LINGUAGEM DE PROGRAMAÇÃO

AULA 1

Prof. Sandro de Araujo

CONVERSA INICIAL

Esta aula tem como base o livro *Treinamento em Linguagem C*, do autor Victorine Viviane Mizrahi, livro que pode ser acessado em nossa **Biblioteca Virtual Pearson**. Não deixe de revisá-lo e ou consultá-lo em caso de dúvidas.

A aula apresenta a seguinte estrutura de conteúdo:

1. Compilador;
2. Estrutura de um programa em C;
3. Função *main*;
4. Principais características de uma função;
5. Pré-processador e diretivas.

O objetivo desta aula é introduzir os principais conceitos e temas das abordagens sobre compiladores; a estrutura de um programa em linguagem de programação C; a função *main*; as principais características de uma função; e o pré-processador e diretivas a serem utilizadas nesta disciplina.

TEMA 1 – COMPILADOR

Criado na década de 1950, o nome **compilador** se refere ao processo de tradução da linguagem de programação para linguagem de máquina (de uma linguagem fonte para uma linguagem que o computador entenda). No entanto, a tradução de uma linguagem fonte não é a única função do compilado; ele também reporta ao seu usuário a presença de erros no programa origem (Mizrahi, 2008).

Definido em Aho et al. (1995), um compilador consiste em um programa que lê outro programa escrito em uma linguagem – a linguagem de origem – e a traduz em um programa equivalente em outra linguagem – a linguagem de destino. Sendo importante no processo de tradução, o compilador reporta ao seu usuário a presença de erros no programa origem.

Ao longo dos anos 1950, os compiladores foram considerados programas notoriamente difíceis de escrever. O primeiro compilador Fortran, por exemplo, consumiu 18 homens/ano para ser implementado (Backus, 1957). Desde então, foram descobertas técnicas sistemáticas para o tratamento de muitas das mais importantes tarefas desenvolvidas por um compilador.

A variedade de compiladores nos dias de hoje é muito grande. Existem inúmeras linguagens fontes, que poderiam ser citadas em várias páginas deste trabalho. Isso se deve, principalmente, ao fato de que, com o aumento do uso dos computadores, aumentaram também as necessidades de cada indivíduo, sendo essas específicas, exigindo, por sua vez, linguagens de programação diferentes.

Este processo, com a evolução da tecnologia de desenvolvimento de compiladores, levou à criação de várias técnicas para a construção de um compilador, ou seja, passaram a existir diferentes maneiras de se implementar um compilador.

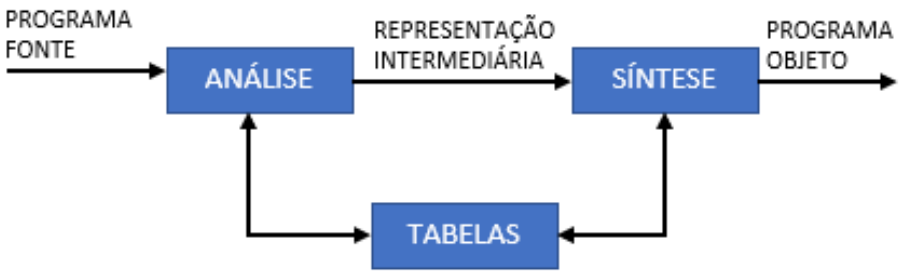
No entanto, a despeito dessa aparente complexidade, as tarefas básicas que qualquer compilador precisa realizar são essencialmente as mesmas. A grande maioria dos compiladores de hoje faz uso da técnica chamada tradução dirigida pela sintaxe. Nessa técnica, as regras de construção do programa fonte são utilizadas para guiar todo o processo de compilação. Algumas das técnicas mais antigas utilizadas na construção dos primeiros compiladores (da linguagem Fortran) podem ser obtidas em Rosen (1967).

Existem duas tarefas principais executadas por um compilador no processo de tradução:

1. **Análise** – momento em que o texto de entrada (código escrito na linguagem de programação) é examinado, verificado e compreendido;
2. **Síntese** (ou geração de código) – momento em que o texto de saída (texto objeto) é gerado.

A análise cria uma representação intermediária desse texto, e a síntese constrói o texto objeto (programa objeto) por meio da representação intermediária criada pela análise, conforme mostrado na Figura 1.

Figura 1 – Etapas do processo de tradução



A análise retorna como uma representação intermediária do código fonte. Sendo assim, deve conter as informações necessárias para a geração do código objeto que o corresponda. Quase sempre, essa representação tem como complemento tabelas com informações adicionais sobre o programa fonte. Existem casos em que a representação intermediária toma a forma de um programa em uma linguagem intermediária, tornando mais fácil a tradução para a linguagem objeto desejada (Mizrahi, 2008).

