UAlg FCT

UNIVERSIDADE DO ALGARVE

FACULDADE DE CIÊNCIAS E TECNOLOGIA

Programação Imperativa

Lição n.º 4

Tipos numéricos: int e double

Aritmética em C

- Aritmética em C.
- Aritmética int.
- Aritmética double.
- Aritmética mista.
- Funções matemáticas de biblioteca.
- Funções max e min.

Aritmética em C

- As regras da aritmética do C são semelhantes às da aritmética da matemática, que aprendemos na escola primária.
- Mas há diferenças subtis, que frequentemente nos apanham desprevenidos.
- Primeira observação importante: os números inteiros são representados pelo tipo int, mas o tipo int não representa todos os números inteiros!
- Só representa os número inteiros do intervalo [-2147483648..2147483647].

Testando a adição de ints, de novo

- Eis um programa com uma função de teste que faz repetidamente a adição de dois números int:

```
#include <stdio.h>

void test_addition(void)
{
  int x;
  int y;
  while (scanf("%d%d", &x, &y) != EOF)
  {
    int z = x + y;
    printf("%d\n", z);
  }
}

int main(void)
{
  test_addition();
  return 0;
}
```

```
$ ./a.out
3 7
10
3000 7000
10000
3000000 7000000
10000000
3000000000 7000000000
1410065408
2000000000 1
2000000001
2000000000 2000000000
-294967296
3000000000 0
-1294967296
```

Conclusão: quando uma das parcelas ou o resultado sai do intervalo dos int, está tudo estragado.

[-2147483648..2147483647] ou [$-2^{31}..2^{31}-1$]

- Em C, cada número int ocupa uma *palavra* de 32 bits.
- Uma palavra são quatro *bytes* consecutivos, cada um com 8 bits.
- A sequência dos valores dos bits da palavra constitui a representação binária do número.
- Logo, com 32 bits, podem ser representados no máximo 2^{32} = 4294967296 números diferentes.
- Metade serão negativos, um é o zero e metade menos um serão positivos.
- Por isso, o intervalo dos números int é [$-2^{31}..2^{31}-1$], ou [-2147483648..2147483647].

Overflow

- Há *overflow* de inteiros quando o resultado de um cálculo com números inteiros cai fora do intervalos dos números int.
- Quando há *overflow*, os cálculos aritméticos ficam errados, irremediavelmente.

```
sources pedro$ ./a.out
2147483647 1
-2147483648
2147483647 10
-2147483639
2147483647 20
-2147483629
-2147483648 -1
2147483647
```

Repare, 2147483647 + 1 dá -2147483648. É como se o sucessor do maior número fosse o menor número. Analogamente -2147483648 – 1 dá 2147483647, como se o predecessor do menor número fosse o maior número.

Operações aritméticas, tipo int

- Adição: x + y
- Subtração: x – y
- Multiplicação: x * y
- Quociente da divisão inteira: x / y
- Resto da divisão inteira: x % y

Note bem: ambos os operandos, x e y, representam expressões cujo valor é um número int. O resultado, se houver, é um valor de tipo int.

Cuidados:

- Não deixar dar *overflow*.
- Não deixar o divisor ser zero. Se o divisor for zero, o programa "estoira".
- Não usar operandos com valor negativo na operação resto da divisão inteira.

Testando as operações aritméticas, int

```
void test_operations_int(void)
{
  int x;
  int y;
  while (scanf("%d%d", &x, &y) != EOF)
  {
    int z1 = x + y;
    printf("%d\n", z1);
    int z2 = x - y;
    printf("%d\n", z2);
    int z3 = x * y;
    printf("%d\n", z3);
    int z4 = x / y;
    printf("%d\n", z4);
    int z5 = x % y;
    printf("%d\n", z5);
  }
}
```

```
$ ./a.out
20 7
27
13
140
2
6
33 50
83
-17
1650
0
33
14 3
17
11
42
4
2
```

Operações aritméticas, tipo double

- Adição: x + y
- Subtração: x – y
- Multiplicação: x * y
- Quociente da divisão: x / y

Note bem: ambos os operandos, x e y, representam expressões cujo valor é um número double. O resultado, se houver, é de tipo double.

Cuidados:
- Não deixar o divisor ser zero.
- Não contar com precisão ilimitada na representação do resultado.

