



DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

Elementos da Linguagem C

K&R: Capítulo 2

IAED

Elementos da Linguagem C

- Identificadores
- Tipos
- Conversão de tipos
- Constantes
- Declarações
- Operadores aritméticos, lógicos e relacionais
- Operadores de incremento e decremento
- Operações de atribuição e expressões
- Expressões condicionais
- Precedência e ordem de avaliação

Identificadores

- Sequências de letras, underscore, ou dígitos
- Primeiro caracter é letra ou underscore
- Identificadores são case-sensitive
 - `int i, I; /* Duas variáveis diferentes */`
- Frequentemente nomes nas bibliotecas começam com underscore para minimizar possíveis conflitos
- Por convenção usa-se o nome de **variaveis** em minúsculas e **CONSTANTES** em maiúsculas
- Nomes reservados: `if`, `else`, `int`, `float`, etc.

Tipos de Dados

- `char`, `int`, `float`, `double`
- **`short int`** **`int`** **`long int`**
- `signed/unsigned char`, `short`, `int`, `long`
- `unsigned` obedece aritmética módulo 2^n (n =número *bits*)
- `signed char` entre -128 e 127
- `Tamanho(char) = 1 byte (=8 bits)`
- `Tamanho-típico(int) = 4 bytes (=32 bits)`
- `Tamanho(short) <= Tamanho(int) <= Tamanho(long)`
- **`long double`**
- `Tamanho(float) <= Tamanho(double) <= Tamanho(long double)`
- Obter o tamanho de um tipo: `sizeof()`

Tipos de Dados (formatos de leitura & escrita)

- char : **%c**
- int: **%d** ou **%i** (base decimal)
- int: **%x** (base hexadecimal)
- **short** int: **%hd**
- **long** int: **%ld**
- **unsigned short** int: **%hu**
- **unsigned** int: **%u**
- **unsigned long** int: **%lu**
- float e double: **%f**

Conversão de Tipos

- Argumentos de operadores de diferentes tipos provocam transformação de tipos dos argumentos
- Algumas **conversões automáticas**: de representações "estreitas" para representações mais "largas". Exemplo: conversão de `int` para `float` em `f + i`
- `char` é um inteiro pequeno e podem-se fazer operações aritméticas com caracteres

Conversão de Tipos

- Exemplo de operações aritméticas sobre caracteres
- Objectivo:
 - Função que recebe como argumento uma string composta apenas por dígitos e devolve o inteiro correspondente

```
int atoi(char s[]) {  
    int i, n;  
  
    n = 0;  
    for (i = 0; s[i] >= '0' && s[i] <= '9'; i++)  
        n = 10 * n + (s[i] - '0');  
  
    return n;  
}
```

Conversão de Tipos

- Quando operador binário (+, *, etc) tem operandos de tipos diferentes, tipos dos operandos convertidos
- Quando não há argumentos unsigned:
 - Se algum dos operandos é `long double`, converte outro para `long double`
 - Caso contrário, se um dos operandos é `double`, converte outro para `double`
 - Caso contrário, se um dos operandos é `float`, converte outro para `float`
 - Caso contrário, converte `short` para `int` e se algum dos operandos for `long`, converte o outro para `long`

Conversão forçada de tipos

- Conversão forçada de tipos: utilização de operador `cast`
`(<tipo>) <expressão>`
- Valor `<expressão>` convertido para tipo `<tipo>` como se tratasse de atribuição
- Exemplo: `int i = (int) 2.34;`
- Conversão de `float` para `int`: *truncagem*
- Conversão de `double` para `float`: *truncagem ou arredondamento*
- Nas chamadas a funções, não é necessário recorrer a uma conversão forçada de tipos (ex: `double sqrt (double x);`):
`root2 = sqrt(2);` é equiv. a
`root2 = sqrt(2.0);`