Uma das formas mais comuns de tabela utilizada na representação intermediária é a tabela de símbolos (*tokens*). A função dessa tabela é guardar cada token usado no programa na informação correspondente.

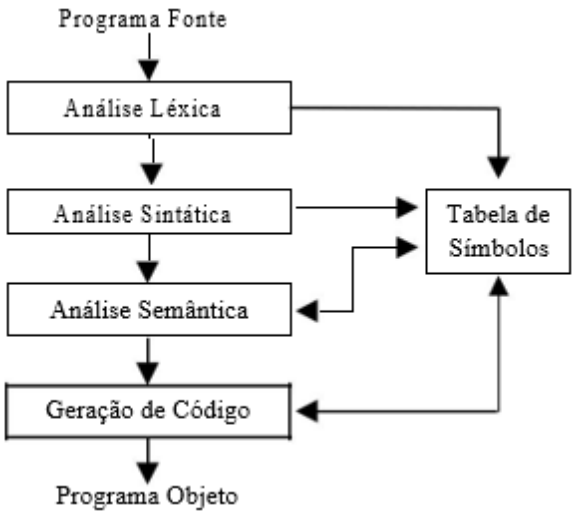
1.1 Componentes de um compilador

Geralmente, um compilador é dividido em quatro fases. São elas: 1) analisador léxico; 2) analisador semântico; 3) analisador sintático; 4) geração/otimização do código. Cada uma dessas partes tem uma função específica, conforme mostrado abaixo:

- **Analisador léxico** – separa cada símbolo do código fonte que tenha algum significado para a linguagem, ou avisa quando encontra um símbolo que não faz parte da linguagem.
- **Analisador semântico** – verifica se os aspectos semânticos estão corretos, ou seja, se não existem incoerências quanto ao significado das construções utilizadas pelo programador. A análise semântica depende de uma tabela de símbolos em que são armazenadas as informações de variáveis declaradas, funções, entre outros.
- **Analisador sintático** – é responsável por verificar se a sequência de símbolos existentes no programa fonte forma um programa válido ou não. É construído sobre uma gramática composta de uma série de regras que descrevem as construções válidas da linguagem.
- **Geração e otimização de código** – após a verificação da não existência de erros sintáticos ou semânticos, o compilador realizará a tarefa de criar o código objeto correspondente, mediante instruções de baixo nível. Nessa etapa, também pode realizar otimizações no código, em que serão aplicadas diversas técnicas para otimizar algumas características do código objeto, como tamanho ou velocidade.

Essas fases são conceituais e especificam atividades que todos os compiladores executam, embora frequentemente as atividades de várias fases possam ser combinadas e executadas simultaneamente. Essas fases estão resumidamente ilustradas na Figura 2.

Figura 2 – Fases das atividades de um compilador



1.2 Os quatro estágios da compilação de um programa em C

Saber como funciona a compilação pode ser muito útil na hora de escrever um código e ao depurá-lo. Compilar um programa C é um processo que pode ser dividido em quatro estágios separados: pré-processamento, compilação, montagem e linkagem.

Para melhor entendimento, vamos percorrer cada um dos quatro estágios de compilação do seguinte programa em C, mostrado na Figura 3.

Figura 3 – Arquivo helloWorld.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      printf("Hello world!\n");
7      return 0;
8  }
9
```

1.2.1 Pré-processamento

O primeiro estágio da compilação é chamado de pré-processamento. Nele, as linhas que começam com um #caractere são interpretadas pelo pré-processador como comandos. Esses comandos formam uma linguagem macro simples com sua própria sintaxe e semântica (Mizrahi, 2008).

Essa linguagem é usada para reduzir a repetição no código-fonte, fornecendo funcionalidade para arquivos embutidos, definindo macros e omitindo o código condicionalmente.

A primeira fase da tradução de código fonte para código de máquina inclui:

- Remoção de comentários;
- Expansão de macros;
- Expansão dos arquivos incluídos.

Antes de interpretar os comandos, o pré-processador retira os comentários e produz o conteúdo da biblioteca stdio.h e stdlib.h, que está no cabeçalho do nosso código, que será associado ao conteúdo do helloWorld.c.

Os comentários fornecem uma forma de incluir um texto descritivo nos códigos, facilitando a compreensão do programa por outras pessoas que fizeram parte de seu desenvolvimento.

Vantagens de se comentar um programa:

- Facilidade na organização e escrita do algoritmo.
- Auxilia a detecção e correção de erros.

Os comentários operam, na prática, como uma documentação interna dos sistemas. Sua utilização é considerada de grande importância.

Um comentário em C consiste em uma sequência de caracteres que começa com uma barra e um asterisco (/*) e termina com um asterisco e uma barra (*/). Os comentários são tratados pelo compilador como um espaço em branco, isto é, são ignorados. Um comentário pode incluir qualquer grupo de caracteres, incluindo mudanças de linha, exceto o indicador de fim de comentário, ou seja /*. Em virtude deste fato, os comentários descritos não podem ser encadeados (um comentário dentro de outro comentário) (Intprogc, 2018).

Outra forma de formar comentários em C consiste em escrever duas barras (//) na coluna em que queremos iniciar o comentário. Todos os caracteres, até ao final dessa linha, serão considerados comentários (Intprogc, 2018).

Os comentários podem aparecer em qualquer ponto do programa em que seja permitida a utilização de um espaço em branco.

Os exemplos seguintes ilustram algumas das formas possíveis de um comentário.

