UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"

Faculdade de Engenharia e Ciências de Guaratinguetá
Departamento de Engenharia Elétrica

Sistemas Microcomputadorizados

Laboratório 1

Revisão sobre programação em C

Sumário

1 Objetivos

- Revisar conceitos importantes da linguagem de programação C;
- Estudar o processo de compilação de programas em C.

2 Introdução

Em 1967, Martin Richards desenvolveu a BCPL. Esta linguagem de programação foi criada com o intuito de ser uma linguagem para desenvolver sistemas operacionais e compiladores. Baseando-se em algumas das funcionalidades da BCPL, Ken Thompson desenvolveu a linguagem B e, em 1970, utilizou-a para criar as primeiras versões do sistema operacional UNIX.

Dennis Ritchie, então, criou a versão inicial de C a partir da linguagem B, em 1972. Inicialmente, a linguagem C tornou-se amplamente conhecida como a linguagem utilizada no desenvolvimento do sistema operacional UNIX. Muitos dos sistemas operacionais mais utilizados atualmente são escritos em C e/ou C++.

2.1 Por que aprender C?

A grande vantagem em aprender a linguagem de programação C é que esta nos obriga a entender como a arquitetura de um computador funciona. É considerada uma linguagem de *nível intermediário*, no sentido de que está localizada entre as chamadas linguagens de alto nível (como, por exemplo, JavaScript) e as de muito baixo nível, que trabalham diretamente com o hardware do computador (como, por exemplo, Assembly) [1].

Vamos utilizar uma analogia para entender este conceito. Suponha que você esteja fazendo aulas para aprender a dirigir um carro, e que o veículo disponibilizado pela auto-escola possui câmbio automático. Naturalmente, enquanto você aprende a dirigir usando este carro, você não precisa se preocupar com todo o processo envolvido na direção para a troca manual de marchas. Após a conclusão das aulas, chega o momento do exame para obter a permissão para dirigir. Entretanto, ao chegar no local da avaliação, você nota que o carro que será utilizado no exame final possui câmbio manual. Ou seja, todo o procedimento para operação deste tipo de veículo ficou oculta durante o seu aprendizado.

Algo similar acontece quando aprendemos apenas a programar utilizando as chamadas linguagens de alto nível, pois não temos conhecimento do que está acontecendo em um nível fundamental: não sabemos como um computador trata estruturas e tipos de dados, como é gerenciada a memória ou como os dados são armazenados [2]. Se desejamos construir programas eficientes, que sejam executados o mais rápido possível, utilizando a menor quantidade de recursos possível do sistema, é preciso que tenhamos em mente o que acontece nesse nível mais fundamental.

C é considerada uma das linguagens de programação "modernas" mais poderosas, na medida em que permite acesso direto à memória e diversas outras operações computacionais de "baixo nível", enquanto mantém uma legibilidade razoável para seres humanos [3].

Por esses motivos, a linguagem C é amplamente utilizada para o desenvolvimento de sistemas que necessitam de alta performance, como, por exemplo, sistemas operacionais, sistemas embarcados, sistemas de tempo-real e sistemas de comunicação.

- **Sistemas Operacionais**: o desempenho apresentado por sistemas escritos em C faz com que a linguagem torne-se desejável para a implementação de sistemas operacionais. Sistemas tais como Linux, alguns trechos do Microsoft Windows e Android são escritos em C. O sistema OS X, da Apple, é construído utilizando a linguagem Objective-C, a qual é derivada de C.

- **Sistemas Embarcados**: a grande maioria dos processadores produzidos todos os anos está presente em sistemas embarcados, e não nos computadores de uso pessoal. Tipos de sistemas embarcados incluem sistemas de navegação, sistemas de segurança, smartphones, tablets, robôs entre outros. C é uma das linguagens mais populares para o desenvolvimento deste tipo de aplicação, uma vez que estas precisam ser executadas o mais rápido possível, consumindo a menor quantidade de recursos possível.
- **Sistemas de tempo-real**: são tipicamente utilizados em aplicações chamadas "críticas" (aquelas que trazem riscos, principalmente, à integridade física das pessoas), as quais necessitam de garantias temporais e comportamento previsível.
- **Sistemas de comunicação**: este tipo de sistema precisa ser capaz de tratar um grande volume de dados.

Neste laboratório, vamos revisar alguns dos principais conceitos da linguagem de programação C, os quais serão extremamente importantes para os tópicos que serão abordados ao longo do curso. Este roteiro é fortemente baseado em [4].

3 Variáveis

Variáveis nada mais são do que posições de memória onde valores podem ser armazenados para serem usados pelo programa. Cada variável deve ser definida com um nome (identificador) e um tipo de dados.

3.1 Declaração e atribuição de valores

Todas as variáveis devem ser definidas (declaradas) antes de serem utilizadas por um programa. O Bloco de Código 1 apresenta três exemplos de declaração de variáveis.

Bloco de código 1: Exemplo de definição de variáveis.

```
1 int var1;
2 char first_letter;
3 double result;
```

No exemplo acima, a variável `var1` foi declarada com o tipo `int`, o que indica que será utilizada para armazenar um número inteiro. Já a variável `first_letter`, foi declarada com o tipo `char`, o que indica que será utilizada para armazenar um caractere. Por fim, a variável `result` possui tipo de dados `double`, o que indica que será utilizada para armazenar o valor de um número de ponto flutuante.

O padrão C permite que variáveis sejam declaradas em qualquer ponto do código, desde que sejam declaradas antes do seu primeiro uso ao longo do programa (embora alguns compiladores antigos não permitam esta condição). Entretanto, é interessante declarar novas variáveis próximas ao local onde serão utilizadas pela primeira vez [4].

