Aula 2: Primeiros passos em C ¶

Nesta aula você vai aprender conceitos básicos da linguagem de programação C, que será usada no curso. Estes são conceitos fundamentais que se repetem em muitas outras linguagens de programação. No futuro, quando você for aprender outras linguagens, os conceitos vistos aqui serão muito úteis. Ao final desta aula espera-se que você saiba escrever alguns programas simples em C.

O que é uma linguagem de programação?

Uma linguagem de programação é uma linguagem *artificial* e *formal* usada para comunicar instruções a um computador. Nós vimos na aula passada um pequeno exemplo de um programa em *linguagem de montagem*, que é muito próxima à linguagem de máquina do computador. Por ser muito próxima à linguagem de máquina, é difícil escrever programas grandes em linguagem de montagem. Para isso existem linguagens de nível mais alto, como C.

Como uma linguagem natural, uma linguagem de programação possui uma *sintaxe*. No caso de linguagens de programação, a sintaxe é formalmente definida e deve ser seguida rigorosamente. Qualquer desvio torna o programa impossível de ser interpretado pelo computador. Em nosso curso, vamos aprender a sintaxe de C através de exemplos. Seria muito tedioso e pouco produtivo ensinar a sintaxe através de sua definição formal, mas é importante saber que tudo com o que lidamos está formalmente definido, não há nenhuma ambigüidade.

O que é um programa em C?

Um programa em C é um *texto* cujo conteúdo segue a sintaxe da linguagem C. Um exemplo curto é o seguinte:

```c
#include <stdio.h>

int main()
{
  printf("Hello World!\n");

  return 0;
}
```

Para ser executado pelo computador, o programa acima deve primeiro ser *compilado*. Um *compilador* para a linguagem C é um programa capaz de interpretar um programa em C, como o programa acima, e traduzí-lo para linguagem de máquina, gerando um *arquivo executável*.

Tipos de dados e variáveis

Um programa minimamente útil precisa manipular dados na memória do computador. A memória guarda apenas uma imensa seqüência de bits e o que eles representam é indiferente para o funcionamento da máquina. Linguagens como C, entretanto, facilitam a vida do programador disponibilizando *tipos de dados*.

Um tipo de dado é um conjunto de possíveis valores e operações que podem ser efetuadas sobre esses valores. A linguagem C nos oferece alguns tipos de dados *primitivos*. Um deles é o tipo `int`, usado para representar números inteiros.

A linguagem C nos permite declarar *variáveis*, que possuem um tipo determinado. Declarar uma variável é como dar um nome a um pedaço da memória que guardará um valor de determinado tipo. Daí em diante podemos acessar o valor guardado e manipulá-lo através do nome da variável. Para declarar uma variável de tipo `int` e nome `x`, fazemos assim:

```c
int x;
```

Note o ponto-e-vírgula. Sentenças em C, como acima, terminam sempre em ponto-e-vírgula. Você pode escrever quantas sentenças quiser na mesma linha, contanto que elas sempre terminem em ponto-e-vírgula.

Após declarar a variável `x`, podemos usar o seu nome para acessar o valor nela contido. Por exemplo, podemos usar o *operador de atribuição* `=` para guardar um valor em `x`:

```
int x;
x = 5;
```

A primeira linha *declara* a variável `x` e a partir daí podemos usar o nome `x` para nos referirmos a ela. A segunda linha coloca o número 5 na variável `x`. Note que, antes da atribuição, o valor contido em `x` é indefinido; ele pode ser qualquer coisa que estivesse na memória no lugar que foi associado ao nome `x`. Muitos erros de programação decorrem de variáveis *não inicializadas*. Por isso, geralmente é uma boa idéia declarar e inicializar uma variável ao mesmo tempo:

```
int x = 5;
```

O tipo `int` suporta diversas operações, como soma, produto, multiplicação, divisão inteira, comparações. As operações possíveis seguem a notação usual e a precedência dos operadores é a usual em matemática, ou seja, produtos e divisões são feitos antes que somas e subtrações:

```
int x = 5;
int y;

y = 1 + 2 * x * (5 - 8 * x);
x = x * y;
```

Na última linha, `x` aparece do lado esquerdo e do lado direito da atribuição. Isso significa sempre o seguinte: primeiro, encontra-se o valor da expressão do lado direito usando-se o valor atual de `x`, depois coloca-se esse valor em `x`. Qual o valor de `x` ao final da execução das sentenças acima?

Você deve experimentar o comportamento dos operadores aritméticos `+`, `-`, `*` e `/`. Desses, o operador de divisão `/` funciona diferente do esperado. Tente descobrir o que ele faz.

