1.Algoritmos e pseudocódigos

A solução de qualquer problema computacional envolve a execução de uma série de ações segundo uma ordem específica. Um procedimento para resolver o problema em termos de:

1. As ações a serem executadas e;
2. a ordem em que essas ações devem ser executadas

Isto é chamado de ***algoritmo***

Pseudocódigo é uma linguagem artificial e informal que ajuda os programadores a desenvolver algoritmos. Pseudocódigo é similar à linguagem do dia-a-dia. Ela é convenientemente e amigável, embora seja uma linguagem real de programação.

Algoritmo 1 Exemplo de Pseudocódigo.

```
leia (x, y) {Esta linha é um comentário}
se x > y então
    escreva ("x é maior")
senão
    se y > x então
        escreva ("y é maior")
    senão
        escreva ("x e y são iguais")
    fim-se
fim-se
```

Exemplo de pseudocódigo

Os programas em ***pseudocódigos*** não são na verdade executados em computadores. Em vez disso, ele simplesmente ajuda o programador a "pensar" em um programa antes de tentar escrevê-lo em uma linguagem de programação como o C. O pseudocódigo consiste exclusivamente em caracteres. Desse modo, os programadores podem digitar convenientemente programas em pseudocódigo em um computador, utilizando um programa editor de textos. O computador pode exibir ou imprimir uma cópia recente do programa em pseudocódigo quando o usuário desejar. Também consiste apenas em instruções de ação. As declarações não são instruções executáveis. Elas são mensagens para o compilador.

As instruções em um programa são executadas uma após a outra, na ordem em que foram escritas. Isto é chamado de ***execução sequencial***. Várias instruções em C permitem que o programador especifique que a próxima instrução a ser executada seja diferente da próxima na sequência. Isto é chamado de ***transferência de controle***.

Um trabalho de Bohm e Jacopini demonstrou que todos os programas podiam ser escritos em termos de apenas três estruturas de controle: ***estruturas de sequência*** (sequencial), a ***estrutura de seleção*** e a ***estrutura de repetição***.

2.O surgimento das bibliotecas padrão do C

A partir da década de 1980, os problemas de compatibilidade por conta de diferentes bibliotecas se tornaram cada vez mais aparentes. Em 1983 a ANSI formou um comitê para estabelecer uma especificação formal da linguagem conhecida como ANSI C. Esse trabalho culminou na criação padrão C89 em 1989. Parte do padrão que havia surgido era um conjunto de bibliotecas chamado biblioteca padrão do ANSI C. Revisões posteriores do padrão da linguagem C adicionaram diversos novos cabeçalhos e funcionalidades à biblioteca padrão. Entretanto, o suporte para essas novas extensões variam entre implementações. A biblioteca padrão ANSI C consiste de 24 cabeçalhos, cada um contendo uma ou mais declarações de funções, tipos de dados e macros. Esta biblioteca padrão é minúscula. Ela fornece um conjunto básico de operações matemáticas, manipulação de cadeias de caracteres, conversão de tipos de dados e entrada e saída de arquivos e da tela. Não contém um conjunto padrão de containers como a biblioteca padrão do C++, nem suporta interface gráfica do utilizador. A vantagem desse sistema minimalista é que fornecer um ambiente funcional de ANSI C é muito mais simples que em outras linguagens, e, consequentemente, a portabilidade de C entre diferentes plataformas é uma tarefa relativamente simples.

3.A função *main* e o primeiro Hello World

Cada programa C tem uma função principal que precisa ser nomeada como ***main***. Esta função 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. Os parênteses após a palavra indicam que é um bloco de construção do programa chamado ***função***. Os programas em C contêm uma ou mais funções, e uma delas deve ser o ***main***.

```
main(){}
```

O símbolo de chaves deve começar o corpo de todas as funções. Uma chave direita é equivalente e deve terminar cada função. Este par de chaves e a parte do programa entre elas também é chamado um bloco.

Existem algumas restrições que se aplicam à função ***main*** que não se aplicam a nenhuma outra função C. Esta função também não tem uma declaração, pois é incorporada à linguagem.

- Não pode ser declarada como ***inline***
- Não pode ser declarada como ***static***
- Não pode ter seu endereço usado
- Não pode ser chamada pelo programa

As ***funções*** do programa de origem executam uma ou mais tarefas específicas. A função ***main*** pode chamar outras funções para executar respectivas tarefas. Quando chamadas, ***main*** passa para elas o controle da execução, assim, a execução começa na primeira instrução da

função. Uma função devolve o controle a ***main*** quando uma instrução ***return*** é executada ou quando o término da função é atingido.

Um programa geralmente interrompe a execução quando é retornado ou atinge o final de ***main***, embora possa terminar em outros pontos do programa. É possível utilizar o comando ***return*** para encerrar a função ***main*** e passar um inteiro para o sistema operacional. O retorno de 0 indica ao OS que o programa foi bem sucedido. Cada outro número retornado indica o código de uma condição de erro.

```
int main{
  [código]
  return (0);
}
```

Quando uma função chama outra função, essa função chamada recebe valores para os respectivos **parâmetros** da função que chamou. Esses valores são denominados de **argumentos**. É possível declarar parâmetros formais para ***main***, o que permitirá a definição de argumentos da linha de comando usando o formato mostrado na assinatura da função. O protótipo da função ***main*** depende do sistema operacional subjacente.

