Introdução à linguagem C

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

Introdução à linguagem C

Introdução:

- Alguma história. Aspectos gerais da linguagem.
- "Hello world!" - um primeiro programa em C, uso do GCC.
- Variáveis, tipos primitivos e expressões em C.
- Fluxo de controlo: chamadas a funções, escolha, ciclos.
- Apontadores: conceito base, passagem de valores a funções por referência.

Alguma história. Aspectos gerais da linguagem.

História

- **1972/73**: 1ª versão por Dennis Ritchie. Uso para programação de de variantes de sistema operativos Unix.
- **1978**: 1ª edição do livro "The C programming language", Brian Kernighan e Dennis Ritchie.
- **Anos 80**: Surgem C++ e Objective-C.
- **1989**: ANSI C standard.
- **1999/2011/2017**: standards ISO C99, C11, C17.

Características gerais e uso

Características gerais:

- Programação imperativa (também chamada de procedimental): programas definidos por funções contendo instruções imperativas.
- Linguagem estaticamente tipada: variáveis, funções, expressões, ..., têm tipos associados, validados em tempo de compilação.

Usado para programação de vários tipos sistemas, incluindo:

- sistemas operativos (ex. Linux)
- motores de bases de dados (ex. MySQL, SQLite)
- sistemas embutidos (ex. Arduino, Microbit)
- computação paralela (ex. as bibliotecas MPI, OpenMP)
- GPUs (ex. biblioteca CUDA), ...
- ...

Vantagens

- Linguagem pequena e sem necessidade estrita de biblioteca de sistema tornando comparativamente fácil a implementação de um compilador e o uso de C em vários contextos.
- Permite trabalhar a "baixo nível": aceder a hardware e memória directamente, embeber código "assembly", ...
- Há compiladores de C para todo o tipo de sistemas e arquitecturas. Compiladores maduros (GCC, Clang, ...) conseguem gerar código máquina altamente optimisado.

Desvantagens

- Acessos inválidos a memória não são validados nem têm comportamento previsível, levando a programas que podem ser pouco fiáveis, e em particular a uma grande quantidade de bugs de segurança ao longo da história.
- Memória dinamicamente alocada tem de ser gerida pelo programador. Não há "garbage collection".
- Vários comportamentos indefinidos na semântica da linguagem são deixados ao critério do compilador, podendo comprometer a fiabilidade e portabilidade de programas.

A "família C": C++ e Objective-C

- C++ e Objective-C são outras linguagens da "família do C", inicialmente desenvolvidas na década de 80, e com suporte para programação orientada-a-objectos.
- C++ é uma linguagem de uso relativamente geral, Objective-C esteve sempre mais associado ao desenvolvimento de programas no ecosistema da Apple (ex. macOS e iOS).
- Código C é inter-operável / pode ser integrado em C++ e Objective-C tipicamente com poucas ou nenhumas mudanças. As construções imperativas funcionam da mesma forma e com as mesmas vantagens e desvantagens.

A "família C": outros parentes

- Outras linguagens de amplo uso como C# e Java também se dizem da "família C", mas mais por afinidade na sintaxe de construções imperativas.
- Por exemplo, tanto C# e Java têm um ambiente de execução baseado numa máquina virtual e empregam "garbage collection" para gerir memória automaticamente. As semânticas de C# e Java não têm comportamentos indefinidos, não permitem acesso directo à memória, etc.

“Hello world!”

Um primeiro programa em C

Podemos editar um ficheiro `hello.c` com o seguinte conteúdo:

```
/*
  A simple program that prints "Hello world!"
*/
#include <stdio.h>

int main(void) {
  printf("Hello world!\n"); // Print the message
  return 0;
}
```

