assim, os argumentos da função têm offset negativo, a começar em -1, e as variáveis locais têm offset positivo, a começar em 3. No diagrama assume-se que a pilha cresce para cima.

```
          <- SP
Variável local n
...
Variável local 1
Variável local 0
PC
SL
DL        <- FP
Argumento m
...
Argumento 1
Argumento 0
```

A principal complicação envolvida na activação duma função e na criação do seu registo de activação é o calculo do static link (que representa o ambiente de definição dessa função). O static link tem de ser guardado no registo de activação da função para permitir o subsequente acesso a variáveis não-locais. Para determinar o static link é preciso conhecer a **diferença de nível (nesting)** entre o ponto da declaração da função e o ponto da sua chamada. A diferença de nível representa o número de vezes que é necessário descer pelos static links para se atingir o registo de activação onde se encontra o static link a copiar. Numa chamada recursiva a diferença de nível é zero. Na chamada duma função filha, a diferença de nível é -1.

No nosso modelo a chamada e retorno de funções estão organizados da seguinte forma:

- Antes da função ser chamada, os seus argumentos são empilhados;

- No momento da chamada são empilhados o dinamic link, o static link e o endereço de retorno;
- À entrada da função, esta reserva espaço para as suas variáveis locais.
- Quando a função termina ela desempilha o registo de activação completo, incluindo os argumentos.
- A função deixa no topo da pilha o seu resultado.

Concretização em C:

```
#define DL(fp)      ((fp)[0])
#define SL(fp)      ((fp)[1])
#define PC(fp)      ((fp)[2])

void Call(int nesting, Pt addr) {
    Push(FP) ;          // push dynamic link
    if( nesting == -1 )
        Push(FP) ;      // push static link
    else {
        for( Pt pt = FP ; nesting-- ; pt = SL(pt) ) ;
        Push(SL(pt)) ;  // push static link
    }
    Push(PC) ;          // push return address
    PC = addr ;         // jump to the function
}

void Enter(int locSpace) {
    if( STACK_SIZE - (SP - stack) < locSpace + 20 ) {
        fprintf(stderr, "Stack overflow at PC = %d\n", PC) ;
        exit(1) ;
    }
    FP = SP - 3 ;    // FP points to the dynamic link inside the frame
    SP += locSpace ;
}

void Exit(int argSpace) {
    Word res = Pop() ;    // save the result here
    SP = FP - paramSpace; // discard the entire frame, including the arguments
    PC = PC(FP) ;         // restore the program counter
    FP = DL(FP) ;         // restore the previous frame pointer
    Push(res) ;           // push the result
}
```

Passagem de funções como argumento - closures

Já sabemos que a passagem duma função como argumento requer efectivamente a passagem duma closure como argumento. A principal complicação envolvida na criação da closure é o calculo do static link. Mas agora não há qualquer novidade: ele calcula-se exactamente da mesma forma que na chamada duma função normal.

A activação duma função através duma closure é mais simples do que a activação directa duma função pois o static link já se encontra calculado na closure.

Concretização em C:

```
void PushClosureArg(int nesting, Pt addr) {
    if( nesting == -1 )
        Push(FP) ;      // push static link
    else {
        for( Pt pt = FP ; nesting-- ; pt = SL(pt) ) ;
        Push(SL(pt)) ;  // push static link
    }
    Push(addr) ;
}

void CallClosure(int nesting, int offset) {
    for( Pt pt = FP ; nesting-- ; pt = SL(pt) ) ;  // find the frame of the closure-
argument
```

```
    Pt sl = pt[offset] ;       // get the static link from the closure
    Pt addr = pt[offset + 1] ; // get the function address from the closure
    Push(FP) ;                 // push dynamic link
    Push(sl) ;                 // push static link
    Push(PC) ;                 // push return address
    PC = addr ;                // jump to the function
}
```

Funções retornadas por outras funções

O estudo mais detalhado desta questão é um tópico avançado que está fora do âmbito da cadeira de LAP.

Exercícios

- Relativamente ao primeiro exemplo desta aula, mostre o estado da pilha de execução à entrada da função h.
- Relativamente ao segundo exemplo desta aula, mostre o estado da pilha de execução à entrada da função h.

Para efeitos da criação do registo de activação inicial, convém imaginar que cada programa em C está embebido numa função chamada start, sem argumentos. É importante tratar todas as entidades globais do programa como sendo locais à função imaginária start.

Assuma também que a primeira célula da pilha de execução é identificada como posição 0, a segunda célula da pilha de execução é identificada como posição 1, etc.

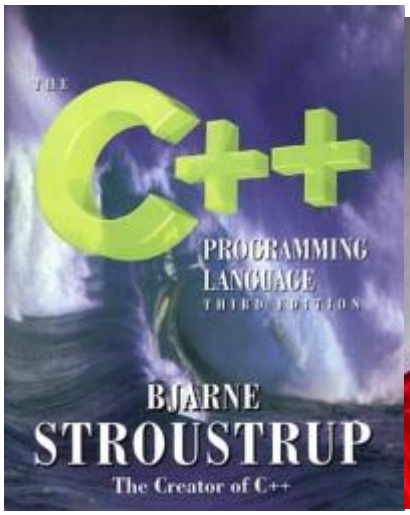
```
Resolução do primeiro exercício:

              <- SP
25: i  3 --h--
    PC ?
    SL 10
    DL 15     <- FP
    l  7 --g--
20: k  7
    j  7
    i  7
    PC ?
    SL 10
15: DL 10
    j  2 --f--
    i  2
    PC ?
    SL 0
10: DL 7
    PC ? --main--
    SL 0
    DL 0
    l  1 --start--
 5: k  1
    j  1
    i  1
    PC ?
    SL ?
 0: DL ?
```

Linguagens e Ambientes de Programação (2010/2011)

Teórica 18 (27/Abr/2011)

Introdução à linguagem C++. Objectivos do C++. Comparação do C com C++.
Suporte para múltiplos estilos de programação em C++. Programação orientada pelos objectos contrastada com programação baseada em valores.

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.

#60

Linguagens e Ambientes de Programação (2010/2011)

Teórica 19 (02/Mai/2011)

Mecanismos do C++ não relacionados com a definição de classes.

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.

#70

Linguagens e Ambientes de Programação (2010/2011)

Teórica 20 (04/Mai/2011)

Introdução às classes em C++. Dois exemplos.

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 ;
}
```

#70

Linguagens e Ambientes de Programação (2010/2011)

Teórica 21 (09/Mai/2011)

A extensibilidade como ideia orientadora da escrita de programas em C++ ou Java.

O uso de testes explícitos de classe compromete a extensibilidade dos programas.

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)
```

Linguagens e Ambientes de Programação (2010/2011)

Teórica 22 (11/Mai/2011)

Mecanismos do C++ relacionados com a definição de classes.

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.

Há mais alguns detalhes relativos à resolução de nomes perante herança múltipla, que não são aqui discutidos.

#60

Linguagens e Ambientes de Programação (2010/2011)

Teórica 23 (16/Mai/2011)

Sistemas de tipo e sua utilidade.
Tipificação estática. Tipificação dinâmica.
Sistemas de tipos seguros.

Tipos

Um **tipo** representa uma coleção de elementos de dados e tem associados um conjunto de *literais* mais um conjunto de *operações*.

Por exemplo, em ML o tipo `int` representa o conjunto do inteiros, tem associados os literais ..., -2, -1, 0, 1, 2, etc., e as operações `+`, `-`, `*`, `div`, `mod`, `succ`, etc.

Numa linguagem sem tipos, e.g. Assembler, uma operação pode ser aplicada a quaisquer dados sem que qualquer validação seja feita. Os dados são vistos como simples sequências de bits e cada operação interpreta uma sequências de bits da maneira que lhe convém.

Numa linguagem com tipos, cada elemento de dados tem um tipo associado. Ao tentar aplicar uma operação a valores que não têm o tipo esperado obtém-se um **erro de tipo**.

Géneros de tipos

Cada linguagem de programação oferece um conjunto de tipos simples e um conjunto de construtores de tipos. É assim possível distinguir dois géneros de tipos:

- **Tipos simples:** tipos pré-definidos cujos valores são escalares atómicos inteiros, caracteres, reais, booleanos, enumerados.
- **Tipos estruturados:** tipos compostos definidos pelo programador aplicando determinadas construções a tipos existentes: arrays, registos, listas, tuplos, classes, funções, apontadores.

Utilidade dos tipos

Para que servem os tipos? No caso das linguagens com tipificação estática:

- Permitem a deteção antecipada de erros, em tempo de compilação.
- São úteis na otimização de código gerado.
- Ajudam a documentar os programas, tornando mais claras as intenções do programador (exceto se for usada inferência de tipos).
- Suportam a definição de módulos que ocultam a representação interna dos valores.

