Tipos básicos do C

CAP Diogo Silva

Table of contents

Tipos básicos do C

Inteiros

Até agora usámos apenas o tipo `int` para representar números inteiros, mas existem outros:

```
short int
unsigned short int

int
unsigned int

long int
unsigned lont int
```

Diferentes tipos podem representar números inteiros em diferentes intervalos e ocupam mais ou menos espaço em memória.

Estes intervalos e espaço ocupado em memória pode variar de máquina para máquina, mas é garantido que:

`short int < int < long int`

Processadores com arquitecturas de 64 bits começam a ser comuns e os intervalos de valores comuns são:

Uma forma rápida de verificar os limites de um determinado tipo numa máquina, é usar a biblioteca `<limits.h>`.

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

int main(void) {
  int v;
  printf("max long=%ld\n", LONG_MAX);
}
```

Lista de todas as constantes na documentação da biblioteca.

https://man7.org/linux/man-pages/man0/limits.h.0p.html

Inteiros

Constantes

Até agora definimos constantes de inteiros no formato decimal simples, i.e. usando 10 digitos distintos.

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

int main(void) {
  int v = 42;  // 42 é constante de inteiro
}
```

Mas existem outras formas.

É possível definir constantes em formato octal e hexadecimal. Não iremos explorar estas bases, mas veremos como escrever constantes de inteiros diferentes.

```
15L  -> interpretar 15 como um long int
15U  -> interpretar 15 como um unsigned int
15UL -> interpretar 15 como um unsigned long int
```

Também se pode escrever U e L em minúsculas. Pode-se escrever UL ou LU.

Overflow

Como vimos anteriormente, os tipos inteiros têm um valor máximo e mínimo que podem representar.

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

int main(void) {
```

```c
  int v = INT_MAX;
  printf("v=%d\n", v);
  printf("v=%d\n", v+1);
  return 0;
}
```

- v = ?
- v + 1 = ?

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

int main(void) {
  int v = INT_MAX;
  printf("v=%d\n", v);
  printf("v=%d\n", v+1);
  return 0;
}
```

- v = 2147483647
- v + 1 = -2147483648

---

printf e scanf

```c
scanf("%u", &v);  // unsigned int
printf("%u", v);

scanf("%hd", &v);  // short int
printf("%hd", v);

scanf("%hu", &v);  // unsigned short int
printf("%hu", v);

scanf("%ld", &v);  // long int
printf("%ld", v);
```

```
scanf("%lu", &v);  // unsigned long int
printf("%lu", v);
```

Reais

```
float
double
long double
```

–

O long double não aparece porque os intervalos variam bastante de máquina para máquina.

---

constantes

Diferentes formas de escrever o número 57:

```
57.0
57.
57.0e0
57E0
5.7e1
5.7e+1
.57e2
570.e-1
```

---

printf scanf

```
double d;
scanf("lf", &d);
printf("lf", d);
```

```c
long double ld;
scanf("%Lf", &ld);
printf("%Lf", ld);
```

Texto

char

---

char

Até agora usámos texto apenas no contexto do scanf e print.

O C tem um tipo para texto: o char.

```c
char letra = 'C';
```

- Um char guarda 1 letra.
- As constantes de char são escritas com plicas ' '. Não confundir com as aspas " " usadas para strings (e.g. o printf e scanf).

Internamente, um char é apenas um número inteiro que pode ser interpretado como letras, através da tabela ASCII.

ASCII = American standard Code for Information Interchange

Não precisamos de saber esta correspondência, porque podemos sempre interpretar um char como um inteiro e vice-versa.

```c
char letra = 'A';
printf("A letra %c tem o valor %d\n", letra, letra);  // A 65
```

A especificação de conversão para char é o %c.