Compilação do programa

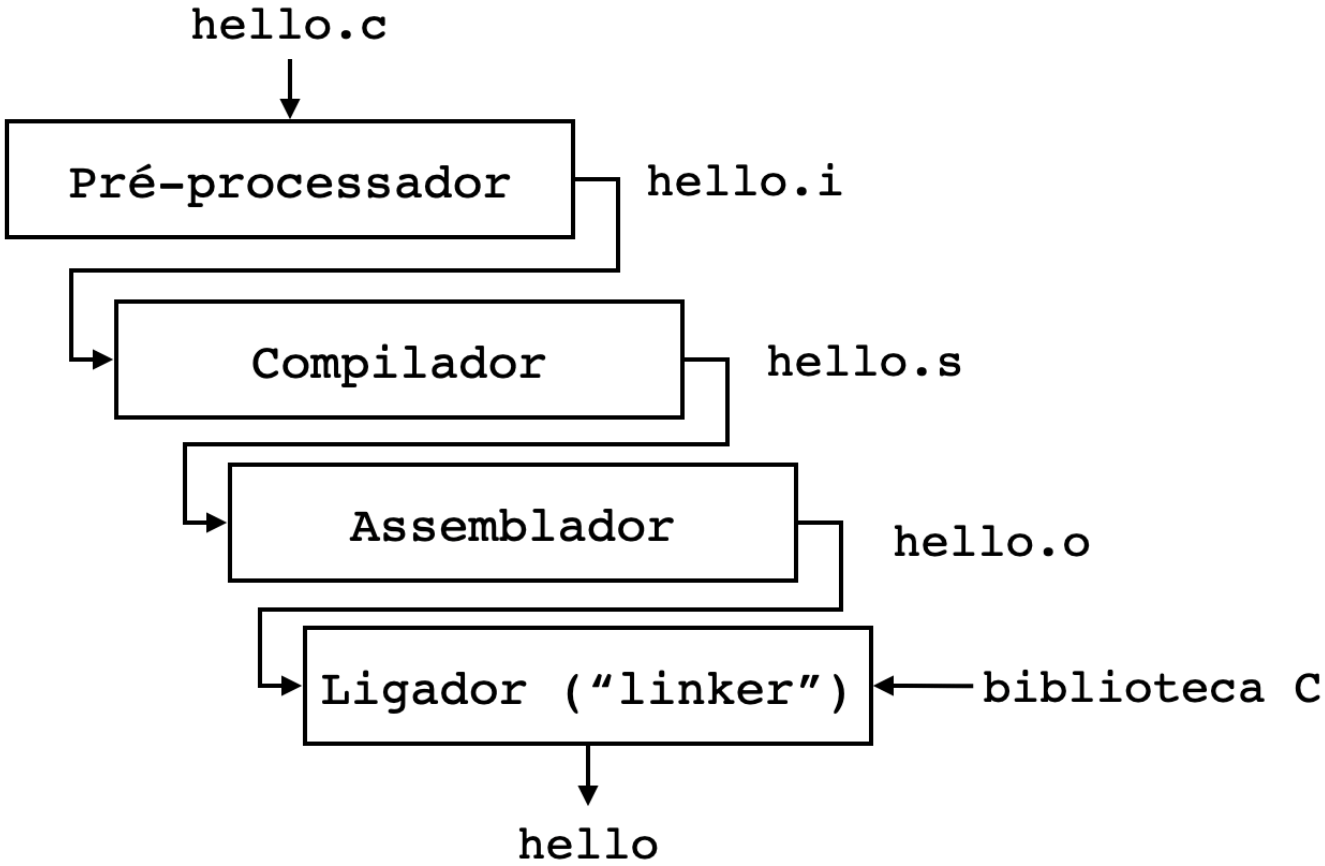
De seguida podemos compilar `hello.c` para gerar um ficheiro binário executável `hello`. Por exemplo, usando o GCC:

```
$ gcc hello.c -Wall -o hello
```

O ficheiro `hello` pode depois ser executado:

```
$ ./hello
Hello world!
```

Estágios da compilação de código C



Pode ver os ficheiros intermédios usando a opção `-save-temps` com o GCC.

```
$ gcc -save-temps -Wall hello.c -o hello
```

Estágios da compilação de código C (cont.)

Pré-processamento

Processa directivas de pré-processamento, emite código C sem estas.

Compilação

Processa código C pré-processado para gerar código "assembly" .

Assemblagem

Gera código máquina (binário) a partir de código "assembly".

Ligação

Liga código máquina com bibliotecas necessárias e produz executável.

“Hello world!” dissecado

Alguns dos aspectos essenciais da sintaxe C são ilustrados por este programa simples:

- **Comentários**, iniciados por `/*` e terminados por `*/` ou iniciados por `//` e com término até ao fim da linha.
- **O uso de caracteres “brancos”** como a mudanças de linha ou espaços, relevantes para a indentação de programas facilmente legíveis.
- **Directivas de pré-processamento**, iniciadas por `#` como a directiva `#include`.
- **Definição de funções**, no caso em questão de `main`, que é o ponto de entrada de um programa C.
- **Uso de caracteres “separadores” ou “marca”** como `;`, `(`, `)`, `{`, ou `}`.