O nome de uma variável em C pode ser qualquer identificador considerado válido pela linguagem. Um identificador válido é um conjunto de caracteres composto por letras, digitos e *underscore* (_), desde que o identificador não comece com um dígito. C é uma linguagem *case sensitive*, ou seja, letras maiúsculas e minúsculas de um mesmo caractere são consideradas letras diferentes. Por exemplo, `a1` e `A1` são considerados identificadores diferentes.

Para armazenar valores em variáveis utilizamos o operador de atribuição (`=`). O Bloco de Código 2 apresenta um exemplo de como atribuir valores às variáveis apresentadas pelo Bloco de Código 1.

Bloco de código 2: Exemplo de atribuição de valores à variáveis.

```
var1 = 10;
first_letter = 't';
result = 3.1415;
```

É possível, também, atribuir um valor à uma variável durante a sua declaração – esta situação é chamada de **inicialização** da variável. O Bloco de Código 3 apresenta um exemplo de inicialização de variáveis.

Bloco de código 3: Exemplo de declaração e atribuição de valores à variáveis.

```
int var1 = 10;
char first_letter = 't';
double result = 3.1415;
```

3.2 Escopo

Em programação, de forma geral, o escopo de variáveis refere-se à visibiliadde e acessibilidade de uma variável em diferentes trechos do código. Em C, existem dois tipos principais de escopo: escopo local e escopo global.

Variáveis declaradas dentro de um bloco (como uma função, por exemplo) têm escopo local, o que significa que elas só podem ser acessadas dentro do bloco onde foram declaradas. O Bloco de Código 4 apresenta um exemplo desta situação.

Bloco de código 4: Exemplo de variável com escopo local.

```
#include <stdio.h>

void umaFuncaoQualquer() {
  int x = 10; // variável com escopo local
  printf("O valor de x dentro da função: %d\n", x);
}

int main() {
  umaFuncaoQualquer();
  /*
  printf("%d\n", x); Resultaria em um erro,
                     pois x não está definido neste escopo
  */

  return 0;
}
```

Neste exemplo, a variável `x` só pode ser acessada dentro da função `umaFuncaoQualquer` e qualquer tentativa de acessá-la fora dela resultaria em um erro de compilação.

Por outro lado, as variáveis globais têm escopo global, o que significa que elas podem ser acessadas de qualquer lugar no programa. O Bloco de Código 5 apresenta um exemplo do uso de variáveis globais.

Bloco de código 5: Exemplo de variável com escopo global.

```
#include <stdio.h>

int y = 20; // variável com escopo global

void umaFuncaoQualquer() {
```

```
6   printf("O valor de y dentro da função: %d\n", y);
7 }
8
9 int main() {
10  printf("O valor de y no escopo global: %d\n", y);
11  umaFuncaoQualquer();
12
13  return 0;
14 }
```

Neste caso, a variável y pode ser acessada tanto na função `umaFuncaoQualquer` quanto na função `main`, pois é globalmente definida fora de qualquer função específica.

Entender o conceito de escopo é de extrema importância para que seja possível gerenciar de forma correta e cuidadosa o escopo de uma variável a fim de se evitar *bugs* e garantir um comportamento previsível do programa. De forma geral, são consideradas boas práticas de programação manter o escopo de uma variável o mais limitado possível. Isso ajuda a evitar a poluição do espaço de nomes e minimizar conflitos de variáveis. É interessante, também, evitar o uso de variáveis globais. O uso excessivo de variáveis globais pode tornar o código menos legível, menos modular, mais difícil de entender e depurar. Prefira passar variáveis como argumentos para funções ou usar variáveis locais sempre que possível.

4 Funções

Programas de computador que resolvem problemas do "mundo real" são, em sua maioria, muito maiores que os programas que abordaremos ao longo do curso. Entretanto, diversos autores demonstram que a melhor maneira de desenvolver e, posteriormente, manter um grande projeto, é construí-lo a partir de pequenos módulos, onde cada um destes possui uma funcionalidade específica.

Na linguagem C, funções são utilizadas para criar estes pequenos módulos dentro de programas complexos. Programas são, normalmente, construídos ao combinarmos as funções que elaboramos com aquelas disponibilizadas pela biblioteca padrão C. A biblioteca padrão disponibiliza uma grande quantidade de funções para executar diversos tipos de tarefas, tais como operações matemáticas, manipulação de strings e caracteres, operações de entrada e saída, entre outras.

Para estudarmos os conceitos envolvidos nas definições de uma função em C, considere como exemplo o código apresentado pelo Bloco de Código 6.

Bloco de código 6: Definição da função `square`.

```
1  #include <stdio.h>
2
3  // Function Prototype
4  int square(int y);
5
6  int main(void) {
7    int result = 0;
8
9    for (int x = 1; x <= 10; ++x) {
10     result = square(x);
11     printf("%d ", result);
12   }
13
14   puts("");
15 }
16
17 // Function Definition
```

```
18 int square(int y) {
19   int sq_result = y * y;
20   return sq_result;
21 }
```

Resultado

```
1  4  9  16  25  36  49  64  81  100
```

4.1 Definição de uma função

A definição da função `square` é apresentada pelas linhas 18 a 21 do Bloco de Código 6. Nesta definição, podemos observar que a função espera receber como argumento um número do tipo inteiro (`int y`). A palavra-chave (*keyword*) `int`, à esquerda do nome da função, indica que a função `square` retornará como resultado um número do tipo `int`. A palavra-chave `return`, dentro da função `square` (linha 20), passa o valor armazenado na variável `sq_result` (ou seja, o resultado do cálculo de y^2) de volta para a função que fez a chamada à função `square`.

