



UAlg FCT

UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Programação Imperativa

Lição n.º 2

A natureza da programação

A natureza da programação

- Problemas de programação.
- Decomposição funcional.
- Funções em C.
- A função **main**.
- Algoritmo da soma.



Problemas de programação

- Tipicamente, cada novo projeto de programação começa quando surge um novo problema de programação.
- A primeira tarefa do programador é entender o problema, intimamente.
- Há problemas simples e há problemas complicados, mas tudo é relativo.
- Eis um problema simples: somar dois números.
- Isto quer dizer, implicitamente, que alguém quer um programa que aceite dois números da consola e que mostre depois na consola a soma desses números.

Presunções

- Presumimos que os dois números são números inteiros quaisquer, representáveis no computador, e que a soma também é representável.
- Presumimos que não é responsabilidade do programa verificar que esses números são representáveis.
- Presumimos que os números são digitados pelo utilizador utilizando a representação decimal dos números a somar, um e depois o outro, separados por *espaço em branco*.
- Presumimos que não é responsabilidade do programa verificar isso.
- Presumimos que o utilizador quer ver a soma também representada decimalmente.
- Presumimos, mas convém confirmar...

Aritmética de inteiros **int**, em C

Existem em C as seguintes operações aritméticas:

- 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 estoura.
- Não usar operandos com valor negativo na operação resto da divisão inteira.

Solução

- Uma vez que o C tem a operação que nos pedem, podemos programar diretamente:

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

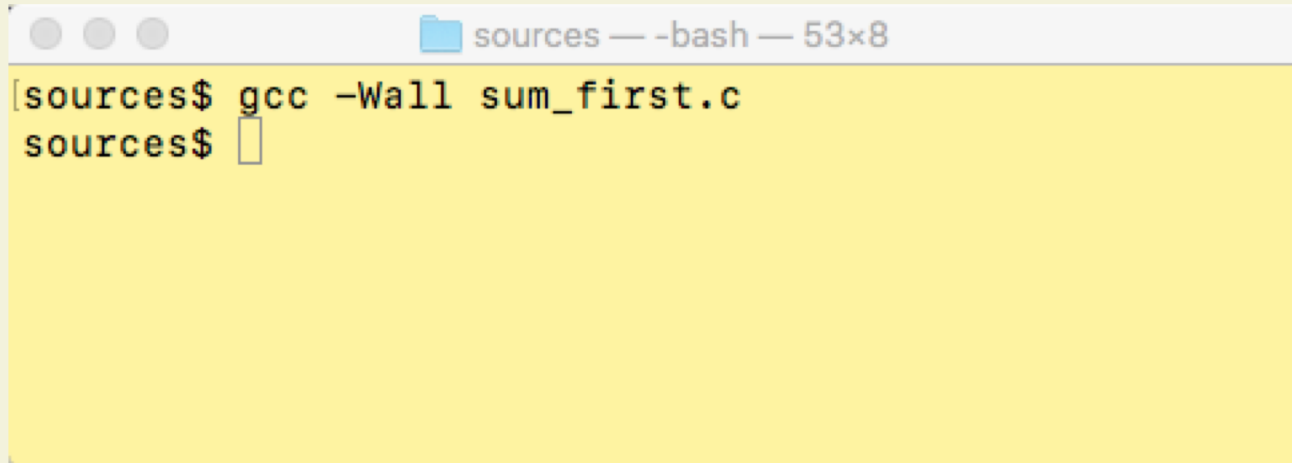
```
int main(void)
{
    int x;
    int y;
    scanf("%d%d", &x, &y);
    int z = x + y;
    printf("%d\n", z);
    return 0;
}
```

A função **main** é onde o programa começa e acaba.

Acaba com **return 0;**.

Compilação

- Depois de escrever o programa, compilamo-lo:

A terminal window titled 'sources — -bash — 53x8' with a yellow background. It shows the command 'gcc -Wall sum_first.c' being executed, followed by a new prompt line 'sources\$' with a cursor.

```
sources$ gcc -Wall sum_first.c
sources$
```

- Felizmente, não deu erros de compilação.
- Logo, podemos começar a experimentar.

Teste

- Eis a transcrição de uma sessão de teste:

Estes três casos não dão bem, e ainda bem que não dão, pois são inválidos. O programa não está preparado para os processar.