“Hello world!” dissecado (cont.)

```c
/*
  A simple program that prints "Hello world!"
*/
#include <stdio.h>

int main(void) {
  printf("Hello world!\n"); // Print the message
  return 0;
}
```

Os aspectos anteriores são postos em evidência ao usarmos um editor ou IDE (“Integrated Development Environment”) por via de **“syntax highlighting”**, i.e., o uso de cores e/ou estilos de formatação diferentes em associação aos diferentes elementos sintácticos que tipicamente facilitam a leitura e edição do código.

Comentários

Comentários em C podem ser iniciados por `/*` e terminados por `*/`

```
/*
  A simple program that prints "Hello world!"
*/
```

ou (desde o C99) iniciados por `//` e com término no final da linha.

```
  printf("Hello world!\n"); // Print the message
```

Comentários (cont.)

Erros comuns:

```
/*
  Comment section
printf("Hello world!\n");
```

(comentário não é terminado)

```
/*
  Comment section
*/ */
printf("Hello world!\n");
```

(fim de comentário não tem início correspondente)

```
printf("Hello world!\n"); // Comment
                                section
```

(comentário estende-se por mais de uma linha)

Indentação

```c
                                    /*  A simple program
that prints "Hello world!"
*/#include                                  <stdio.h>
int main(                                                  void
){printf                        ("Hello world!\n"
); // Print the message
  return                                                  0;}
```

A indentação não altera o significado do programa (ao contrário de programas em Python). Devemos no entanto indentar o programa convenientemente para boa legibilidade. A variante de `hello.c` acima é pouco clara :(

Caracteres brancos (mudança de linhas, espaços, ...) são ignorados pelo compilador, excepto a mudança de linha em alguns casos, ex. comentários de linha simples (`// ...`), a definição de sequências de caracteres (ex. `"Hello world!\n"`) ou o uso de directivas de pré-processamento (ex. `#include`).

Directivas de pré-processamento

Directivas de pré-processamento são iniciadas por `#` como em

```
#include <stdio.h>
```

A directiva `#include` tem como efeito expandir no corpo do programa as definições de outro ficheiro C, neste caso `stdio.h`.

Ficheiros com extensão `.h`, chamados de "header files" ou simplesmente de "headers", tipicamente contêm apenas declarações que são necessárias a um programa, mas não a definição de código em si. Veremos depois exemplos.

No caso em questão `stdio.h` contém as definições da biblioteca standard de C ("C standard library") para entrada e saída de dados. O programa usa a função `printf` que é declarada neste "header".

Directivas de pré-processamento (cont.)

Iremos depois ver exemplos de uso de outras directivas de pré-processamento.

```
#define
#undef
#if
#ifdef
#ifndef
#error
#pragma
```

Definição de funções

Programas C são definidos por sequências de instruções, agrupadas em **funções**.

```
int main(void) {
   ...
}
```

A **assinatura ou declaração da função** indica-nos o tipo de retorno, e os argumentos e respectivos tipos. No caso `int main(void)` indica-nos que a função deve retornar um valor de tipo `int` e que não tem argumentos - `void` é o "tipo vazio". Como veremos depois uma função tem no caso geral uma assinatura com a forma:

```
return_type function_name(type_1 arg_1,
                          type_2 arg_2 ...,
                          type_n arg_n)
```

Definição de funções (cont.)

```c
int main(void) {
  printf("Hello world!\n"); // Print the message
  return 0;
}
```

O **corpo da função** entre `{` e `}` contém a sequência de instruções a executar por `main`. No caso temos apenas duas instruções simples:

- uma chamada à função `printf` da biblioteca de C (declarada em `stdio.h`)
- a instrução `return 0` termina a execução de `main`, retornando o valor `0`.
- instruções simples são terminadas por `;` (ponto-e-vírgula), sequências de instruções (como o corpo de `main`) são agrupadas com `{` e `}`.

Significado de `main`

```
int main(void) {
  ...
}
```

A execução de um programa começa sempre pela invocação da função `main`.

O valor de retorno representa o código de erro para o sistema operativo. A convenção é que um retorno do valor `0` indica a ausência de qualquer erro.

Veremos depois que **main** pode ser alternativamente declarada por forma a receber argumentos de invocação do programa (o uso de `void` leva a que estes sejam ignorados):

```
int main(int argc, char** argv)
```

Palavras-chave ("keywords")

Em `hello.c`, empregamos `int`, `void` e `return` que são **palavras-chave** da linguagem (**"keywords"**), também chamadas de palavras reservadas ("reserved words"), e que têm um significado especial.