No caso das linguagens com tipificação dinâmica:

- Suportam um modelo de execução no qual se aborta a execução do programa logo que seja detetado algum erro de tipo.

Sistemas de tipos

Um **sistema de tipos** é um conjunto de regras que:

- Associa tipos aos literais e às expressões duma linguagem. Por exemplo o literal `2`, e a expressão `2+5*3` são de tipo inteiro.
- Determina quais são as manipulações permitidas dos valores de cada tipo. Por exemplo, os inteiros podem ser manipulados usando apenas as operações associadas ao tipo inteiro.
- Define como os valores dos vários tipos interagem. Por exemplo, em C a expressão `2 + 5.2` provoca para o valor `2` uma conversão automática de tipo (uma *coerção*).

Para exemplificar, eis três regras que fazem parte do sistema de tipos da linguagem OCaml:

```
                                        exp : bool    exp' : bool
--------------        -------------     ---------------------------
 false : bool          true : bool          exp&&exp' : bool
```

Nestas regras, por cima da barra são colocadas pré-condições e por baixo da barra são colocadas conclusões. As duas primeiras regras dizem que `false` e `true` são valores válidos de tipo `bool`; a terceira regra diz que se `exp` e `exp'` forem expressões válidas de tipo `bool`, então `exp&&exp'` é também uma expressão válida de tipo `bool`.

Não mostramos mais regras, pois a formalização de sistemas de tipo não é matéria de LAP. Mas, pelo menos fica uma ideia.

Estas regras são úteis por várias razões. Por exemplo, servem para provar que o sistema de tipos é consistente (ou seja, que a cada expressão é associado um tipo único), e também servem como base para a escrita duma parte importante do compilador.

Verificação de tipos (Type checking)

Chama-se **verificação de tipos** ao processo de validação dum programa face às regras dum sistema de tipos.

Tipificação estática

Diz-se que uma linguagem usa **tipificação estática** se a verificação de tipos for efetuada em tempo de compilação. Os erros de tipo são detetados antes do programa começar a correr. Nestas linguagens declaram-se os tipos das variáveis, dos argumentos e do resultado das funções para o compilador ter uma base para fazer as suas validações.

As seguintes linguagens usam tipificação estática: ML, C, C++ e Java.

Numa linguagem com tipificação estática, os tipos são associados às expressões que aparecem no texto dos programas e é o próprio texto que é validado. Mais tarde, durante a execução, os tipos dos valores são ignorados de forma geral, porque o compilador já garantiu a ausência de erros de tipo.

Mas atenção: Num programa validado, ou seja sem erros de tipo, podem mesmo assim ocorrer diversos problemas durante a execução:

1. Certos **valores particulares** podem gerar exceções: **zero** usado como denominador duma divisão, ou **null** usado como recetor de mensagem.
2. O programa pode entrar num **ciclo infinito**.
3. O programa pode estar **incorreto**, ou seja mal feito, e não fazer o que o programador pretendia.

Costuma dizer-se que os sistemas de tipos das linguagens com tipificação estática **são conservadores**. Quando se usa tipificação estática, os tipos representam informação incerta, correspondendo a "aproximações" dos valores que realmente serão usados em tempo de execução (e que não se podem prever). Para garantir segurança, o compilador tem sempre de assumir o pior caso, mesmo que o pior caso nem sempre ocorra. Por exemplo, no código C++ que se segue, a variável `a`, de tipo `Animal *` recebe um gato. Depois, mais adiante, tenta-se fazer miar o animal apontado por `a`. Mas o compilador tem considerar que, nessa altura, a variável `a` poderá já não apontar para um gato e, por isso, emite um erro de compilação; no entanto, em tempo de execução até poderia até não haver problema...

```
Animal *a = new Cat ;
...
a->Meou() ;   // ERRO DE TIPO
```

Tipificação dinâmica

Diz-se que uma linguagem usa **tipificação dinâmica** se a verificação de tipos for efetuada em tempo de execução. Os erros de tipo são detetados durante a execução dos programas. Nestas linguagens não de declaram os tipos das variáveis, dos argumentos e do resultado das funções. Nestas linguagens uma mesma variável pode conter valores de tipos diferentes, em diferentes momentos da execução.

As seguintes linguagens usam tipificação dinâmica: JavaScript, Prolog, Lisp, Perl, Python, Ruby, APL e Smalltalk.

Numa linguagem com tipificação dinâmica, os tipos são associados aos valores que são usados em tempo de execução. Portanto, em tempo de execução tem de existir informação de tipo associada aos valores. Sempre que se aplica uma operação a alguns argumentos, o tipo dos argumentos é testado pelo sistema de execução.

Os sistemas de tipos das linguagens com tipificação dinâmica **não precisam de ser conservadores** pois exercem a sua influência durante a execução dos programas, quando os argumentos exatos das operações são conhecidos e estão disponíveis para teste.

Elementos de tipificação dinâmica dentro de linguagens com tipificação estática

Há linguagens de programação com tipificação estática que incluem alguns elementos de tipificação dinâmica.

Um exemplo em Java: a operação **instanceof** permite ao programador testar dinamicamente o tipo de qualquer objeto; a operação de cast aplicada a um objeto também executa um teste de tipo implícito em tempo de execução. Para isto funcionar, os objetos em Java precisam de registar a classe a que pertencem.

Outro exemplo em C++: a operação **dynamic_cast** também envolve um teste dinâmico de tipo. Por isso os objetos em C++ também precisam de registar a classe a que pertencem. (Na realidade, alguns de compiladores de C++ requerem a ativação duma flag chamada RTTI (Run-Time Type Information) para que o cast dinâmico funcione. No caso do g++, essa flag está cativada por omissão.)

Duas escolas de programação

Nos dias de hoje, entre as linguagens de programação mais usadas encontram-se linguagens de programação com tipificação estática e linguagens de programação com tipificação dinâmica. Existem adeptos enquadrados em cada uma das duas escolas de programação, os quais defendem renhidamente a sua "dama" (veja por exemplo esta discussão).

Por exemplo, os adeptos da tipificação estática acreditam que os seus programas são mais seguros depois de verificados, mas os adeptos da tipificação dinâmica argumentam que conseguem programar melhor as suas ideias sem constrangimentos artificiais e que apesar de tudo os seus programas têm provado ser robustos e conter poucos erros.

Na verdade é perfeitamente possível ser bons resultados dentro de cada uma das escolas de programação desde que se usem boas técnicas de desenvolvimento de software. Uma das técnica mais importante é certamente a técnica de escrever muitos tests unitários para validar sistematicamente todos os aspetos do software em desenvolvimento.

Só uma curiosidade: O CLIP, o sistema da informação da FCT, é um exemplo de software da escola da tipificação dinâmica - está escrito em Prolog.

Sistemas de tipos seguros

Um sistema de tipos diz-se **seguro** se conseguir garantir que não ocorrem erros de tipo durante a execução dos programas.

Os sistemas de tipos do C e C++ não são seguros porque:

- Alguns casts permitem violar o sistema de tipos de forma muito básica. Por exemplo é possível converter um inteiro num apontador para função.
- Mesmo sem usar um cast, usando aritmética de apontadores é fácil pôr um apontador para int a apontar para uma variável de tipo double e depois escrever nela (ou em parte dela) um inteiro.
- A gestão explícita de memória também é um problema. Suponhamos que temos um apontador para uma variável dinâmica inteira e que libertamos essa variável. Suponhamos que logo a seguir criarmos uma variável dinâmica de tipo double. Suponhamos ainda que, por acaso, a variável é criada na zona de memória anteriormente libertada. Agora a variável real pode ser acedida usando o apontador para inteiro original, pois apesar de ser considerar um apontador inválido, ele manteve o endereço original.

Exemplos de linguagens com sistemas de tipos seguros:

- ML

- Java
- Smalltalk
- Lisp

Repare que na lista anterior aparecem linguagens com tipificação estática e linguagens com tipificação dinâmica.

#50

Linguagens e Ambientes de Programação (2010/2011)

Teórica 24 (18/Mai/2011)

Polimorfismo e variedades de polimorfismo.

Interoperabilidade entre linguagens. Interoperabilidade entre C e C++. Interoperabilidade entre C e Java.

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.

#50

Linguagens e Ambientes de Programação (2010/2011)

Teórica 25 (23/Mai/2011)

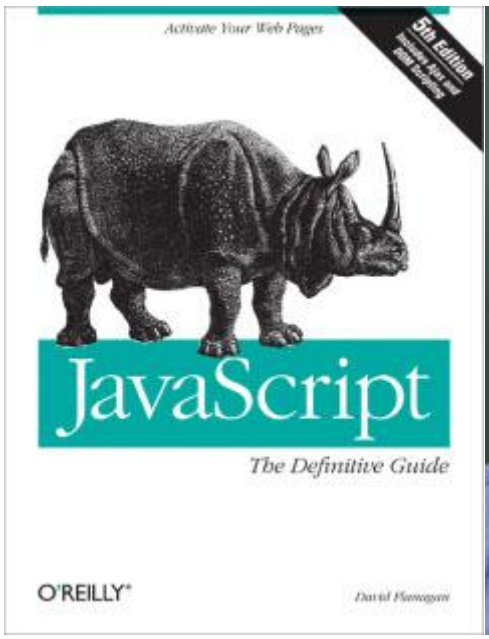
Introdução à linguagem JavaScript.

