# C

Guilherme P. Telles

IC

14 de março de 2023

# Estruturas de controle

# Valores lógicos

- Em C, $0$ é falso e $\neq 0$ é verdadeiro.

# if

```
if (expressão)
  uma única sentença ou um bloco
```

```
if (expressão)
  uma única sentença ou um bloco
else
  uma única sentença ou um bloco
```

```c
if (i == 1)
  x++;
```

```c
if (i == 3) {
  x++;
  y--;
}
```

```c
if (i == 5) {
  x++;
}
else
  y--;
```

exm-if.c

```c
int main() {

  int var = 1;

  if (var == 1)
    puts("if var == 1\n");

  if (var)
    puts("if var\n");

  if (1)
    puts("if 1\n");

  if (-1)
    puts("if -1\n");

  return 0;
}
```

- Quando há mais de um `if`, o `else` se liga ao `if` mais próximo.

- Quando há mais de um `if`, o `else` se liga ao `if` mais próximo.

```
if (i == 1)
  if (j == 2)
    x++;
  else
    y++;
```

- Quando há mais de um `if`, o `else` se liga ao `if` mais próximo.

```
if (i == 1)
  if (j == 2)
    x++;
  else
    y++;
```

```
if (i == 1) if (j == 2) x++; else y++;
```

# while

```
while (expressão)
  uma única sentença ou um bloco
```

```c
int var = 0;
while (var <= 10) {
  printf("while %d\n", var);
  var++;
}
```

```c
int var = 0;
while (var <= 10) {
  printf("while %d\n", var);
  var++;
}
```

```c
while (1) {
  printf("infinito\n");
}
```

# for

```
for (expressão1; expressão2; expressão3)
  uma única sentença ou um bloco
```

# for

```
for (expressão1; expressão2; expressão3)
  uma única sentença ou um bloco
```

- É equivalente a

```
expressão1;
while (expressão2) {
  sentenças
  expressão3;
}
```

- Qualquer expressão pode ser omitida. Se `expressão2` for omitida, seu valor é considerado igual a 1.

```c
for (var = 0; var <= 10; var++)
  printf("for %d\n", var);
```

```c
for (var = 0; var <= 10; var++)
  printf("for %d\n", var);
```

```c
for (i=0; i<n; i++)
  printf("for %d\n", i);

for (; i<2*n; i++)
  printf("for %d\n", i);
```

```c
for (var = 0; var <= 10; var++)
  printf("for %d\n", var);
```

```c
for (i=0; i<n; i++)
  printf("for %d\n", i);

for (; i<2*n; i++)
  printf("for %d\n", i);
```

```c
for (int j=0; i<m; j++)
  printf("for %d\n", j);
```

```c
for (var = 0; var <= 10; var++)
  printf("for %d\n", var);
```

```c
for (i=0; i<n; i++)
  printf("for %d\n", i);

for (; i<2*n; i++)
  printf("for %d\n", i);
```

```c
for (int j=0; i<m; j++)
  printf("for %d\n", j);
```

```c
for (i=0; ; i++)
  printf("infinito\n");
```

# for

- O controle do laço pode ser feito em função de duas variáveis, usando o operador `,`.

```
for (i=0, j=0; i+n>=j; i++, j+=2)
```

# do

```
do {
  sentenças
} while (expressão);
```

```c
var = 0;
do {
  printf("do %d\n", var);
  var++;
} while (var <= 10);
```

exm-loops.c

```c
int main() {

  int var = 0;
  while (var <= 10) {
    printf("while %d\n", var);
    var++;
  }

  for (var = 0; var <= 10; var++)
    printf("for %d\n", var);

  var = 0;
  do {
    printf("do %d\n", var);
    var++;
  } while (var <= 10);

  return 0;
}
```

# switch

```
switch (expressão-integral) {
  case constante-integral: sentenças
                           break;
  ...
  case constante-integral: sentenças
                           break;
  default: sentenças
}
```

# switch

```
switch (expressão-integral) {
  case constante-integral: sentenças
                           break;
  ...
  case constante-integral: sentenças
                           break;
  default: sentenças
}
```

- As sentenças do primeiro case que for igual à expressão-integral são executadas até um break ou até o fim do switch.

```c
int main() {
  int i;
  scanf("%d", &i);

  switch (i) {
  case 1:
  case 2:
    printf("Igual a 1 ou 2\n");

  case 3:
  case 4:
    printf("Menor que 5\n");
    break;

  case 5:
    printf("Igual a 5\n");
    break;

  default:
    printf("Menor que zero ou maior que 5\n");
  }

  return 0;
}
```

- Deve haver pelo menos um `case`.
- O `break` pode ser omitido em qualquer `case`.
- O default é opcional e é tomado (i) se nenhum case for tomado ou (ii) se o fluxo continuar de um case anterior sem break.