O nome de uma variável pode ser qualquer seqüência de letras e números e alguns símbolos como `_`. Um nome não pode começar com números e também não pode conter espaços. O uso de maísculas e minúsculas é permitido e maísculas são consideradas diferentes de minúsculas, de modo que `soma` e `sOma` são variáveis diferentes! Veja alguns nomes válidos:

```
int soma, x, y, x12, x_12, soma_abc, nome_bem_grande, NomeBemGrande;
```

Note também que você pode declarar várias variáveis ao mesmo tempo, separando os nomes por vírgulas.

Entrada e saída

Um programa que não consegue se comunicar com o usuário do computador também não é muito útil. Para imprimir algo na tela do computador, usamos em C a *função* `printf`. Mais adiante você vai entender bem o que é a função `printf`, no momento basta saber como usá-la para mostrar algo na tela.

Para mostrar uma linha de texto, fazemos assim:

```
printf("Hello World!");
```

Note que o texto vem entre aspas duplas. O texto pode ter qualquer tamanho, mas você não pode colocar uma quebra de linha no meio do texto. Por exemplo, o seguinte programa é **inválido**:

```
printf("Hello
World!");
```

Mas e se você quiser imprimir

```
Hello
World
```

em vez de `Hello World`? Basta usar a *seqüência de escape* (*escape sequence*) `\n`:

```
printf("Hello\nWorld");
```

faz exatamente o que você quer.

Como fazemos para mostrar um número inteiro? A função `printf` também pode ser usada para tanto:

```
printf("Eis um número: %d", 17);
```

O resultado é que `%d` será substituído por 17. A saída será:

```
Eis um número: 17
```

Você pode mostrar mais de um número ao mesmo tempo:

```
int a = 2;
int b = 3;
int resultado = a * b;

printf("%d vezes %d = %d", a, b, resultado);
```

Para ler um número inteiro digitado pelo usuário, usamos a função `scanf`. É preciso usar uma variável do tipo `int` para guardar o número lido. Por exemplo:

```
int x;
scanf("%d", &x);
```

Assim, o computador vai esperar que o usuário digite um número e pressione ENTER, e então esse número será colocado na variável `x`. Você notou o símbolo `&` antes de `x`? Ele é essencial; mais adiante vamos ver por que.

Um segundo programa

Vejamos agora um exemplo completo. O programa abaixo pede ao usuário que digite dois números e mostra o produto deles:

```
#include <stdio.h>

int main()
{
  int x, y;

  printf("Digite dois números: ");

  scanf("%d", &x);
  scanf("%d", &y);

  printf("O produto é %d\n", x * y);

  return 0;
}
```

O programa acima possui o formato básico de um programa em C a ser usado no início do nosso curso, a saber:

```
#include <stdio.h>

int main()
{
  /* Declaração de variáveis */

  /* Sentenças */

  return 0;
}
```

No momento, você não precisa compreender o que `#include <stdio.h>` quer dizer, o que `int main()` ou `return 0;` quer dizer. Basta copiar essa estrutura básica e saber que é preciso declarar todas as varáveis que vai usar logo de início e que depois deve colocar as demais sentenças do programa.

Por exemplo, o seguinte programa está errado:

```
#include <stdio.h>

int main()
{
  int x;

  scanf("%d", &x);

  int y;

  scanf("%d", &y);

  return 0;
}
```

O problema é que as variáveis não foram todas declaradas no início do bloco.

Laços

Com o que sabemos até agora é possível escrever programas que manipulam números inteiros, mas cada sentença em nossos programas será executada apenas uma única vez. Suponha entretanto que queiramos escrever um programa que imprima na tela o fatorial de um número digitado pelo usuário. Como podemos fazer?

Para escrever um programa desses precisamos de laços. Um *laço* (*loop*) é uma parte do código que é executada repetidas vezes, uma após a outra. Para fazer laços, precisamos de comandos de repetição. O primeiro que vamos aprender é o comando `while`. O formato de um comando `while` é o seguinte:

```
while (expr) A;

B;  /* Primeira sentença após o laço */
```

Aqui, `expr` é qualquer expressão que resulte num número inteiro e `A` é qualquer sentença.

O comando `while` repete a sentença `A` enquanto a expressão `expr` for verdadeira. O que significa dizer que `expr` é verdadeira? Em C, uma expressão é *verdadeira* se ela resulta num número inteiro **diferente de ZERO**. Caso contrário, a expressão é *falsa*.

Cada execução de `A` é uma *iteração* do laço. Em detalhes, o comando `while` funciona assim:

1. Verifica se a expressão `expr` é verdadeira. Se ela for falsa, então pula para `B`, a primeira sentença após o laço.
2. Se a expressão for verdadeira, executa a sentença `A` e pula para o passo 1.

Vejamos um exemplo. O trecho abaixo imprime na tela a mesma frase para sempre:

```
while (1) printf("Hello World!\n");
```

Temos então um *laço infinito*. Já o laço a seguir nunca executa `printf`, e portanto não imprime nada na tela:

```
while (0)
  printf("Hello World!\n");
```

Note que colocamos `printf` na linha seguinte. Isso é irrelevante em C: o espaçamento entre comandos e as quebras de linha não importam.