JavaScript embebido numa consola.

JavaScript embebido num WEB browser.

DOM - Document Object Model.

Introdução à linguagem JavaScript




Brendan Eich

Algumas características

- O principal responsável pelo desenho e implementação foi Brendan Eich. Esse trabalho começou a ser realizado no final de 1995 na empresa Netscape Communications. Inicialmente ele foi contratado para tornar a linguagem Java, já integrada no browser Netscape, mais fácil de usar por não-programadores. Mas rapidamente ele decidiu que que era necessário criar uma linguagem de scripting nova para facilitar a gestão dos elementos das páginas WEB. Acima de tudo era importante tornar a linguagem acessível a WEB designers com poucos conhecimentos de programação. (Correntemente, Brendan Eich é o CTO da Mozilla Corporation.)
- O grande objetivo inicial do Java e depois do JavaScript foi adicionar interatividade a páginas WEB.
- Hoje em dia, a linguagem JavaScript está integrada em diversos tipos de aplicações, não só em browsers. Só alguns exemplos: a aplicação Acrobat usa-a para manipular ficheiros PDF; os controlos remotos programáveis topo-da-gama da Philips usam-na; potencialmente, qualquer aplicação escrita em Java 6.0, ou versão mais recente, pode integrar JavaScript usando o package `javax.script`.
- É uma linguagem padronizada com standard ECMA.
- A especificação diz que a linguagem deve poder ser interpretada, por forma a que seja possível correr scripts gerados dinamicamente.
- Tem uma sintaxe semelhante à do Java e alguns dos seus aspetos superficiais imitam o Java (e.g. classe Math). Contudo todo o resto da linguagem é diferente do Java: trata-se duma linguagem com tipificação dinâmica, que suporta funções que retornam funções (ou seja programação funcional), e ao nível dos objetos ela baseia-se nas ideia de "protótipo" da linguagem Self e não usa classes como a linguagem Java. A escolha do nome JavaScript foi apenas uma decisão de marketing tomada em 1995.
- A maioria das implementações assume que os programas correm dentro dum ambiente, e.g. um browser, que disponibiliza os objetos e os métodos com os quais a linguagem vai interagir.
- Suporta o uso de expressões regulares na manipulação de texto (inspirado no Awk e Perl).
- Tem gestão automática de memória, tal como o Java, mas opcionalmente pode ser usada uma primitiva `delete` para apagar explicitamente objetos, tal como em C++.
- O JavaScript é uma linguagem híbrida que suporta os paradigmas imperativo (procedimental), funcional e orientado pelos objetos.

Padronizações do JavaScript

O esforço de padronização teve início em 1996 e o padrão é conhecido pelo nome ECMA 262. Atualmente, o padrão vai na versão 5.

Versões do JavaScript

A implementação original de Brendan Eich evoluiu com o tempo, e deu origem a duas implementações que atualmente são *open source* e são mantidas pela Mozilla Foundation:

- SpiderMonkey - Escrito em C.
- Rhino - Escrito em Java para permitir usar scripting dentro de programas Java. Serviu de base ao novo package `javax.script`, introduzido no Java 6.0.

O JavaScript tem vindo a evoluir a par com a evolução do padrão ECMA. A versão mais recente do JavaScript implementa a versão 3 do padrão, mas adiciona algumas extensões.

- JavaScript 1.0 - 1995
- JavaScript 1.1 - 1996
- JavaScript 1.2 - 1997
- JavaScript 1.3 - 1998
- JavaScript 1.4 - 1999
- JavaScript 1.5 - 2000
- JavaScript 1.6 - 2006
- JavaScript 1.7 - 2007 - Introduz geradores, iteradores, *compreensões* usando arrays, expressões `let`, etc.
- JavaScript 1.8 - 2008
- JavaScript 1.8.1 - 2008
- JavaScript 1.8.5 - 2010
- JavaScript 2 - Futuro

Algumas implementações disponíveis

A página da bibliografia de LAP disponibiliza implementações do SpiderMonkey JavaScrip para várias plataformas. Tratam-se de interpretadores que funcionam em ambiente de consola e que são ótimos para desenvolver e testar programas. Na mesma página encontra-se disponível um extensão para o Firefox que inclui uma consola de JavaScript e que se destina a fazer debugging de páginas WEB.

Além disso, na coluna esquerda da página de LAP, no link "JavaScript Shell!" está disponível uma consola de JavaScript que funciona dentro do browser e é muitíssimo prática de usar. Escreva os seus programas JavaScript usando um editor de texto e depois vá fazendo copy&paste para dentro da consola para ir testando.

JavaScript embebido numa consola

O JavaScript só pode ser usado na prática, integrado num ambiente de execução que forneça objetos e métodos para interação com o exterior.

A versão de consola do SpiderMonkey corre num ambiente que fornece diversas primitivas, das quais destacamos as seguintes:

```
print      readline      load
build      help          quit
```

Examine bem esta pequena sessão de trabalho com a versão de consola do interpretador SpiderMonkey. Constitui um bom primeiro contacto com a linguagem.

```
// Correr js
$js

//Matemática simples
js> 1 + 1
2

js> d = Math.PI * 2 * 2
```

```
12.566370614359172

js> Math.random()
0.15057766821669438

js> typeof(1)
number

js> typeof(1.0)
number

// Strings
js> "hello, world"[0]
hd

js> "hello, world".replace("hello", "goodbye")
goodbye, world

js> x = 12.4 + "144"
12.4144

js> typeof(x)
string

js> typeof(readline())
> 123
string

js> typeof(parseInt("123", 10))
number

js> parseInt("aaa", 10)    // conversão base 10
NaN

js> parseInt("aaa", 16)    // conversão base 16
2730

// Ciclos
js> for (i = 0; i < 5; i++)
        print(i) ;
0
1
2
3
4

// Arrays
js> a = ["dog", "cat", "hen"]
dog,cat,hen

js> a.length
3

js> a[0] = 123.45
123.45

js> a
123.45,cat,hen

js> typeof(a[0])
number

js> typeof(a)
object


// Funções
js> function f(x) { return x + 1 ; }
```

```
js> typeof(f)
function

js> f(7)
8

js> function curriedAdd(x)
     { return function(y) { return x + y ; } }

js> var g = curriedAdd(5) ;

js> g(1) ;
6

// Fim
js> quit()    // ou CTRL-D

$
```

JavaScript embebido num WEB browser

Virtualmente todos os browsers atuais têm um interpretador de JavaScript embebido. Cada browser proporciona ao JavaScript um ambiente de execução com objetos e métodos que permitem usar essa linguagem para criar páginas WEB interativas.

Chama-se DOM - Document Object Model ao ambiente que qualquer browser é obrigado a disponibilizar ao JavaScript, caso queira suportar a linguagem. O DOM implementa a visão que o JavaScript tem das páginas escritas em HTML e do estado interno do browser. Usando o DOM, o JavaScript consegue examinar e modificar dinamicamente qualquer página WEB e ainda examinar e modificar o estado do browser.

DOM é um padrão controlado pela organização W3C - World Wide Web Consortium, a mesma organização que controla o padrão HTML, XML e muitos outros padrões da WEB.

Para usar JavaScript integrado numa página WEB é necessário saber um pouco de HTML. Nas nossas aulas vamos restinguir-nos à programação de HTML Forms.

Exemplo 1

O seguinte exemplo é uma form simples que permite somar números. A form é constituída por três caixas de texto, a terceira das quais é *read-only*, e por um botão. Por favor, brinque um pouco com a form para perceber o seu comportamento.