```
// Isto é um exemplo de um comentário uma única linha.
/*
Isto é outro exemplo
de um comentário
que ocupa mais de
uma linha
*/
```

1.2.2 Compilação

Nesse estágio, o código pré-processado é traduzido para instruções de montagem específicas da arquitetura do processador de destino. Alguns compiladores também suportam o uso de um montador integrado, no qual o estágio de compilação gera diretamente o código de máquina, evitando a sobrecarga de gerar as instruções de montagem intermediárias e invocando o montador (Mizrahi, 2008).

Nessa fase, é gerado um arquivo de saída intermediário, o helloWorld.s, arquivo já preparado com as instruções no nível da montagem.

1.2.3 Montagem

Nessa fase, o helloWorld.s é tomado como entrada e transformado em helloWorld.o pelo montador, já com instruções no nível da máquina (Mizrahi, 2008). Somente o código existente é convertido em linguagem de máquina, as chamadas de função, como printf (), não são resolvidas nesse momento, e sim na próxima fase, na linkagem ou vinculação.

1.2.4 Linkagem

Essa é a fase final, em que todas as chamadas de função e suas definições são resolvidas (Mizrahi, 2008). O *Linker* sabe onde todas essas funções são implementadas e, caso tenha algum código extra, também será adicionado ao nosso programa.

Para produzir um programa executável, as peças existentes precisam ser reorganizadas; as que faltam, são preenchidas. No caso do programa "Hello, World!", O vinculador adicionará o código do objeto para a função printf ().

TEMA 2 – ESTRUTURA DE UM PROGRAMA EM C

Todos os computadores suportam alguma linguagem nativa que especifica um conjunto de instruções para serem executadas diretamente. Traduzida para representação binária, escrever o algoritmo necessário para resolver o problema computacional, nesta linguagem, por um programador humano, é um tanto fastidiosa e sujeita a erros (Mizrahi, 2008).

Desde o desenvolvimento dos primeiros computadores, foram feitas tentativas de tornar o processo de programação mais simples, por meio da redução do conhecimento das características internas do computador, necessárias para a escrita de programas. Daí a necessidade de criar sistemas em uma linguagem mais compreensível para o humano (Mizrahi, 2008).

As linguagens de programação de alto nível realizadas pelos humanos, em suas línguas nativas, possibilitam a especificação de soluções em conceitos mais próximos dos empregados. Essas linguagens foram criadas para tornar a criação de algoritmos mais simples, sem ter a necessidade de os programadores conhecerem a estrutura e o funcionamento interno um máquina.

A linguagem de Programação C é de alto nível e apresenta sintaxe estruturada e flexível. Com a linguagem de Programação C, são criados programas compilados, gerando programas executáveis.

A linguagem de Programação C tem estrutura simples e gera códigos mais enxutos e velozes se comparada a outras linguagens, pois permite a inclusão de uma farta quantidade de rotinas.

A linguagem C foi criada por Dennis Ritchie, nos laboratórios Bell, em 1972. Hoje, é uma das linguagens mais utilizada no mundo (Mizrahi, 2008).

Um código fonte em C pode ser formado por um ou mais arquivos fonte. Em um arquivo fonte, encontramos declarações e definições de funções e identificadores. Tais declarações e definições podem estar contidas em arquivos fonte, arquivos cabeçalho, bibliotecas e outros arquivos necessários ao programa. Para se obter um arquivo executável, é necessário compilar cada um dos arquivos do projeto e gerar arquivos objeto destes.

Um programa em C é constituído de:

- Um cabeçalho – que contém inclusão de bibliotecas, as diretivas de compilador onde se define o valor de constantes simbólicas, a declaração de variáveis, a declaração de funções, entre outros.
- Um bloco principal de instruções e outros blocos de rotinas.
- Documentação do programa em forma de comentários.

Os comentários podem ser escritos em qualquer parte do algoritmo. Para que seja identificado como tal, ele deve ter um /* antes e um */ depois.

A seguir, veremos, na Figura 4, um programa em C que vai mostrar no monitor a frase “Olá Alunos!”:

Figura 4 – Código C olaAlunos.c