Não podemos usar palavras-chave como identificadores, por exemplo dar o nome de `int` a uma função.

Em C99 temos as seguintes palavras-chave:

```
auto      enum      restrict  unsigned
break     extern    return    void
case      float     short     volatile
char      for       signed    while
const     goto      sizeof    _Bool
continue  if        static    _Complex
default   inline    struct    _Imaginary
do        int       switch
double    long      typedef
else      register  union
```

Variáveis, tipos primitivos, e expressões

Variáveis - tipo

```c
#include <stdio.h>
const int g = 100;

int main(void) {
  int a = 10;
  int b;
  b = 20;

  int c = a + b + g;
  printf("%d %d %d %d\n", a, b, c, g);
  return 0;
}
```

Uma **variável** em C tem sempre associado um **tipo** na sua declaração.

O tipo de todas as variáveis acima é `int`. Podemos associar à declaração de um valor inicial, ex. `a`, `c` e `g` acima).

Variáveis - âmbito

```c
const int g = 100; // global scope

int main(void) {
  int a = 10; // local scope to main
  ...
}
```

Uma **variável** em C tem um **âmbito** de utilização ("scope").

Pode ser **local** a um bloco de instruções (entre `{` e `}`), como o corpo de uma função `main`, ou **global** como no caso de `g` acima.

Variáveis globais são caso geral desaconselhadas porque tendem a criar uma lógica "paralela" e "obscurecer" o fluxo de um programa.

A declaração de `g` acima é mais "pacífica" / "aceitável". Declaramos uma variável com valor constante: o valor de variáveis declaradas com o modificador `const` não pode ser re-definido após inicialisação.

Variáveis - regras básicas de declaração e uso

- Uma variável tem de ser sempre declarada e apenas uma vez.
- Uma variável só pode ser usada no seu âmbito e após a sua declaração.
- Não podemos declarar duas variáveis com o mesmo identificador dentro do mesmo âmbito.
- Valores atribuídos a uma variável têm de ter tipos compatíveis com o declarado.

Tipos primitivos inteiros

Tipo	Bytes (x86_64)	Valor mínimo	Valor máximo
`char`	1	-2^7 (-128)	$2^7 - 1$ (127)
`short`	2	-2^{15}	$2^{15} - 1$
`int`	4	-2^{31}	$2^{31} - 1$
`long`	8	-2^{63}	$2^{63} - 1$
`unsigned char`	1	0	$2^8 - 1$ (255)
`unsigned short`	2	0	$2^{16} - 1$
`unsigned int`	4	0	$2^{32} - 1$
`unsigned long`	8	0	$2^{64} - 1$

Além de `int` existem os tipos inteiros `char`, `short` e `long`, bem como as suas variantes `unsigned` para inteiros positivos.

O tamanho em bytes (e correspondente âmbito de valores) é **dependente** da arquitectura / compilador. Acima são listados os valores usados para arquitecturas de 64 bits (ex. Intel x86_64).

Constantes inteiras

Representação	Exemplos
Decimal	`10` `65` `-1` ...
Caracter ASCII	`'\n'` (10) `'A'` (65) ...
Octal	`012` (10) `0101` (65)
Hexadecimal	`0x0A` (10) `0x41` (65)

Tipos de vírgula flutuante

`float` e `double` são tipos de vírgula flutuante ("floating point"), normalmente representados de acordo com standard IEEE 754.

- `float`: precisão simples, 32 bits em Intel x86_64, precisão de 10^{-38} a 10^{38};
- `double`: precisão dupla, 64 bits em Intel x86_6, precisão de 10^{-308} a 10^{308};

Constantes

Representação	Exemplos
Decimal	0.01 -1.23 1230.0
Notação científica	1e-2 -123e-02 123e+1

Expressões e operadores

Uma **expressão** em C pode ser constituída por constantes, variáveis, chamadas a função, e sua combinação via **operadores**.

Exemplo:

```
y = (1.0 + a) * b * c / f(1e-02, 2, x - 2);
```

Atribuição

```
a = b;
```

A expressão `a` é chamado o "l-value", designando a localização de memória a ser escrita, e a expressão `b` é chamado o "r-value" designando o valor a ser escrito.