Se observarmos, ainda, a definição da função, vamos notar que a declaração da variável `sq_result` é feita dentro da função `square`. Todas as variáveis declaradas dentro da definição de uma função são variáveis locais, ou seja, estas podem ser acessadas apenas dentro da função onde foram instanciadas. Funções, em C, possuem uma lista de argumentos, os quais são um meio para que funções recebam informações externas. Os argumentos de uma função também são variáveis locais para a própria função.

4.2 Protótipo de uma função

Na linha 4, encontramos a seguinte instrução:

```
4 int square(int y);
```

Esta instrução é chamada de protótipo da função (também é conhecida como declaração da função). A palavra-chave `int`, dentro dos parênteses, informa ao compilador que a função `square` espera receber um valor do tipo inteiro. O compilador utiliza o protótipo da função para validar a chamada à mesma. Isto é feito ao comparar a chamada à função `square` (linha 10 do Bloco de Código 6) com o protótipo da função. Esta comparação garante que:

- o número de argumentos está correto;
- os argumentos possuem os tipos de dados corretos;
- os tipos de dados dos argumentos estão na ordem correta;
- o tipo de dados do retorno da função está de acordo com o contexto no qual a função foi chamada.

4.3 Passando argumentos por valor e por referência

Em diversas linguagens de programação, existem duas maneiras para passarmos argumentos para funções: **por valor** e **por referência**.

Quando argumentos são passados por valor, uma cópia do valor do argumento é criada e passada para a função que foi chamada. Passagem de argumentos por valor deve ser utilizada sempre que a função chamada não precise alterar o conteúdo da variável que a função chamadora passou para

ela. Como exemplo, no Bloco de Código 6, a função `main` chama a função `square` e passa como argumento a variável `x`. Neste ponto, uma cópia do conteúdo da variável `x` é criada e enviada para a função `square`, a qual, por sua vez, armazena esta cópia em sua variável local `y`. Qualquer alteração feita no valor da variável `y` não será reproduzida pela variável `x`, a qual faz parte do escopo da função `main`.

Quando um argumento é passado por referência, como o próprio nome indica, uma referência à variável é passada para a função que recebe o argumento. Nesta situação, é possível que a função chamada realize alterações na variável da função chamadora.

Na seção 6, onde o conceito de ponteiros é abordado, estudaremos como realizar a passagem de argumentos por referência.

4.4 Funções Matemáticas

A biblioteca padrão da linguagem C inclui diversas funções para a realização de cálculos matemáticos. Para utilizá-las, deve-se incluir o cabeçalho `math.h`. A Tabela 1 apresenta algumas das funções matemáticas mais utilizadas.

Tabela 1: Funções matemáticas mais utilizadas (definidas em `math.h`)

Função	Descrição
`sqrt(x)`	Raiz quadrada de `x`.
`exp(x)`	Exponencial de `x` (e^x).
`log(x)`	Logaritmo natural (base e) de `x` ($ln(x)$).
`log10(x)`	Logaritmo de `x` (base 10) ($log_{10}x$).
`ceil(x)`	Arredonda `x` para o menor inteiro não menor que `x`.
`floor(x)`	Arredonda `x` para o maior inteiro não maior que `x`.
`pow(x, y)`	`x` elevado a potência `y` (x^y).
`fmod(x, y)`	Resto da operação x/y (retorna um `float`).
`sin(x)`	Seno de `x`.
`cos(x)`	Cosseno de `x`.
`tan(x)`	Tangente de `x`.

5 Arrays

Arrays são estruturas de dados que consistem em dados relacionados do mesmo tipo. Do ponto de vista de armazenamento, arrays são posições de memória adjacentes que armazenam o mesmo tipo de dados. Para fazer referência a um determinado elemento do array, especificamos o nome do array e o número da posição (também chamado de índice, ou *index*) do elemento em questão dentro do array.

A Figura 1 apresenta um exemplo de um array de números inteiros, denominado `c`, o qual contém 12 elementos. Pode-se referenciar qualquer um destes elementos ao utilizarmos o nome do array seguido pelo número da posição do elemento específico dentro de colchetes ([ ]). Em qualquer array, o primeiro elemento possui número de posição 0 (zero).

Se quisermos, por exemplo, armazenar o número inteiro 1000 no terceiro elemento do array `c`, apresentado pela Figura 1, devemos proceder da seguinte forma:

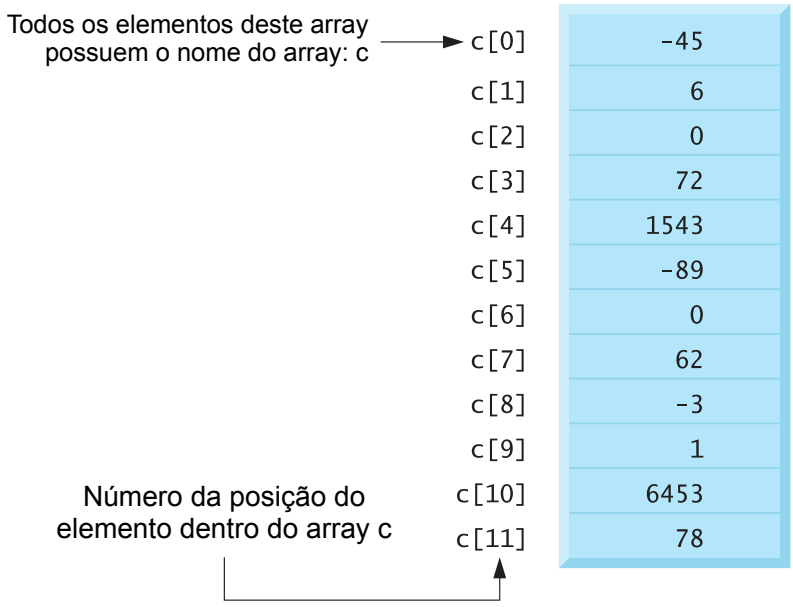

Figura 1: Array `c` com 12 posições. Adaptado de [4].

```
c[2] = 1000;
```

De forma semelhante, supondo que `a = 5` e `b = 6`, então podemos adicionar `2` ao elemento `c[11]` da seguinte forma:

```
c[a + b] += 2;
```