Constantes - Tipos Enumerados

- Tipo enumerado definido por sequência de constantes
- `enum resposta { NAO, SIM };`
- Tipo `resposta` tem duas constantes: `NAO` e `SIM`
- Constantes de tipo enumerado têm valor inteiro (`int`): a primeira constante vale 0, a segunda vale 1, etc
- Tipo `resposta`: `NAO` vale 0 e `SIM` vale 1
- Pode-se especificar valores para as constantes ou não
`enum meses { JAN=1, FEV=2, MAR, ABR, MAI, JUN, JUL, AGO, SET, OUT, NOV, DEZ };`
- Permite criar uma abstracção dos valores quando se programa usando apenas os nomes do tipo enumerado

Constantes - Tipos Enumerados

```
#include <stdio.h>

enum meses { JAN=1, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SET, OUT,
            NOV, DEZ };

int main ()
{
    enum meses mes;

    mes=FEV;
    mes++;

    if (mes==MAR)
        puts("Estamos em Março");

    return 0;
}
```

Constantes - Tipos Enumerados

```
#include <stdio.h>

enum meses { JAN=1, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SET, OUT,
            NOV, DEZ };

int main ()
{
    enum meses mes;

    mes=FEV;
    mes++;

    if (mes==MAR)
        puts("Estamos em Março");

    return 0;
}
```

Declarações de Variáveis

- Precedem utilização e especificam tipo e lista das variáveis
- Sequência de declarações:

```
int superior, inferior, passo;  
char c, linha[1000];
```

- Alternativa:

```
int superior;  
int inferior;  
int passo;  
char c;  
char linha[1000];
```

Declarações de Variáveis

- Inicialização de variáveis externas (**globais**) e estáticas:
`<tipo> <variável> = <expressão constante>;`
- Variáveis globais são declaradas fora das funções
- Variáveis estáticas podem ser locais a uma função, mas mantêm o valor entre chamadas à função
- **Caso de omissão: valor 0**
 - Em C só as variáveis globais e estáticas são inicializadas automaticamente a 0, se o utilizador não fornecer nenhuma inicialização explícita
- Exemplo: `int pi = 3.14159;`

Declarações de Variáveis

- Inicialização de variáveis automáticas (**locais**):
`<tipo> <variável> = <expressão>;`
- Variáveis automáticas são reinicializadas sempre que a função é invocada
- **Caso de omissão: valor indefinido**
- Exemplo: `int i, j = f(5);`

Declarações de Variáveis

```
int global;
```

Variáveis globais são inicializadas automaticamente a zero

```
int contador() {  
    static int i = 1;  
    return i++;  
}
```

Uma variável "static" é inicializada apenas na primeira invocação da função e depois mantém o valor entre invocações

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

- Quais os valores escritos no standard output para `a`, `b` e `c`? Porquê?

\$ a = 1, b = 2, c = 3

Declarações de Variáveis

- **const** pode anteceder qualquer declaração
- Significa que valor não vai mudar
- Se tentar modificar o valor, o compilador detecta como sendo um erro
- Compilador pode tirar partido e fazer optimizações

- Exemplos:

```
const double e = 2.71828182845905;  
const char msg[] = "bem vindo ao C";  
int strlen(const char[]);
```

Inicialização de Vectores

- Exemplo anterior:

```
char msg[] = "bem vindo ao C";
```

- Posso fazer a mesma coisa com vectores inteiros

```
int numbers[] = {1, 44, 12, 567};
```

- ...e até com vectores de vectores (strings, neste caso)

```
char codes[][3] = {"AA", "AB", "BA", "BB"};
```



2 char's + '\0'

- Para imprimir o "AB" basta escrever

```
printf("%s", codes[1]);
```

Inicialização de Vectores

- E que tal assim?

```
int numbers[10] = {1, 44, 12, 567};
```

Se definirmos o tamanho

...sem inicializar todos os campos

... os restantes inteiros são inicializados a 0 !