Podemos ter várias atribuições encadeadas, embora tal não seja muito comum, ex.:

```
i = j = k = 123;
```

Operadores aritméticos

Expressão	Significado
`a + b`	Soma
`a - b`	Subtração
`a * b`	Multiplicação
`a / b`	Quociente (valores inteiros) ou divisão (vírgula flutuante)
`a % b`	Módulo (resto de divisão).

Além disso `-` e `+` podem ser usados como operadores unários, ex.

```
+a
-a
- (a + 1)
```

Operadores relacionais

Expressão	Avalia para 1 se ...
`a == b`	`a` tiver valor igual a `b`
`a != b`	`a` tiver valor diferente de `b`
`a < b`	`a` tiver valor menor que `b`
`a <= b`	`a` tiver valor menor ou igual que `b`
`a > b`	`a` tiver valor maior que `b`
`a >= b`	`a` tiver valor maior ou igual que `b`

Operadores lógicos

Expressão	Avalia para 1 se ...
`a && b`	`a` e `b` forem diferentes de 0
`a || b`	`a` ou `b` forem diferentes de 0
`!a`	`a` for 0

Ao contrário de outros operadores binários, são garantidas para os operadores `&&` e `||` uma ordem estrita de avaliação da esquerda para a direita e também a avaliação condicional do segundo argumento:

- `a && b` avalia `a` primeiro e apenas `b` só se `a != 0`.
- `a || b` avalia `a` primeiro e apenas `b` só se `a == 0`.

Operador ternário `?:`

Uma expressão `a ? b : c` devolve o resultado de `b` ou `c` consoante `a != 0` ou não.

```
x = y > 100 ? 1 : 2;
```

O valor atribuido a `x` será de `1` se `y > 100` e `2` caso contrário.

Operadores de manipulação de bits

Expressão	Resultado
`a & b`	Conjunção de bits (AND)
`a | b`	Disjunção de bits (OR)
`a ^ b`	Disjunção exclusivo de bits (XOR).
`~a`	Inversão de bits (NOT)
`a << b`	Deslocamento de bits para a esquerda.
`a >> b`	Deslocamento de bits para a direita.

Atribuição composta

Expressão	Equivalente a ...
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`
`a %= b`	`a = a % b`
`a &= b`	`a = a & b`
`a |= b`	`a = a | b`
`a ^= b`	`a = a ^ b`
`a <<= b`	`a = a << b`
`a >>= b`	`a = a >> b`

Pré/pós incremento e decremento

```
++a     a++     --b     b--
```

Os operadores são frequentemente úteis para expressar o incremento e decremento de variáveis de forma concisa. Por exemplo:

```
a++;
--b;
```

é equivalente a

```
a += 1;
b -= 1;
```

Pré/pós incremento e decremento (cont.)

O seu uso em conjunto com outros operadores será de evitar, pois pode tornar um programa confuso ou até levar a comportamento indefinido. Será de evitar o uso destes operadores nesses casos.

O fragmento

```
int a = 1;
int b = ++a + 1;
```

resulta no incremento de `a` **antes** da atribuição a `b`, portanto `b` fica o com valor `3`. Ao invés, o uso de pós-incremento em

```
int a = 1;
int b = a++ + 1;
```

resulta no incremento de `a` **depois** da atribuição a `b`, portanto `b` fica com valor `2`.

Pré/pós incremento e decremento (cont.)

Exemplo de comportamento indefinido:

```
int a = 1;
int b = a + ++a;
```

Como C não garante uma ordem estrita de avaliação da esquerda para a direita excepto quando estão em causa os operadores lógicos `&&` e `||`, o fragmento acima pode resultar na atribuição do valor `3` a `b` (se `a`, o operando à esquerda da soma, é avaliado primeiro) como do valor `4` (se ao invés `++a` é avaliado primeiro).

Operadores - precedência e associatividade

Precedência	Operadores	Associatividade
...		
3	`* / %`	Esquerda
4	`+ -`	Esquerda
5	`<< >>`	Esquerda
6	`< > <= >=`	Esquerda
7	`== !=`	Esquerda
...		
14	`=`	Direita

Acima identificamos a **precedência** e a **associatividade** de alguns dos operadores que apresentamos até agora, usadas para determinar o significado de expressões com vários operadores. Referência completa: por exemplo aqui.

Operadores - precedência e associatividade (cont.)