5.1 Declaração de arrays

Quando declaramos um array, especificamos o tipo de dados de cada elemento e a quantidade de elementos que este array conterá. Desta forma, o computador poderá reservar a quantidade de memória adequada para esta estrutura de dados. No exemplo a seguir, fazemos a declaração do array `c`, onde reservamos memória para 12 elementos do tipo inteiro. Lembrando que este array conterá elementos com índices de 0 a 11.

```
int c[12];
```

Se quisermos reservar memória para outros dois arrays, denominados `b` e `x`, por exemplo, procedemos da mesma forma, conforme apresentado abaixo.

```
int b[100], x[27];
```

Neste último exemplo, lembrar que os arrays `b` e `x` possuem 100 e 27 elementos, enquanto os índices de seus elementos vão de 0 a 99 e 0 a 26, respectivamente. Apesar de ser possível declarar mais de um array na mesma linha, é aconselhável fazê-lo em linhas separadas.

Podemos criar arrays de outros tipos de dados. Um array do tipo `char`, por exemplo, pode armazenar um array de caracteres (ou uma string). Veremos outros exemplos de arrays com outros tipos de dados ao longo deste roteiro.

5.2 Inicializando arrays

Podemos inicializar um array de algumas formas diferentes.

5.2.1 Utilizando um loop

Podemos inicializar um array utilizando um loop para atribuir um valor válido a cada um de seus elementos. O Bloco de Código 7 apresenta um exemplo deste método.

Bloco de código 7: Inicialização de um array usando um loop `for`.

```
1  #include <stdio.h>
2
3  int main() {
4    int n[5];
5
6    for (size_t i = 0; i < 5; ++i) {
7      n[i] = 0;
8    }
9
10   printf("%s%13s\n", "Element", "Value");
11
12   for (size_t i = 0; i < 5; ++i) {
13     printf("%7lu%13d\n", i, n[i]);
14   }
15
16   return 0;
17 }
```

No exemplo acima, o primeiro laço `for` (linhas 6 a 8) realiza a inicialização de cada elemento do array com o número 0 (zero). Já o segundo laço `for` (linhas 12 a 14) exibe na tela o índice e o valor armazenado em cada elemento.

O resultado da execução do programa exibido pelo Bloco de Código 7 é apresentado abaixo e o código está disponível neste link.

Resultado

```
Element        Value
      0            0
      1            0
      2            0
      3            0
      4            0
```

5.2.2 Utilizando uma lista de valores para inicialização

Neste método, passamos os valores dos elementos do array na forma de uma lista. Cada item desta lista é atribuído a um elemento do array. O Bloco de Código 8 apresenta a declaração e inicialização dos arrays `n` (linha 4), `o` (linha 13) e `p` (linha 22). Repare como estes arrays são inicializados de formas ligeiramente diferentes entre si.

Bloco de código 8: Inicialização de arrays usando lista de valores.

```
1  #include <stdio.h>
2
3  int main() {
4    int n[5] = {32, 27, 64, 18, 95};
5
6    printf("Printing array \"n\":\n");
7    printf("%s%13s\n", "Element", "Value");
8
9    for (size_t i = 0; i < 5; ++i) {
```

```
10      printf("%7u%13d\n", i, n[i]);
11    }
12
13    int o[10] = {3};
14
15    printf("\nPrinting array \"o\":\n");
16    printf("%s%13s\n", "Element", "Value");
17
18    for (size_t i = 0; i < 10; ++i) {
19      printf("%7u%13d\n", i, o[i]);
20    }
21
22    int p[] = {3, 4, 5, 6, 7, 8, 9};
23
24    printf("\nPrinting array \"p\":\n");
25    printf("%s%13s\n", "Element", "Value");
26
27    for (size_t i = 0; i < 7; ++i) {
28      printf("%7u%13d\n", i, p[i]);
29    }
30
31    return 0;
32  }
```

A declaração e inicialização do array `n` são feitas na linha 4. Neste caso, o array é declarado com 5 elementos (`int n[5]`). A lista de valores do lado direito do operador de atribuição também possui 5 elementos: `32, 27, 64, 18, 95`. Nesta situação, o primeiro item da lista (número 32) é atribuído ao primeiro elemento do array (elemento `c[0]`). O segundo item da lista (número 27) é atribuído ao segundo elemento do array (`c[1]`) e, assim, sucessivamente.

Já no caso do array `o` (linha 13), a lista de inicialização possui apenas 1 item (número 3), enquanto o array possui 10 elementos. Neste caso, o valor 3 (três) é atribuído ao elemento `o[0]`, enquanto os outros elementos do array são inicializados automaticamente com 0 (zero).

Por fim, no caso do array `p` (linha 22), não é especificado seu número de elementos dentro dos colchetes durante sua declaração. Nesta situação, o array terá número de elementos igual a quantidade de itens presentes na lista de inicialização. Como a lista de inicialização do array `p` possui 7 itens (`3, 4, 5, 6, 7, 8, 9`), este também possuirá 7 elementos, onde `o[0]` receberá como valor o primeiro item da lista (número 1), `o[1]` receberá o segundo valor da lista (número 4) e, assim, sucessivamente.

O resultado da execução do programa exibido pelo Bloco de Código 8 é apresentado abaixo e o código está disponível neste link.