# Condicional ternário

```
expressão1 ? expressão2 : expressão3
```

- O valor do condicional é igual ao valor da `expressão2` se a `expressão1` for diferente de 0. Caso contrário é igual ao valor da `expressão3`.

exm-cond.c

```c
#include <stdio.h>

int main() {

  int i;
  scanf("%d", &i);

  int is_even = i%2 == 1 ? 0 : 1 ;

  printf("%d\n", is_even);

  return 0;
}
```

```
question = ( to ) ?  be :  !  be;
```

– Wm. Shakespeare

[fortune]

# Desvios incondicionais

- `break`, `continue`, `goto`
- Vários autores não recomendam o uso porque quebram o fluxo convencional dos outros comandos e pioram a legibilidade.
- `break` e `continue` são bastante usados para construir loops mais compactos.

# `break`

- Causa o término de uma repetição ou `switch`.
- A execução do programa continua na sentença que vem imediatamente após a repetição ou `switch`.

exm-break.c

```c
int main() {

  int i = 0;

  while (i < 5) {

    if (i == 3)
      break;

    printf("i %d\n",i);

    i = i+1;
  }

  printf("i depois do while %d\n",i);

  return 0;
}
```

# `continue`

- Causa a interrupção de uma iteração em uma repetição.
- A execução do programa continua no início da próxima iteração.
- Pode ser usado em `for`, `while` e `do`.
- No caso do `for`, a `expressão-3` é executada antes da primeira sentença da próxima iteração.

exm-continue.c

```c
int main() {

  int i = 0;

  while (i < 5) {

    if (i == 2) {
      i = i + 1;
      continue;
    }

    printf("i %d\n",i);

    i = i+1;
  }

  printf("i depois do while %d\n",i);

  return 0;
}
```

# goto

- Causa um salto incondicional para uma sentença rotulada dentro da função corrente.
- A sintaxe do rótulo é

```
identificador : sentença
```

- A sintaxe do comando é

```
goto identificador;
```

- O escopo do rótulo é o corpo da função.

```c
int i,j,k;

graph* G = 0;

FILE* f = fopen(filename,"r");
if (!f) return NULL;

char* buff = malloc(256*sizeof(char));
if (!buff) goto ENOMEMH;
size_t buffs = 256;

// Get the number of vertices from problem line:
buff[0] = 0;
while (buff[0] != 'p' || buff[1] != ' ')
  if (getline(&buff,&buffs,f) == -1) goto EILSEQH;

k = 2;
while (buff[k] != ' ')
  k++;

int st = sscanf(buff+k,"%d %d",&i,&j);
if (st != 2) goto EILSEQH;

G = gr_alloc('u',i);
if (!G) goto ENOMEMH;
```

# Operadores

# Operadores relacionais

```
> >= < <=
```

- Retornam valores `int` 0 ou 1.
- Lembre-se que em C 0 é falso e diferente de 0 é verdadeiro.

# Operadores de igualdade

```
==  !=
```

- Retornam valores `int` 0 ou 1.

# Operadores lógicos

```
&&  ||  !
```

- São **e**, **ou** e **não** lógicos.
- Expressões com valor diferente de 0 são operadas como se fossem iguais a 1. P.ex. `!7` é igual a zero.
- Expressões com os operadores `&&` e `||` deixam de ser calculadas assim que o resultado puder ser decidido, mesmo que uma parte da expressão não tenha sido executada.

- O operador e lógico (and, conjunção, $\wedge$) é definido pela tabela-verdade:

| $x$ | $y$ | $x \wedge y$ |
|---|---|---|
| V | V | V |
| V | F | F |
| F | V | F |
| F | F | F |

- O operador e lógico (and, conjunção, $\wedge$) é definido pela tabela-verdade:

| $x$ | $y$ | $x \wedge y$ |
|---|---|---|
| V | V | V |
| V | F | F |
| F | V | F |
| F | F | F |

- Em C é `&&`.