Por exemplo `*` tem precedência sobre `+` e `-` portanto

```
a * b + c * d - e
```

é implicitamente equivalente a

```
(a * b) + (c * d) - e
```

e portanto tem um significado diferente de

```
a  * (b + c) * (d - e)
```

Operadores - precedência e associatividade (cont.)

A associatividade de um operador, à esquerda ou à direita, determinam a interpretação expressões no contexto de operadores com igual precedência.

Por exemplo, `*` e `/` associam à esquerda, portanto

```
a * b / c
```

é equivalente a

```
(a * b) / c
```

Ao invés, o operador de atribuição `=` associa à direita portanto

```
a = b = 10;
```

é equivalente a

```
a = (b = 10);
```

Chamada de funções

Chamada de funções

Esqueleto de programa para imprimir o máximo de dois valores `int` passados como dados de entrada:

```
#include <stdio.h>
int max(int a, int b) {
  ...
}
int main(void) {
  int a, b;
  scanf("%d %d", &a, &b);
  int r = max(a, b);
  printf("%d\n", max);
  return 0;
}
```

Chamada de funções (cont.)

```
scanf("%d %d", &a, &b); // 1
int r = max(a, b);      // 2
printf("%d\n", r);      // 3
return 0;               // 4
```

Fluxo de execução:

1. `main` chama `scanf` (da biblioteca de C) para ler `a` e `b`. **Apontadores** para `a` e `b` são passados como argumentos (`&a` e `&b` - veremos depois o que significa em detalhe) para que `scanf` possa escrever a memória de `a` e `b` em `main`.

2. Depois de `scanf` retornar, `main` chama `max` passando os **valores** de `a` e `b`.

3. O resulto obtido de `max` é atribuído a `r` e impresso com mais uma chamada, desta vez a `printf`.

4. `main` termina a sua execução retornando o valor 0.

Chamada de funções (cont.)

```
#include <stdio.h>
int max(int a, int b) {
   ...
}
int main(void) {
  int a, b;
   ...
  int max = max(a, b);
    ...
}
```

Coincidentemente, tanto `main` como `max` definem `a` e `b`, mas o âmbito das declarações é independente, por ex. qualquer escrita em `a` ou `b` em `max` (caso haja) não é reflectida nas variáveis de `main` com o mesmo nome.

Escolha

Instruções `if` e `if-else`

```
if (cond)
  corpo
```

Avalia se `cond` é verdadeira (valor diferente de 0) e (só) nesse caso executa `corpo`.

```
if (cond)
  corpo1
else
  corpo2
```

Avalia se `cond` é verdadeira (valor diferente de 0) e (só) nesse caso executa `corpo1`. Ao invés, se `cond` é falsa (valor 0) então executa `corpo2`.

Escolha usando `if` e `if-else`

Para a implementação de `max` podemos usar uma instrução `if`:

```
int max(int a, int b) {
  int r = a;
  if (a < b)
    r = b;
  return r;
}
```

Em alternativa, podemos usar uma instrução `if-else`

```
int max(int a, int b) {
  int r ;
  if (a < b)
    r = b;
  else
    r = a;
  return r;
}
```

Escolha com operador ternário `? :`

Podemos ainda empregar o operador ternário `? :`

```
int max(int a, int b) {
  int r =  a < b ? b : a;
  return r;
}
```

A variável `r` é redundante (só é usada uma vez depois de definida), podemos na verdade ter apenas:

```
int max(int a, int b) {
  return a < b ? b : a;
}
```

Escolha - 2º exemplo

```
#include <stdbool.h>

bool is_leap_year(int y) {
   ...
}
```

Um ano é bissexto ("leap year"), i.e., tem 366 dias em vez de 365 se o seu valor for divisível por 4 excepto quando é também divisível também por 100 mas não por 400. Por exemplo:

- 2020 é um ano bissexto, 2021 não.
- 2100 não é um ano bissexto (2100 divisível por 100 mas não por 400).
- 2000 é um ano bissexto (divisível por 100 e por 400).

Nota: o "header" `stdbool.h` introduz definições para `bool` (tipo inteiro), e constantes inteiras `true` (1) e `false` (0). Poderiamos usar `int` em alternativa e sem problema, o uso de `bool` é uma forma de "açúcar sintáctico".

Escolha - 2º exemplo (cont.)

Podemos recorrer a instruções `if-else` para implementar `is_leap_year`:

```
bool is_leap_year(int y) {
  bool r;
  if (y % 4 != 0) {
    r = false;
  } else if (y % 100 == 0 && y % 400 != 0) {
    r = false;
  } else {
    r = true;
  }
  return r;
}
```

Escolha - 2º exemplo (cont.)