Resultado

```
Printing array "n":
Element         Value
      0            32
      1            27
      2            64
      3            18
      4            95

Printing array "o":
Element         Value
      0             3
      1             0
      2             0
```

```
      3            0
      4            0
      5            0
      6            0
      7            0
      8            0
      9            0

Printing array "p":
Element        Value
      0            3
      1            4
      2            5
      3            6
      4            7
      5            8
      6            9
```

5.3 Trabalhando com arrays de caracteres

Até agora, trabalhamos apenas com arrays de números inteiros. Entretanto, como mencionado anteriormente, arrays podem armazenar dados de qualquer tipo de dados. Nesta seção, vamos aprender como armazenar *strings* em arrays de caracteres (tipo `char`).

5.3.1 Inicializando um array de caracteres com um string

Arrays de caracteres têm algumas características únicas. Por exemplo, este tipo de array pode ser inicializado utilizando-se um "*string literal*".

```
char string1[] = "First";
```

No exemplo acima, o array `string1` é declarado e inicializado com o string "`First`", onde cada elemento do array recebe um dos caracteres desse string. Neste caso, o tamanho do array é determinado pelo compilador baseado na quantidade de caracteres do string. Assim sendo, o array `string1` possui 5 elementos para acomodar cada um dos caracteres do string "`First`", mais um caractere especial que indica o fim de um string. Este caractere especial é chamado de null character e é representado por "\0". Todos os strings em C terminam com este caractere. Desta forma, o array `string1` possui, na verdade, 6 elementos no total.

5.3.2 Inicializando um array de caracteres com uma lista de inicialização

Assim como estudamos na seção 5.2.2, arrays de caracteres também podem ser inicializados com uma lista de valores. Neste caso, os valores da lista são caracteres. Segue um exemplo de como inicializar um array desta forma.

```
char string1[] = {'F', 'i', 'r', 's', 't', '\0'};
```

Veja que, neste caso, precisamos incluir o caractere de terminação como último elemento do array.

5.3.3 Exemplo de uso de arrays de caracteres

O Bloco de Código 9 demonstra a declaração e inicialização dos arrays, a leitura e armazenamento de um string em um array de caracteres e como acessar cada caractere individualmente neste tipo de array. Repare que o laço `for` verifica explícitamente se o caractere sendo avaliado é diferente do null character.

Bloco de código 9: Exemplo do uso de arrays de caracteres.

```c
#include <stdio.h>

#define SIZE 20

int main (void) {
  char string1[SIZE];
  char string2[] = "string literal";

  printf("%s", "Enter a string (no longer than 19 characters): ");
  scanf("%19s", string1);

  printf("string1 is: %s\nstring2 is: %s\n", string1, string2);

  printf("string1 with spaces between characters is:\n");
  for (size_t i = 0; i < SIZE && string1[i] != '\0'; i++) {
    printf("%c ", string[i]);
  }

  puts("");

  return 0;
}
```

O resultado da execução do programa exibido pelo Bloco de Código 9 é apresentado abaixo e o código está disponível neste link.

Resultado

```
Enter a string (no longer than 19 characters): Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
```

5.4 Passando arrays para funções

Para passar um array como argumento para uma função, utilizamos o identificador deste array sem os colchetes. Vamos considerar que um array chamado `hourlyTemperatures` tenha sido definido, conforme exemplificado abaixo.

```c
int hourlyTemperatures[HOURS_IN_A_DAY];
```

Com o array declarado, vamos considerar, também, que precisamos passar este array para uma função chamada `modifyArray`.

```c
modifyArray(hourlyTemperatures, HOURS_IN_A_DAY);
```

No exemplo acima, passamos o array `hourlyTemperatures` e o tamanho deste array (número de elementos) para a função `modifyArray`. Em C, arrays são passados para funções por referência.

O nome do array passado para a função é traduzido para o endereço da posição de memória do primeiro elemento do array. Assim sendo, o que a função `modifyArray` realmente recebe é o endereço do elemento `hourlyTemperatures[0]`. Desta forma, quando a função `modifyArray` realiza alguma modificação em algum elemento do array, estará modificando o conteúdo original do elemento do array.

No Bloco de Código 10 é apresentado um exemplo que demonstra que o endereço armazenado pelo identificador do array ("nome do array"), nada mais é que o endereço da posição de memória do primeiro elemento do array.

Bloco de código 10: Endereços de memória de arrays.

```c
#include <stdio.h>

int main(void) {
  char array[5];

  printf("array = %p\n&array[0] = %p\n&array = %p\n",
    array, &array[0], &array);
}
```

O resultado da execução do programa exibido pelo Bloco de Código 10 é apresentado abaixo e o código está disponível neste link.

Resultado

```
    array = 0x7fffc7bcf543
&array[0] = 0x7fffc7bcf543
   &array = 0x7fffc7bcf543
```

6 Ponteiros

Nesta seção, vamos estudar um dos conceitos mais importantes, e que também gera muitas dúvidas, da linguagem C: ponteiros. Ponteiros permitem que programas passem parâmetros por referência para funções e passem funções entre funções. Permitem, também, criar e manipular estruturas dinâmicas de dados, as quais podem aumentar e diminuir de tamanho durante a execução de um programa. Neste laboratório, vamos nos ater aos conceitos básicos de ponteiros. Conceitos mais avançados serão revisados ao longo do curso.

6.1 Declaração e inicialização de ponteiros

Ponteiros são variáveis cujos valores são endereços de memória. Normalmente, uma variável contém um determinado valor de um determinado tipo de dados. Um ponteiro, no entanto, contém o endereço de uma variável, a qual, por sua vez, contém um valor específico. Neste sentido, um nome de variável faz referência a um valor diretamente, enquanto um ponteiro faz referência a um valor indiretamente. A Figura 2 exibe uma representação visual deste conceito.