Este caso dá bem, mesmo sendo inválido!

```
sources$ gcc -Wall sum_first.c
[sources$ ./a.out
1 1
2
[sources$ ./a.out
3333 8888
12221
[sources$ ./a.out
300 -122
178
[sources$ ./a.out
1000000000 1000000000
2000000000
[sources$ ./a.out
1000000000 2000000000
-1294967296
[sources$ ./a.out
2147483647 1
-2147483648
[sources$ ./a.out
2147483647 -2147483648
-1
[sources$ ./a.out
abc 123
88764880
[sources$ ./a.out
3000000000 -1000000000
2000000000
[sources$ ./a.out
3000000000 1000000000
-294967296
sources$
```

Overflow!

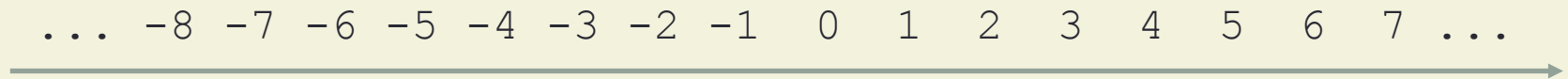
Dados inválidos

Números inteiros

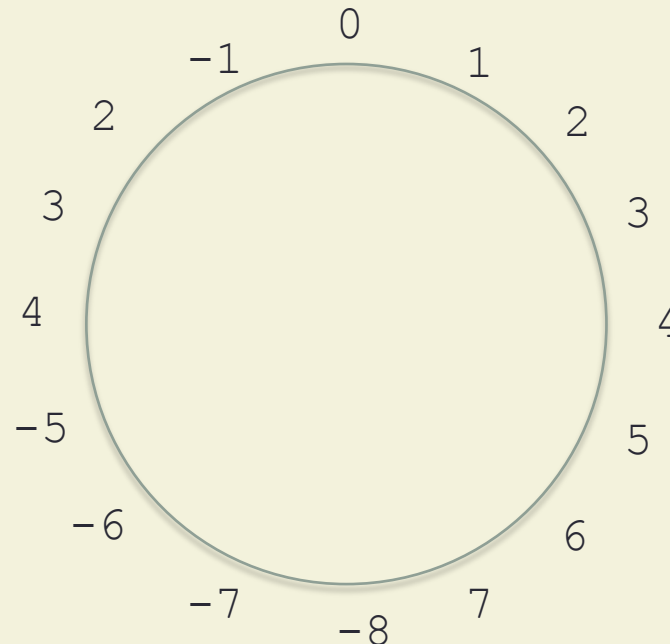
- Em matemática, os números inteiros constituem um conjunto infinito.
- Em programação, os números inteiros constituem um conjunto finito.
- Logo, os números inteiros da programação não são os números inteiros da matemática!
- Por conseguinte, a adição em programação é diferente da adição em matemática.

Reta, círculo dos números inteiros

- Imaginamos os números inteiros dispostos sobre uma reta.



- Em programação, é melhor imaginá-los dispostos sobre um círculo.



Para calcular a soma de x com y , partimos de x e avançamos y unidades para a direita, na reta ou y unidades no sentido dos ponteiros do relógio.