Testando as operações aritméticas, double

```
void test_operations_double(void)
{
  double x;
  double y;
  while (scanf("%lf%lf", &x, &y) != EOF)
  {
    double z1 = x + y;
    printf("%f\n", z1);
    double z2 = x - y;
    printf("%f\n", z2);
    double z3 = x * y;
    printf("%f\n", z3);
    double z4 = x / y;
    printf("%f\n", z4);
  }
}
```

Note bem: o operador %, resto da divisão inteira, não existe com para números double.

```
$ ./a.out
25.0 4.0
29.000000
21.000000
100.000000
6.250000
14.0 3.0
17.000000
11.000000
42.000000
4.666667
6.125 0.5
6.625000
5.625000
3.062500
12.250000
0.333333 0.5
0.833333
-0.166667
0.166666
0.666666
```

Aritmética mista

- Quando numa expressão do tipo x+y, x-y, x*y ou x/y um dos operandos é double e o outro é int, este é "convertido" automaticamente para double e aplicam-se as regras da aritmética de doubles.
- Esta conversão é pacífica, pois o valor aritmético mantém-se
- A conversão inversa, de double para int, é mais delicada, pois, em geral, o valor aritmético muda.
- Nos casos em que muda, temos de ter a certeza de que é isso que queremos.
- A conversão de double para int é traiçoeira, porque pode ocorrer inadvertidamente, por erro de programação.

O tipo double

- Informalmente dizemos que o tipo double representa os números reais, e que o tipo int representa os números inteiros.
- Mas, em rigor não é bem isso.
- Já sabemos que o tipo int "só" representa os inteiros do intervalo $[-2^{31}..2^{31}-1]$.
- E representa todos os números desse intervalo.
- O tipo double representa números *racionais*: alguns desses números são inteiros e outros não.
- Mas o tipo double não representa todos os números racionais, nem sequer o todos os números racionais de um certo intervalo.
- Nem poderia, porque em qualquer intervalo numérico não vazio, há um número infinito de números racionais.

Funções matemáticas de biblioteca

O C traz um pequeno conjunto de funções matemáticas, operando sobre números double:

Para usar, inserir #include <math.h>.

Note bem: o resultado é double, mesmo quando representa um número inteiro.

Função	Significado
sin(x)	Seno de x.
cos(x)	Cosseno de x.
tan(x)	Tangente de x.
asin(x)	Arco seno de x, no intervalo [-π/2, π/2].
acos(x)	Arco cosseno de x, no intervalo [0, π].
atan2(y, x)	Arco tangente de y/x, no intervalo [-π, π].
exp(x)	Exponencial de x.
log(x)	Logaritmo natural de x.
pow(x, y)	Potência: x elevado a y.
sqrt(x)	Raiz quadrada de x.
floor(x)	Maior número inteiro menor ou igual a x.
ceil(x)	Menor número inteiro maior ou igual a x.
round(x)	O número inteiro mais próximo de x.
fabs(x)	Valor absoluto de x.

Arredondamento

- No problema da nota, precisamos de arredondar a nota exata, para o inteiro mais próximo, tendo o cuidado de arredondar para cima as meias unidades.
- Podemos usar a função round para isso.
- Mas atenção que o resultado da função round, sendo um número inteiro, é representado por um valor de tipo double!
- Note bem, o tipo do resultado da função round é double, mas o resultado é um número inteiro.

Tipo double e tipo int

- Não confunda: há números double que são números inteiros (por exemplo, 2.0, -314.0, 0.0, 5e3).
- Note bem: 5e3 é uma maneira compacta de escrever o número dado pela expressão $5*10^3$.
- E claro, todos os números int são inteiros (por exemplo, 2, -314, 0, 5000).
- Na matemática 2.0 e 2 representam o mesmo número, que pertence ao conjunto dos números reais e também ao conjunto dos números inteiros.
- Em programação, o tipo double e o tipo int são tipos "disjuntos".
- Um valor numérico tem um tipo, e um só tipo, em C: não pode ser ao mesmo tempo double e int.

Nota final

- A nota final é o arredondamento da nota exata.
- Observe:

```
double final_grade(double lb, double ex)
{
  return round(grade(lb, ex));
}
```

Note que o resultado é um número inteiro expresso por um número double (e não por um número int), pois toda a aritmética é aritmética de doubles.

