Instituto Federal de Educação, Ciência e Tecnologia da Paraíba
Campus Campina Grande
Coordenação de Informática
Bacharelado em Engenharia de Computação

Estruturas de Dados e Algoritmos em C

Prof. Ruan Delgado Gomes, D.Sc.

2018

1 Introdução à Linguagem C

A linguagem C foi criada na década de 70, mas continua até hoje sendo muito utilizada devido à sua versatilidade e poder. A linguagem C, bem como a sua variante orientada a objetos (C++), é utilizada para o desenvolvimento de uma grande gama de aplicações, que vão desde softwares para controlar dispositivos embarcados (televisão, câmera digital, forno microondas etc), desenvolvimento de planilhas eletrônicas e processadores de texto, até sistemas operacionais.

C é uma linguagem de alto nível, embora apresente características de baixo nível. Dessa forma, é necessário a utilização de um compilador, para que programas escritos em C sejam convertidos para linguagem de máquina. No sistema operacional Windows o compilador usualmente utilizado é o MinGW. No Linux usualmente se usa o *GNU Compiler Collection* (GCC), e nos sistemas operacionais BSD e Mac OS usualmente é utilizado o CLANG/LLVM.

Antes de iniciarmos a demonstração de programas escritos em C, é importante comentar que a linguagem C é *case-sensitive*, ou seja, ela faz diferenciação entre letras maiúsculas e minúsculas. Dessa forma, os identificadores "A" e "a", por exemplo, são diferentes.

1.1 Primeiro Programa em C

```
1 #include <stdio.h>
2 int main() {
3     printf("Oi Mundo!!");
4     return 0;
5 }
```

Esse programa é muito simples e tem como resultado a saída na tela da string "Alô Mundo!!". Vamos agora analisar linha por linha o significado dos comandos presentes nesse programa em C.

Na primeira linha temos ***#include <stdio.h>***, que serve para incluir a biblioteca *stdio.h*. Bibliotecas implementam um conjunto de funções que podem ser usadas pelos programas C. A biblioteca *stdio.h* implementa as funções de entrada e saída de dados padrão. Para usar funções de uma biblioteca é necessário incluí-la no seu programa a partir de um comando *#include*. Existem diversas bibliotecas de funções disponíveis em C, como a biblioteca de funções matemáticas (*math.h*), por exemplo.

Diferença em relação a python: em python não é necessário importar pacotes para usar as funções *print()* e *input()*. Em C é obrigatório importar *stdio.h* para usar as funções de entrada padrão (*scanf()*) e saída padrão (*printf()*).

Na segunda linha temos o início da função *main*, que é a função principal do programa escrito em C. O programa será executado seguindo a definição dos comandos escritos dentro dela. Ou seja, entre as chaves de abertura e fechamento ({ e }) que delimitam a função

main deve-se colocar os comandos necessários para a resolução do problema. É importante notar que dentro da função *main* pode haver a chamada de outras funções, como por exemplo, a utilização da função *printf* para escrita de informações na saída padrão.

A partir da terceira linha temos as definições dos comandos que formam o programa descrito. Na terceira linha temos o primeiro comando do programa em questão. O comando é uma chamada à função *printf*, que coloca a string "Oi Mundo!!" na saída padrão (terminal de execução do programa).

Diferença em relação a python: em python o final de uma linha de comando é definido pelo caractere que representa uma nova linha (inserido ao clicar enter). Em C é necessário informar explicitamente o final de uma linha de comando, usando ponto e vírgual (**;**). Note que não é necessário colocar **;** após abertura e fechamento de chaves e também não é necessário incluir **;** após o *#include*.

Na quarta linha temos o comando *return 0*. O comando *return* força a saída da função e retorna algum valor (exceto se a função não tiver retorno [*void*]). Ou seja, quando um comando *return* é executado a função é encerrada, e o valor colocado após o *return* é retornado ("jogado para fora da função"). Por enquanto, tenham apenas em mente que o *return* marca o fim da função main e, consequentemente, o fim do programa.

Na quinta linha temos o fechamento da chave, que delimita o final do escopo da função *main*.

Diferença em relação a python: em python o escopo é delimitado usando identação. Em C o escopo é delimitado usando abertura e fechamento de chaves. Apesar de não ser obrigatório o uso de identação, uma vez que a delimitação do escopo é feita usando as chaves, é uma boa prática de programação (eu diria uma prática obrigatória) usar a identação, de modo a tornar o código mais legível e fácil de entender.

1.2 Variáveis

Uma variável representa um espaço na memória e serve para armazenar dados de um determinado tipo. Na linguagem C, todas as variáveis devem ser explicitamente declaradas antes de serem usadas. Para realizar uma declaração de uma variável na linguagem C deve-se informar o tipo da variável seguido do seu identificador. Existem algumas regras para a formação dos identificadores, por exemplo:

- não pode iniciar com número
- não pode conter espaços nem caracteres especiais (com exceção do *underline*)
- não pode ser uma palavra reservada da linguagem (ex: **int**, **include**, **void**, **return**, **if**, **while** etc).

Uma variável de determinado tipo apenas pode armazenar valores desse mesmo tipo, caso contrário haverá um erro de semântica no programa. Por exemplo, uma variável do tipo inteiro só pode armazenar valores numéricos inteiros no seu espaço de memória.

Diferença em relação a python: python usa tipagem dinâmica; ou seja, as variáveis podem armazenar valores de diferentes tipos no decorrer do programa. C usa tipagem estática; ou seja, uma variável que é declarada como sendo de um tipo, permanece sendo do mesmo tipo dentro do escopo a que ela pertence. Além disso, em C é necessário declarar explicitamente o tipo da variável antes de começar a usa-la. Considere os exemplos de código a seguir para somar dois números inteiros, em python e em C, respectivamente.

```python
a = 10
b = 20
soma = a+b
print (soma)
```

```c
#include <stdio.h>

int main() {
    int a = 10;
    int b = 20;
    int soma = a + b;
    printf("%d",soma);
    return 0;
}
```

Em python o tipo de uma variável é definido de acordo com o tipo do valor que é atribuído a ela. Em C deve-se informar o tipo antes da definição do identificador da variável (**int** no exemplo). Após a declaração da variável com um determinado tipo, caso ocorra uma atribuição posteriormente de um valor de tipo diferente a essa variável, isso causará um erro em tempo de compilação. Em python não ocorre erro se um valor de tipo diferente for atribuído às variáveis "a" ou "b" posteriormente.

O exemplo de código a seguir mostra exemplos de variáveis sendo declaradas. Note que é opcional atribuir um valor inicial à variável após a declaração (no exemplo, apenas a variável **b** recebe um valor inicial junto da declaração). Também é possível declarar variáveis de um mesmo tipo, uma ao lado da outra, com os identificadores separados por vírgula (como foi o caso das variáveis **n1** e **n2**).

```c
#include <stdio.h>

int main() {
    int a;
    float b = 1.5;
    char c;
    int n1,n2;
    return 0;
```

```
9 }
```

Um erro comumente cometido por iniciantes na linguagem C é a utilização de variáveis não inicializadas. Quando se declara uma variável, não se sabe o valor que está armazenado nela, a não ser que haja uma inicialização durante a declaração. Quando não existe uma inicialização, as variáveis armazenam no momento da declaração um valor que estava previamente na memória e que chamamos de "lixo". Por exemplo, considere o seguinte trecho de código:

```
1 ...
2 int a, c;
3 c = 10 + a;
4 ...
```

O valor de **a** não foi inicializado, portanto é um erro utilizarmos ele na atribuição à variável **c**. Como não inicializamos **a** no momento da declaração, ele pode conter qualquer valor. Portanto, é importante inicializar uma variável antes de utilizá-la, com o objetivo de evitar erros no programa. A inicialização poderia ser feita da seguinte forma:

```
1 ...
2 int a = 5, c;
3 c = 10 + a;
4 ...
```

Nesse caso é garantido que será atribuído o valor 15 (10 + a) à variável **c**, já que a variável **a** foi inicializada com o valor 5.

1.2.1 Tipos Primitivos

Os tipos primitivos de uma linguagem de programação são os tipos básicos que fazem parte da definição da linguagem. À princípio apenas pode-se declarar variáveis dos tipos primitivos. No entanto, é possível criar novos tipos compostos, a partir da combinação de variáveis de tipos primitivos ou pela combinação de outras variáveis de tipos compostos. Mais na frente será mostrado como construir novos tipos compostos em C.

A Tabela 1 mostra os tipos disponíveis em C para valores inteiros, o tamanho em memória das variáveis de cada tipo, o intervalo de valores admitidos para cada tipo, e também o código de formatação para cada um.

Para armazenamento de valores inteiros, a linguagem C oferece cinco tipos primitivos: ***char***, ***short***, ***int***, ***long*** e ***long long***. O que muda de um tipo para o outro é a quantidade de bytes que uma variável irá ocupar na memória e, consequentemente, o intervalo de valores que é possível armazenar na variável. Por exemplo, variáveis do tipo short ocupam dois bytes (16 bits) na memória. Portanto, é possível armazenar $2^{16} = 65536$ valores diferentes na memória. Caso sejam admitidos valores positivos e negativos, metade dos possíveis valores são números negativos e a outra metade números positivos (além do zero). Portanto, variáveis do tipo ***short*** podem armazenar valores inteiros entre -32768 e 32767.