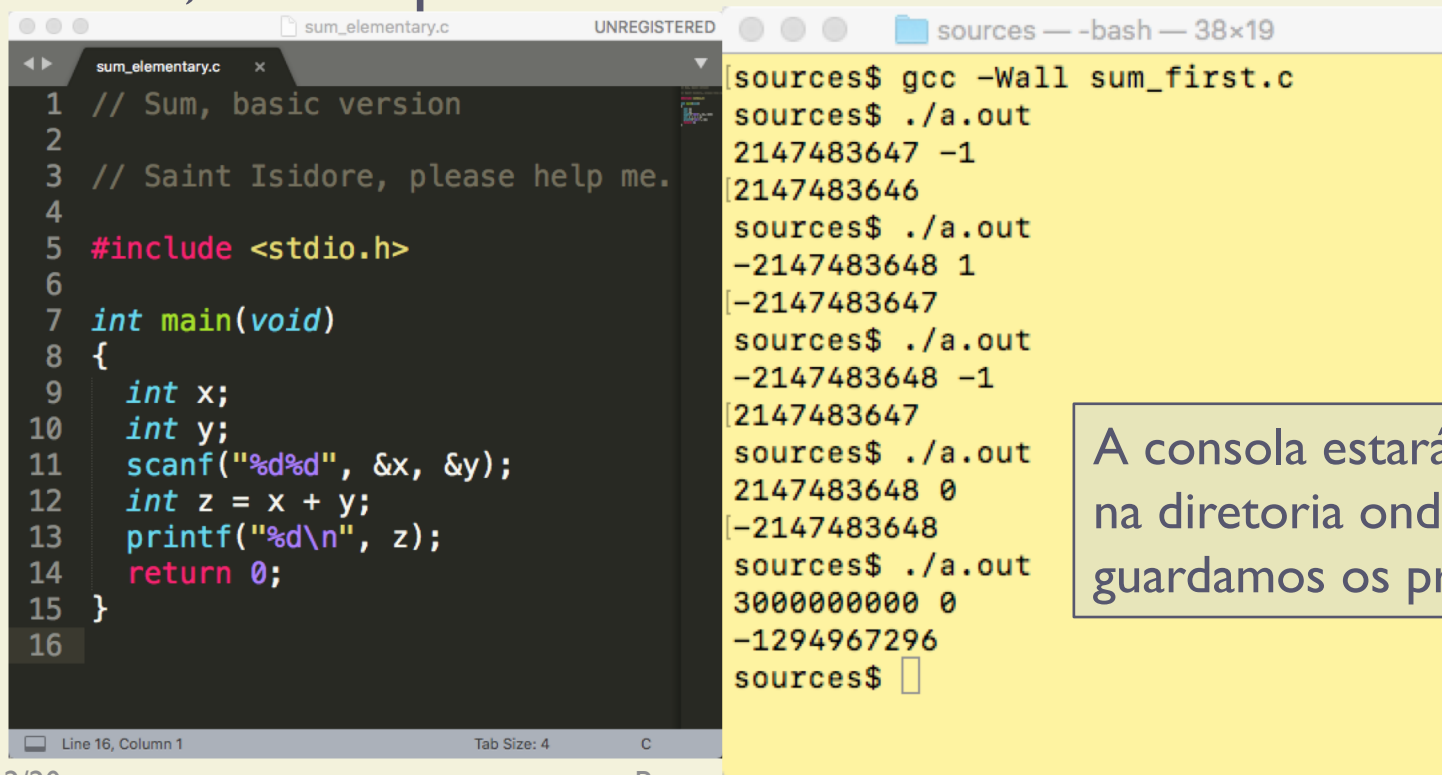
Memória

- Os números sobre os quais os computadores operam estão na **memória**.
- Se cada posição de memória tivesse quatro bits, daria para 16 números, entre -8 e 7.
- Na verdade, nos computadores habituais, cada **posição** de memória, chamada **byte**, tem quatro bits, e, portanto, dá para 256 números, entre -128 e 127.
- Usando quatro posições de memória, dá para 4294967296 números, entre -2147483648 e 2147483647.
- O maior número inteiro representável com 4 bytes é 2147483647.

2 1 4 7 4 8 3 6 4 7

Ambiente de programação

- Programaremos escrevendo os nossos programas num editor de texto (por exemplo, Sublime Text 3) e compilando numa consola.
- Teremos numa janela o editor e noutra a consola, assim, em esquema:



The screenshot displays two windows side-by-side. The left window is a code editor titled 'sum_elementary.c' with a dark theme. It contains the following C code:

```
1 // Sum, basic version
2
3 // Saint Isidore, please help me.
4
5 #include <stdio.h>
6
7 int main(void)
8 {
9     int x;
10    int y;
11    scanf("%d%d", &x, &y);
12    int z = x + y;
13    printf("%d\n", z);
14    return 0;
15 }
16
```

The right window is a terminal titled 'sources — -bash — 38x19'. It shows the compilation and execution of the program:

```
sources$ gcc -Wall sum_first.c
sources$ ./a.out
2147483647 -1
[2147483646]
sources$ ./a.out
-2147483648 1
[-2147483647]
sources$ ./a.out
-2147483648 -1
[2147483647]
sources$ ./a.out
2147483648 0
[-2147483648]
sources$ ./a.out
3000000000 0
-1294967296
sources$
```

A consola estará situada na diretoria onde guardamos os programas.