Adder

1.2 + 3.4 = 4.6

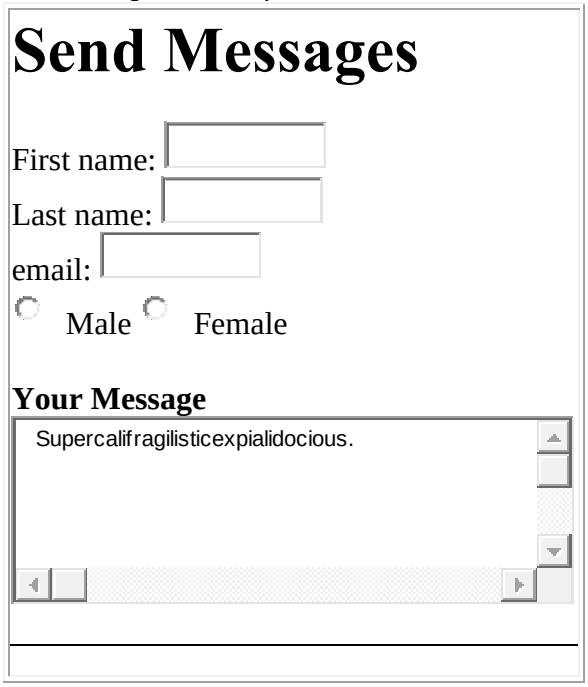
Agora examine o código HTML e JavaScript que implementa a form anterior:

```
<HTML>

<HEAD>
<TITLE>MyDocument</TITLE>

<SCRIPT TYPE="text/javascript">
function Compute(n1, n2) {
    return n1 + n2 ;
}
function RunForm1(form) {
    form.text3.value = Compute(parseFloat(form.text1.value),
```

```
parseFloat(form.text2.value)) ;
}
</SCRIPT>
</HEAD>

<BODY>
<H1>Adder</H1>
<FORM NAME="form1">
     <INPUT TYPE="text" NAME="text1" VALUE="1.2" SIZE=10 style="{ text-align: right }">
   + <INPUT TYPE="text" NAME="text2" VALUE="3.4" SIZE=10 style="{ text-align: right }">
   = <INPUT TYPE="text" NAME="text3" VALUE="4.6" SIZE=25 READONLY style="{ text-align:
right; font-weight : bold }">
 <p> <INPUT TYPE="button" NAME="button1" VALUE="Add" OnClick="RunForm1(form)">
</FORM>
</BODY>

</HTML>
```

Exemplo 2

O seguinte exemplo é uma *form* contendo botões de rádio e uma área de texto. Por favor, teste a form.



Eis o código HTML e JavaScript correspondentes:

```
<HTML>

<HEAD>
<TITLE>MyDocument</TITLE>

<SCRIPT>
function RunForm2(form) {
    var msg = "Sent by: " ;
    msg += form.firstname.value + " " + form.lastname.value ;
    msg += " (" + form.email.value + ") " ;
    if( form.sex1.checked )
        msg += " [male]" ;
    else if( form.sex2.checked )
        msg += " [female]" ;
    else
        msg += " []" ;
    msg += "\nMesg: " + form.textarea1.value ;
    alert(msg) ;
}
</SCRIPT>
</HEAD>
```

```
<BODY>
<H1>Send Messages</H1>
<FORM NAME="form2">
    First name: <INPUT TYPE="text" NAME="firstname"><BR>
    Last name: <INPUT TYPE="text" NAME="lastname"><BR>
    email: <INPUT TYPE="text" NAME="email"><BR>
    <INPUT TYPE="radio" NAME="sex1" VALUE="Male" OnClick="sex2.checked=false"> Male
    <INPUT TYPE="radio" NAME="sex2" VALUE="Female" OnClick="sex1.checked=false">
Female<BR>
    <P><B>Your Message</B><BR>
    <TEXTAREA NAME="textarea1" ROWS="5"
COLS="50">Supercalifragilisticexpialidocious.</TEXTAREA>
    <P><INPUT TYPE="button" NAME="button1" VALUE="Send" OnClick="RunForm2(form)">
    <INPUT type="reset">
</FORM>
</BODY>

</HTML>
```

Exemplo 3

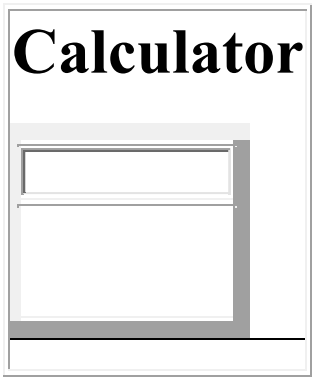
O seguinte exemplo mostra como se pode carregar um novo URL, mudando assim completamente o conteúdo do documento corrente. Também ilustra a criação duma caixa de alerta.

Eis o código HTML e JavaScript correspondentes:

```
<HTML>

<HEAD>
<TITLE>MyDocument</TITLE>

<SCRIPT>
function RunForm3(form) {
    var url = form.url.value ;
    if( url == "" )
        alert("Error: Empty URL") ;
    else
        document.location = form.url.value ;    // Change URL
}
</SCRIPT>
</HEAD>

<BODY>
<H1>Goto WEB page</H1>
<FORM NAME="form3">
    Enter URL: <INPUT TYPE="text" NAME="url" SIZE=40>
    <P><INPUT TYPE="button" NAME="button1" VALUE="Goto" OnClick="RunForm3(form)">
    <INPUT type="reset">
</FORM>
</BODY>

</HTML>
```

Exemplo 4

O seguinte exemplo serve para ilustrar a operação eval do JavaScript. Como já foi dito antes, todas as implementações de JavaScript devem ser capazes de interpretar código gerado dinamicamente. Isso pressupõe à existência duma função `eval` para correr código arbitrário.



Eis o código HTML e JavaScript que implementa a calculadora. A implementação é verdadeiramente simples. Recria no mostrador da calculadora, a pouco e pouco, a expressão introduzida pelo utilizador. No final, quando o utilizador carrega na tecla '=', invoca a função `eval` para avaliar a expressão recebida, sob a forma de string. Este truque funciona bem porque as expressões aritméticas do JavaScript têm a sintaxe habitual da matemática.

```
<HTML>

<HEAD>
<TITLE>MyDocument</TITLE>
</HEAD>

<BODY>
<H1>Calculator</H1>
<FORM NAME="form4">
    <TABLE BORDER=10>
    <TR><TD>
        <INPUT TYPE="text" NAME="display" Size="18">
    </TD></TR>
    <TR><TD>
        <INPUT TYPE="button" NAME="1" VALUE="1" OnClick="form.display.value += '1'">
        <INPUT TYPE="button" NAME="2" VALUE="2" OnCLick="form.display.value += '2'">
        <INPUT TYPE="button" NAME="3" VALUE="3" OnClick="form.display.value += '3'">
        <INPUT TYPE="button" NAME="+" VALUE="+" OnClick="form.display.value += '+'">
    <br>
        <INPUT TYPE="button" NAME="4" VALUE="4" OnClick="form.display.value += '4'">
        <INPUT TYPE="button" NAME="5" VALUE="5" OnCLick="form.display.value += '5'">
        <INPUT TYPE="button" NAME="6" VALUE="6" OnClick="form.display.value += '6'">
        <INPUT TYPE="button" NAME="-" VALUE="-" OnClick="form.display.value += '-'">
    <br>
        <INPUT TYPE="button" NAME="7" VALUE="7" OnClick="form.display.value += '7'">
        <INPUT TYPE="button" NAME="8" VALUE="8" OnCLick="form.display.value += '8'">
        <INPUT TYPE="button" NAME="9" VALUE="9" OnClick="form.display.value += '9'">
        <INPUT TYPE="button" NAME="*" VALUE="*" OnClick="form.display.value += '*'">
    <br>
        <INPUT TYPE="button" NAME="C" VALUE="C" OnClick="form.display.value = ''">
        <INPUT TYPE="button" NAME="0" VALUE="0" OnClick="form.display.value += '0'">
        <INPUT TYPE="button" NAME="=" VALUE="=" OnClick="form.display.value =
eval(form.display.value)">
        <INPUT TYPE="button" NAME="/" VALUE="/" OnClick="form.display.value += '/'">
    <br>
    </TD></TR>
    </TABLE>
</FORM>
</BODY>

</HTML>
```

#70

Linguagens e Ambientes de Programação (2010/2011)

Teórica 26 (25/Mai/2011)

Linguagens de scripting. A linguagem Bash.

Tudo sobre a linguagem JavaScript.

Linguagens de scripting

As **linguagens de programação clássicas** são concebidas para criar estruturas de dados e algoritmos a partir do zero, usando os elementos primitivos da linguagem. Exemplos: ML, C, C++, Java.

Em constaste, nas **linguagens de scripting** assume-se que já existe uma coleção, pronta a ser usada, de componentes escritas noutras linguagens. As linguagens de scripting são concebidas para permitir ligar e organizar componentes existentes **de forma simples, expedita e flexível**. Exemplos: JavaScript, Bash, Tcl, Perl, PHP, Ruby, Python.

Chama-se **script** a um programa escrito numa linguagem de scripting.

Linguagens clássicas versus linguagens de scripting

As linguagens clássicas são geralmente estaticamente tipificadas para permitir detetar cedo os erros que podem ocorrer quando se utilizam elementos primitivos na construção de estruturas de dados complexas. Em constaste, as linguagens de scripting são sempre tipificadas dinamicamente pois um sistema de tipos estático seria uma complicação burocrática que só serviria para atrasar a escrita de scripts.