Tabela 1: Tipos primitivos para valores inteiros.

Tipo	Tamanho em Memória (bytes)	Intervalo de Valores	Código de Formatação
char	1	-128 a 127	%hhd (como número) ou %c (como caractere)
unsigned char	1	0 a 255	%hhd (como número)
short	2	-32768 a 32767	%hd
unsigned short	2	0 a 65535	%hu
int	4	-2147483648 a 2147483647	%d
unsigned int	4	0 a 4294967296	%ud
long	8	-9223372036854775807 a 9223372036854775808	%ld
unsigned long	8	0 a 1,844674407370955 $\times 10^{19}$	%lu
long long	8	-9223372036854775807 a 9223372036854775808	%lld
unsigned long long	8	0 a 1,844674407370955 $\times 10^{19}$	%llu

É importante notar que a quantidade de bytes associada a cada tipo depende da plataforma. Essa tabela considera um computador com processador Intel de 64 bits e o sistema operacional Mac OS 10.13. Por exemplo, em microcontroladores de 8 bits da família PIC o tipo ***int*** ocupa apenas dois bytes (ver página 149 do manual do compilador [1]). Portanto, caso sua aplicação seja sensível a essas diferenças, é importante verificar a definição das variáveis primitivas utilizada para a plataforma para qual a aplicação está sendo desenvolvida. O exemplo de código a seguir mostra como saber a quantidade de bytes ocupada em memória por um tipo de dados em C (usando a função ***sizeof()***), o que pode ser útil para desenvolver programas genéricos e que se adaptem à plataforma sendo utilizada, em alguns casos. Compile e execute esse código em sua máquina para testar.

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Tamanho de um char: %lu byte\n", sizeof(char));
5      printf("Tamanho de um short: %lu bytes\n", sizeof(short));
6      printf("Tamanho de um int: %lu bytes\n", sizeof(int));
7      printf("Tamanho de um long: %lu bytes\n", sizeof(long));
8      printf("Tamanho de um long long: %lu bytes\n", sizeof(long long));
9      return 0;
10 }
```

Também é importante notar que às vezes dois tipos diferentes podem ser equivalentes. Por exemplo, para a plataforma considerada para gerar a Tabela 1 as variáveis dos tipos ***long*** e ***long long*** são equivalentes (ocupam 8 bytes). No Sistema Operacional Windows, uma variável do tipo ***int*** é equivalente a uma variável do tipo ***long***, ambas ocupando 4

bytes, enquanto que uma variável do tipo ***long long*** ocupa 8 bytes [2].

Como mostrado na Tabela 1, é possível definir variáveis que só consideram valores positivos, usando a palavra chave ***unsigned*** antes do tipo. Nesse caso, o intervalo de valores é dobrado para o lado positivo e os números negativos não são representados. Por exemplo, uma variável do tipo ***unsigned short*** pode assumir valores entre 0 e 65535.

Finalmente, os códigos de formatação são úteis para a formatação dos dados de entrada e saída, como será melhor explicado na Seção 2. Como exercício, observe os exemplos de código C mostrados até o momento e identifique como os códigos de formatação foram utilizados em conjunto com a função *printf()*. No exemplo de código que mostra como utilizar a função ***sizeof()***, é utilizado o código de formatação "%lu" para representar o valor retornado pela função ***sizeof()***. Isso quer dizer que a resposta fornecida por essa função é do tipo ***unsigned long*** (ver Tabela 1).

Ao implementar um algoritmo para resolver algum problema deve-se ficar atento às restrições sobre os valores que podem ser atribuídos às variáveis. Por exemplo, o valor máximo que uma variável do tipo ***int*** pode armazenar é igual a $2^{31} - 1 = 2147483647$. Tendo isso em mente, analise o trecho de código a seguir.

```
1 #include <stdio.h>
2
3 int main() {
4     long a = 3000000000;
5     int b = 10;
6     int soma = a + b;
7     printf("%d", soma);
8     return 0;
9 }
```

Ao executar o programa gerado a partir desse código, é colocado na saída o valor -1294967286. Isso ocorre pois os valores atribuídos às variáveis **a** e **soma** são maiores que o valor máximo permitido para variáveis do tipo ***int***. Isso é chamado de *overflow*. Esse tipo de erro é muito perigoso, uma vez que o código compila e executa, mas o resultado obtido é diferente do esperado. Além disso, o código pode funcionar para vários casos de teste (que não chegam a provocar *overflow*), o que pode dar a falsa impressão de que o código está correto. Troque os tipos das variáveis **a** e **soma** por ***long long*** nesse exemplo de código e execute novamente para ver o resultado obtido. Lembre de modificar o código de formatação na função *printf()*.

Diferença em relação a python: no python 2 também existe os tipos ***int*** e ***long*** (no python 3 essa diferença desaparece), mas não é necessário explicitar se uma determinada variável é de um desses tipos. Dependendo do valor atribuído à variável, o tipo dela é alterado pelo interpretador. Caso os valores fiquem abaixo do limite definido para variáveis do tipo ***int*** (esse valor é específico da plataforma em uso), as operações com os valores inteiros são realizadas utilizando as instruções para operações aritméticas disponibilizadas pelo processador. Para valores que excedam o limite máximo para variáveis ***int***,

o interpretador precisa tratar as operações via software. À princípio, não existe limite para o tamanho dos valores inteiros usados em python, no entanto para valores muito grandes o tempo de processamento pode ser alto, devido à necessidade de realizar processamento em software para as operações aritméticas [3].

Sobre o tipo *char*

Variáveis o tipo ***char*** são mais comumente utilizadas para armazenar caracteres. Os caracteres em C são representados utilizando aspas simples (’), por exemplo: ‘a’, ‘2’,‘#’. Existe uma tabela chamada “Tabela ASCII” que faz uma associação de todos os caracteres dessa tabela com um valor numérico correspondente. Dessa forma, uma variável do tipo ***char*** pode armazenar o valor inteiro correspondente a um determinado caractere, seguindo a tabela. Por exemplo, o caracter ‘A’ é representado pelo valor 65 na tabela ASCII. Assim, uma variável do tipo ***char*** que contenha o caractere ‘A’, na realidade armazena em memória o valor inteiro 65. O exemplo de código a seguir mostra como é possível armazenar um caractere a uma variável do tipo ***char*** e como utilizar o *printf()* para colocar na saída o caractere armazenado pela variável, bem como o valor numérico correspondente. Como demonstrado na Tabela 1, o padrão de codificação para a variável do tipo ***char*** no modo numérico é “%hhd”, enquanto que no modo caractere é “%c”.

```
1  #include <stdio.h>
2
3  int main() {
4      char c1 = 'C';
5      char c2 = 'G';
6      printf ("IFPB - %c%c\n", c1, c2);
7      printf ("Valores inteiros: %hhd %hhd\n", c1,c2);
8
9      printf ("%d\n", '#');
10
11     return 0;
12 }
```

Ao ser executado, o programa gerado por esse código coloca as seguintes infomações na saída:

```
1          IFPB - CG
2          Valores inteiros: 67 71
```

Na primeira linha os conteúdos das variáveis *c1* e *c2* são combinados para compor a palavra “IFPB - CG”, enquanto que na segunda linha os valores inteiros correspondentes aos caracteres ‘C’ e ‘G’ na tabela ASCII são mostrados (67 e 71).

É importante notar que devido à ampliação do conjunto de caracteres que podem ser representados, o tipo ***char*** não é mais capaz de representar todos os caracteres existentes, mas continua sendo usado para representar o alfabeto ASCII [4]. No entanto, grande

parte dos caracteres que estamos habituados a utilizar nas línguas portuguesa e inglesa podem ser representados pela tabela ASCII. Para se informar mais sobre a representação de caracteres, pesquise sobre Tabela *Unicode*.

A linguagem C oferece três tipos básicos para representação de números de ponto flutuante: ***float***, ***double*** e ***long double***. A diferença entre elas também é a quantidade de bytes que uma variável ocupa e, consequentemente, o intervalo de valores que pode ser armazenado, bem como a precisão. A Tabela 2 mostra os tipos disponíveis em C para valores reais, o tamanho em memória das variáveis de cada tipo, o intervalo de valores admitidos para cada tipo, e também o código de formatação para cada um. Essa tabela também foi gerada considerando a mesma plataforma usada para gerar os dados da Tabela 1. Portanto, em outras plataformas as definições com relação à quantidade de bytes das variáveis de cada tipo podem variar.

Tabela 2: Tipos primitivos para valores reais.

Tipo	Tamanho em Memória (bytes)	Intervalo de Valores	Código de Formatação
float	4	1.175494×10^{-38} a 3.402823×10^{38}	%f (notação normal) ou %e (notação científica)
double	8	$2.225074 \times 10^{-308}$ a 1.797693×10^{308}	%lf (notação normal) ou %le (notação científica)
long double	16	$3.362103 \times 10^{-4932}$ a $1.189731 \times 10^{4932}$	%Lf (notação normal) ou %Le (notação científica)

De forma similar como discutido para os números inteiros, antes de decidir qual tipo de variável utilizar para valores reais, deve-se analisar as características dos valores que precisam ser armazenados e processados pela sua aplicação. Em geral, quando uso de memória não é uma forte restrição, pode-se utilizar ***double*** para conseguir uma boa precisão. No entanto, caso a aplicação possua requisitos críticos de memória (como pode ocorrer em muitos sistemas embarcados), deve-se avaliar se o uso de variáveis ***float*** atendem aos requisitos de precisão dos cálculos que serão realizados, uma vez que variáveis ***float*** ocupam a metade do espaço em memória ocupado por uma variável do tipo ***double***. Outra alternativa é usar variáveis do tipo ***float*** para armazenar os dados e utilizar variáveis do tipo ***double*** para realizar os cálculos. Nesse último cenário pode-se minimizar as perdas de precisão durante a realização de cálculos. Isso é especialmente útil quando se realiza uma grande quantidade de cálculos sucessivos utilizando valores reais como operandos. O tempo de processamento para variáveis com maior precisão também é maior, mas nem sempre a diferença será significante. O tipo ***long double*** é o que apresenta maior precisão e deve ser usado em aplicações que requeiram uma precisão muito grande. Na grande maioria dos casos o uso de variáveis ***double*** resolve o problema.

Para entender melhor as diferenças, deve-se levar em consideração que variáveis do tipo

float possuem aproximadamente 7 dígitos de precisão, variáveis do tipo ***double*** possuem aproximadamente 16 dígitos de precisão e variáveis do tipo ***long double*** possuem aproximadamente 34 dígitos de precisão. Em resumo, caso o sistema não possua requisitos críticos de memória ou precisão, usar ***double*** é a melhor opção. Caso o sistema possua requisitos críticos de memória, deve-se avaliar a possibilidade de utilizar variáveis do tipo ***float***, ao menos para armazenamento. Caso a aplicação apresente requisitos críticos de precisão, deve-se avaliar se é necessário utilizar ***long double***, ao menos durante o processamento dos dados (e manter ***double*** para armazenamento).

String e booleano não possuem representação direta na linguagem C por meio de tipos primitivos. Em C, uma string é representada por um conjunto (array) de caracteres e um booleano pode ser representado por uma variável inteira. Valores inteiros diferentes de 0 equivalem ao valor booleano "verdadeiro" e o valor 0 equivale ao valor booleano "falso". Arrays e strings (arrays de ***char***) serão estudados em mais detalhes na Seção **??**.

2 Entrada e Saída

Nessa seção serão apresentados mais detalhes sobre a função de saída padrão (*printf()*) e será apresentada a função de entrada padrão (*scanf()*).

A função *printf()* permite que um conjunto de valores (constantes, variáveis ou resultados de expressões) sejam exibidos de acordo com um determinado formato. O formato geral da função *printf()* é:

```
1 printf(<formato>, <lista de valores>);
```

O primeiro parâmetro da função *printf()* (o formato) contém a cadeia de caracteres que será exibida na tela. Após o primeiro parâmetro, deve-se colocar uma lista de 0 ou mais valores (separados por vírgula). No parâmetro "formato" são definidos os locais dentro da cadeia de caracteres onde os valores que fazem parte da lista de valores passados como parâmetro (constantes/variáveis/resultados de expressões) serão apresentados. Para definir os locais onde os valores aparecem dentro da cadeia de caracteres, deve-se utilizar os códigos de formatação das variáveis, de acordo com os seus tipos (os códigos de formatação das variáveis primitivas foram mostrados nas tabelas 1 e 2). Considere o exemplo a seguir:

```
1 #include <stdio.h>
2
3 int main() {
4         int idade = 35;
5         float altura = 1.8;
6         printf("Tenho %d anos e %f metros de altura", idade, altura);
7         return 0;
8 }
```

Neste exemplo, são colocados dois valores na saída (um do tipo ***int*** e um do tipo ***float***). No primeiro parâmetro da função *printf()* é informado o formato da saída, que define a

frase que será colocada na saída e os locais onde irão aparecer os dois valores (das variáveis *idade* e *altura*), que são informados nos parâmetros seguintes. Ao executar o programa gerado a partir desse código, é obtida a seguinte saída:

```
1 Tenho 35 anos e 1.800000 de altura
```

No local onde foi colocado o "%d" aparece o valor que estava dentro da variável *idade* e no local onde foi colocado o "%f" aparece o valor que estava dentro da variável *altura*. Para limitar a quantidade de casas decimais dos valores reais na saída, pode-se escrever o código de formatação na forma: "%.<qtd-digitos>f", em que <qtd-digitos> deve ser substituído pela quantidade de dígitos que devem aparecer, como mostrado no exemplo de código a seguir, em que o valor da altura aparece na saída com apenas duas casas decimais. No exemplo a seguir também foi colocado um caractere '\n' ao final da string que define o formato da saída. Um caractere '\n' serve para pular uma linha.

```
1 #include <stdio.h>
2
3 int main() {
4         int idade = 35;
5         float altura = 1.8;
6         printf("Tenho %d anos e %.2f metros de altura\n", idade, altura);
7         return 0;
8 }
```

A função de entrada padrão de C funciona de forma semelhante à função de saída. No primeiro parâmetro deve-se informar o formato da entrada, indicando os tipos dos valores que serão lidos (por meio dos códigos de formatação). Após o primeiro parâmetro, deve-se colocar os endereços das variáveis que receberão os dados da entrada. O conceito de "endereço de variável" será melhor explorado na Seção **??** durante o estudo sobre ponteiros. Neste momento, apenas lembre que o endereço de uma variável é obtido colocando um & antes do identificador da variável. Considere o exemplo a seguir:

```
1 #include <stdio.h>
2
3 int main() {
4         int idade;
5         float altura;
6
7         scanf("%d", &idade);
8         scanf("%f", &altura);
9
10        printf("Tenho %d anos e %.2f metros de altura", idade, altura);
11        return 0;
12 }
```

Nesse exemplo, os valores de *idade* e *altura* são lidos da entrada usando a função *scanf()*. Para a variável *idade* foi usado o código de formatação "%d" (variável do tipo ***int***) e para

a variável *altura* foi usado o código de formatação "%f" (variável do tipo ***float***). Note que o segundo parâmetro da função *scanf()* é o endereço das variáveis que receberão os valores lidos da entrada (&idade e &altura). É importante ter atenção a esse detalhe, pois caso não seja colocado o & antes do nome da variável, o código compila normalmente (provavelmente será mostrado apenas um *warning*), mas ocorre um erro durante a execução.

O código a seguir mostra um exemplo de como declarar uma string em C, como ler da entrada a string e como colocar na saída o conteúdo da string. Existe um código de formatação para string, que é o "%s".

```
1 #include <stdio.h>
2
3 int main() {
4         char nome[100];
5         printf ("Digite seu nome: ");
6         scanf("%s", nome);
7         printf("Nome: %s", nome);
8         return 0;
9 }
```

Nesse exemplo a variável *nome* é um array (o conceito equivalente em python seria "lista") de 100 posições do tipo ***char***. Essa variável pode armazenar strings de até 99 caracteres, uma vez que um caractere deve ser reservado para indicar o final da string (esses detalhes serão melhor explorados em uma seção específica). Por enquanto, é apenas importante aprender a declarar strings como um array de ***char***, ler da entrada strings e colocar na saída o conteúdo de uma string. Um diferença importante da string com relação às variáveis primitivas é que na função *scanf()* não é necessário usar o & antes do nome da variável. O motivo disso também será entendido na seção que descreve arrays e strings em mais detalhes. Nesse momento, apenas lembre que não é necessário colocar o & no *scanf()* para ler strings, mas é necessário colocar o & para ler variáveis dos tipos primitivos.

3 Operadores Aritméticos

Não há muita diferença entre os operadores aritméticos básicos utilizados em C e em Python. Em C existem os seguintes operadores aritméticos: soma (+), subtração (-), multiplicação (*), divisão (/) e resto da divisão (%). Os operadores de multiplicação, divisão e resto de divisão possuem maior precedência em relação aos operadores de soma e subtração. Para operadores que possuem a mesma precedência, a avaliação da expressão ocorre da esquerda para a direita. Também é possível utilizar parênteses para forçar precedência entre os operadores.

Em C também são definidos operadores de incremento (++), de decremento (−−) e operadores de atribuição. O exemplo de código a seguir exemplifica o uso desses operadores.

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

```
2
3  int main() {
4          int n1 = 0;
5          int n2 = 2;
6
7          n1++; //equivale a n1 = n1 + 1;
8          n2--; //equivale a n2 = n2 - 1;
9
10         n1 += n2; //equivale a n1 = n1 + n2;
11         n1 /= n2; //equivale a n1 = n1/n2;
12         n2 *= 3; //equivale a n2 = n2*3;
13         n2 %= 2; //equivale a n2 = n2%2;
14
15         printf("N1: %d\nN2: %d\n", n1,n2);
16         return 0;
17 }
```

Ao executar o programa gerado por esse código, obtemos a saída mostrada a seguir. Como exercício, simule o código no papel para verificar como esse resultado foi obtido.