Operadores e Precedências

- Operadores Aritméticos: $+$, $-$, $*$, $/$, $\%$
 - Arredondamentos de $/$ e $\%$ para números negativos é machine-dependent
 - Precedências: $*$, $/$, $\%$ **>>>** $+$, $-$
- Operadores Relacionais: $>$, $>=$, $<$, $<=$, $==$, $!=$
 - Precedências: $>$, $>=$, $<$, e $<=$ **>>>** $==$ e $!=$
- Operadores Lógicos: $!$, $\&\&$, $||$
 - $\&\&$ e $||$ avaliam argumentos da esquerda para direita e param se os argumentos são suficientes para definir valor
 - Precedências: $\&\&$ **>>>** $||$
 - Precedências: $!$ **>>>** Aritméticos

Operadores e Precedências

- Precedências:

! >>> Aritméticos >>> Relacionais >>> Lógicos

Valores de Verdade

- Valor numérico de expressão lógica é 1 se expressão for verdadeira e 0 se é falsa
- Qualquer valor diferente de 0 é interpretado como verdadeiro
- $!0$ é o valor 1
- $!x$ é o valor 0 para qq valor $x \neq 0$
- `if (x) {}` é equivalente a `if (x != 0) {}`
- `if (!x) {}` é equivalente a `if (x == 0) {}`

Operadores de Incremento e Decremento

- Operador incrementar variável ($++$) e decrementar variável ($--$) e retorna valor variável
- Operadores prefixos ($++<var>$, $--<var>$) primeiro incrementa/decrementa e depois retorna valores
- Operadores posfixos ($<var>++$, $<var>--$) primeiro retorna valor e depois incrementa/decrementa
- Se n é 5, qual é o valor de x depois de $x = n++$?
- Se n é 5, qual é o valor de x depois de $x = ++n$?

Operações Bit a Bit

- Em C é possível efectuar operações sobre a representação binária
- Manipular bits em inteiros (char, short, int, long):
 - `&` AND bit a bit
 - `|` OR bit a bit
 - `^` XOR (OR exclusivo) bit a bit
 - `<<` shift left
 - `>>` shift right

Operações Bit a Bit

15 = 1111

- $n = n \ \& \ 15;$
 - Põe a zero todos os *bits* de n que não os 4 de ordem mais baixa
- $n = n \ | \ 10;$
 - Põe a um os *bits* de n que estão a um na representação binária do número 10
- $72 \ | \ 184 = ??$
 - $01001000 \ |$
 $10111000 =$
 - $11111000 = 248$
- $72 \ \& \ 184 = ??$
 - $01001000 \ \&$
 $10111000 =$
 - $00001000 = 8$

Operações Bit a Bit

- Qual o output do seguinte exemplo?

```
int x = 1, y = 2;  
int z = x & y;  
int w = x && y;  
  
printf("z = %d w = %d \n", z, w);
```

x: 01

y: 10

z: 00

\$ z = 0 w = 1

Operações Bit a Bit

- $n = x \hat{\ } y;$
 - Põe em n a um (zero) os *bits* que em x e y são diferentes (iguais) — "ou" exclusivo.
- $n = x \ll 2;$
 - Desloca *bits* 2 posições esquerda. Espaço preenchido com 0
 - Cada *bit* deslocado equivale a uma multiplicação por 2
- $n = x \gg 1;$
 - Desloca *bits* 1 posição p/ direita. Espaço preenchido com 0
 - Cada *bit* deslocado equivale a uma divisão por 2

1 << 2 = ??

1=1

=

100 = 4

4 >> 1 = ??

4=100

=

10 = 2

Atribuições e Expressões

- $i = i + 1;$ pode ser reescrito como $i += 1;$
- $+=$ é um operador atribuição
- Existem outros operadores correspondentes usando $-, *, /, \%, >>, <<, \&, \wedge, |$
- $\langle \text{expr1} \rangle \langle \text{op} \rangle = \langle \text{expr2} \rangle$ equivale a
 $\langle \text{expr1} \rangle = (\langle \text{expr1} \rangle) \langle \text{op} \rangle (\langle \text{expr2} \rangle)$
- Vantagens operadores atribuição:
 - Mais próxima maneira de pensar de humanos; Ex: $i += 2;$
 - Simplifica leitura de expressões complicadas; Ex:
`yyval[yyvsp[p3+p4]+yyvsp[p1+p2]] += 2;`