```
1  #include <stdio.h>
2
3  void main()
4  {
5      printf("Olá Alunos!");
6
7  }
8
```

A primeira linha do programa #include <stdio.h> informa ao compilador a biblioteca que deve incluir à biblioteca stdio (standard input/output) no programa (Intprogc, 2018).

Na segunda linha, foi declarada a única função, a função *main*, e nessa existe apenas uma única instrução, que é a função printf() na linha 4 (disponível na biblioteca stdio.h da linguagem C) para escrever uma mensagem no monitor.

As palavras *include*, *int*, pertencem ao léxico do C. São palavras com significado especial na linguagem C.

Algumas palavras apresentam um significado especial para o compilador e, portanto, não podem ser utilizadas para descrever entidades definidas pelo programador. A linguagem C utiliza um conjunto extenso de palavras-chave em

relação à linguagem C (Intprogc, 2018). No entanto, as seguintes palavras-chave, um subconjunto das palavras chave utilizadas pela linguagem C, são as únicas que utilizaremos, conforme mostrado na Tabela 1.

Tabela 1 – Palavras reservadas da linguagem C

auto	break	case	char
continue	default	do	double
else	enum	extern	false
float	for	goto	if
int	struct	long	register
return	short	signed	sizeof
static	struct	switch	typedef
union	unsigned	struct	void
volatile	while		

Fonte: Intprogc, 2018.

TEMA 3 – A FUNÇÃO *MAIN*

A função *main* serve como o ponto de partida para a execução do programa. Em geral, ela controla a execução, direcionando as chamadas para outras funções no programa. Normalmente, um programa para de ser executado no final de *main*, embora possa ser encerrado precocemente em outros pontos por diversos motivos.

Portanto, a função *main* é a primeira função de um programa C a ser executada e a única que não precisa ser explicitamente chamada.

A seguir, os formatos usados na função *main*:

- int main()
- int main(void)
- int main(int argc, char * argv[ ])
- int main(int argc, char * const argv[ ], char * const envp[ ])

Os formatos int main( ) e int main(void) são usados quando nenhum argumento é passado ao programa. Por sua vez, os formatos int main(int argc, char * argv[ ]) e int main(int argc, char * const argv[ ], char * const envp[ ]) fornecem, respectivamente, dois e três argumentos. Os argumentos, quando declarados, são sempre os seguintes:

- **argc** – é o contador de argumentos. Ele informa quantos argumentos foram passados juntos com o nome do programa. Se o valor é 1, então argumentos não foram fornecidos com o nome do programa. Observe as figuras 5 e 6.

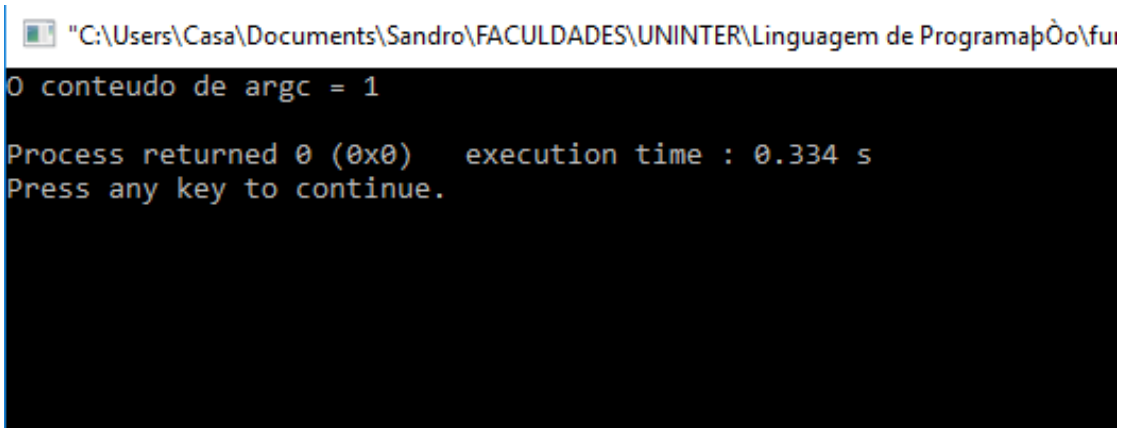
Figura 5 – Algoritmo que imprime o conteúdo de argc

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

int main(int argc, char * const argv[ ], char * const envp[ ])
{
    printf("O conteudo de argc = %d\n", argc);

    return 0;
}
```