```
1 N1: 2
2 N2: 1
```

Os operadores de incremento e decremento também podem ser usados antes da variável (ex: $--$n1). Quando utilizados de forma isolada (é o caso do exemplo de código mostrado anteriormente), não faz diferença colocar o operador antes ou depois da variável, mas quando utilizado dentro de uma expressão maior, a posição do operador de incremento influencia no resultado final obtido. Mais especificamente, quando colocado antes da variável, primeiro é realizado o incremento/decremento e só após isso o valor da variável é utilizado na expressão. Quando o operador é colocado após a variável, o valor atual da variável é utilizado na expressão e só depois é realizado o incremento/decremento. Para entender melhor, considere o exemplo de código a seguir:

```
1  #include <stdio.h>
2
3  int main() {
4          int n = 2;
5          int a,b;
6
7
8          a = n++ - 1;
9          b = ++n - 1;
10
11         printf("a: %d\nb: %d\n", a,b);
12         return 0;
13 }
```

O programa gerado por esse código coloca na saída o seguinte:

```
1 a: 1
2 b: 3
```

Na expressão da linha 9 o incremento à variável n é feito após a subtração. Logo, ao final dessa expressão a variável a recebe o valor 1 (2 - 1) e depois a variável n passa a armazenar o valor 3. Na expressão da linha 10 o incremento à variável n ocorre antes. Logo, primeiro n passa a armazenar o valor 4 e depois o resto da expressão é analisada, o que culmina na atribuição do valor 3 à variável b.

3.1 Exercícios no URI

Resolver os seguintes problemas no URI, usando a linguagem de programação C:

1. 1009 – Salário com Bônus – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1009`
2. 1015 – Distância Entre Dois Pontos – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1015`
3. 1017 – Gasto de Combustível – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1017`
4. 1019 – Conversão de Tempo – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1019`
5. 1018 – Cédulas – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1018`

4 Operadores Relacionais e Lógicos

Os operadores relacionais são usados para comparar dois valores. A linguagem C oferece os seguintes operadores relacionais (iguais aos usados em python): < (menor que), > (maior que), <= (menor ou igual a), >= (maior ou igual a), == (igual a), != (diferente de).

Os operadores relacionais mostrados servem para comparar dois valores. O resultado produzido por um operador relacional é 0 ou 1. Como em C não existe o tipo booleano, são usados inteiros para representar valores lógicos. O valor 0 corresponde a *falso* e qualquer valor diferente de zero corresponde a *verdadeiro*.

Os operadores lógicos servem para combinar expressões relacionais e valores booleanos. A Tabela 3 mostra os operadores lógicos existentes na linguagem C, bem como o operador equivalente em python, para fins de comparação.

É importante notar que os operadores lógicos *e* e *ou* são representados por dois caracteres (&& e ||). Em C existem operadores para manipular valores em nível de bit, chamados

Tabela 3: Operadores lógicos.

Operador	Em C	Em python
não	!	**not**
e	&&	**and**
ou	||	**or**

de operadores *bitwise*. No caso dos operadores *bitwise* é usado apenas um caractere para o *e* e para o *ou* (& e |). O uso de operadores *bitwise* está fora do escopo desta seção, mas é importante conhecer a diferença de sintaxe entre eles e os operadores lógicos. Alguns exemplos de uso dos operadores lógicos e relacionais serão mostrados nas próximas seções, que apresentam as estruturas de decisão e de repetição.

Diferença em relação a python: diferente de C, em python as expressões do tipo $a < b < c$ possuem a interpretação que é convencional na matemática [5]. Em C, não é possível usar um mesmo operando para dois operadores relacionais (na expressão $a < b < c$, o b é comparado tanto com o a como com o c). Para escrever uma expressão como essa em C, seria necessário combinar dois operadores relacionais, usando o &&, da seguinte forma: $a < b$ && $b < c$.

5 Estruturas de Decisão

Uma estrutura de decisão permite decidir por executar ou não certo conjunto de comandos, de acordo com uma determinada condição. Em C existem duas formas de representar um processo de decisão, usando a estrutura ***if − else*** ou a estrutura ***switch − case***.

5.1 Estrutura de decisão *if-else*

A palavra chave ***if*** é utilizada para codificar uma tomada de decisão em C com base em uma condição (expressão booleana). A seguir é exemplificado o uso do ***if*** em um algoritmo que verifica se um número inteiro lido da entrada é positivo.

```
1  #include <stdio.h>
2
3  int main() {
4          int n1;
5          scanf("%d",&n1);
6          if(n1 > 0) {
7                  printf("valor positivo");
8          }
9          return 0;
10 }
```

Após a palavra chave ***if*** deve-se colocar a condição entre parênteses. O escopo da estrutura de decisão pode ser delimitado usando abertura e fechamento de chaves. Isso quer dizer que todos os comandos que ficarem dentro do escopo da estrutura de decisão ***if*** apenas serão executados se a condição for verdadeira. O uso e abertura e fechamento de chaves pode ser dispensado quando existe apenas um comando dentro do escopo da decisão. Se houver dois ou mais comandos dentro do escopo é obrigatório o uso de abertura e fechamento de chaves. Lembrando que diferente de python, em C o uso de identação não implica em definição de escopo.

A palavra chave ***else*** pode ser utilizada para executar um conjunto de comandos alternativos aos definidos dentro do ***if***. Ou seja, caso a condição definida no ***if*** seja falsa, os comandos colocados dentro do escopo do ***else*** serão executados. No exemplo a seguir, é mostrado um código que verifica e informa ao usuário se um determinado número inteiro é positivo ou não, usando ***if-else***.

```
1  #include <stdio.h>
2
3  int main() {
4          int n1;
5          scanf("%d",&n1);
6          if(n1 > 0) {
7                  printf("valor positivo");
8          }
9          else {
10                 printf("valor negativo ou igual a zero");
11         }
12         return 0;
13 }
```

Ao usar o ***if-else*** como no exemplo mostrado, pode-se definir duas opções de fluxo de processamento no código. Caso a condição definida no ***if*** seja verdadeira, os comandos colocados dentro do seu escopo serão executados, caso contrário os comandos colocados dentro do escopo do ***else*** serão executados. Caso seja necessário definir mais de duas opções de fluxo de processamento, pode-se combinar as palavras chaves ***else*** e ***if*** para definir condições alternativas. Considere o exemplo a seguir, em que é verificado e informado ao usuário se um número é positivo, negativo ou igual a zero.

```
1  #include <stdio.h>
2
3  int main() {
4          int n1;
5          scanf("%d",&n1);
6          if(n1 > 0) {
7                  printf("valor positivo");
8          }
9          else if (n1 < 0){
10                 printf("valor negativo");
```

```
11          }
12          else {
13                  printf("valor igual a zero");
14          }
15          return 0;
16  }
```

Diferença em relação a python: em python existe uma palavra chave que representa a combinação de um ***else*** com um ***if***, denominada ***elif***. Em C essa palavra chave não existe.

5.2 Estrutura de decisão *switch-case*

Outra forma de modelar uma decisão é por meio da estrutura ***switch-case***. Nesse tipo de estrutura de decisão, o fluxo de código a ser executado depende do valor de uma expressão que retorna um valor inteiro (ou um tipo com representação inteira direta). Cada valor de interesse da expressão é denominado de "caso" (***case***). Considere o exemplo de código a seguir, que informa ao usuário qual a opção escolhida, de acordo com o número digitado.

```
1  #include <stdio.h>
2
3  int main () {
4     int op;
5     printf("Digite o valor de op: ");
6     scanf("%d",&op);
7     switch(op) {
8        case 1 :
9           printf("Op. 1 escolhida\n" );
10          break;
11       case 2 :
12          printf("Op. 2 escolhida\n" );
13          break;
14       case 3 :
15          printf("Op. 3 escolhida\n" );
16          break;
17       default :
18          printf("Op. invalida\n" );
19    }
20    return 0;
21 }
```

A variável do tipo ***int*** *op* é utilizada como a condição no ***switch*** e três casos (***case***) possíveis são definidos, que são executados quando a variável *op* possui os valores 1, 2 e 3, respectivamente. Caso um valor menor que 1 ou maior que 3 seja digitado, é informado ao usuário que a opção é inválida. O valor de cada caso deve ser sucessido por ":" e os comandos que fazem parte do caso devem ser colocados em seguida. Os comandos são então executados até que se encontre uma palavra chave ***break***.

É importante notar que se não for colocado um ***break***, todos os comandos posteriores ao ***case*** correspondente ao valor da condição serão executados, inclusive os comandos definidos nos casos posteriores. Por exemplo, caso seja retirado o ***break*** da linha 13 do exemplo mostrado, se o valor da variável *op* for igual a 2, a seguinte saída será mostrada na tela:

```
1 Op. 2 escolhida
2 Op. 3 escolhida
```

O caso ***default*** sempre deve ser definido (mesmo que sem comandos) e é executado quando o valor da variável de condição é diferente dos valores definidos em todos os ***case***.

5.3 Exercícios no URI

Resolver os seguintes problemas no URI, usando a linguagem de programação C:

1. 1035 – Teste de Seleção – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1035`
2. 1041 – Coordenadas de um Ponto – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1041`
3. 1042 – Sort Simples – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1042`
4. 1047 – Tempo de Jogo com Minutos – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1047`
5. 1049 – Animal – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1049`

6 Estruturas de Repetição

Durante a construção de algoritmos é comum aparecer casos em que um comando ou um determinado conjunto de comandos deve ser repetido uma certa quantidade de vezes ou repetidos de acordo com alguma condição. Portanto, é necessário algum mecanismo para se representar a repetição de comandos em um algoritmo. Na linguagem C existem três estruturas de repetição: ***while***, ***do while*** e ***for***.

A primeira estrutura de repetição a ser apresentada é a estrutura ***while***. Essa estrutura permite repetir a execução de um conjunto de comandos de acordo com alguma condição. Se a condição for verdadeira, o bloco de comandos definido dentro do escopo do ***while*** será executado. Ao final da execução de todos os comandos do escopo, a condição é avaliada novamente. Caso ela seja verdadeira, o bloco de comandos é executado novamente. Esse processo se repete até que a condição seja falsa, e nesse caso o bloco de comandos não é

executado e o fluxo de execução do algoritmo continua a partir do primeiro comando depois do fechamento de chaves (}) do ***while***. Caso a condição nunca fique falsa, o programa entrará em uma condição de *loop* infinito. Considere o exemplo a seguir, que usa uma estrutura de repetição ***while***.