Expressões Condicionais

- Expressão condicional: expressão cujo valor depende de uma outra expressão

`<expr1> ? <expr2> : <expr3>`

- Se `<expr1>` for verdadeiro, valor da expressão é `<expr2>`
- Se `<expr1>` for falso, valor da expressão é `<expr3>`

```
int maior (int a, int b) {  
    if (a > b)  
        return a;  
    else  
        return b;  
}
```

```
int maior (int a, int b) {  
    return (a > b ? a : b);  
}
```

Tabela de Precedência e Ordem de Avaliação

() [] -> .
! ~ ++ -- * & (tipo) sizeof
* / %
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
?:
= += -= *= /= %= &= ^= |= <<= >>=
,

ED

DE

ED

ED

ED

ED

ED

ED

ED

ED

ED

ED

DE

DE

ED





DEI

DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA

TÉCNICO LISBOA

Controlo de Execução

K&R: Capitulo 3

IAED

Controlo de Execução

- Instruções e Blocos
- `if`
 - `else-if`
- `switch`
- Ciclos:
 - `while` e `for`
 - `do-while`
- `break` e `continue`
- `goto` e labels

Instruções e Blocos

- Instrução
 - Expressão terminada por ' ; '
 - Caracter ' ; ' denota o fim de uma instrução
 - Exemplo: `x = 0; i++;`
- Bloco
 - Chavetas, { }, permitem agrupar declarações e instruções
 - instruções de uma função
 - conjuntos de instruções em `if`, `for`, `while`, etc.
 - Exemplo:

```
{ int x, i = 1; x = 0; i++; printf("%d %d\n"); }
```

Instruções e Blocos

```
int main ()
{
    int a = 2, b = 3;
    if (a > 0)
    {
        int aux = a;
        a = b;
        b = aux;
    }
    printf("%d %d %d\n", a, b, aux);
    return 0;
}
```



- Qual o resultado deste exemplo?

teste.c: In function 'main':

teste.c:12: error: 'aux' undeclared (first use in this function)

Execução Condicional: `if`

- Permite exprimir decisões:

```
if (<expressao>)  
    <instrucao1>  
else  
    <instrucao2>
```

- Se `<expressao>` tem valor diferente de 0
então `<instrucao1>` é executada
- Se `<expressao>` tem valor igual a 0
então `<instrucao2>` é executada

Execução Condicional: `if` – `else if`

- Permite exprimir decisões:

```
if (<expressao1>)  
    <instrucao1>  
else if (<expressao2>)  
    <instrucao2>
```

- Se `<expressao1>` tem valor diferente de 0
então `<instrucao1>` é executada
- Se `<expressao1>` tem valor igual a 0, **e**
 `<expressao2>` é diferente de 0
então `<instrucao2>` é executada

Execução Condicional: `if` – `else if` – `else`

- Permite exprimir decisões:

```
if (<expressao1>)  
    <instrucao1>  
else if (<expressao2>)  
    <instrucao2>  
else if (<expressao3>)  
    <instrucao3>  
...  
else  
    <instrucao_default>
```

Execução Condicional: `switch`

- Decisão com opções múltiplas; testa se uma expressão assume um de um conjunto de valores constantes

```
switch ( <expressao> ) {  
    case <const-expr>:  
        <instrucoes1>  
    case <const-expr>:  
        <instrucoes2>  
    default:  
        <instrucoes3>  
}
```

- **default** é opcional e é executado se a expressão é diferente de qualquer dos outros casos

Execução Condicional: `switch`

- Decisão com opções múltiplas; testa se uma expressão assume um de um conjunto de valores constantes

```
switch (c = getchar()) {  
    case 'a':  
        <instrucoes1>  
    case 'b':  
        <instrucoes2>  
    default:  
        <instrucoes3>  
}
```

- **default** é opcional e é executado se a expressão é diferente de qualquer dos outros casos

Execução Condicional: `switch`

- Decisão com opções múltiplas; testa se uma expressão assume um de um conjunto de valores constantes

```
switch (c = getchar()) {  
    case 'a':  
        <instrucoes1>  
    case 'b':  
    case 'B':  
        <instrucoes2>  
    default:  
        <instrucoes3>  
}
```

- **default** é opcional e é executado se a expressão é diferente de qualquer dos outros casos