Outra diferença é o facto das linguagens de scripting serem geralmente interpretadas e não compiladas. Um dos objetivos disso é acelerar o desenvolvimento do código. Outro objetivo é facilitar a geração dinâmica de scripts que possam ser executados imediatamente pelo interpretador.

Nas linguagens de scripting dá-se pouca importância à questão eficiência. Em contrapartida dá-se a máxima importância à simplicidade, flexibilidade de utilização e a um grande poder expressivo que permita escrever scripts compactos. A questão da eficiência é pouco importante porque os scripts tendem a ser pequenos e porque a eficiência da linguagem é dominada pela eficiência das componentes, as quais são normalmente implementadas numa linguagens de programação clássica.

Algumas linguagem de scripting são também concebidas para serem usadas no interior duma aplicação de software, com o objetivo de fornecer ao utilizador um elevado grau de controlo do comportamento da aplicação, incluindo a adição de novas funcionalidades. Se é verdade que o utilizador não pode alterar o código de base da aplicação, ele pode escrever scripts para adaptar a aplicação às suas necessidades. Exemplos: Emacs Lisp é a linguagem de scripting do editor de texto emacs; JavaScript é linguagem de scripting mais usada nos browsers da WEB.

Como escolher entre linguagem clássicas e de scripting

Quando estão em causa aplicações que envolvem acima de tudo a coordenação de componentes já implementadas, podemos escolher programar essa aplicação numa linguagem clássica, e.g. Java, ou numa linguagem de scripting, e.g. JavaScript. Mas estudos mostram que se escolhermos uma linguagem de scripting, tanto o tempo de desenvolvimento como o tamanho da aplicação se reduzem num fator de 5 a 10, em média!

Quando estão em causa aplicações com algoritmos e estruturas de dados complexas, o melhor é usar uma linguagens de programação clássica. Usando uma linguagem de scripting, o script não ficaria mais pequeno, não haveria o

benefício da tipificação estática para ajudar a apanhar antecipadamente erros subtis na utilização das estruturas de dados e, no final, o programa correria 10 vezes mais devagar.

Complementaridade dos dois tipos de linguagens

Se puderem ser usadas em conjunto, os dois tipos de linguagens permitem a criação de ambientes de desenvolvimento e execução de programas particularmente poderosos e flexíveis.

As linguagens de scripting sempre tiveram alguma popularidade, mas ultimamente a sua importância tem aumentado. A principal razão é a tendência atual para escrever aplicações baseadas em componentes já disponíveis. É o que se passa, por exemplo, quando está em causa o desenvolvimento de interfaces gráficas e de aplicações que correm sobre a WEB.

Exemplo: linguagem Bash

Bash (bourne-again shell) é uma linguagem de scripting muito usada no Linux na qual as "componentes" são as aplicações disponíveis. Em Bash, um script implementa uma nova funcionalidade usando as aplicações do sistema. Uma das construções mais importantes do Bash é o **pipe**, que permite ligar o output duma aplicação ao input de outra aplicação. Em Bash também é possível testar o código se saída duma aplicação e tomar decisões em conformidade (zero significa que a aplicação terminou sem erro; um valor diferente de zero representa um código de erro particular). Também é possível fazer é recolher o output duma aplicação numa variável e processar o conteúdo da variável a seguir.

Eis um exemplo de script em bash, retirado daqui. Este script lista na consola o nome de todos os ficheiros HTML que se encontram na diretoria corrente e, além disso, escreve a primeira linha de cada um desses ficheiros num ficheiro chamado File_Heads.

```
#!/bin/sh
# This is a comment
echo "List of files:"
ls -lA

FILE_LIST="`ls *.html`"
echo FILE_LIST: ${FILE_LIST}

RESULT=""
for file in ${FILE_LIST}
do
    FIRST_LINE=`head -1 ${file}`
    RESULT=${RESULT}${FIRST_LINE}
done

echo ${RESULT} | cat > FILE_HEADS

echo "'$RESULT' written Script done. "
```

No Windows, a linguagem de scripting chama-se PowerShell e foi introduzida em 2006 na versão Windows XP SP2. Antes do PowerShell usava-se a linguagem de scripting implementada pelo programa COMMAND.COM.

Comparação do JavaScript com o Java

A primeira é uma linguages de scripting. A segunda é uma linguagem clássica.

JavaScript	**Java**
Tipificação dinâmica	Tipificação estática
Baseada em protótipos	Baseada em classes
Herança usando o mecanismo dos protótipos	Herança através da hierarquia de classes

Podem ser adicionados novos membros a objetos individuais	Não é possível a adição dinâmica de novos membros
Estilo livre onde a maioria das declarações são opcionais	Estilo rígido a pensar na segurança

---

Palavras reservadas do JavaScript

Esta é a lista das principais palavras reservadas em JavaScript:

```
break     const      delete   for        import   new      this     void
case      continue   do       function   in       return   typeof   while
default   export     if       else       switch   var      with
```

Muitas das palavras anteriores são também palavras reservadas em Java.

As restantes palavras reservadas do Java também estão reservadas em JavaScript, apesar de não serem usadas de momento. Todos os interpretadores de JavaScript deveriam proibir a utilização destas palavras. Contudo alguns não o fazem.

Variáveis

O JavaScript é uma linguagem **dinamicamente tipificada**. Uma consequência disso é o facto das **variáveis não terem tipos associados**. Os tipos ficam associados aos os valores e não variáveis. Exemplo:

```
var x = 34 ;
x = "Hello!" ;   // x pode conter um valor de qq tipo
```

Declaração de variáveis

A palavra `var` permite declarar variáveis:

- Dentro duma função, `var` declara uma **variável local**.
- O **argumento duma função**, também declara implicitamente uma variável local.
- Fora duma função, `var` declara uma **variável global**.
- Usa-se **escopo estático** na resolução de nomes.
- Uma função declarada dentro dum bloco têm como âmbito toda a função envolvente, ou seja, **não há âmbito de bloco**.

Exemplo com variáveis globais e variáveis locais:

```
var x = 5 ;
var z = 7 ;

function f(x) {
    print(x) ;
    print(y) ;   // Não inicializada ainda
    x = 1 ;
    var y = 2 ;
    print(x) ;
    print(y) ;
    print(z) ;
    return x ;
}

f(6) ;
print(x) ;

// Output: 6  undefined  1  2  7  5
```

Exemplo com aninhamento de funções:

```
function f(x) {
    function g(y) {
        var x = 10 ;
        print(y) ;
        print(x) ;    // Imprime o x local
        return 0 ;
    }
    print(x) ;
    g(1) ;
    print(x) ;
}

f(6) ;

// Output: 6  1  10  6
```

Exemplo relativo ao facto de não existir âmbito de bloco:

```
function f() {
    var x = 1 ;
    {
       var x = 2 ;
    }
    print(x) ;     // Escreve 2
}
```

Variáveis indefinidas

Variáveis declaradas num determinado âmbito, ficam com o valor `undefined` enquanto não forem inicializadas.

Exemplos sobre variáveis indefinidas:

```
print(zzz) ;   // Lança a exceção ReferenceError se zzz nao estiver declarada
var zzz ;
if( zzz === undefined )   // Pode testar-se se uma variável está indefinida.
    print("zzz is undefined") ;
if( !zzz )     // undefined comporta-se como false num contexto booleano
    print("zzz is undefined") ;
```

Atribuição a variáveis

A atribuição efetua-se usando o operador =. É possível efetuar uma atribuição a um nome ainda não definido. Nesse caso é automaticamente criada uma variável global inicializada.

```
x = 5 ;
y = 7 + 5 ;
```

Constantes

A palavra `const` permite declarar constantes:

```
const x = 5 ;
```

Tipos primitivos

Os tipos primitivos são os seguintes:

- **number** - Mistura reais e inteiros. O maior valor é 1.7976931348623157e+308. 0377 é um valor em octal e 0xFF é um valor em hexadecimal.
- **boolean** - Tem os valores `false` e `true`.
- **string** - Por exemplo `""`, `''`, `"Hello"`, `'Hello'`, `"inner 'string' "`, `'inner "string" '`.
- **null** - Este tipo só tem o valor `null` e serve para atribuir a uma variável para indicar que esta não tem valor.
- **undefined** - Este tipo só tem o valor `undefined`, que é valor das variáveis não inicializadas.

Note que em JavaScript, uma string não é um objeto. No entanto também há objetos de tipo String que simplesmente encapsulam valores primitivos de tipo string. Em Javascript as strings são imutáveis, tal como em Java.

O operador `typeof` pode ser usado para saber o tipo de qualquer valor.