Função de teste para a nota exata

- Acrescentamos o novo cálculo à função test_grade:

```
void test_grade(void)
{
  double lb;
  double ex;
  while (scanf("%lf%lf", &lb, &ex) != EOF)
  {
    double v = weighted_average(lb, ex);
    printf("%f\n", v);
    double z = grade(lb, ex);
    printf("%f\n", z);
    int g = final_grade(lb, ex);
    printf("%d\n", g);
  }
}
```

Em geral, é prudente observar também os resultados intermédios nas funções de teste.

Nota de exame necessária

- Problema: para passar com y como nota final, não arredondada, quanto precisa conseguir no exame um aluno cuja nota da prática é x?
- Para começar, temos de resolver em ordem a z a inequação 0.3 * x + 0.7 * z >= y.
- Mas o resultado exato não basta, pois a nota do exame é expressa com uma casa decimal.
- Por exemplo, se der z >= 12.73, será preciso 12.8 no exame; 12.7 não seria suficiente para passar, com a nota pretendida!
- E, para mais, se der z maior que um número menor que 8.5, esse número não serve: para passar é preciso pelo menos 8.5 no exame.

Nota necessária exata

- Calculemos primeiro a nota necessária com a precisão possível, e sem considerar a questão do 8.5.
- A função exame_exact resolve a inequação, em ordem a z, com x representado por lab e y representado por goal:

```
double exam_exact(double lab, int goal)
{
  return (goal - 0.3 * lab) / 0.7;
}
```

Se o resultado for maior que 20.0, isso significa que é impossível passar com a nota desejada!

Arredondamento para cima às décimas

- Para arredondar para cima, às unidades, temos a função ceil.
- Como fazer para arredondar às décimas?
- Eis o truque: multiplica-se por 10, arredonda-se às unidades e divide-se por 10:

```
double ceiling_one_decimal(double x)
{
  return ceil(x * 10.0) / 10.0;
}
```

E se quiséssemos arredondar para cima às milésimas, como faríamos? E arredondar para cima, às dezenas?

Nota necessária com uma casa decimal

- Arredonda-se a nota exata, às décimas, para cima:

```
double exam_one_decimal(double lab, int goal)
{
  return ceiling_one_decimal(exam_exact(lab, goal));
}
```

Temos ainda de considerar o caso em que esta nota é menor que 8.5, pois um tal valor não daria para passar.

Máximo e mínimo, de dois números

- A função fmax retorna o valor do maior dos seus (dois) argumentos.
- A função fmin, idem, para o menor.
- Estas funções existem na biblioteca do C, para doubles.
- Para ints, não existe função de biblioteca; logo teremos de programar, quando forem precisas:

Para usá-las, inserir #include <math.h>.

```
int max(int x, int y)
{
  return x >= y ? x : y;
}

int min(int x, int y)
{
  return x <= y ? x : y;
}
```

Estas duas funções, tal como as funções fmax e fmin, são muito úteis, muitas vezes.

Nota necessária

- Se a nota exata arredondada for menor do que 8.5 a nota necessária é 8.5; caso contrário, a nota necessária é a nota exata arredondada.
- Por outras palavras: a nota necessária é o máximo entre a nota exata arredondada e 8.5:

```
double exam(double lab, int goal)
{
  return fmax(exam_one_decimal(lab, goal), 8.5);
}
```

Função de teste

- Na função de teste, observamos os cálculos intermédios e recalculamos a nota final, e também a nota final se tivéssemos uma décima a menos no exame:

```
void test_exam(void)
{
  double lb;
  int gl;
  while (scanf("%lf%d", &lb, &gl) != EOF)
  {
    double z1 = exam_exact(lb, gl);
    printf("%f\n", z1);
    double z2 = exam_one_decimal(lb, gl);
    printf("%f\n", z2);
    double z = exam(lb, gl);
    printf("%f\n", z);
    int x1 = grade(lb, z);
    printf("%d\n", x1);
    int x2 = grade(lb, z - 0.1);
    printf("%d\n", x2);
  }
}
```

```
$ ./a.out
15 10
7.857143
7.900000
8.500000
10.450000
8.400000
12.0 15
16.285714
16.300000
16.300000
15.010000
14.940000
12.0 18
20.571429
20.600000
20.600000
18.020000
17.950000
```