Figura 6 – Resultado da execução do algoritmo da figura 5



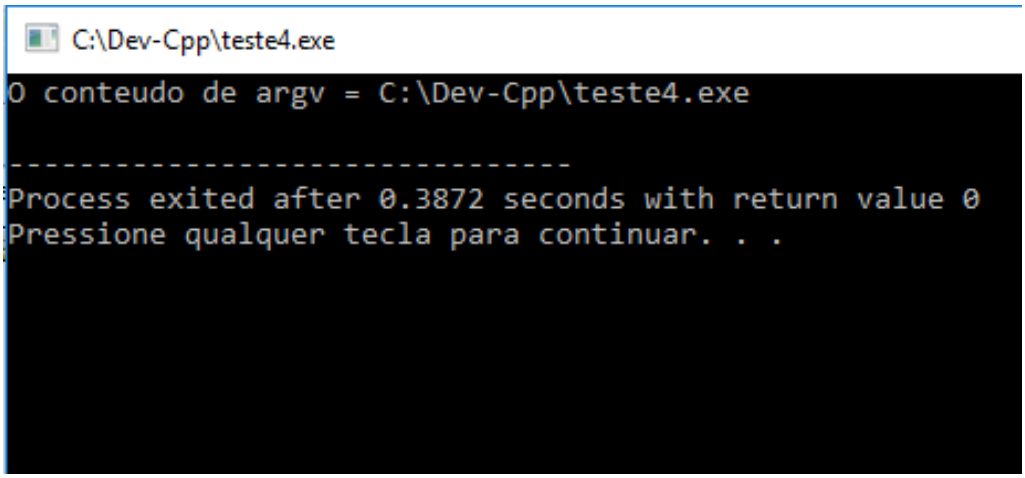
- **argv** – o nome do programa é armazenado em argv[0]. Observe as figuras 7 e 8.

Figura 7 – Algoritmo que imprime o conteúdo de argv

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

int main(int argc, char * const argv[ ], char * const envp[ ])
{
    printf("O conteudo de argv = %s\n", argv[0]);
    return 0;
}
```

Figura 8 – Resultado da execução do algoritmo da figura 7



A Figura 8 traz como resultado o endereço onde foi armazenado o arquivo executável **teste4.exe** após a compilação do algoritmo. Nesse exemplo, o endereço apresentado foi:

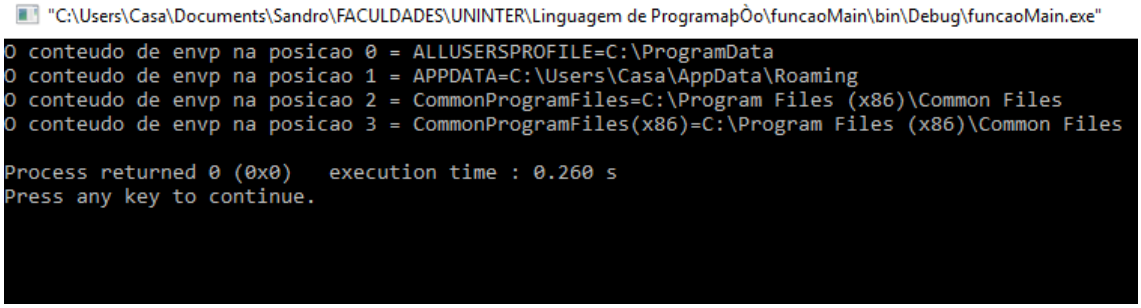
C:\Dev-Cpp\teste4.exe

- **envp** – apresenta informações sobre o ambiente do processo. Observe as figuras 9 e 10.

Figura 9 – Algoritmo que imprime o conteúdo de envp em diferentes posições no vetor

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(int argc, char * const argv[ ], char * const envp[ ])
5   {
6       printf("O conteudo de envp na posicao 0 = %s\n", envp[0]);
7       printf("O conteudo de envp na posicao 1 = %s\n", envp[1]);
8       printf("O conteudo de envp na posicao 2 = %s\n", envp[2]);
9       printf("O conteudo de envp na posicao 3 = %s\n", envp[3]);
10      return 0;
11  }
```

Figura 10 – Resultado da execução do algoritmo da figura 9



O ambiente do processo é substituído pelo parâmetro envp, sendo possível guardar vários processos em diferentes posições no vetor.