São efetuadas **conversões automáticas de tipo**, entre os tipos primitivos. Exemplos:

```
"The answer is " + 42       // produz "The answer is 42"
42 + " is the answer"       // produz "42 is the answer"
"37" - 7                    // produz 30
"37" + 7                    // produz "377"
true + 7                    // produz 8
```

Operadores

Eis a tabela de operadores do JavaScript ordenada por prioridade decrescente:

```
member                    .  []
call / create instance    ()  new
negation/increment        !  ~  -  +  ++  --  typeof  void  delete
multiply/divide           *  /  %
addition/subtraction      +  -
bitwise shift             <<  >>  >>>
relational                <  <=  >  >=  in  instanceof
equality                  ==  !=  ===  !==
bitwise-and               &
bitwise-xor               ^
bitwise-or                |
logical-and               &&
logical-or                ||
conditional               ?:
assignment                =  +=  -=  *=  /=  %=  <<=  >>=  >>>=  &=  ^=  |=
comma                     ,
```

Operadores de igualdade

```
==          Igualdade, produz true se os argumentos forem iguais (após possíveis
conversões automáticas de tipo).
!=          Desigualdade, produz true se os argumentos forem diferentes.
===         Igualdade estrita, produz true se os argumentos forem iguais e do mesmo tipo.
!==         Desigualdade estrita, produz true se os argumentos forem diferentes ou se
forem de tipos diferentes.
```

Expressões regulares

O JavaScript suporta expressões regulares semelhantes às da linguagem Perl.

A seguinte expressão regular representa dois "a"s seguidos de zero ou mais dígitos:

```
re = /aa\d*/ ;
re = new RegExp("aa\\d") ;          // Equivalente
```

A próxima expressão regular, mais abaixo na caixa, representa um "d" seguido de um ou mais "b"s seguido dum "d". As flags "i" e "g" indicam que o emparelhamento deve ignorar a caixa das letras e que deve ser global.

O método `test` determina se uma string emparelha com a expressão regular.

O método `exec` produz um array com o resultado do emparelhamento na posição 0 do array, mais os resultados dos emparelhamentos das sub-expressões entre parêntesis. Se a expressão regular tiver a flag "g" ligada, então sucessivas chamadas de `exec` produzem sucessivos resultados de emparelhamentos até ser retornado `null`; sem a flag "g" apenas o resultado do primeiro emparelhamento é retornado.

```
var re = /d(b+)(d)/ig ;
re = new RegExp("d(b+)(d)", "ig") ;        // Equivalente
var b = re.test("cdbBdbsbz") ;             // resultado: true
var arr = re.exec("cdbBdbsdbdz") ;         // primeiro resultado: ["dbBd", "bB", "d"]
```

As expressões regulares suportam ainda os métodos `match`, `search`, `replace`, `split`.

Arrays

Em JavaScript os arrays podem ser inicializados, pelo menos de duas maneiras diferentes. Exemplo:

```
var colors = ["Red", "Green", "Blue"] ;
var colors = new Array("Red", "Green", "Blue") ;    // Equivalente
```

Tal como em Java os índices começam em zero e existe uma propriedade `length`.

```
var len = colors.length ;                           // Vale 3
```

Eis um exemplo dum array de comprimento 6 com apenas 4 elementos. Dois elementos estão indefinidos.

```
var colors = ["Red", , , "Green", "Blue","Yellow"] ;
```

Ao contrário do Java, os arrays crescem automaticamente. Basta atribuir a uma posição inexistente para o array crescer.

```
var colors = [] ;                                   // Array vazio
colors[2] = "Blue" ;
var len = colors.length ;                           // Vale 3
var t = typeof(colors[0]) ;                         // Vale undefined
```

Para fazer crescer um array na primeira posição livre, fazer assim:

```
colors[colors.length] = "Yellow" ;
colors.push("Yellow") ;                           // Equivalente
```

Para aceder e remover o último elemento dum array fazer:

```
var last = colors.pop() ;
```

É possível escrever diretamente na propriedade `length` dum array para fazer um array crescer, ou para truncar o array:

```
colors.length = 2 ;
```

Para percorrer os elementos dum array pode usar-se um `for`, mas também se pode fazer assim:

```
var colors = ["Red", "Green", "Blue"] ;
colors.forEach(function(c) { print(c) ; }) ;      // Iteração usando função anónima
```

... ou assim:

```
var colors = ["Red", "Green", "Blue"] ;
for( i in colors ) print(colors[i]) ;         // Iteração usando for..in
```

Eis um array a duas dimensões, 2x3:

```
var table = [[0, 1, 2],
             [3, 4, 5]] ;
var r = table[0][2] ;                             // Vale 2
```

Outros métodos disponíveis para arrays: `join`, `reverse`, `shift`, `slice`, `splice`, `sort`, e muitos outros.

Funções

O JavaScript suporta o paradigma de programação funcional pois inclui funções anónimas, funções de ordem superior e funções que retornam outras funções.

```
function square(n) { return n * n ; }
var square = function(n) { return n * n ; } ;        // Equivalente
```

Eis um exemplo duma função de ordem superior, que depois é chamada usando uma função anónima como argumento:

```
function map(f, a) {
    var result = [] ;
    for( var i = 0 ; i < a.length ; i++ )
        result[i] = f(a[i]) ;
    return result;
}

var a = map(function(x) { return x * x ;}, [0, 1, 2, 3]) ;   // Vale [0, 1, 4, 9]
```

Nas chamadas das funções o número de argumentos não é validado: argumentos a mais na chamada são ignorados; argumentos a menos na chamada ficam indefinidos.

Dentro da cada função há um array predefinido chamado `arguments` que representa a sequência de argumentos realmente usados na chamada. Assim é fácil implementar funções com um número variável de argumentos, como no seguinte exemplo:

```
function allAll() {
    var result = 0 ;
    for( var i = 0 ; i < arguments.length ; i++ )
        result += arguments[i] ;
    return result;
}

var i = allAll(1,2,3,4,5) ;                    // Vale 15
```

A passagem de argumentos de tipos primitivos é feita por valor. Os objeto-argumento são passados por referência.

Eis algumas funções predefinidas em JavaScript:

- **eval(string)** - Avalia uma string contendo código JavaScript.
- **isFinite(number)** - Testa se um número é finito.
- **isNaN(number)** - Teste se um número é a constante NaN.
- **parseInt(string, radix)** - Converte string em número inteiro.
- **parseFloat(string)** - Converte string em número real.
- **Number(obj)** - Converte um objeto num número.
- **String(obj)** - Converte um objeto numa string.

Objetos

Em JavaScript para além dos tipos primitivos, temos os tipos objeto. Os arrays são considerados objetos.

Como habitualmente, um objeto é um elemento de dados que possui identidade e que interage com outros objetos através da troca de mensagens.

Em JavaScript os objetos predefinidos principais são os seguintes: Date, Array, Boolean, Function, Math, Number, RegExp e String . Mas no ambiente de execução envolvente, estão geralmente disponíveis muitos mais objetos predefinidos. Por exemplo, no ambiente dum browser, todos os tipos de objetos previstos no DOM estão disponíveis: Document, Window, Form, Link, etc.

Objetos literais

Em JavaScript, os objetos comportam-se em grande medida como simples dicionários. Eis um exemplo de **objeto literal**, que define uma pessoa:

```
var p = {name: "Pedro", address: "Lisboa", age: 42} ;
```

Para aceder a uma componente dum objeto, há duas notações disponíveis:

```
var n = p.name ;
var n = p["name"] ;                 // Equivalente
p.name = "Pedrinho" ;               // Muda nome
```

Se atribuirmos a um membro inexistente dum objeto, esse membro passa imediatamente a existir para esse objeto individual:

```
p.born = "Porto" ;
```

Para apagar um membro, usa-se a palavra `delete`:

```
delete p.born ;
```

Eis um objeto mais complexo:

```
var myStructure = {
  name: {
    first: "Mel",
    last: "Smith"
  },
  age: 33,
  hobbies: [ "chess", "jogging" ]
} ;
```

Construtores

Os objetos literais são muito úteis, mas não são suficientes quando é preciso definir diversos objetos do mesmo tipo: por exemplo, para partilha dos mesmos métodos, ou para efeitos de utilização do operador `instanceof`.

A definição de novos **tipos-objeto** faz-se através de **construtores**. Um construtor é uma função JavaScript que se preocupa em inicializar objetos dum dado tipo. A única particularidade determinante dum construtor é a seguinte:

- O construtor é uma função que se destina a ser chamada no contexto do operador `new`.

Dentro do construtor usa-se o nome `this` para inicializar os membros de cada objeto. O nome do construtor acaba por ser também o nome de tipo que pode ser usado do lado direito do operador `instanceof`.

Abaixo define-se um construtor chamado Car. Os métodos do objeto são membros funcionais atribuídos dentro do construtor. Note que os métodos usam a palavra `this` para referir o objeto a que são aplicados.

```
function carAsString() {
   return "A Beautiful " + this.year + " " + this.make + " " + this.model ;
}

function Car(make, model, year) {
   this.make = make ;
   this.model = model ;
   this.year = year ;
   this.asString = carAsString ;    // Também se podia ter usado uma função anónima
}

var car1 = new Car("Toyota", "Corolla", 2002) ;
```

Um objeto definido através dum objeto literal, considera-se que foi definido usando o construtor Object.