- O operador ou lógico (or, disjunção, $\vee$) é definido pela tabela-verdade:

| $x$ | $y$ | $x \vee y$ |
|---|---|---|
| V | V | V |
| V | F | V |
| F | V | V |
| F | F | F |

- O operador ou lógico (or, disjunção, $\vee$) é definido pela tabela-verdade:

| $x$ | $y$ | $x \vee y$ |
|---|---|---|
| V | V | V |
| V | F | V |
| F | V | V |
| F | F | F |

- Em C é `||`.

- O operador não lógico (not, negação, $\neg$) é definido pela tabela-verdade:

| $x$ | $\neg x$ |
|---|---|
| V | F |
| F | V |

- O operador não lógico (not, negação, $\neg$) é definido pela tabela-verdade:

| $x$ | $\neg x$ |
|---|---|
| V | F |
| F | V |

- Em C é `!`.

```
ano = 2000;

bisexto = 0;
if (ano % 400 == 0 || (ano % 4 == 0 && !(ano % 100 == 0)))
  bisexto = 1;
```

```
ano = 2000;

bisexto = 0;
if (ano % 400 == 0 || (ano % 4 == 0 && !(ano % 100 == 0)))
  bisexto = 1;
```

```
bisexto = 0;
if (ano % 400 == 0 || (ano % 4 == 0 && ano % 100 != 0))
  bisexto = 1;
```

# Operadores de incremento

```
++ --
```

- São unários e podem ser pré-postos ou pós-postos.
- Quando pré-posto, o valor da variável é incrementado antes de dar valor à expressão.
- Quando pós-posto, o valor da variável é incrementado depois de dar valor à expressão.

exm-inc.c

```c
#include <stdio.h>

int main() {

  int i;

  i = 5;
  printf("i++ %d\n", i++);

  i = 5;
  printf("++i %d\n", ++i);

  i = 5;
  printf("i-- %d\n", i--);

  i = 5;
  printf("--i %d\n", --i);

  return 0;
}
```

- É muito comum vermos os operadores de incremento nas expressões de laços:

```
int var = -1;
while (var++ < 10)
  printf("while %d\n", var);
```

- É muito comum vermos os operadores de incremento nas expressões de laços:

```c
int var = -1;
while (var++ < 10)
  printf("while %d\n", var);
```

```c
var = 0;
do {
  printf("do %d\n", var);
} while (++var <= 10);
```

# Operadores bit a bit

```
&  |  ~  ^  <<  >>
```

- Se aplicam a tipos integrais.
- Aplicam a operação sobre cada par de bits correspondentes dos operandos.
- Cuidado para não usar como operadores lógicos.

```
   1011 0101
 & 1111 0110
-------------
   1011 0100
```

```
   1011 0101
 & 1111 0110
-------------
   1011 0100
```

```
   1011 0101
 & 0000 0001
-------------
   0000 0001
```

```
  1011 0101
& 1111 0110
-----------
  1011 0100
```

```
  1011 0101
& 0000 0001
-----------
  0000 0001
```

```
  1011 0101
& 1111 1011
-----------
  1011 0001
```

```
  1011 0101
| 1111 0110
-----------
  1111 0111
```

```
   1011 0101
 | 1111 0110
-------------
   1111 0111
```

```
 ~ 1011 0110
-------------
   0100 1001
```

```
   1011 0101
 | 1111 0110
-------------
   1111 0111
```

```
 ~ 1011 0110
-------------
   0100 1001
```

```
   1011 0101
 ^ 1111 0110
-------------
   0100 0011
```

- O operador ou exclusivo (xor, disjunção exclusiva, $\oplus$) é definido pela tabela-verdade:

| $x$ | $y$ | $x \oplus y$ |
|---|---|---|
| V | V | F |
| V | F | V |
| F | V | V |
| F | F | F |

- O operador ou exclusivo (xor, disjunção exclusiva, $\oplus$) é definido pela tabela-verdade:

| $x$ | $y$ | $x \oplus y$ |
|---|---|---|
| V | V | F |
| V | F | V |
| F | V | V |
| F | F | F |