Variante com instruções `if` imbricadas:

```
bool is_leap_year(int y) {
  bool r = false;
  if (y % 4 == 0) {
    if ( y % 100 != 0 || y % 400 == 0 ) {
      r = true;
    }
  }
  return r;
}
```

Escolha - 2º exemplo (cont.)

O anterior exemplo é equivalente a:

```
bool is_leap_year(int y) {
  bool r = false;
  if (y % 4 == 0 && (y % 100 != 0 || y % 400 == 0 )) {
      r = true;
  }
  return r;
}
```

... mas ao fim cabo `r = y % 4 == 0 && (y % 100 != 0 || y % 400 == 0 )` pelo que podemos reduzir o código a:

```
bool is_leap_year(int y) {
  return y % 4 == 0 &&  (y % 100 != 0 || y % 400 == 0);
}
```

Escolha - 3º exemplo

```
int is_leap_year(int y) { ... }
int days_in_month(int m, int y) { ... }
```

Consideremos o problema de calcular o número de dias de um mês em determinado ano, ex. Outubro de 2021 tem 30 dias.

O mês de Fevereiro é um caso especial, o único que depende do ano em questão tem 29 dias em anos bissextos e 28 dias nos restante.

Todos os outros meses têm sempre 30 (ex. Novembro) ou 31 (ex. Dezembro), independentemente do ano.

Escolha - 3º exemplo (cont.)

Possível implementação:

```
int days_in_month(int m, int y) {
  int r;
  if (m == 2) {
    r = is_leap_year(y) ? 29 : 28; // February
  } else if (m == 1 || m == 3 ||
             m == 5 || m == 7 ||
             m == 8 || m == 10 ||
             m == 12) {
    r = 31; // Months with 31 days
  } else {
    r = 30; // All others have 30 days
  }
  return r;
}
```

Instrução `switch-case`

```
switch (expr) {
  case constante1:
     corpo1
     break;
  case constante2:
     corpo2
     break;
  ...
  default:
     corpo_default
     break;
}
```

Avalia valor de `expr`, uma expressão de tipo inteiro, e executa: `corpo1` se `expr` tem valor `constante1`, `corpo2` se `expr` tem valor `constante2`, ..., ou `corpo_default` (se definido) se `expr` tem outro valor.

Escolha - 3º exemplo (cont.)

Implementação alternativa com `switch-case`:

```
int days_in_month(int m, int y) {
  int r;
  switch (m) {
    case 2:
      r = is_leap_year(y) ? 29 : 28; // February
      break;
    case 1: case 3: case 5:
    case 7: case 8: case 10:
    case 12:
      r = 31; // Months with 31 days
      break;
    default:
      r = 30; // All others have 30 days
      break;
  }
  return r;
}
```

Escolha - 3º exemplo (cont.)

Para melhor legibilidade, podemos empregar constantes com identificadores associados via directiva `#define` do pré-processador.

```
#define JANUARY  1
#define FEBRUARY 2
#define MARCH    3
...
#define DECEMBER 12
```

... ou via enumeração de valores inteiros para o mesmo efeito

```
enum {
  JANUARY = 1,
  FEBRUARY, /* implicitly 2 */ MARCH, /* 3 */
  /* etc */
  APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER,
  OCTOBER, NOVEMBER, DECEMBER
};
```

Escolha - 3º exemplo (cont.)

Empregando as contantes podemos ter:

```
int days_in_month(int m, int y) {
  int r;
  switch (m) {
    case FEBRUARY:
      r = is_leap_year(y) ? 29 : 28; // February
      break;
    case JANUARY: case MARCH: case MAY:
    case JULY: case AUGUST: case OCTOBER:
    case DECEMBER:
      r = 31; // Months with 31 days
      break;
    default:
      r = 30; // All others have 30 days
      break;
  }
  return r;
}
```

Iteração (ciclos)

Instruções de iteração `while` e `do-while`

```
while(cond)
  corpo
```

Enquanto condição `cond` for verdadeira (`!= 0`) executa `corpo` do ciclo.

```
do {
  corpo
} while (cond);
```

Executa (pelo menos uma vez) `corpo` até que `cond` seja falsa.