Outro problema, menos imediato

- Calcular a soma de todos os números entre dois números dados.
- Presumimos que se trata de números inteiros representáveis e a soma também.
- Presumimos que o primeiro número é menor ou igual ao segundo.
- Presumimos que os números dados entram na soma.

Progressão aritmética

- Trata-se da soma dos termos de uma progressão aritmética.
- Se não nos lembrarmos da fórmula, podemos ir ver à Wikipedia, por exemplo:

Soma dos termos de uma progressão aritmética

A soma dos termos de uma progressão aritmética situados no intervalo fechado de a_p até a_q é dada pelo produto do número de termos no intervalo, $(q - p + 1)$, pela média aritmética dos extremos do intervalo. Ou seja, pela seguinte fórmula:

$$S_{(p,q)} = \frac{(q - p + 1) \cdot (a_p + a_q)}{2}.$$

Soma do intervalo

- Programamos a fórmula numa função e depois chamamos a função:

```
#include <stdio.h>

int sum_from_to(int p, int q)
{
    return (q - p + 1) * (p + q) / 2;
}

int main(void)
{
    int x;
    int y;
    scanf("%d%d", &x, &y);
    int z = sum_from_to(x, y);
    printf("%d\n", z);
    return 0;
}
```

Aqui transcrevemos a fórmula da Wikipedia, mantendo as variáveis.

A seguir compilamos, testamos, etc.

Reformulação

- Dizer “somar todos os números entre x e y ”, é ambíguo, porque hesitamos se x e y entram na soma...
- Preferimos “somar a sequência de n números consecutivos a partir de x ”.
- Isso pode programar-se assim:

```
int sum_numbers_from(int x, int n)
{
    return (x + (x + n - 1)) * n / 2;
}
```

Moral da história

- Se não temos na linguagem, ou em alguma biblioteca de funções associada, uma operação que resolva o nosso problema, programamos uma função nova, recorrendo às que já existem.
- Porventura, programaremos não só a função que resolve o problema mas também outras, que tornam a programação dessa mais compreensível.
- De entre essas que programarmos guardaremos na nossa biblioteca pessoal aquelas que nos pareça poderem voltar a ser úteis no futuro.

Exercício

- Admitamos que o C não tem operações aritméticas.
- Admitamos que tem apenas três operações
 - calcular o *sucessor* de um número, expressa por $x+1$, para o número representado por x .
 - calcular o *predecessor* de um número, expressa por $x-1$, para o número representado por x .
 - verificar se um número é zero, expressa por $x==0$ para o número representado por x . O resultado será 1, se o número x valer 0 e valerá 0, se não.
- Como programar as operações aritméticas?

Adição elementar

- Já sabemos:

```
int sum(int x, int y)
{
    return y == 0 ? x : sum(x+1, y-1);
}
```

Programa completo

- O programa completo:

```
#include <stdio.h>

int sum(int x, int y)
{
    return y == 0 ? x : sum(x+1, y-1);
}

int main(void)
{
    int x;
    int y;
    scanf("%d%d", &x, &y);
    int z = sum(x, y);
    printf("%d\n", z);
    return 0;
}
```

Testando

- Eis a transcrição de uma função de teste:

Observamos três casos de *segmentation fault*.

O primeiro percebe-se: se a segunda parcela é um número negativo, o cálculo nunca acabaria. Ora, antes disso, esgotou-se a memória para fazer contas, por assim dizer, e isso fez ocorrer o *segmentation fault*.

Nos outros dois casos, os cálculos acabariam, mas tal como no caso anterior, a memória esgotou-se entretanto.

```
sources — -bash — 33x29
[sources$ gcc -Wall sum_3.c
sources$ ./a.out
3 5
8
sources$ ./a.out
1000 2000
3000
sources$ ./a.out
-5 9
4
sources$ ./a.out
9 -5
Segmentation fault: 11
sources$ ./a.out
10000 10000
20000
sources$ ./a.out
1000000 1000000
Segmentation fault: 11
sources$ ./a.out
1000000 1
1000001
sources$ ./a.out
1 1000000
Segmentation fault: 11
sources$ ./a.out
2147483647 1
-2147483648
sources$ ]
```

Outro exercício, multiplicação

- Admitindo que só temos aquelas três operações primitivas — somar 1, subtrair 1 e verificar se vale 0 — como programar a multiplicação?
- Podemos usar a função sum, é claro, pois essa está programada usando só aquelas três operações.
- Na verdade, fica bem simples:

```
int product(int x, int y)
{
    return y == 0 ? 0 : sum(x, product(x, y-1));
}
```