```
1  #include<stdio.h>
2
3  int main() {
4          int N;
5          printf("Digite um valor para N\n");
6          scanf("%d",&N);
7          while(N != 5) {
8                  printf("Digite um novo valor para N\n");
9                  scanf("%d",&N);
10         }
11         printf("Finalmente foi digitado 5\n");
12         return 0;
13 }
```

No código mostrado é lido um valor inteiro da entrada, que é atribuído à variável N. Após isso, são lidos novos valores para N de forma contínua enquanto o valor lido for diferente de 5. É importante notar que se o primeiro valor lido para N for igual a 5 (fora do ***while***) nenhuma repetição é realizada. Dessa forma, a condição que define se uma repetição vai ocorrer ou não é a verificação se N é diferente de 5 (N != 5).

A estrutura de repetição ***do while*** é muito semelhante à estrutura ***while***. A única diferença é que no ***do while*** o bloco de comandos no interior da estrutura de repetição é executado pelo menos uma vez. Apenas após uma iteração o teste da condição é realizado. Se a condição for verdadeira ocorre a repetição dos comandos. O exemplo a seguir mostra o mesmo algoritmo descrito no exemplo anterior, mas agora utilizando o ***do while***.

```
1  #include<stdio.h>
2
3  int main() {
4          int N;
5          do {
6                  printf("Digite um novo valor para N\n");
7                  scanf("%d",&N);
8          } while(N != 5);
9          printf("Finalmente foi digitado 5\n");
10         return 0;
11 }
```

A terceira estrutura de repetição definida na linguagem C é a estrutura ***for***. Em C a definição do for é muito mais flexível do que em python, uma vez que a repetição também ocorre com base em uma condição e não com base em uma lista de valores a serem percorridos. No entanto, esse tipo de estrutura de repetição é geralmente a melhor escolha quando se sabe de antemão a quantidade de repetições a serem realizadas. O modo mais comum

de uso da estrutura ***for*** é por meio do uso de uma variável de controle, que geralmente serve para contar a quantidade de repetições já realizadas e também é útil para construção de algoritmos que manipulam arrays, como será melhor discutido mais na frente neste material. O exemplo de código a seguir mostra como repetir 100 vezes um conjunto de comandos usando ***for***.

```
1 #include<stdio.h>
2
3 int main() {
4         int i;
5         for(i = 0; i < 100; i++) {
6                 printf("Valor de i: %d\n", i);
7         }
8         return 0;
9 }
```

Ao executar o programa gerado por esse código são mostradas 100 frases na saída, no formato mostrado a seguir.

```
1 Valor de i: 0
2 Valor de i: 1
3 Valor de i: 2
4 Valor de i: 3
5 Valor de i: 4
6 Valor de i: 5
7 ...
8 Valor de i: 99
```

O código mostrado nesse exemplo é equivalente ao seguinte código que usa ***while***:

```
1 #include<stdio.h>
2
3 int main() {
4         int i = 0;
5         while(i < 100) {
6                 printf("Valor de i: %d\n", i);
7                 i++;
8         }
9         return 0;
10 }
```

Em geral, tudo que se pode fazer com um ***for***, é possível de se fazer com um ***while***, e vice-versa. A estrutura ***for***, entretanto, é mais compacta e mais fácil de ser usada em situações em que já se sabe a quantidade de repetições de antemão e quando o uso de uma variável de controle que conta a quantidade de repetições é necessária.

Dentro do ***for*** são definidas três partes: a inicialização da variável de controle (i = 0). a condição de repetição (i < 100) e por último o incremento da variável de controle (i++). Nenhuma das três partes é obrigatória, mas é necessário colocar de toda forma

os ; dividindo as três partes [ex: (; i < 100;)]. Tanto a condição de repetição como o incremento também podem ser escritos de forma diferente do que foi mostrado no exemplo. Por exemplo, pode-se definir que a variável de controle será incrementada de 2 em 2 (i += 2) ou mesmo que ela será decrementada. Também é possível inicializar e realizar incremento/decremento em mais de uma variável dentro da mesma estrutura ***for***. Considere o exemplo a seguir, em que duas variáveis de controle (i e k) são inicializadas e incrementadas/decrementadas dentro do ***for***.

```
1  #include<stdio.h>
2
3  int main() {
4          int i;
5          int k;
6          for(i = 0, k = 99; i < 100; i++, k--) {
7                  printf("Valor de i: %d\n", i);
8                  printf("Valor de k: %d\n", k);
9          }
10         return 0;
11 }
```

Ao executar o programa gerado por esse código são mostradas 200 frases na saída, no formato mostrado a seguir.

```
1  Valor de i: 0
2  Valor de k: 99
3  Valor de i: 1
4  Valor de k: 98
5  Valor de i: 2
6  Valor de k: 97
7  Valor de i: 3
8  Valor de k: 96
9  ...
10 Valor de i: 99
11 valor de k: 0
```

6.1 Exercícios no URI

Resolver os seguintes problemas no URI, usando a linguagem de programação C:

1. 1156 – Sequência S II – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1156`
2. 1157 – Divisores I – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1157`
3. 1158 – Soma de Ímpares Consecutivos III– `https://www.urionlinejudge.com.br/judge/pt/problems/view/1158`

4. 1160 – Crescimento Populacional – https://www.urionlinejudge.com.br/judge/pt/problems/view/1160

5. 1165 – Número Primo – https://www.urionlinejudge.com.br/judge/pt/problems/view/1165

6. 2682 – Detector de Falhas – https://www.urionlinejudge.com.br/judge/pt/problems/view/2682

7. 2415 – Consecutivos – https://www.urionlinejudge.com.br/judge/pt/problems/view/2415

8. 1895 – Jogo do Limite – https://www.urionlinejudge.com.br/judge/pt/problems/view/1895

7 Arrays

Existem problemas em que é necessário armazenar e processar um conjunto (possivelmente grande) de valores do mesmo tipo. Por exemplo, considere um programa que leia da entrada 1000 valores reais e depois informe a quantidade de valores acima da média. Para resolver esse problema, é necessário armazenar os 1000 valores, calcular a média aritmética desses 1000 valores, e por fim verificar quantos desses 1000 valores são maiores que a média que foi calculada. Utilizando apenas os conhecimentos que foram discutidos até o momento neste material, nós precisaríamos declarar 1000 variáveis e depois escrever milhares de linhas de código para conseguir ler da entrada os 1000 valores, calcular a média aritmética e verificar quantos valores são maiores que a média. Naturalmente, essa é uma solução que demandaria muito tempo e teria grande probabilidade de apresentar erros. O código a seguir ilustra de maneira compacta como seria essa solução pouco inteligente para o problema.

```
1  #include <stdio.h>
2
3  int main() {
4          double v1,v2,v3...v1000;
5          double media;
6          int qtd = 0;
7          scanf("%lf",&v1);
8          scanf("%lf",&v2);
9          ... // mais 997 linhas
10         scanf("%lf",&v1000);
11         media = (V1 + V2 + V3 +...+v1000)/1000;
12         if(v1 > media) qtd++;
13         if(v2 > media) qtd++;
14         ... // mais 997 linhas
15         if(v1000 > media) qtd++;
16         printf("%d\n",qtd);
17         return 0;
```

```
18 }
```

Felizmente, as linguagens de programação geralmente oferecem algum mecanismo para declararmos um grande conjunto de variáveis de um mesmo tipo utilizando pouco código. Também é possível manipular esse conjunto de valores utilizando estruturas de repetição, quando o mesmo processamento é realizado em todos os valores do conjunto, como é o caso do exemplo mostrado. Em C existe o conceito de *array*, que consiste em um conjunto de elementos de um mesmo tipo.

Diferença em relação a python: em python não existe o conceito de array no formato encontrado em C. Em python existe uma estrutura similar denominada *list*, que também serve para representar um conjunto de valores. No entanto, existem diferenças importantes entre um *array*, como definido na linguagem C, e um *list*, como definido na linguagem python. Primeiramente, o *array* é estático, ou seja, a quantidade de elementos é definida no momento da declaração e não pode variar posteriormente (a não ser que sejam usados ponteiros e alocação dinâmica de memória, que são assuntos de futura seções). Outra diferença importante é que um array armazena sempre valores de um mesmo tipo. Na Seção **??** iremos aprender a implementar em C estruturas de dados similares ao *list* de python.

Para declarar um array, precisamos informar o tipo dos elementos que serão armazenados dentro do array e a quantidade de elementos, seguindo a sintaxe mostrada a seguir:

```
1 <tipo> <nome_do_array> [quantidade_de_elementos];
```

O código a seguir mostra alguns exemplos de declaração e manipulação de arrays.