Decimal	Escape Sequence Oct	Hex	Char	Character							
0	\0	\x00		*nul*	32		64	@	96	`
1	\1	\x01		*soh* (^A)	33	!	65	A	97	a
2	\2	\x02		*stx* (^B)	34	"	66	B	98	b
3	\3	\x03		*etx* (^C)	35	#	67	C	99	c
4	\4	\x04		*eot* (^D)	36	$	68	D	100	d
5	\5	\x05		*enq* (^E)	37	%	69	E	101	e
6	\6	\x06		*ack* (^F)	38	&	70	F	102	f
7	\7	\x07	\a	*bel* (^G)	39	'	71	G	103	g
8	\10	\x08	\b	*bs* (^H)	40	(	72	H	104	h
9	\11	\x09	\t	*ht* (^I)	41	)	73	I	105	i
10	\12	\x0a	\n	*lf* (^J)	42	*	74	J	106	j
11	\13	\x0b	\v	*vt* (^K)	43	+	75	K	107	k
12	\14	\x0c	\f	*ff* (^L)	44	,	76	L	108	l
13	\15	\x0d	\r	*cr* (^M)	45	-	77	M	109	m
14	\16	\x0e		*so* (^N)	46	.	78	N	110	n

Figure 1: ascii

Por serem números inteiros, podemos realizar operações aritméticas sobre char.

```
char letra = 'C';
printf("A letra depois do %c = %c\n", letra, letra+1);  // C D
```

```
// abecedário completado em maísculas
for(char l='A'; l<'Z'; l++)
  printf("%c,"l);
printf("\n");
```

Tal como vimos para os tipos inteiros, o char também pode ser signed ou unsigned.

```
char n = 127+1;
printf("n=%d\n", n); // -128

unsigned char n2 = 255+1;
```

```
printf("n2=%d\n", n2);  // 0
```

Também está sujeito a overflow.

- Já usámos anteriormente caracteres especiais, e.g. `'\n'` e `'\t'`. Existem outros.
- Alguns só podem ser especificados em formato octal ou hexadecimal.

---

scanf

- Quando o scanf acaba de processar um determinado input, existem caracteres que ficam por consumir.
- Como todos os caracteres são válidos para o tipo char, isso pode ser um problema.

---

scanf

```
char c1, c2;
printf("Introduza um caracter:");
scanf("%c", &c1);

printf("Introduza outro caracter:");
scanf("%c", &c2);

printf("c1=%c   c2=%c  \n", c1, c2);
```

```
Introduza um caracter:a
Introduza outro caracter:c1=a   c2=
```

- Aparentemente, o segundo scanf foi ignorado:
  - o utilizador não escreveu nada
  - o que estaria depois de c2= está vazio

- Na verdade, o primeiro scanf deixa um enter `'\n'` por consumir.
- No scanf seguinte, o que é pedido é um char.
- Como `'n'` é um char válido, a especificação de conversão aceita-o como input
- Deixa de ser necessário pedir input ao utilizador porque as especificações de conversão já foram satisfeitas.

Como confirmar? Vamos interpretar c2 como um inteiro.

```
char c1, c2;
printf("Introduza um caracter:");
scanf("%c", &c1);

printf("Introduza outro caracter:");
scanf("%c", &c2);

printf("c1=%c   c2=%d  \n", c1, c2);
```

```
Introduza um caracter:a
Introduza outro caracter:c1=a   c2=10
```

---

alternativas para ler e escrever char

Existem outras formas de ler e escrever um char. - getchar - putchar

---

putchar

A função putchar escreve um caracter na consola.

```
putchar('C');
```

getchar

A função getchar lê um único char.

```
l = getchar();
```

Tal como no scanf, o getchar não salta espaços em branco quando lê um char.

E se quisermos ler vários chars?

Usamos um ciclo.

```
// lê chars até encontrar \n
char l;
do{
  scanf("%c", &l);
} while (l != '\n');
```

```
char l;
while ((l = getchar()) != '\n')
  ;
```

---

exercicio

Calcular comprimento de mensagem

Conversão de tipos

- No C, é possível converter de uns tipos para outros.
- Na verdade, nós já usámos esta funcionalidade sem saber, porque existem conversões que são automáticas e implicitas.
- Outras têm de ser explicitamente declaradas.