Modificação dinâmica dum tipo-objeto (Herança dinâmica)

Um tipo-objeto tem um membro chamado `prototype` através da qual é possível alterar a funcionalidade desse tipo, e consequentemente de todos os objetos desse tipo.

```
car1.changeMake("TTT") ;                                    // Gera um erro
Car.prototype.changeMake = function(make) { this.make = make ; } ;
car1.changeMake("TTT") ;                                   // Agora já não gera um erro
car1.asString() ;                                          // Permite confirmar que a marca
mudou
```

O código anterior funciona porque **todos os objeto herdam dinamicamente do respetivo protótipo**. Quando se tenta aceder a um membro dum objeto, se esse membro não estiver diretamente disponível no objeto, então a procura continua no protótipo.

O protótipo correspondente a um construtor é criado quando se chama o construtor pela primeira vez. Dessa primeira vez, além do protótipo é criado ainda um objeto normal; a partir daí só são criados objetos normais.

Usando esta técnica é possível alterar inclusivamente os membros dos tipos-objeto predefinidos, sendo por exemplo possível adicionar novos métodos ao tipo Array.

Criação de hierarquias

Manipulando diretamente o membro `prototype` de um ou mais tipos-objeto é possível criar uma hierarquia de tipos com herança dinâmica.

No seguinte exemplo definem-se dois tipos-objeto independentes, `Super` e `Sub`. Depois atribui-se uma instância de `Super` a `Sub.prototype`. Na prática isto tem o seguinte efeito: Quando se tenta aceder a um membro dum objeto de tipo Sub, se esse membro não estiver diretamente disponível no objeto, então a procura continua no respetivo protótipo, que agora é um objeto de tipo Super. Se também não estiver aí definido então a busca prossegue no protótipo desse objeto, ou seja em `Super.prototype`.

Usando esta ideia, vê-se que é fácil criar uma hierarquia de tipos, com tantos níveis quanto se deseje.

```
function Super() {
   this.p = function() { print("super") ; }
   this.q = function() { print("super") ; }
}

function Sub() {
   this.q = function() { print("sub") ; }
}

Sub.prototype = new Super() ;   // Alteração explícita do protótipo para efeitos de
herança

var a = new Super() ;
var b = new Sub() ;

a.p() ;                      // Escreve "super"
a.q() ;                      // Escreve "super"
b.p() ;                      // Escreve "super" - este método foi herdado
b.q() ;                      // Escreve "sub"
```

Dúvida que surgiu no final da aula

Será que os objetos normais têm um campo `prototype`?

O campo `prototype` está definido apenas nos tipo-objeto, estando disponível para ser consultado e alterado, como vimos.

Pode testar-se se um objeto herda dum determinado protótipo usando o operador `instanceof`.

Linguagens e Ambientes de Programação (2010/2011)

Teórica 27 (30/Mai/2011)

Ambientes de programação. Estudo breve de vários casos.

Ambientes de programação

Um **ambiente de programação** é uma coleção de ferramentas usadas no desenvolvimento de software. Um bom sistema de desenvolvimento aumenta a produtividade dos programadores e também contribui um pouco para aumentar a qualidade do software produzido.

Um ambiente minimal é tipicamente constituído por:

- um sistema de ficheiros,
- um editor de texto,
- um interpretador (ou, em alternativa, um compilador e um ligador)

Um ambiente sofisticado é tipicamente constituído por:

- um sistema de ficheiros,
- numerosas ferramentas, incluindo editor, (depurador) debugger, gerador de documentação, framework para testes unitários, etc.
- essas ferramentas estão bem integradas umas com as outras;
- são oferecidas ao utilizador duas interfaces uniformes: uma interfaces gráfica e uma interface baseada em linha de comando.

O ambiente de desenvolvimento JDK para Java

O ambiente de desenvolvimento não-visual JDK (Java Developer's Kit) consiste num conjunto de ferramentas que ajudam a **desenvolver**, **testar**, **documentar** e **executar** programas em Java. Estas ferramentas podem ser usadas na linha de comando, mas também podem ser integradas em ambientes de desenvolvimento gráficos, como por exemplo o Eclipse. O JDK é gratuitamente disponibilizado pela empresa Sun e existem versões para Windows, MacOS, Linux, etc.

Ferramentas do JDK

javac	Compilador
java	Executor de *programas independentes*
javadoc	Gerador de documentação
appletviewer	Executor de *applets*
jdb	Depurador

```
javap          Disassembler
javah          Gerador de header files para métodos nativos escritos em C
```

Como se compilam programas em Java?

- Os programas em Java compilam-se por meio do comando **javac**. Por exemplo: `javac Sets.java`. O comando **javac** permite compilar tanto **programas independentes** como **applets**.
- Os ficheiros compilados pelo comando **javac** têm obrigatoriamente a extensão ".java".
- Quando cativado, o compilador cria na diretoria corrente diversos ficheiros com a extensão ".class", um por cada classe ou interface definidas nos ficheiros fonte. Os ficheiros ".class" contêm código executável, num formato padrão chamado **bytecode**.

Como se executam programas em Java?

- Se se tratarem de **programas independentes**, executam-se usando o comando **java**. Por exemplo: `java TestSets`. Neste exemplo, "TestSets" refere-se ao ficheiro "TestSets.class" na diretoria corrente: repare que o comando `java` não admite a escrita da extensão ".class".
- Se se tratarem de **applets**, executam-se usando o comando **appletviewer** ou então por intermédio dum browser WWW.
- A componente de software que permite executar ficheiros de de *bytecode* chama-se **Java Virtual Machine**. Está incorporada no comando **java**, no comando **appletviewer** e na maioria dos browsers WWW modernos.

Como se documentam programas em Java?

- O gerador de documentação, **javadoc** converte ficheiros ".java" em ficheiros HTML, visualizáveis usando qualquer browser WWW. Estes ficheiros HTML incluem documentação sobre as classes, interfaces, métodos, variáveis e exceções definidos nos ficheiros fonte e ainda o conteúdo de comentários especiais, da forma /** ... */, escritos pelo programador. A vasta documentação sobre a plataforma Java que é fornecida pela Sun foi gerada pelo comando **javadoc**.

As ferramentas do OCaml

A distribuição do OCaml inclui uma vasto conjunto de ferramentas disponíveis no Windows, Linux e MacOS. Estas ferramentas podem ser usadas na linha de comando, mas também podem ser integradas em ambientes de desenvolvimento gráficos.

Repare que a lista de ferramentas abaixo inclui um profiler que serve para o programador descobrir quais as partes do programa onde se gasta mais tempo. Essas são as partes do programa que mais interessa optimizar.

Ferramentas do OCaml

```
ocaml          Interpretador
ocamlc         Compilador para a máquina virtual CAML
ocamlopt       Compilador de código nativo
ocamlrun       Executor da máquina abstrata CAML
ocamlbrowser   Permite navegar e inspecionar o conteúdo dos módulos de biblioteca
ocamldebug     Depurador de código fonte
ocamldoc       Gerador de documentação
ocamldep       Inspeciona um conjunto de ficheiros fonte e gera automaticamente informação de
dependências para a ferramenta make
ocamlcp        Profiler, insere no código fonte a contagem de quantas vezes cada função é
chamada, etc.
ocamlprof      Interpreta o output da execução de programas processados usando ocamlcp
ocamllex       Gera reconhecedores de expressões regulares diretamente a partir dessas
expressões
ocamlyacc      Gera parsers diretamente a partir de gramáticas LALR(1)
ocamlp4        Processador do código fonte de programas ocaml (para pretty-print, por exemplo)
```

ocamldumpobj Disassembler de ficheiros .cmo
ocamlobjinfo Inspeciona ficheiros .cmo, .cmi, .cma

Unix e Gnu/Linux

O Unix é um sistema operativo mas também um sistema de desenvolvimento e manutenção de software. Atualmente a sua versão open-source, chamada de Gnu/Linux, é muito popular e tem sido usada no desenvolvimento de software "open-source" e de software comercial.

O Unix disponibiliza uma grande quantidade de ferramentas de desenvolvimento, mas deve ser dito que essas ferramentas foram nascendo de forma caótica e que não há uma interface consistente entre elas. Algumas ferramentas são também muito complexas, embora cumpram os seus objetivos de forma admirável, na maior parte dos casos.

gnu tools

Os **gnu tools** são um conjunto de ferramentas que ajudam a produzir software portável, que funciona em diversas versões do Unix e noutros sistemas operativos.

autoconf Ferramenta para criar projetos de software portáveis e reconfiguráveis
automake Gerador automático de makefiles
libtool Ferramenta para criar bibliotecas de software

make

A utilitário **make** pode ser usado para gerir automaticamente pequenos projetos. A sua principal utilização é determinar automaticamente que partes do projeto precisam de ser recompiladas e produzir os comandos para concretizar a recompilação. O make considera as dependências entre os diversos ficheiros do projeto para saber o que é precisos fazer em cada momento.