```
1  #include <stdio.h>
2
3  int main() {
4          char nome[50];
5          double notas[4];
6
7          scanf("%s", nome);
8          scanf("%lf", &notas[0]);
9          scanf("%lf", &notas[1]);
10         scanf("%lf", &notas[2]);
11         scanf("%lf", &notas[3]);
12
13         double media = (notas[0] + notas[1] + notas[2] + notas[3])/4;
14
15         printf("A media de %s foi %lf\n", nome, media);
16
17         return 0;
18 }
```

Neste exemplo de código são declarados dois *arrays*. O primeiro é um *array* de ***char*** com 50 elementos, ou seja, uma string de até 49 caracteres (o último elemento é reservado

para um caractere especial que indica o final da string). Em uma seção anterior, já foi mostrado como declarar, ler e escrever strings em C. O segundo é um *array* de valores reais (***double***), que representa um conjunto de notas. No código do exemplo, quatro notas são lidas da entrada, além do nome do aluno, e é calculada a média aritmética das quatro notas que são armazenadas dentro de um *array*.

Acessando elementos do *array*

Para acessar ou modificar os valores armazenados em um *array*, deve-se indexa-lo; ou seja, informar qual a posição do elemento dentro do *array* que se deseja acessar ou modificar. Em C, considerando um *array* de N elementos, ele pode ser indexado nas posições entre $\mathbf{0}$ **e** $\mathbf{N-1}$. Por exemplo, o *array notas*, do exemplo mostrado anteriormente, só pode ser indexado entre 0 e 3, uma vez que ele possui 4 elementos.

A indexação fora dos limites causa um erro em tempo de execução (*runtime error*); ou seja, um erro que só ocorre durante a execução do programa, mas não é identificado em tempo de compilação. Isso pode ser perigoso, pois um mesmo programa pode executar várias vezes sem apresentar erro, mas apresentar erro para um caso de teste específico. Portanto, sempre que realizar a indexação de um *array* em um trecho de código, analise se existe a possibilidade de haver indexação fora dos limites, especialmente quando o valor do índice for oriundo de algum processamento realizado anteriormente e que depende de valores fornecidos pela entrada. Nesse caso, deve-se analisar o intervalo de valores permitidos para a entrada e verificar se em algum caso o processamento resulta em um valor de índice inválido. Nesses casos, deve-se corrigir o código que calcula o índice ou realizar um tratamento para só permitir o acesso ao *array* quando o índice calculado for válido.

Voltando ao exemplo do início da seção, se quisermos ler 1000 valores da entrada de um mesmo tipo, podemos declara-los como um *array* de 1000 elementos. Além disso, podemos usar uma repetição para ler todos os elementos da entrada. Quando usamos a estrutura de repetição ***for***, por exemplo, em geral é usado um contador, que é incrementado em cada repetição, e serve para contar a quantidade de repetições já realizadas e indicar o momento em que o *loop* deve encerrar (por meio de uma condição). Esse contador pode ser usado para indexar os elementos do *array* em cada repetição, de modo que para cada iteração do *loop* um elemento diferente do *array* é acessado. O código a seguir mostra a implementação do problema de motivação (ou seja, ler 1000 valores reais e informar quantos são acima da média) utilizando *array* e repetição para acessar e processar os elementos do *array*.

```
1 #include <stdio.h>
2
3 #define SIZE 1000
4
5 int main() {
6     double val[SIZE];
7     double media = 0;
8     int i = 0;
9     for(i = 0; i < SIZE; i++) {
```

```
10          scanf("%lf", &val[i]);
11          media += val[i];
12      }
13      media = media/SIZE;
14
15      int qtd = 0;
16      for(i = 0; i < SIZE; i++) {
17          if(val[i] > media) qtd++;
18      }
19      printf("Quantidade de valores acima da media: %d\n",qtd);
20
21      return 0;
22  }
```

No exemplo, a variável i, que é usada para controlar as repetições, também é usada para indexar o vetor. Perceba que o i sempre começa com o valor 0 e possui o valor $SIZE - 1$ na última repetição, sendo que $SIZE$ é uma constante que define o tamanho do vetor (ver linha de código 3). Dessa forma, em cada iteração do ***for***, um elemento diferente do *array* é acessado. Utilizando uma constante, como no exemplo mostrado, pode-se modificar esse programa para funcionar com vetores de tamanhos diferentes de forma mais rápida, bastando apenas modificar o valor da constante na linha 3.

Acesso a Elementos da Vizinhança em um *Array*

Em alguns problemas, é necessário realizar um tratamento diferenciado para os elementos que se localizam nas extremidades do *array*. Isso ocorre quando é necessário acessar elementos da vizinhança (anteriores ou posteriores) de cada elemento do *array* para realizar algum processamento. O problema é que o primeiro elemento não possui elemento anterior e o último elemento não possui elemento posterior. Portanto, usualmente a solução desses problemas pode levar a um acesso ilegal ao array à medida em que se processa os elementos próximos das extremidades.

Considere o seguinte problema como exemplo: faça um programa que leia da entrada 10 valores inteiros e armazene-os em um array. Posteriormente, verifique quantos elementos do *array* possuem valor menor que o seu sucessor no *array*. Por exemplo, para a entrada a seguir:

```
1  2 3 5 4 6 7 6 3 7 1
```

Seria fornecida a saída:

```
1  5 valores menores que o sucessor
```

A solução para esse problema é mostrada no código a seguir:

```
1  #include <stdio.h>
2
3  #define SIZE 10
```

```
4
5  int main() {
6      int vet[SIZE];
7      int i;
8      for(i = 0; i < SIZE; i++) {
9          scanf("%d",&vet[i]);
10     }
11
12     int qtd = 0;
13     for(i = 0; i < SIZE; i++) {
14         if(i < SIZE-1) {
15             if(vet[i] < vet[i+1]) qtd++;
16         }
17     }
18     printf("%d valores menores que o sucessor\n", qtd);
19     return 0;
20 }
```

Note que dentro do ***for*** que verifica se os elementos do *array* são menores que os seus sucessores, a verificação só ocorre quando $i < SIZE - 1$. Isso impede de executar o código da linha 15 quando o i é igual a $SIZE - 1$; ou seja, quando se está acessando o último elemento do vetor. Se o código da linha 15 fosse executado para esse caso, ao indexar com $i + 1$ seria realizado um acesso fora dos limites do *array*, o que poderia causar um erro em tempo de execução.

Agora considere outro problema de exemplo: faça um programa que leia da entrada 10 valores inteiros e armazene-os em um *array*. Posteriormente, verifique quantos elementos do *array* possuem valor **menor que a soma dos dois elementos posteriores**. Para esse problema, deve-se considerar que o conjunto de valores possui uma **organização circular**; ou seja, o elemento posterior do último elemento do *array* é o primeiro elemento do *array*. Por exemplo, considerando um *array* de 10 elementos, os dois elementos posteriores ao elemento da posição 8 são os elementos das posições 9 e 0, e os dois elementos posteriores ao elemento da posição 9 (última posição do *array*) são os elementos das posições 0 e 1 (primeiro e segundo elementos do *array*).

Para a entrada a seguir:

```
1  2 3 5 4 1 0 6 4 7 1
```

Seria fornecida a saída:

```
1  7 elementos menores que a soma dos dois sucessores
```

A solução para esse problema é mostrada no código a seguir:

```
1  #include <stdio.h>
2
3  #define SIZE 10
4
```

```
5  int main() {
6      int vet[SIZE];
7      int i;
8      for(i = 0; i < SIZE; i++) {
9          scanf("%d",&vet[i]);
10     }
11
12     int qtd = 0;
13     for(i = 0; i < SIZE; i++) {
14         if(i == SIZE-2) { //penultimo elemento
15             if(vet[i] < (vet[i+1] + vet[0])) qtd++;
16         }
17         else if(i == SIZE-1) { //ultimo elemento
18             if(vet[i] < (vet[0] + vet[1])) qtd++;
19         }
20         else { //caso geral
21             if(vet[i] < (vet[i+1] + vet[i+2])) qtd++;
22         }
23     }
24     printf("%d elementos menores que a soma dos dois sucessores\n",qtd);
25     return 0;
26 }
```

Note que nesse caso foi necessário tratar de forma individual dois casos especiais, o caso de acesso ao penúltimo elemento e o caso de acesso ao último elemento do *array*. Todos os outros elementos entram no caso geral. Uma solução mais elegante para esse problema é mostrada a seguir, usando o operador módulo (resto de divisão). Fica como exercício para o aluno a simulação desse código para verificar como ele funciona.