Programa completo, multiplicação

- Ei-lo:

```
#include <stdio.h>

int sum(int x, int y)
{
    return y == 0 ? x : sum(x+1, y-1);
}

int product(int x, int y)
{
    return y == 0 ? 0 : sum(x, product(x, y-1));
}

int main(void)
{
    int x;
    int y;
    scanf("%d%d", &x, &y);
    int z = product(x, y);
    printf("%d\n", z);
    return 0;
}
```


Adição da escola primária (I)

- A nossa função **sum** soma de 1 de 1.
- Isso não é muito eficiente...
- Desde a escola primária que conhecemos uma maneira melhor de somar números representados decimalmente.
- O algoritmo de adição da escola primária foi talvez o primeiro algoritmo que aprendemos.
- Devemos ser capaz de o programar!

Adição da escola primária (2)

- Tomemos um exemplo: $728+415$.
- Primeiro somamos 8 e 5, o que dá 13; dizemos “e vai 1”, e registamos 3.
- Depois continuamos para a esquerda, nas duas parcelas, somando 72 e 41, usando o mesmo esquema: somar 2 e 1, etc.
- Quando as contas acabarem saberemos que a soma de 72 e 41 é 113.
- Como “ia 1” há bocado, somamos esse 1 a 113, obtendo 114.
- Finalmente, acrescentamos o 3 de há bocado, obtendo 1143.

Adição da escola primária (3)

- Como fazemos para somar 8 e 5? Contamos pelos dedos ou lembramo-nos da tabuada, que memorizámos.
- E como obtemos 8, a partir de 718, numericamente? É o resto da divisão de 718 por 10.
- A divisão por 10 é muito fácil de realizar sobre a representação decimal.
- Aliás, obtemos 71 a partir de 718, calculando o quociente da divisão de 718 por 10 e analogamente para 41 a partir de 415.
- Somar 1 a um número equivale a calcular o seu sucessor, que é uma operação elementar.
- E acrescentar um algarismo a um número equivale a multiplicá-lo por 10 e adicionar o valor do algarismo.

Operações primitivas para a adição decimal

- As três do costume mais as seguintes:
 - Calcular o décuplo, expressa por $x * 10$, para o número representado por x .
 - Calcular o quociente da divisão inteira por 10, expressa por $x / 10$, para o número representado por x .
 - Calcular o resto da divisão inteira por 10, expressa por $x \% 10$, para o número representado por x .
- Além disso usamos a função `sum`, a tal de soma de 1 e 1, mas apenas em casos em que o segundo operando é menor que 10.

Algoritmo da adição decimal

```
int sum_decimal(int x, int y)
{
    return y < 10 ? sum(x, y) :
        sum(sum(sum_decimal(x/10, y/10),
            sum(y%10, x%10) / 10) * 10,
            sum(y%10, x%10) % 10);
}
```

Palavras para quê?

Formulação alternativa

- De início, preferimos uma formulação equivalente mais pausada. Observe:

```
int sum_decimal_alt(int x, int y)
{
    int result;
    if (y < 10)
        result = sum(x, y);
    else
    {
        int r1 = sum(x%10, y%10);
        int r2 = sum_decimal_alt(x / 10, y / 10);
        int carry = r1 / 10;
        result = sum(sum(r2, carry) * 10, r1 % 10);
    }
    return result;
}
```

Algoritmo da adição binária.

- Na nossa cultura, “vemos” os números na sua representação decimal, onde multiplicar e dividir por 10 é muito fácil.
- Os computadores “veem” os números na sua representação binária, onde multiplicar e dividir por 2 é muito fácil.
- Portanto, a nós programadores, o que interessa é o algoritmo da adição binária.
- Basta substituir todos os “10” por “2”.

Algoritmo da adição binária, em C

```
int sum_binary(int x, int y)
{
    int result;
    if (y < 2)
        result = sum(x, y);
    else
    {
        int r1 = sum(x%2, y%2);
        int r2 = sum_binary(x / 2, y / 2);
        int carry = r1 / 2;
        result = (r2 + carry) * 2 + r1 % 2;
    }
    return result;
}
```

Para o computador, qual é a versão preferível? Esta, com certeza!