Da mesma forma que outros tipos de variáveis, ponteiros devem ser declarados antes de serem usados. No exemplo abaixo, fazemos a declaração de um ponteiro para uma variável do tipo `int` (`countPtr`) e, também, fazemos a declaração de uma variável do tipo `int` (`count`).

```c
int *countPtr, count;
```

No exemplo acima, podemos observar que a variável `countPtr` é do tipo `int *`, ou seja, é "um ponteiro para uma variável do tipo `int`". Já a variável `count` é do tipo `int`, e não um ponteiro. Nesta linha de declaração de variáveis, o "*" se aplica apenas à variável `countPtr`.

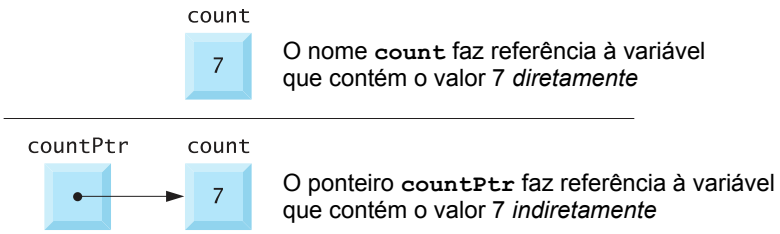

Figura 2: Fazendo referência a uma variável direta e indiretamente. Adaptado de [4].

Um ponteiro pode ser inicializado com um valor `NULL`, `0` (zero) ou um endereço de memória. Um ponteiro com valor `NULL` não aponta para nenhuma variável. `NULL` é uma constante simbólica definida no arquivo de cabeçalho `<stddef.h>`. Este arquivo é incluído por diversos outros arquivos de cabeçalho, entre eles, o arquivo `<stdio.h>`. Inicializar um ponteiro com `0` é equivalente a inicializar com `NULL`. No entanto, é recomendável, nesta situação, inicializar com `NULL`, pois, desta forma, enfatizamos que a variável é do tipo ponteiro. O valor inteiro `0` é o único valor inteiro que pode ser atribuído diretamente a uma variável do tipo ponteiro.

A atribuição de um endereço a um ponteiro será abordada na próxima seção, onde estudaremos sobre o operador de endereço.

6.1.1 O operador de endereço (&)

O operador de endereço, o qual é representado pelo caractere `&`, é utilizado para obter-se o endereço da posição de memória de uma variável. Para exemplificar seu uso, considere o Bloco de Código 11 (código disponível neste link).

Bloco de código 11: Exemplo de declaração e inicialização de ponteiros.

```
1  #include <stdio.h>
2
3  int main() {
4    int y = 5;
5    int *yPtr;
6
7    yPtr = &y;
8
9    printf("Valor da variável y: %d\n", y);
10   printf("Endereço da variável y: %p\n", &y);
11   printf("Valor da variável yPtr: %p\n", yPtr);
12   printf("Endereço da variável yPtr: %p\n", &yPtr);
13
14   return 0;
15 }
```

Como podemos observar, na linha 4, o valor `5` é atribuído à variável `y`. Na linha 5, o ponteiro `yPtr` é declarado. Na linha 7, utilizamos o operador de endereço para atribuir ao ponteiro o endereço da posição de memória ocupada pela variável `y`. Nas linhas 9 a 13, temos um conjunto de instruções `printf` com o objetivo de apresentar na tela os valores das variáveis em uso neste programa. O resultado da execução deste programa é apresentado abaixo.

Resultado

```
Valor da variável y: 5
Endereço da variável y: 0x7ffec7d2c7ac
Valor da variável yPtr: 0x7ffec7d2c7ac
Endereço da variável yPtr: 0x7ffec7d2c7b0
```

É importante prestar bastante atenção no resultado produzido por esse programa. Na primeira linha, temos o valor armazenado pela variável `y` (5). Já na segunda linha, temos o endereço da variável `y`. Se voltarmos ao Bloco de Código 11, na linha 10, podemos observar que obtemos este valor ao utilizar o operador `&` em frente ao nome da variável `y`.

Na terceira linha do resultado, obtemos o conteúdo do ponteiro `yPtr`. Note que o conteúdo do ponteiro é igual ao endereço da variável `y`, como esperado. A quarta linha do resultado apresenta o endereço da variável ponteiro `yPtr`. Da mesma forma como procedemos para obter o endereço da variável `y`, utilizamos também o operador de endereço para obter o endereço da variável `yPtr`. Como esperado, este último endereço é diferente do endereço armazenado por ela própria.

6.1.2 Representação gráfica da posição em memória de um ponteiro

Relembrando a seção anterior, a execução do Bloco de Código 11 nos mostrou que os endereços das posições de memória ocupadas pelas variáveis `y` e `yPtr` são `0x7ffec7d2c7ac` e `0x7ffec7d2c7b0`, respectivamente. Logo, uma representação gráfica do esquemático da memória ocupada por estas variáveis pode ser apresentado pela Figura 3.

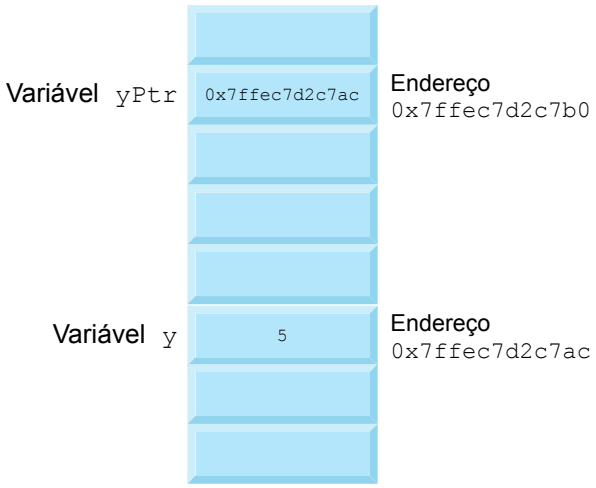

Figura 3: Esquemático representando as posições de memória ocupadas pelas variáveis `y` e `yPtr`. Adaptado de [4].