Ciclos Genéricos: **while**

```
while ( <expressao> )  
    <instrucoes>
```

- Enquanto <expressao> for diferente de zero, a <instrucao> é executada
- Ciclo termina quando valor de <expressao> for zero

Ciclos Contados: for

```
for ( <expr1> ; <expr2> ; <expr3> )  
    <instrucoes>
```

```
<expr1>;  
while ( <expr2> ){  
    <instrucoes>;  
    <expr3>;  
}
```

- Expressão de inicialização: <expr1>
- Condição de ciclo: <expr2>
- Expressão de incremento: <expr3>
- Ciclos com inicialização e incremento simples

Ciclos: do-while

```
do {  
    <instrucoes>  
} while ( <expressao> );
```

- Enquanto <expressao> for diferente de zero, as <instrucoes> são executadas
- Ciclo termina quando valor de <expressao> for zero
- Note-se que <instrucoes> são executadas sempre pelo menos uma vez

```
/* O valor de n tem que ser superior ou igual a 2 */  
int n = 0;  
do {  
    puts("introduza um valor valido");  
    scanf("%d", &n);  
} while (n < 2);
```

Instruções **break** e **continue**

- A instrução **break** permite terminar a execução de um `for`, `while`, `do-while` ou `switch`
- A instrução **continue** desencadeia a execução da próxima iteração de um `for`, `while` ou `do-while`
 - Num ciclo `for`, a execução continua com a expressão de incremento
- O que acontece aqui ? E se fosse **break** em vez de **continue** ?

```
for(i = 0; i < n; i++) {  
    if (a[i] <= 0)  
        continue;  
    printf("%d ", a[i]);  
}
```

Revisitando o `switch` + `break`

- Decisão com opções múltiplas; testa se uma expressão assume um de um conjunto de valores constantes

```
switch (c = getchar()) {  
    case 'a':  
        <instrucoes1>  
        break;  
    case 'b':  
        <instrucoes2>  
        break;  
    default:  
        <instrucoes3>  
}
```

- **default** é opcional e é executado se a expressão é diferente de qualquer dos outros casos

Exemplo: interface para a *command line*

- Vamos supor que queremos fazer um programa com 3 comandos — a, b e x, sendo que x termina o programa

```
int main()
{
    char command;
    while ((command = getchar()) != 'x') {
        switch (command) {
            case 'a':
                /* Chama a funcao responsavel pela execucao do comando a */
                break;
            case 'b':
                /* Chama a funcao responsavel pela execucao do comando b */
                break;
            default:
                printf("ERRO: Comando desconhecido\n");
        }
        getchar(); /* le o '\n' introduzido pelo utilizador */
    }
    return 0; /* done! */
}
```

Exemplo: interface para a *command line*

- Vamos supor que queremos fazer um programa com 3 comandos — a, b e x, sendo que x termina o programa

```
int main()
{
    char command;
    while (1) {
        command = getchar(); /* le o comando */
        switch (command) {
            case 'a':
                /* Chama a funcao responsavel pela execucao do comando a */
                break;
            case 'b':
                /* Chama a funcao responsavel pela execucao do comando b */
                break;
            case 'x':
                return 0; /* Termina o programa com sucesso */
            default:
                printf("ERRO: Comando desconhecido\n");
        }
        getchar(); /* le o '\n' introduzido pelo utilizador */
    }
    return -1; /* se chegou aqui algo correu mal */
}
```

Exemplo: interface para a *command line*

- Vamos supor que queremos fazer um programa com 3 comandos — a, b e x, sendo que x termina o programa

```
int main()
{
    char command;
    while (1) {
        command = getchar(); /* le o comando */
        switch (command) {
            case 'a':
                /* Chama a funcao responsavel pela execucao do comando a */
                executa_a();
                break;
            case 'b':
                /* Chama a funcao responsavel pela execucao do comando b */
                break;
            case 'x':
                return EXIT_SUCCESS; /* Termina o programa com sucesso (STDLIB) */
            default:
                printf("ERRO: Comando desconhecido\n");
        }
        getchar(); /* le o '\n' introduzido pelo utilizador */
    }
    return EXIT_FAILURE; /* se chegou aqui algo correu mal (STDLIB) */
}
```