Instrução de iteração `for`

```
for (inicialização; cond; actualização) {
  corpo
}
```

equivalente a

```
inicialização
while (cond) {
  corpo
  actualização
}
```

Exemplo - iteração com `while`

Função para determinar se um número é primo (algoritmo naive):

```
bool is_prime(int n) {
  if (n <= 1)
    return false;
  int i = 2;
  while (i < n) {
    if (n % i == 0)
      return false;
    i++;
  }
  return true;
}
```

Exemplo - iteração com `for`

Implementação equivalente ao exemplo anterior empregando um ciclo `for`:

```
bool is_prime(int n) {
  if (n <= 1)
    return false;
  for (int i = 2; i < n; i++) {
    if (n % i == 0)
      return false;
  }
  return true;
}
```

Exemplo - iteração com `do-while`

```
bool is_prime(int n) {
  if (n <= 1)
    return false;
  if (n == 2)
    return true;
  int i = 2;
  do {
    if (n % i == 0)
      return false;
    i++;
  } while (i < n);
  return true;
}
```

Uso de `break` em ciclos

No corpo de um ciclo a instrução `break` causa o término da iteração ("salta fora"). Deve ser usado raramente … o fluxo de execução pode ficar mais difícil de perceber.

```
bool is_prime(int n) {
  if (n <= 1)
    return false;
  int i = 2;
  while (i < n) {
    if (n % i == 0)
      break; // step out
    i++;
  }
  return i == n; // prime only if i == n !
}
```

Uso de `continue` em ciclos

No corpo de um ciclo a instrução `continue` leva a que o fluxo de execução salte imediatamente para a avaliação da condição de teste em `while` ou `do-while` ou para o código de actualização e teste em ciclos `for`.

Tal como `break`, o uso de `continue` pode tornar o fluxo de execução mais difícil de perceber.

```
bool is_prime(int n) {
  if (n <= 1)
    return false;
  for (int i = 2; i < n; i++) {
    if (n % i != 0)
      continue; // continue iterating
    return false;
  }
  return true;
}
```

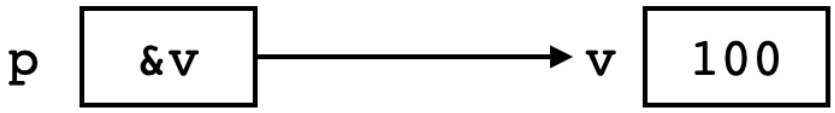
`break` e `continue` - usos mais adequados

Em ciclos com várias condições de saída ou de "re-ingresso" na iteração, o o uso de **`break`** e **`continue`** pode levar a código com menos blocos imbricados e mais legível.

```
while(1) {
  // Múltiplas condições de paragem
  if (stop_ cond1) break;
  ...
  if (stop_cond_n) break;
  // ... e/ou de "reingresso" na iteração
  if (cont_cond_1) continue;
  ...
  if (cont_cond_k) continue;
  ...
}
```

Apontadores - introdução

O que são apontadores?



Considere o seguinte fragmento de código:

```
int v = 100;
int *p = &v;
*p = *p + 1;
printf("%d %d %p %p\n", v, *p, p, &v);
```

Usamos uma variável `p` de tipo `int*`, que se diz um **apontador** para um valor de tipo **int**. `p` é inicializado com o endereço em memória de de v: `&v`.

Sendo `p` um apontador, `*p` permite-nos escrever ou ler um valor da posição referenciado por `p`, ou seja `v`. No final do fragmento acima, `v` ficará com o valor igual a 101.

Passagem de valores por “referência”

Já usamos `&` em associação a chamadas a `scanf`, isto para permitir que a função use os endereços dados para a escrita de valores “de fora”.

C funciona com passagem de argumentos por valor, os apontadores permitem-nos ter um esquema de passagem argumentos por “referência”, por exemplo:

- para permitir que funções tenham múltiplos dados de saída, além do valor retornado;
- para que os dados sejam passados como argumentos ou retornados de forma mais eficiente, a “referência” via apontador evita a cópia de dados para tipos com dimensão maior.

Passagem de valores por "referência" (cont.)

Exemplo - definição e uso de função que devolve o mínimo, máximo, e soma de 2 valores:

```
void min_max_sum
(int a, int b, int* min, int* max, int* sum) {
  *min = a < b ? a : b;
  *max = a > b ? a : b;
  *sum = a + b;
}

void some_func(void) {
  int maior, menor, s;
  min_max_sum(-123, 123, &menor, &maior, &s);
  ...
}
```