Como podemos observar, o conteúdo da variável `yPtr` é o endereço da posição de memória ocupada pela variável `y`.

6.1.3 O operador de derreferência *

O operador de derreferência é utilizado para se obter o valor da variável que é apontada por um ponteiro. Como exemplo, vamos considerar que tivéssemos adicionado a linha abaixo ao programa exibido pelo Bloco de Código 11.

```
printf("Valor da variável apontada pelo ponteiro yPtr: %d\n", *yPtr);
```

Perceba que o segundo argumento que passamos para a função `printf` é `*yPtr`. Neste caso, estamos interessados no valor armazenado na posição de memória apontada pelo ponteiro `yPtr`, ou seja, o valor `5`, uma vez que este é o conteúdo da variável `y`, que é apontada pelo ponteiro `yPtr`.

O resultado da execução desta linha de código é apresentado abaixo.

Resultado

```
Valor da variável apontada pelo ponteiro yPtr: 5
```

6.2 Passando argumentos para funções por referência

Como mencionado anteriormente (seção 4.3), existem duas formas para se passar um argumento para uma função: por valor e por referência. Funções frequentemente necessitam da capacidade de modificar variáveis da função chamadora ou receber um ponteiro para um conjunto de dados muito grande para evitar o *overhead* causado por receber uma cópia deste conjunto.

Na linguagem C, utilizamos os operadores `&` e `*` para aplicar o conceito de passagem por referência. Quando chamamos uma função onde os argumentos devem ter seu conteúdo modificado, passamos os endereços dos argumentos para a função. Para tanto, normalmente, utilizamos o operador de endereço junto à variável que deve ser modificada (a qual está presente na função chamadora). Quando o endereço de uma variável é passado a uma função, podemos utilizar o operador de derreferência dentro da função para obter e modificar o valor armazenado naquela determinada posição de memória.

Para melhor exemplificar este conceito, vamos considerar o programa apresentado pelo Bloco de Código 12.

Bloco de código 12: Cálculo do cubo de um número inteiro utilizando passagem por referência.

```
1  #include <stdio.h>
2
3  void cubeByReference(int *nPtr);
4
5  int main(void) {
6    int number = 5;
7    printf("Valor original da variável \"number\": %d\n", number);
8
9    cubeByReference(&number);
10
11   printf("Novo valor da variável \"number\": %d\n", number);
12
13   return 0;
14 }
15
16 void cubeByReference(int *nPtr) {
17   *nPtr = *nPtr * *nPtr * *nPtr;
18 }
```

Uma função que recebe um endereço como um de seus argumentos deve definir um parâmetro do tipo ponteiro para tal. No Bloco de Código 12, a função `cubeByReference` (linha 14) especifica que esta deve receber o endereço de uma variável do tipo inteiro como um argumento. A função, então armazena este endereço localmente e não retorna nenhum valor (`void`).

Perceba que, internamente à função `cubeByReference`, utilizamos o operador de derreferência (`*`) para que tenhamos acesso ao valor (linha 23) que está armazenado na posição de memória apontada pelo endereço recebido. Cuidado para não confundir este operador com o operador matemático de multiplicação. Apesar de utilizarem o mesmo símbolo gráfico, o operador de derreferência é aquele que está junto ao nome da variável `nPtr`. Note, também, que em nenhum momento fizemos qualquer modificação ao conteúdo da variável `n` diretamente. Fizemos esta alteração indiretamente.

O resultado da execução do programa exibido pelo Bloco de Código 12 é apresentado abaixo e o código está disponível neste link.

Resultado

```
Valor original da variável ‘‘number’’: 5
Novo valor da variável ‘‘number’’: 125
```

7 Structures (Structs)

Structures (ou *Structs*) são uma espécie de coleções de variáveis relacionadas sob o mesmo nome. Um `struct` pode conter variáveis de diversos tipos de dados. Podemos utilizar uma combinação entre `structs` e ponteiros para formar tipos de dados mais complexos.

7.1 Definição de um `struct`

Um `struct` é um tipo de dados derivado de outros tipos de dados. Considere o exemplo abaixo.

```
struct card {
  char *face;
  char *suit;
};
```

A palavra-chave `struct` é utilizada para definirmos um novo structure. O identificador `card` é chamado de tag do structure, qual dá nome à definição do structure e é usado em conjunto com a palavra-chave `struct` para declarar variáveis do tipo do structure, como, neste caso, `struct card`. As variáveis declaradas dentro das chaves que definem o `struct` são chamadas de membros do structure. Cada definição de `struct` deve terminar com um ponto-e-vírgula.

A definição do `struct card` contém as variáveis `face` e `suit`, onde ambos são ponteiros do tipo `char *`. Variáveis membro de um structure podem ser variáveis dos tipos primitivos (`int`, `float`, etc), ou agregados, como arrays e outros structures.

Abaixo, é apresentado mais um exemplo, com variáveis membro de diversos tipos.

```
struct employee {
  char firstName[20];
  char lastName[20];
  unsigned int age;
  double hourlySalary;
};
```

7.2 Declarando variáveis do tipo `struct`

A definição de um novo tipo de dados `struct` não faz a alocação de memória para nenhuma nova variável. Ao invés disso, cada definição cria um novo tipo de dados que pode ser utilizado para definir novas variáveis. Variáveis do tipo `struct` são declaradas da mesma forma que variáveis de outros tipos de dados, conforme exemplificado abaixo.

```
struct card aCard, deck[52], *cardPtr;
```

No exemplo acima, declaramos uma variável do tipo `struct card`, um array do tipo `struct card` com 52 elementos e, por fim, um ponteiro (`cardPtr`) para uma variável `struct card`.