Por padrão, os nomes argc, argv[ ] e envp[ ] são sempre usados. Entretanto, não há problema em você usar outros nomes (ex: contador, argumentos[ ] e ambiente[ ]), desde que você não altere os tipos de dados.

A função *main* pode retornar um inteiro para o sistema operacional. Para isso, utiliza-se o **int main()**, em conjunto com o comando *return*, para encerrar a função e passar um inteiro para o sistema operacional, conforme mostrado no exemplo da Figura 11.

Figura 11 – Exemplo função *main*

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main()
5   {
6       printf("Ola Mundo!\n");
7       printf("Olha o meu algoritmo!!!! \0/\0/\0/\n");
8
9       system("pause");
10      return 0;
11  }
12
```

O retorno zero indica ao Sistema Operacional que o programa foi bem-sucedido. Cada outro número retornado indica o código de uma condição de erro (Mizrahi, 2008). Portanto, lembre-se sempre de colocar o retorno. Isso é uma boa prática de programação.

TEMA 4 – PRINCIPAIS CARACTERÍSTICAS DE UMA FUNÇÃO

Toda função é declarada com uma identificação e parênteses após seu nome. No exemplo main(), a função é identificada com o nome "*main*" e acompanhada de "(" e ")". Essas características que permitem que o compilador saiba que se trata de uma função. Sem os parênteses, o compilador pode tratar o nome como se fosse uma variável, ocasionando um erro, uma vez que a palavra *main* é reservada pelo compilador (Mizrahi, 2008).

Após a chamada da função, vem o bloco de código. Toda função em linguagem de programação C delimita o bloco com chaves. Começar com uma

chave de abertura de bloco ({) e terminar com uma chave de fechamento de bloco (}). Essas chaves determinam o corpo da função.

Em uma função, é permitido inserir espaços em branco, tabulações e pular linhas (esses caracteres são ignorados pelo compilador). É permitido também escrever várias instruções em uma única linha ou escrever uma instrução em várias linhas. Não há um estilo obrigatório para a escrita de programas em C. No entanto, é sempre desejável seguir as boas práticas da programação com indentações sem exagero (Mizrahi, 2008).

Quando desenvolvemos um projeto de desenvolvimento de software, é importante que todo o código seja bem indentado, ou seja alinhando de forma correta.

Em um grande projeto, é importante a definição de um padrão de indentação, que deve ser documentado e disponibilizado aos demais programadores.

A palavra **indentação** é um neologismo e não existe na língua portuguesa. Esse termo foi "abrasileirado" do termo *indentation*, usado na língua inglesa, que significa recuo.

O objetivo da indentação é o de tornar os códigos mais legíveis. A utilização de regras de indentação permite, por meio de uma simples inspeção visual, evidenciar facilmente a estrutura global do programa, tornando claramente visíveis todas as instruções de um ou mais blocos de código.

Nos exemplos das figuras 12, 13, 14, 15, 16 e 17, a seguir, são ilustrados os dois estilos de indentação mais utilizados. O número de espaços utilizado pode variar entre 2 e 8. Escolha o estilo que mais lhe agrada e utilize de forma uniforme em todos os seus programas.

Figura 12 – Indentação IF

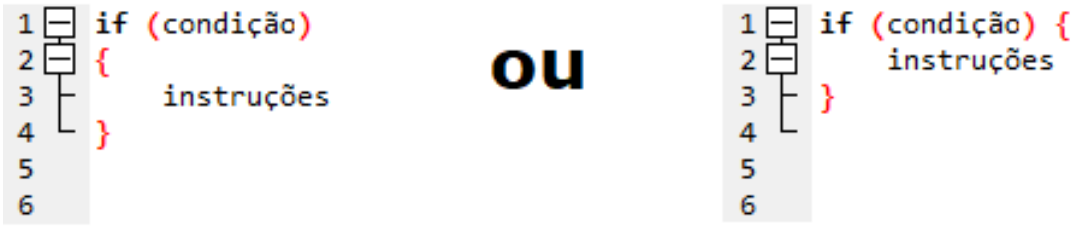


Figura 13 – Indentação IF, ELSE IF, ELSE

```
if (condição)
{
    instruções
}
else if (condição)
{
    instruções
}
else
{
    instruções
}
```

ou

```
if (condição) {
    instruções
} else if (condição) {
    instruções
} else {
    instruções
}
```

Figura 14 – Indentação FOR

```
for (inicialização; teste; incremento)
{
    instruções
}
```

ou

```
for (inicialização; teste; incremento) {
    instruções
}
```

Figura 15 – Indentação WHILE