```
1  #include <stdio.h>
2
3  #define SIZE 10
4
5  int main() {
6      int vet[SIZE];
7      int i;
8      for(i = 0; i < SIZE; i++) {
9          scanf("%d",&vet[i]);
10     }
11
12     int qtd = 0;
13     for(i = 0; i < SIZE; i++) {
14         int i1 = (i+1)%SIZE;
15         int i2 = (i+2)%SIZE;
16         if(vet[i] < (vet[i1] + vet[i2])) qtd++;
17     }
18     printf("%d elementos menores que a soma dos dois sucessores\n",qtd);
19     return 0;
20 }
```

7.1 Exercícios no URI

Resolver os seguintes problemas no URI, usando a linguagem de programação C:

1. 1245 – Botas Perdidas – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1245`
2. 1533 – Detetive Watson – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1533`
3. 1171 – Frequência de Números – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1171`
4. 1089 – Loop Musical – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1089`
5. 2248 – Estágio – `https://www.urionlinejudge.com.br/judge/pt/problems/view/2248`
6. 1225 – Coral Perfeito – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1225`
7. 1107 – Escultura à Laser – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1107`
8. 1125 – Fórmula 1 – `https://www.urionlinejudge.com.br/judge/pt/problems/view/1125`

8 Manipulação de Strings

9 Ponteiros

Toda variável declarada dentro de um código representa um local de memória alocado para armazenar valores de um determinado tipo. A memória é dividida em posições de memória que podem ser alocadas e cada posição de memória possui um endereço. Quando se declara uma variável, o seu identificador abstrai o seu endereço, de modo que o programador pode salvar valores e ler valores de um determinado local de memória a partir do seu identificador. A Figura 1 representa de maneira conceitual a memória principal de um computador.

As variáveis com identificadores A, B, C e D são do tipo ***int***. Perceba que cada variável possui um endereço correspondente, listado na parte de baixo. Por exemplo, o endereço da variável A é 10 e o endereço da variável D é 16.

Em C, assim como em outras linguagens de programação, existe um tipo especial de dados, denominado ponteiro, que pode armazenar o endereço de posições de memória; ou seja, o endereço físico de outras variáveis. Para declarar um ponteiro de um determinado

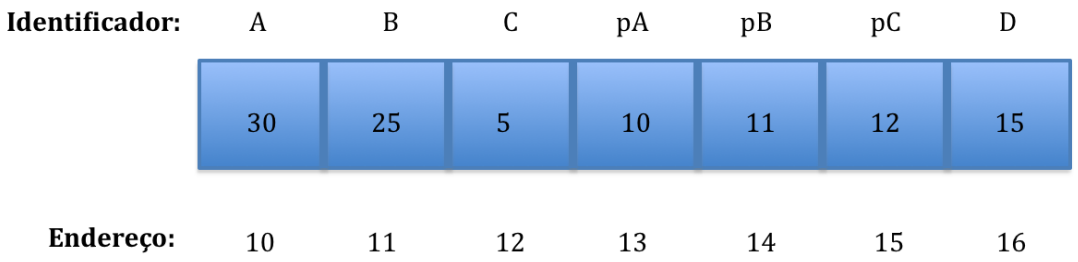

Figura 1: Representação conceitual da memória principal do computador.

tipo, usa-se a seguinte sintaxe:

```
1 <tipo> *<identificador_do_ponteiro> ;
```

Deve-se definir o tipo da variável que o ponteiro é capaz de referenciar e usa-se ***** seguido do identificador do ponteiro. Por exemplo, considere a declaração a seguir:

```
1     float *pR;
```

Nesse caso, um ponteiro denominado pR é declarado e pR é capaz de armazenar o endereço de variáveis do tipo ***float***.

No trecho de código a seguir são exemplificadas a declaração e a utilização de ponteiros. No exemplo, são declaradas quatro variáveis do tipo int (A, B, C e D) e três ponteiros para int (pA, pB, pC).

```
1  #include<stdio.h>
2
3  int main() {
4          int A, B, C, D;
5          int *pA, *pB, *pC;  //int * declara um ponteiro para int
6
7          A = 30;
8          B = 25;
9          C = 5;
10         D = 15;
11
12         pA = &A; //&A retorna o endereço de A
13         pB = &B;
14         pC = &C;
15
16         //* é usado pra acessar a variável apontada por um ponteiro
17         printf("Soma de A + B: %d\n", (A+B));
18         printf("Soma de A + B: %d\n", (*pA+B));
```

```
19          printf("Soma de A + B: %d\n", (*pA+*pB));
20
21          return 0;
22  }
```

As variáveis pA, pB e pC armazenam posições de memória utilizadas para armazenar valores inteiros (do tipo ***int***). É importante notar que ao serem declarados, os ponteiros não apontam para nenhum lugar de memória válido. Em C existe uma constante, denominada *NULL*, que indica que um ponteiro está nulo, ou seja, não aponta para nenhuma posição de memória.

Perceba que o código apresentado no exemplo corresponde exatamente à situação de memória representada pela Figura 1. Foram atribuídos os valores 30, 25, 5 e 15 às variáveis A, B, C e D , respectivamente. Após isso, temos o comando **pA = &A**. O operador **&** retorna o endereço de memória de uma variável. Dessa forma, esse comando atribui ao ponteiro pA o endereço de memória da variável A. Como podemos ver na figura, o endereço da variável A é igual a 10; ou seja, após esse comando pA passa a armazenar o valor 10. Nos comandos seguintes pB passa a armazenar o endereço da variável B (igual a 11) e pC passa a armazenar o endereço da variável C (igual a 12).

Também é possível acessar o valor que está armazenado no local de memória apontado por um ponteiro. Por exemplo, como pA aponta para o endereço de memória de A, podemos utilizar um operador para acessar o valor armazenado em A a partir de pA. Para acessar o valor armazenado no local apontado por um ponteiro, utilizamos o operador ***** antes do identificador do ponteiro. Por exemplo, $*pA$ retorna o valor armazenado pela variável A, uma vez que pA contém o endereço de A; ou seja, $*pA$ vai retornar o valor 30 no nosso exemplo. Dessa forma, perceba que em nosso exemplo as três saídas do final são equivalentes, de modo que a saída desse programa será:

```
1  Soma de A + B: 55
2  Soma de A + B: 55
3  Soma de A + B: 55
```

Nessa seção foi apresentado apenas o conceito de ponteiros e a sintaxe básica para declaração e operação com ponteiros. Mais na frente a utilidade dos ponteiros será melhor esclarecida, mais especificamente para alocação dinâmica de memória, passagem de parâmetros de funções por referência e para a implementação de estruturas de dados dinâmicas.

10 Tipos Estruturados (*structs*)

O sistemas computacionais essencialmente lidam com a manipulação de dados. Geralmente os dados manipulados pelo computador possuem alguma relação com o mundo real; ou seja, eles representam entidades reais existentes dentro do contexto onde o sistema

estará inserido. Por exemplo, em um sistema para controle de uma escola, provavelmente o sistema irá manipular dados relativos a alunos da escola (ex: Nome, Matrícula etc), dados relativos a disciplinas (horário, sala etc), entre outras informações. Dessa forma, ao desenvolver um software é necessário que se defina a representação dessas dados a serem manipulados e também as operações que são realizadas sobre esses dados.

Uma vez que os sistemas computacionais lidam com dados, devem existir mecanismos que permitam a representação desses dados no computador. Em nível de hardware, o computador é capaz de manipular apenas conjuntos de *bits*. Um *bit* pode possuir apenas dois valores: 0 ou 1. Dessa forma, apenas um conjunto restrito de informações podem ser representadas utilizando um bit. Por exemplo, no sistema de controle de escola, suponha que exista uma funcionalidade para registrar a presença de alunos. Para representar a presença ou ausência de um aluno em uma determinada aula é necessário apenas um bit ($0 \rightarrow$ ausente; $1 \rightarrow$ presente), uma vez que essa informação possui apenas dois estados possíveis.

No entanto, geralmente os computadores precisam manipular informações que possuem bem mais do que dois estados. É muito comum que sistemas computacionais manipulem números inteiros, números reais e conjuntos de caracteres (strings). Dessa forma, para que seja possível representar essas informações, deve-se agrupar um conjunto de bits, de modo a representar os vários estados requeridos. Por exemplo, o conjunto de números inteiros entre 0 e 255 pode ser representado utilizando 8 *bits* (1 *Byte*). O hardware dos computadores dão suporte à manipulação de dados do tipo inteiro e, em muitos casos, também dão suporte à manipulação de dados do tipo real. Dessa forma, é possível declarar e realizar operações sobre valores inteiros e reais diretamente em linguagem de baixo nível, utilizando as instruções providas pelo hardware do computador.

Apesar de ser possível manipular valores inteiros e reais diretamente a partir das instruções do hardware, muitos sistemas computacionais devem lidar com informações mais abstratas, que representam as entidades pertencentes ao contexto do problema a ser resolvido pelo sistema. Dessa forma, além das representações suportadas diretamente pelo hardware do computador, pode-se definir por software novos tipos de dados, que permitam representar uma grande quantidade de informações.

As próprias linguagens de programação podem oferecer tipos de dados mais complexos. Por exemplo, as linguagens de programação geralmente oferecem um tipo para representar caracteres. Nesse caso, esses caracteres são representados internamente como valores inteiros. Por exemplo, em algumas linguagens o caractere 'a' é representado pelo valor 97 (1100001 em binário), mas para o programador uma variável do tipo Caractere pode ser vista como um espaço de memória capaz de armazenar caracteres, não importando a forma como esses dados são realmente manipulados pelo computador. Algumas linguagens também fornecem um tipo string, que representa um conjunto de caracteres. Nesse caso o programador não precisa se preocupar como uma string é representada internamente no computador, embora seja importante, em muitos casos, conhecer mais detalhes sobre a representação e a manipulação desses dados mais abstratos.

Os tipos fornecidos nativamente por uma linguagem de programação são chamados de tipos primitivos. Os tipos primitivos da linguagem C foram descritos na Seção 1.2.1. No

geral, as linguagens possuem um ou mais tipos primitivos para representar números inteiros e números reais. Algumas linguagens fornecem abstrações para tipos mais complexos. Em C++ (assim como em Java), por exemplo, string não é um tipo primitivo, mas existe uma biblioteca que define uma classe para representação de Strings. Em geral, a manipulação de strings no código ocorre de forma similar à manipulação de variáveis primitivas, devido às abstrações fornecidas pela biblioteca (incluindo sobrecarga de operadores). A classe já possui a implementação para representar conjuntos de caracteres, além de um conjunto de operações (métodos) que podem ser utilizadas para manipular um conjunto de caracteres.

Da mesma forma que as linguagens de programação podem fornecer tipos mais complexos do que os suportados diretamente pelo hardware, é possível definir novos tipos com um nível de abstração ainda maior, por meio de uma combinação de tipos primitivos. Por exemplo, suponha que no sistema de controle acadêmico seja de interesse a manipulação de dados sobre alunos. Como as linguagens de programação não oferecem um tipo primitivo para representar um aluno no sistema (pelo menos as linguagens que eu conheço), é necessário criar um tipo de dados mais abstrato que permita representar esse aluno, contendo todas as informações importantes sobre um aluno para o sistema. Esse novo tipo mais abstrato de dados pode ser definido por meio da criação de um tipo estruturado, como descrito em mais detalhes na Seção 10.1.1.

Um tipo abstrato de dados define um conjunto de valores e um conjunto de operações que podem ser aplicadas nesses valores. Por exemplo, um tipo abstrato de dados pode definir um novo tipo capaz de representar Alunos ou uma Lista de Alunos. Também pode-se definir um conjunto de operações sobre o tipo abstrato de dados. Por exemplo, pode-se definir uma função para alterar a matrícula de um aluno ou para retornar a quantidade de alunos com média acima de 70 em uma lista de alunos.

10.1 Definindo uma *struct*

Como já discutido na seção anterior, algumas informações que devem ser processadas por um sistema computacional são compostas por múltiplos dados de diferentes tipos. Por exemplo, o Aluno em um sistema acadêmico poderia ser representado pelos seguintes atributos:

- nome (*array* de ***char***)
- idade (***int***)
- curso (*array* de ***char***)

Em um protocolo de rede, geralmente deve-se definir alguma abstração para representar uma mensagem ou pacote transmitido na rede. Um pacote de rede poderia ser representado pelos seguintes atributos:

- endereço de origem (***int***)
- endereço de destino (***int***)

- carga útil do pacote (array de ***uint8_t***)

Note que apenas os atributos necessários para cada entidade a ser representada devem ser incluídos. Na prática, geralmente os sistemas armazenam e manipulam mais informações relacionadas a Alunos ou pacotes de rede, mas manteremos essa definição reduzida por simplicidade. O tipo ***uint8_t*** (definido na biblioteca *inttypes.h*), usado na definição do pacote de rede, representa um inteiro de 8 *bits* sem sinal (equivalente a um ***unsigned char***). Essa notação é comum, por exemplo, na programação de sistemas embarcados.

O exemplo de código a seguir mostra como criar um tipo estruturado para Aluno, como declarar variáveis do novo tipo definido e como acessar atributos individuais das variáveis de tipos estruturados.

```
1  #include <stdio.h>
2
3  struct Aluno {
4      char nome[50];
5      int idade;
6      char curso[20];
7      char cpf[12];
8  };
9
10 int main() {
11     struct Aluno al;
12     al.idade = 30;
13     scanf("%s", al.nome);
14     return 0;
15 }
```

Para definir um novo tipo estruturado usa-se a palavra chave ***struct***. No exemplo mostrado, após essa palavra chave foi colocado um nome para o novo tipo estruturado (Aluno, no exemplo). Após a definição do nome do tipo estruturado, deve-se definir todos os atributos (ou membros) dentro do escopo da estrutura (entre as chaves), separados por ponto e vírgula. Para cada atributo, deve-se definir o tipo do atributo e um nome para o atributo. Após o fechamento das chaves, deve-se colocar um ponto e vírgula.

Para declarar uma variável do novo tipo estruturado deve-se usar a seguinte forma geral:

struct <nome_da_estrutura> <identificador_da_variável>;

Na linha 11 do exemplo é declarada uma variável do tipo ***struct*** Aluno, chamada *al*. Para acessar atributos individuais de variáveis de tipos estruturados deve-se usar a seguinte forma geral:

<nome_da_variavel>.<nome_do_atributo>;

Na linha 12 do exemplo é atribuído o valor 30 ao atributo *idade* da variável *al*. Na linha 13 é lida da entrada uma string para ser armazenada pelo atributo *nome*.

Para evitar usar sempre a palavra chave ***struct*** sempre que for necessário se referir ao novo tipo estruturado criado, pode-se definir um novo nome para o tipo, usando a palavra chave ***typedef***. Essa palavra chave permite definir novos nomes (apelidos) para qualquer tipo, incluíndo os tipos primitivos. Por exemplo, pode-se criar um novo tipo chamado *boolean*, com a seguinte definição:

typedef unsigned char boolean;

Após essa definição, pode-se declarar variáveis do tipo *boolean* (que na realidade poderá armazenar valores inteiros entre 0 e 255).

O exemplo de código a seguir define um novo nome para a ***struct*** Aluno, chamando-a simplesmente de Aluno. A definição do novo nome para o tipo pode ocorrer antes ou depois da definição da ***struct***.

```
1  #include <stdio.h>
2
3  typedef struct Aluno Aluno;
4
5  struct Aluno {
6      char nome[50], curso[20], cpf[12];
7      int idade;
8
9  };
10 int main() {
11     Aluno al;
12     al.idade = 30;
13     scanf("%s", al.nome);
14     return 0;
15 }
```

Uma vez definido o novo nome para o tipo estruturado, pode-se declarar variáveis usando apenas o nome Aluno para o tipo (ver linha 11 do exemplo).

Pode-se ainda definir de forma mais compacta o novo tipo estruturado e um nome para ele, como no exemplo a seguir. O código a seguir é equivalente ao exemplo que acabou de ser descrito, apenas é escrito de uma forma diferente.

```
1  #include <stdio.h>
2
3  typedef struct {
4      char nome[50], curso[20], cpf[12];
5      int idade;
6
```

```
7  } Aluno; //define o nome apenas no final
8
9  int main() {
10     Aluno al;
11     al.idade = 30;
12     scanf("%s", al.nome);
13     return 0;
14 }
```

Um tipo estruturado pode ser definido combinando atributos de diferentes tipos, incluíndo outros tipos estruturados. Portanto, é possível fazer uma composição entre tipos estruturados diferentes. No exemplo a seguir, além do tipo estruturado Aluno, também foi definido um novo tipo estruturado denominado Turma, que contém dois atributos: um array com 50 variáveis do tipo Aluno. Perceba na linha 19 como é feito o acesso a um atributo específico de um determinado aluno dentro do array.

```
1  #include <stdio.h>
2
3  typedef struct {
4      char nome[50], curso[20], cpf[12];
5      int idade;
6
7  } Aluno; //define o nome apenas no final
8
9  typedef struct {
10     Aluno alunos[50]; //possui um array de Aluno como membro
11     char codigo[10];
12 } Turma;
13
14 int main() {
15     Aluno al;
16     al.idade = 30;
17     Turma t;
18     t.alunos[0] = al;
19     t.alunos[1].idade = 40;
20     return 0;
21 }
```

10.1.1 Composição de Tipos Estruturados

Referências

[1] Microchip. "Manual do Compilador XC8".
Disponível em `http://ww1.microchip.com/downloads/en/DeviceDoc/50002053G.pdf`. Acessado em: 26/03/2018.

[2] Microsoft. “Data Type Ranges for Visual Studio 2015”. Disponível em https://msdn.microsoft.com/en-us/library/s3f49ktz.aspx. Acessado em: 26/03/2018.

[3] H. Fangohr. “Performance of Python’s long data type”. Disponível em http://www.southampton.ac.uk/~fangohr/blog/performance-of-pythons-long-data-type.html. Acessado em: 26/03/2018.

[4] P. Feofiloff. “Caracteres ASCII e o tipo char”. Disponível em https://www.ime.usp.br/~pf/algoritmos/aulas/char.html. Acessado em: 26/03/2018.

[5] H. Fangohr. “The Python Language Reference – Expressions”. Disponível em https://docs.python.org/2/reference/expressions.html#not-in. Acessado em: 03/09/2018.