- Em C é bit-a-bit, ^.
- Xor lógico é equivalente a !=.

```
  1011 0101
       << 1
-------------
  0110 1010
```

```
  1011 0101
       << 1
-------------
  0110 1010
```

```
  1011 0101
       >> 3
-------------
  0001 0110
```

# Operadores de atribuição

```
= += -= *= /= %= &= |= ^= <<= >>=
```

- `var op= expressão` equivale a `var = var op expressão`

  p.ex.

```
i /= 2
i = i/2
```

# Operador ,

```
expr1, expr2
```

- Operador binário que permite agrupar expressões.
- O valor de `expr1` é calculado primeiro.
- O valor da expressão como um todo é o valor de `expr2`.

# Operador `sizeof`

- Retorna o número de bytes necessários para armazenar na memória:
  - um tipo ou
  - uma expressão ou
  - uma variável escalar, vetor ou registro.

exm-sizeof-1.c

```c
#include <stdio.h>

int main() {

  printf("int %ld\n",sizeof(int));
  printf("double %ld\n",sizeof(double));

  printf("5+9 %ld\n",sizeof(5+9));
  printf("3.1*2.0 %ld\n",sizeof(3.1*2.0));

  float f;
  printf("f %ld\n",sizeof(f));

  return 0;
}
```

exm-sizeof-3.c

```c
#include <stdio.h>

int main() {

  printf("char %ld\n",sizeof(char));
  printf("short int %ld\n",sizeof(short));
  printf("int %ld\n",sizeof(int));
  printf("long int %ld\n",sizeof(long));
  printf("long long int %ld\n",sizeof(long long));

  printf("float %ld\n",sizeof(float));
  printf("double %ld\n",sizeof(double));
  printf("double %ld\n",sizeof(long double));

  return 0;
}
```

# Operadores

- Operadores têm regras de precedência e associatividade em expressões.
  - A precedência determina qual a ordem de aplicação dos operadores.
  - A associatividade determina em qual ordem operadores de mesma precedência são aplicados.

| Precedence | Operator | Description | Associativity |
|---|---|---|---|
| 1 | ++ -- | Suffix/postfix increment and decrement | Left-to-right |
| | () | Function call | |
| | [] | Array subscripting | |
| | . | Structure and union member access | |
| | -> | Structure and union member access through pointer | |
| | (*type*){*list*} | Compound literal(C99) | |
| 2 | ++ -- | Prefix increment and decrement[note 1] | Right-to-left |
| | + - | Unary plus and minus | |
| | ! ~ | Logical NOT and bitwise NOT | |
| | (*type*) | Cast | |
| | * | Indirection (dereference) | |
| | & | Address-of | |
| | sizeof | Size-of | |
| | _Alignof | Alignment requirement(C11) | |
| 3 | * / % | Multiplication, division, and remainder | Left-to-right |
| 4 | + - | Addition and subtraction | |
| 5 | << >> | Bitwise left shift and right shift | |
| 6 | < <= | For relational operators < and ≤ respectively | |
| | > >= | For relational operators > and ≥ respectively | |
| 7 | == != | For relational = and ≠ respectively | |
| 8 | & | Bitwise AND | |
| 9 | ^ | Bitwise XOR (exclusive or) | |
| 10 | \| | Bitwise OR (inclusive or) | |
| 11 | && | Logical AND | |
| 12 | \|\| | Logical OR | |
| 13 | ?: | Ternary conditional | Right-to-left |
| 14 | = | Simple assignment | |
| | += -= | Assignment by sum and difference | |
| | *= /= %= | Assignment by product, quotient, and remainder | |
| | <<= >>= | Assignment by bitwise left shift and right shift | |
| | &= ^= \|= | Assignment by bitwise AND, XOR, and OR | |
| 15 | , | Comma | Left-to-right |

[cppreference.com]

- Pares de parênteses (também) são pontuação que alteram a precedência dos operadores.
- Precedência e associatividade de operadores não implicam em ordem de execução. Por exemplo, em

```
w = f1(x) + f2(z) + f3(y);
```

a associatividade entre os + especifica em que ordem a soma vai ser executada, mas não garante que `f1` será executada antes de `f2` que será executada antes de `f3`.