Geralmente, cada projeto contém um ficheiro chamado **Makefile** onde se declaram as dependências entre os ficheiros, e onde se definem regras que dizem como os diversos ficheiros devem ser atualizados. Para além da recompilação, podem ser definidas outras ações, tais como apagar os ficheiros auxiliares do projeto, ou produzir o arquivo da distribuição do projeto.

O seguinte exemplo é um ficheiro Makefile que poderia ter sido usado no primeiro projeto de LAP (o projeto das árvores de decisão). O ficheiro declara as dependências entre ficheiros e tem regras que dizem como obter uns ficheiros a partir de outros (as regras começam obrigatoriamente por um tab).

```
APP = dt
VERSION = 0.1
COMP = ocamlc
INTERFS = DTree.mli
OBJS = DTree.cmo dt.cmo
GROUP = 123_456

$(APP): $(OBJS)
        $(COMP) -o $(APP) $(INTERFS) $(OBJS)

%.cmo: %.ml
        $(COMP) $(INTERFS) -c $<

clean:
        rm -f $(APP) *.cmo *.cmi

dist: clean
        mkdir $(GROUP)
        cp *.ml *.mli $(GROUP)
        zip -r $(GROUP).zip $(GROUP)
        rm -rf $(GROUP)
```

Eis uma pequena sessão interativa, que pressupõe a existência do ficheiro `Makefile` na diretoria corrente:

```
$ ls
dt.ml  DTree.ml  DTree.mli  Makefile
$ make DTree.cmo
ocamlc DTree.mli -c DTree.ml
$ make
ocamlc DTree.mli -c dt.ml
ocamlc -o dt DTree.mli DTree.cmo dt.cmo
$ make clean
rm -f dt *.cmo *.cmi
$ make dist
rm -f dt *.cmo *.cmi
mkdir 123_456
cp *.ml *.mli 123_456
zip -r 123_456.zip 123_456
  adding: 123_456/ (stored 0%)
  adding: 123_456/DTree.ml (deflated 72%)
  adding: 123_456/dt.ml (deflated 67%)
  adding: 123_456/DTree.mli (deflated 65%)
rm -rf 123_456
$ ls
123_456.zip  dt.ml  DTree.ml  DTree.mli  Makefile
```

gdb

O Gnu Debugger trabalha com linguagens implementadas nativamente e tem um funcionalidade extensíssima. Mostramos só como usar o `gdb` para descobrir qual a função onde um programa está a *rebentar*.

O seguinte programa está errado:

```
char str[128] ;

void f(void)
{
    char *pt = str ;
    int i ;
    for( i = 0 ; i < 10000 ; i++ ) <-- Provoca estoiro
        *pt++ = 'a' ;
}

void g(void) {
    f() ;
}

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

O `gdb` permite mostrar o conteúdo da pilha de execução no momento do estoiro, o que geralmente é muito útil para descobrir as razões do erro.

```
$ ./a
Segmentation fault (core dumped)
$ gdb ./a
GNU gdb 6.6-debian
Copyright (C) 2006 Free Software Foundation, Inc.
(gdb) run
Starting program: /media/EXTERN/amd1/z/a

Program received signal SIGSEGV, Segmentation fault.
0x0804835d in f ()                    <-- Estoiro na função f
(gdb) backtrace
#0  0x0804835d in f ()
#1  0x0804837b in g ()
#2  0x08048390 in main ()             <-- Pilha de execução no momento do estoiro
```

```
(gdb)
```

cvs

O **cvs** (Concurrent Versions System) um sistema de apoio ao desenvolvimento de projetos de software. Tem duas funcionalidades essenciais:

- Suporta o **registo cronológico** das alterações introduzidas nos ficheiros-fonte do projeto;
- Suporta a **colaboração** entre os diversos participantes de projeto, e sabe lidar com o problema da submissão de **alterações conflituantes**.

Atualmente, o CVS (ou o SVN) é usado na maioria dos projetos de *open source*. Podemos confirmar isso visitando o site Sourceforge: trata-se do principal site hospedeiro deste tipo de projetos. Poderá observar que, inclusivamente, projetos de software livre para o Windows (e.g. Dev-Pascal, Dev-C++) usam o CVS.

Uma ferramenta como o CVS é essencial para a viabilidade do software livre, pois este tipo de software é tipicamente desenvolvido coletivamente por um grande número de voluntários geograficamente espalhados e não pagos. Nenhum desses voluntários teria tempo nem paciência para processar manualmente as numerosas submissões de código novo ou corrigido.

Uma ferramenta do tipo do CVS também faz falta no desenvolvimento de software proprietário, mas no caso do software livre a questão é mais evidente.

Exemplos de software desenvolvido com a ajuda do CVS

- Apache web server
- X window system
- GNOME desktop
- Mozilla web browser
- OpenOffice
- GCC gnu compiler collection
- Debian linux distribuition
- Anjuta IDE for rapid application development

Alternativas ao CVS

Existem diversas alternativas ao CVS, o qual já começa a mostrar a sua longa idade:

- Alternativas *open source*: SVN - Subversion, GNU arch, Monotone, .
- Alternativas comerciais: BitKeeper.

Atualmente o uso do Subversion está em expansão muito rápida. Não admira: introduz melhorias significativas e foi exatamente criado para substituir o CVS. O eSvn é um excelente front-end gráfico para o Subversion (veja os screenshots).

Serviço de SNV do DI, para ser usado por alunos e docentes.

IDEs

Um IDE, ou ambiente de desenvolvimento integrado, é um programa único dentro do qual se processa todo o desenvolvimento do software. Geralmente a interface é gráfica.

Exemplos: Eclipse, Visual Studio .Net, Anjuta, Code::Blocks.

Os IDEs mais sofisticados são estendíveis através de plugins. Isso permite que o mesmo sistema possa ser usado para trabalhar em diversas linguagens.

#30

Linguagens e Ambientes de Programação (2010/2011)

Teórica 28 (01/Jun/2011)

Escolha duma linguagem de programação.
Discussão sobre as linguagens usadas na cadeira.

Escolha duma linguagem de programação

A escolha duma linguagem de programação para usar num projeto de programação depende de muitas circunstâncias. Em contextos diferentes deve dar-se maior ou menor importância a diferentes aspetos. Em algumas situações é conveniente usar mais do que uma linguagem.

- Experiência prévia com a linguagem.
- Facilidade em aprender a linguagem.
- Facilidade em usar a linguagem.
- Suporte de boas práticas de programação.
- Suporte de tecnologias de desenvolvimento de software recentes.
- Qualidade das ferramentas de desenvolvimento disponíveis.
- Disponibilidade de grande número de programadores.
- Suporte para deteção atempada de bugs.
- Padronização da linguagem.
- Expressividade e produtividade.
- Riqueza das bibliotecas disponíveis.
- Portabilidade.
- Segurança.
- Velocidade de execução.
- Custo.
- Facilidade em escrever scripts.
- Adequação a programação de sistemas.
- Suporte para concorrência.
- Suporte para introspeção.

Discussão sobre as linguagens usadas na cadeira

OCaml

- Linguagem simples, fácil de usar, suporta abstrações muito poderosas.
- A programação pode ser de muito alto-nível e os programas tendem a ser compactos.
- Tem um conjunto de bibliotecas rico e há muitas ferramentas desenvolvidas para serem usadas com o OCaml.
- Suporta programação funcional, imperativa e orientada pelos objetos.

C

- Grande liberdade para explorar os recursos da máquina -> programação de sistemas.
- Linguagem insegura mas muito eficiente.
- Nível excelente de interoperabilidade - quase todas as linguagens suportam mecanismos de ligação ao C.
- Depuração difícil devido ao uso de apontadores e à necessidade de gerir a memória explicitamente. Os programas podem ser difíceis de ler.

C++

- C com objetos + templates + bibliotecas mais ricas.
- Tem muitas das características do C mas ajuda no desenvolvimento de projetos maiores.

Java

- Segurança em aplicações orientadas para a WEB.
- Portabilidade dos executáveis.
- Biblioteca imensamente rica.
- Suporta boas práticas de programação e tecnologias de desenvolvimento de software recentes: testes unitários, objetos, componentes (Java Beans), serviços WEB, frameworks de programação (Spring).
- Mais lenta do que o C++.

JavaScript

- Linguagem muito prática de usar, dá grande liberdade ao programador, mantendo um razoável nível de segurança quando o que está em causa é a escrita de pequenos scripts.
- Serve para programar páginas WEB interativas, mas também é usada como linguagens de scripting em muitas outras aplicações.