7.3 Inicializando structures

Structs podem ser inicializados utilizando-se listas de valores de inicialização, assim como fizemos com arrays. Como exemplo, considere a declaração de uma variável do tipo `struct card`, com identificador `aCard`.

```
struct card aCard = {"Three", "Hearts"};
```

Neste exemplo, a variável `aCard` é criada. À sua variável membro `face` é atribuído o string "`Three`", enquanto sua variável membro `suit` recebe o valor "`Hearts`". Caso o número de itens da lista de inicialização seja menor que o número de variáveis membro de um structure, então os demais membros são inicializados com 0 (ou `NULL` caso a variável seja um ponteiro).

O Bloco de Código 13 apresenta dois exemplos de como inicializar uma variável do tipo `struct`. Além destas formas, é possível inicializar cada uma das variáveis que compõem o structure de forma individual.

Bloco de código 13: Exemplo de inicialização de variáveis do tipo `struct`

```c
#include <stdio.h>

struct employee {
  char firstName[20];
  char lastName[30];
  unsigned int age;
  float hourlySalary;
};

void printEmployeeData(struct employee emp);

int main() {
  struct employee fulano = { "Fulano", "de Tal", 45, 30.5 };
  printEmployeeData(fulano);

  struct employee beltrano = beltrano;
  printEmployeeData(fulano);

  return 0;
}

void printEmployeeData(struct employee emp) {
  printf("Full Name: %s %s\nAge: %d\nSalary: %.2f\n",
    emp.firstName, emp.lastName, emp.age, emp.hourlySalary);
}
```

O resultado da execução do programa exibido pelo Bloco de Código 13 é apresentado abaixo e o código está disponível neste link.

Resultado

```
Full Name: Fulano de Tal
Age: 45
Salary: 30.50
Full Name: Fulano de Tal
Age: 45
Salary: 30.50
```

7.4 Acessando membros de structures com os operadores . e ->

Dois operadores são usados para acessar membros de estruturas:

. *structure member operator*;

-> *structure pointer operator* (também chamado de *arrow operator*).

O primeiro operador (`.`) acessa um membro do structure a partir do nome da variável do tipo `struct`. Por exemplo, para exibir na tela o conteúdo da variável membro `suit` do structure `aCard`, devemos fazê-lo conforme no exemplo abaixo.

```
printf("%s", aCard.suit);
```

Já o segundo operador (`->`) fornece acesso ao conteúdo de uma variável através de um ponteiro para um structure. Para ilustrar seu uso, considere o exemplo abaixo.

```
struct card *cardPtr = &aCard;
printf("%s", cardPtr->suit);
```

No exemplo acima, o ponteiro `cardPtr` é declarado como sendo do tipo `struct card *` e é inicializado com o endereço da variável `aCard`.

O Bloco de Código 14 apresenta um exemplo demonstrando o uso dos operadores `.` e `->`.

Bloco de código 14: Exemplo de uso dos operadores `.` e `->`.

```
1  #include <stdio.h>
2
3  struct card {
4    char *face;
5    char *suit;
6  };
7
8  int main() {
9    struct card aCard;
10
11   aCard.face = "Ace";
12   aCard.suit = "Spades";
13
14   struct card *cardPtr = &aCard;
15
16   printf("%s%s%s\n%s%s%s\n%s%s%s\n",
17     aCard.face, " of ", aCard.suit,
18     cardPtr->face, " of ", cardPtr->suit,
19     (*cardPtr).face, " of ", (*cardPtr).suit);
20
21   return 0;
22 }
```

O resultado da execução do programa exibido pelo Bloco de Código 14 é apresentado abaixo e o código está disponível neste link.

Resultado

```
Ace of Spades
Ace of Spades
Ace of Spades
```

7.5 typedef

A palavra-chave `typedef` fornece um mecanismo para criar sinônimos (também chamados de *aliases*) para tipos de dados definidos anteriormente. Nomes para novos tipos de dados criados a partir de structures são frequentemente definidos com `typedef`.

```
typedef struct card Card;
```

No exemplo acima, criamos um novo tipo de dados, chamado `Card`, como um sinônimo para o tipo `struct card`. Frequentemente utilizamos `typedef` para definir um novo tipo de structure, conforme apresentado pelo exemplo abaixo.

```
typedef struct {
  char *face;
  char *suit;
} Card;
```

A partir de agora, `Card`, pode ser utilizada para declarar variáveis do tipo `struct card`, como exemplificado abaixo.

```
Card deck[52];
```

No exemplo acima, um array com 52 elementos do tipo `struct card` foi criado. Criar um novo nome utilizando `typedef` não cria um novo tipo de dados. `typedef` simplesmente cria um novo nome para um tipo de dados, o qual pode ser usado como um *alias* para um tipo de dados existente.

Referências

[1] H. Agarwal, "Why learning c programming is a must?." https://www.geeksforgeeks.org/why-learning-c-programming-is-a-must. Último acesso em: 21/02/2024.

[2] K. Capuzzo, "What is c programming?." https://blog.qwasar.io/blog/why-you-should-learn-c-programming. Último acesso em: 21/02/2024.

[3] H. J. de St. Germain, "The c programming language." https://www.cs.utah.edu/~germain/PPS/Topics/C_Language/the_C_language.html. Último acesso em: 21/02/2024.

[4] H. M. Deitel and P. J. Deitel, *C How to Program*. Londres: Pearson Education Inc, 8 ed., 2016.