---

conversões implicitas

- Quando realizamos operações binárias, o C consegue detectar se os 2 operandos são do mesmo tipo.
- Se não forem, um dos tipos é convertido no outro, porque as operações são feitas com operandos do mesmo tipo.
- O resultado da operação será do tipo "superior".

---

conversões implicitas

```c
int i;
float f, p;
p = f + i;
```

- Neste caso, o valor de i será convertido para float.
- Se o contrário ocorresse, perdiamos por completo a componente decimal de f.
- Desta forma, o pior que pode acontecer é o valor de i perder precisão depois de convertido.

---

conversões implicitas > ambos operandos da mesma "classe"

```
  reais                  inteiros

long double          unsigned long int
    ^                         ^
    |                         |
  double                 long int
    ^                         ^
    |                         |
  float                unsigned int
                              ^
                              |
                             int
```

---

conversões implicitas > exemplos

```
char c;
short int s;
int i;
unsigned int u;
long int l;

i = i + c; // c convertido para int
i = i + s; // s convertido para int
u = u + i; // i convertido para unsigned int
l = l + u; // u convertido para long int
```

---

conversões implicitas > mais exemplos

```
long int l;
unsigned long int ul;
float f;
double d;
long double ld;

ul = ul + l; // l convertido para unsigned long int
f = f + ul; // ul convertido para float
d = d + f; // f convertido para double
ld = ld + d // d convertido para long double
```

---

conversões implicitas > atribuição

```
char c;
int i;
float f;
double d;

i = c; // c convertido para int
f = i; // i convertido para float
d = f; // f convertido para double
```

```
i = 3.14; // 3.14 convertido para 3
c = 10000; // overflow
f = 1.0e100; // excede limite
```

---

conversões explicitas > casting

- Para fazer uma conversão explicita, escrevemos o nome do tipo final entre parêntises, seguido do valor que queremos converter.

```
float f = 3 / 2; // 1.0 -> divisão de inteiros dá inteiro
f = (float) 3 / 2; // 1.5 -> converter 3 para float e dividir por 2
```

- O operador de casting é uma operação unária.
- Operações unárias têm precedência sobre operações binárias.

---

conversões explicitas > casting

Quando realizamos algumas operações aritméticas, pode ser necessário fazer uma conversão explicita.

```
long i;
int j = 10000; // 10000 * 10000 -> 100 000 000

i = j * j;
```

- O resultado da multiplicação na linha 4 cabe na variável i de tipo long.
- Contudo, o resultado da operação será um int e em algumas máquinas o resultado pode levar a overflow.

---

conversões explicitas > casting

```
long i;
int j = 10000; // 10000 * 10000 -> 100 000 000
```

```c
i = j * j;

i = (long) j * j;

i = (long) (j * j); // ERRADO
```

- Para resolver isso, podemos fazer o cast da linha 6.
- Na linha 8, a multiplicação é feita antes da conversão porque está entre ().

Definições de tipos

O C permite a definição de novos tipos com o comando `typedef`.

```c
typedef int Altura;
```

- typedef é seguido do nome original do tipo
- e depois do novo nome que queremos usar

---

typedef

```c
typedef int Altura;
typedef int Massa;
Altura a = 180;
Massa m = 75;
```

- Essencialmente, o que fizemos foi criar um int com um novo nome.
- Útil para tornar o código mais legível

---

typedef

```c
typedef int Altura;
typedef int massa;  // aceite, mas não é convenção
```

```c
int main(){
  //...
}
```

- As definições de tipo ocorrem fora de qualquer função, tipicamente após os #include.
- Os nomes dos tipos obedecem às mesmas regras dos nomes das variáveis.
- É convenção no C, capitalizar os nomes dos tipos.

---

sizeof

A função sizeof recebe um valor ou um tipo e indica qual é o tamanho, em bytes, que esse tipo ocupa em memória.

```c
char c;
printf("size of int = %lu bytes\n", sizeof(int));  // 4
printf("size of char = %lu bytes\n", sizeof(3.14));  // 8 -> double
printf("size of char = %lu bytes\n", sizeof(c));  // 1
```