```
while (condição)
{
    instruções
}
```

ou

```
while (condição) {
    instruções
}
```

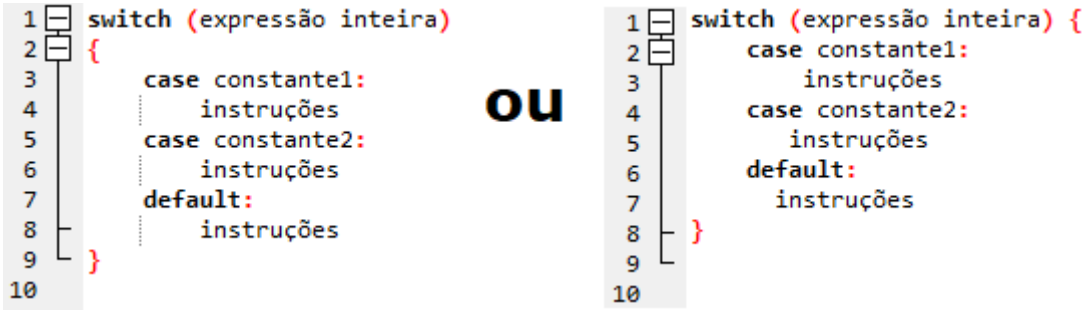
Figura 16 – Indentação DO WHILE

```
do
{
    instruções
}
while (condição);
```

ou

```
do {
    instruções
} while (condição);
```

Figura 17 – Indentação SWITCH CASE



TEMA 5 – PRÉ-PROCESSADOR E DIRETIVAS

As primeiras linhas de um programa não são instruções da linguagem C (observe que não há ponto-e-vírgula ao seu final), mas sim diretivas do pré-processador.

Um pré-processador executa um conjunto de processos preliminares sobre os arquivos fonte antes de estes serem fornecidos para o compilador (Mizrahi, 2008).

O pré-processador é um programa que examina o programa fonte em C e executa certas modificações com base em instruções, as quais chamamos de diretivas. Toda diretiva é iniciada pelo símbolo (#), código especial, e o seu texto deve ser escrito em uma única linha. Como descrito no capítulo 1.2, as diretivas do pré-processador não fazem parte da linguagem C, elas servem para auxiliar no desenvolvimento do código fonte.

Uma diretiva ensina o pré-processador no sentido de efetuar determinada ação sobre o texto do programa antes da compilação.

Na linguagem de programação C, existem comandos processados durante a compilação do programa, conhecidos como diretivas de compilação. Esses comandos informam ao compilador as constantes simbólicas usadas na linguagem de programação e as bibliotecas que devem ser anexadas ao programa compilado (Mizrahi, 2008). Observe os exemplos mostrados abaixo:

- A diretiva ***#include*** diz ao compilador para incluir na compilação do programa outros arquivos. Geralmente estes arquivos contém bibliotecas de funções ou rotinas do usuário.
- A diretiva ***#define*** diz ao compilador quais são as constantes simbólicas que serão usadas no programa.

O código da Figura 18 mostra o uso das diretivas *#include* e *#define*.

Figura 18 – Código C com diretivas

```c
//A diretiva (# include) inclui a biblioteca stdio.h
#include <stdio.h>
//A diretiva (# include) inclui a biblioteca stdlib.h
#include <stdlib.h>
//A diretiva (# define) define a constante
#define QUANTIDADE_MAXIMA 15

int main()
{

    int quantidade;
    printf ("Digite um numero para a quantidade desejada: \n");
    scanf("%d",&quantidade);
    printf("A quantidade maxima: %d\n", QUANTIDADE_MAXIMA);
    printf("A quantidade digitada: %d\n", quantidade);
    return 0;