Como podemos executar mais de uma sentença a cada iteração do laço? Em C, sentenças podem ser agrupadas em *blocos*, que se comportam como uma única sentença para o compilador. Blocos são

definidos usando-se chaves:

```
{
  printf("Hello World!\n");
  printf("Olá Mundo!\n");
}
```

Tudo que escrevemos entre `{` e `}` comporta-se como uma única sentença. Então podemos escrever:

```
while (1)
{
  printf("Hello World!\n");
  printf("Olá Mundo!\n");
}
```

Como quebras de linha e espaços não importam para o compilador, a maioria dos programadores prefere escrever:

```
while (1) {
  printf("Hello World!\n");
  printf("Olá Mundo!\n");
}
```

o que é exatamente o mesmo.

Agora podemos escrever um programa um pouco mais interessante. O programa a seguir faz o seguinte: o usuário digita uma seqüência de números terminada em zero e o programa imprime na tela a soma da seqüência (**código**):

```
#include <stdio.h>

int main()
{
  int x = 1;
  int soma = 0;

  while (x) {
    scanf("%d", &x);
    soma = soma + x;
  }

  printf("Soma = %d\n", soma);

  return 0;
}
```

Note que, na iteração que sucede aquela em que o usuário digita 0, o laço é quebrado, pois a expressão `x` vale 0, sendo portanto falsa. Por que `x` começa valendo 1?

Operadores de comparação

Para um exemplo um pouco mais elaborado, precisamos saber comparar números inteiros. Isso podemos fazer utilizando os *operadores de comparação*. São eles: `<`, `>`, `<=`, `>=`, `==` e `!=`:

```
int a = 5, b = 6;
int x;

x = a < b;
x = a > b;
x = a <= b;
x = a >= b;
x = a == b;
x = a != b;
```

As operações acima são, respectivamente: `a` é *menor* que `b`, *maior*, *menor ou igual*, *maior ou igual*, *igual* e *diferente*. Note que o operador de igualdade é `==` e não `=`, que significa atribuição.

O resultado de uma operação de comparação é ZERO, se a comparação for falsa, ou um número diferente de ZERO, se a comparação for verdadeira. Por exemplo:

```
int a = 5, b = 6;
int x, y;

x = a < b;
y = a >= b;
```

Ao final, temos que `x` contém um número diferente de ZERO e `y` contém ZERO.

Vejamos como resolver então o seguinte problema: dados números inteiros n e k, com $k \geq 0$, determinar n^k. O seguinte programa resolve esse problema (código):

```
#include <stdio.h>

int main()
{
  int n, k;
  int prod = 1;

  scanf("%d", &n);
  scanf("%d", &k);

  while (k > 0) {
    prod = n * prod;
    k = k - 1;
  }

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

  return 0;
}
```

Eficiência do código

Quantos produtos o código acima faz para calcular n^k? Você consegue imaginar um programa mais eficiente, ou seja, que faça menos multiplicações?

Esse tipo de pergunta, sobre a eficiência de um programa, é fundamental em programação. Um mesmo problema pode ser resolvido de diversas maneiras, algumas mais e outras menos eficientes. Muitas vezes, um algoritmo mais rápido é a diferença entre resolver um problema em segundos ou em séculos!

Pense a respeito: você consegue calcular $n^{1000000}$ usando menos de 100 multiplicações?

Vejamos ainda mais um exemplo: dado um número inteiro $n \geq 0$, calcular $n!$. O seguinte programa resolve esse problema (código):

```
#include <stdio.h>

int main()
{
  int n;
  int fatorial = 1;

  scanf("%d", &n);

  while (n > 0) {
    fatorial = fatorial * n;
    n = n - 1;
  }

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

  return 0;
}
```

Comentários e leiaute

Em alguns dos trechos de código acima você deve ter notado pedaços de texto como no seguinte:

```
int n = 2;  /* Um número inteiro */

/* Elevamos n ao quadrado */
n = n * n;
```

Todo texto colocado entre `/*` e `*/` é ignorado pelo compilador. Esses trechos de texto são chamados de *comentários*, e são usados para ajudar os programadores que precisam ler e entender o programa. É importante comentar o seu código, mas é bom não exagerar. Por exemplo, o comentário acima `/* Elevamos n ao quadrado */` é bem inútil.

Também foi mencionado que quebras de linha e espaçamento são irrelevantes para o compilador. De fato são, mas são muito importantes para tornar seu programa fácil e agradável de ler! Toda língua tem suas convenções: em português, sempre colocamos um espaço após uma vírgula ou um ponto. Começamos um parágrafo com um espaço, etc. Numa linguagem de programação, não seria diferente.

O professor Paulo Feofiloff escreveu uma página com diretrizes de estilo para programas em C. Dê uma olhada!