}
```

No código da Figura 18, a constante QUANTIDADE_MAXIMA será executada antes da compilação do texto escrito pelo programador. A diretiva #include provoca a inclusão das bibliotecas *stdio.h* e *stdlib.h* em nosso programa fonte, não interferindo no tempo de execução da programação, mas impactando no tempo de compilação.

Na verdade, o compilador substitui a linha que contém essa diretiva pelo conteúdo do arquivo indicado. Essa substituição também é executada antes de o programa ser compilado. Assim, o efeito obtido é a apresentação de um texto, como se tivéssemos digitado todo o conteúdo de arquivo *stdio.h* e do arquivo *stdlib.h* na posição em que escrevemos as linhas:

Primeiro o ***#include <stdio.h>*** seguido do ***#include <stdio.h>***.

O arquivo *stdlib.h* contém as definições e declarações necessárias para o uso da função *printf*(), já o *stdlib.h* contém as declarações necessárias para o uso da função *system().*

A diretiva #include aceita uma segunda sintaxe:

#include "meuarq.h".

Quando usamos os sinais < e >, o arquivo é procurado somente na pasta include, criada na instalação do seu compilador. Quando usamos " e " (aspas duplas), o arquivo é procurado primeiramente na pasta atual e, depois, se não for encontrado, na pasta include.

Embora a linguagem C disponibilize um grande conjunto de diretivas, que estão enunciadas na tabela seguinte, no âmbito deste curso, utilizaremos fundamentalmente a diretiva #include (Intprogc, 2018).

Tabela 2 – Diretivas em C

#define	#endif	#ifdef	#line
#elif	#error	#ifndef	#pragma
#else	#if	#include	#undef

Fonte: Intprogc, 2018.

5.1 Códigos especiais

Além do comando #, existem vários outros caracteres usados para auxiliar o programador em seu código. A barra invertida (****) é um comando usado no momento em que o programador precisa digitar algo que não pode ser digitados diretamente do nosso teclado. Exemplo: [ENTER], uma quebra de linha, a tabulação, entre outros. Esses caracteres são codificados em C por meio da combinação do sinal \ (barra invertida) com outros caracteres. Observe a Tabela 3.

Tabela 3 – Códigos especiais

CÓDIGOS ESPECIAIS	SIGNIFICADO
\n	Nova linha
\t	Tabulação
\b	Retrocesso (usado para impressora)
\f	Salto de página de formulário
\a	Beep – Toque do alto-falante
\r	CR – Retorno do cursor p/ o início da linha
\\	\ – Barra invertida
\0	Zero
\’	Aspas simples (apóstrofo)
\”	Aspas dupla
\xdd	Representação hexadecimal
\ddd	Representação octal

Fonte: Adaptado de Mizrahi, 2008.

O exemplo da Figura 19 traz um algoritmo com os códigos especiais \t e o \n.

Figura 19 – Algoritmo com códigos especiais \t e \n

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

int main() {

    printf("Texto com \t tab com \n quebra de linha ");
    return 0;
}

```

Figura 20 – Resultado da execução do algoritmo da figura 19

```
Texto com        tab com
 quebra de linha
--------------------------------
Process exited after 0.3271 seconds with return value 0
Pressione qualquer tecla para continuar. . .
```

O exemplo da Figura 21 traz o mesmo algoritmo da Figura 19 sem os códigos especiais \t e o \n. Analise os dois resultados e veja as diferenças.

Figura 21 – Algoritmo sem códigos especiais \t e \n

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

int main() {

    printf("Texto sem tab e sem quebra de linha ");
    return 0;
}
```

Figura 22 – Resultado da execução do algoritmo da figura 21

```
Texto sem tab e sem quebra de linha
--------------------------------
Process exited after 0.3569 seconds with return value 0
Pressione qualquer tecla para continuar. . .
```

Portanto, na linguagem de programação C, sempre que aparecer, em um conjunto de caracteres (texto literal no código), uma barra invertida ('\') ou um porcento ('%'), será um comando. Vejamos na Tabela 4 alguns comandos mais usados na função printf().

Tabela 4 – Códigos para impressão formatada com printf().

Códigos de impressão formatada com printf().	**SIGNIFICADO**
%c	Caractere simples
%d	Inteiro decimal com sinal
%i	Inteiro decimal com sinal
%e	Notação científica (e minúsculo)
%E	Notação científica (e maiúsculo)
%f	Ponto flutuante em decimal

Fonte: Adaptado de Mizrahi, 2008.

FINALIZANDO

Nesta aula, aprendemos o conceito de compiladores, a estrutura de um programa em linguagem de programação C, a função *main*, as principais características de uma função e o pré-processador e diretivas estudadas nesta aula.

REFERÊNCIAS

AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compilers, Principles, Techniques and Tools**. Company Reading, Massachusetts, 1995.

BACKUS, J. W. The FORTRAN Automatic Coding System. **Western joint computer conference**: techniques for reliability. Los Angeles, 1957.

INTPROGC. **Programação em C**. Disponível em: <http://intprogc.pbworks.com/w/page/11211411/FrontPage>. Acesso em: 5 out. 2018.

MIZRAHI, V. V. **Treinamento em Linguagem C**. 2. ed. São Paulo: Pearson, 2008.

RANGEL, J. L. **Compiladores**. Rio de Janeiro: PUC-Rio, 1999.

ROSEN, S. **Programming Systems and Languages**. USA, 1999.