Quando informações precisam ser passadas à função ***main***, os parâmetros são tradicionalmente nomeados como ***argv*** e ***argc***, embora o compilador não exija estes nomes. Tradicionalmente, um terceiro parâmetro é passado para ***main***, ele é nomeado ***envp***. Os tipos são definidos pela própria linguagem C.

- ***argc*** (ARGument Count)**:** significa número de parâmetros na linha de comando que lançou a execução.
- ***argv*** (ARGument Vector)**:** significa vetor de strings que contém os argumentos de linha de comando, finalizado por um ponteiro nulo.
- ***envp*****:** significa vetor de strings na forma "nome=valor" que contém as variáveis de ambiente *shell* que lançou a execução do programa. Também finalizado por um ponteiro nulo.

```
main (int argc, char **argv, char **envp);
main (int argc, char **argv[], char **envp);
```

argv[0] possui o nome do próprio programa e ***argv[1]*** é um ponteiro para o primeiro argumento passado durante o começo da execução e ****argv[n]*** é o último argumento. Se nenhum argumento é passado para a função ***main***, ***argc*** terá valor 1 e se passar apenas 1 argumento, ***argc*** tem valor 2.

O programa abaixo imprime na tela os argumentos usados no lançamento do programa

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

```
int main (int argc, char **argv, char **envp){
   int i;
   printf ("\n Numero de parâmetros: %d\n", argc);
   for (i = 0; i < argc; i++){
      printf ("\n argv[%d]: %s\n",  i,  argv[i]);
   }
   return (0);
}
```

Os símbolos // e /**/ são **indicadores de comentários**, que são usados para documentar os programas e melhorar sua legibilidade. Eles não fazem com que o computador realize qualquer ação quando o programa é executado e são ignorados pelo compilador C e não fazem com que seja gerado código-objeto algum.

4.Diretivas pré-processador e introdução à stdio.h

As linhas que começam com # são processadas pelo ***pré-processador*** antes do programa ser compilado. Esta linha em particular diz ao pré-processador para incluir o conteúdo do arquivo de cabeçalho de entrada/saída padrão. Esse arquivo de cabeçalho contém informações e instruções usadas pelo compilador ao compilar funções de entrada/saída da biblioteca padrão. O arquivo de cabeçalho também contém informações que ajudam o compilador a determinar se as chamadas às funções da biblioteca foram escritas corretamente.

No cabeçalho ***stdio.h*** localizam-se as funções referentes às operações nas quais os mecanismos operam em função da entrada e da saída padrão, bem como em arquivos também. Por meio de variáveis, macros e funções, é possível realizar tarefas de leitura e escrita, tanto por meio de entrada quanto por meio de saída, disponibilizada pela máquina. As duas principais funções são ***printf*** e ***scanf***.

5.A função printf()

A função printf recebe uma string que será impressa na saída padrão, as variáveis indicadas na string para a devida formatação. Ela então retorna o número de caracteres escritos, ou, caso a operação não tenha sido bem sucedida, um valor negativo. Ela não faz parte do C padrão. Ela é apenas uma função útil incluída e definida pela padronização ANSI.

```
int printf (const char* format, …); // função definida na biblioteca
printf ("%", variable);  // utilização em um programa
```

Opcionalmente, a função pode conter ***especificações de formato*** que são substituídos por valores específicos em argumentos subsequentes adicionais e formatações específicas. Ela segue o seguinte protótipo: **%[flags][largura][.precisão][comprimento]especificador**

Especificador	Saída	Exemplo

d ou i	Inteiro decimal com sinal	965
u	Inteiro decimal sem sinal	7852
o	Octal sem sinal	610
x	Inteiro hexadecimal sem sinal	7fa
X	Inteiro hexadecimal sem sinal maiúsculo	7FA
f	Ponto flutuante decimal	36.9
e	Notação científica (mantissa/expoente)	3.9265e+2
E	Notação científica (mantissa/expoente) maiúscula	3.9265E+2
a	Ponto flutuante hexadecimal	-0xc.90fep-2
A	Ponto flutuante hexadecimal maiúsculo	-0XC.90FEP-2
c	Caractere	a
s	String de caracteres	amostra
p	Ponteiro de endereço	b8000000
%	Um % seguido por outro % irá mostrar um % na string	%

Flags	Descrição
+	Força a preceder o resultado com um sinal de mais ou menos (+,-) até para números positivos
(space)	Se nenhum sinal for escrito, um espaço em branco será inserido antes do valor
#	Usado com os especificadores **o**, **X** ou **x**, o valor é precedido de 0, ox ou OX para valores diferentes de 0; Usado com **a**, **A**, **e**, **E**, **f**, **F**, **g** ou **G**, força a saída escrita a conter um ponto decimal, mesmo que não haja mais dígitos a seguir
0	Preenche o número à esquerda com zeros em vez de espaço quando o preenchimento é especificado

Largura	Descrição
(número)	Número mínimo de caracteres a serem mostrados. Se o valor a ser impresso for menos que este número, o resultado será preenchido com espaços em branco
*	A largura não é especificada na string de formato, mas como um argumento de valor inteiro adicional que precede o argumento que

	deve ser formatado

.Precisão	Descrição
.número	**Para especificadores inteiros (d, i, o, u, x, X):** a precisão especifica o número mínimo de dígitos a serem escritos. Se o valor a ser escrito for menor que este número, o resultado será preenchido com zeros à esquerda; **Para especificadores a, A, e, E, f e F:** este é o número de dígitos a serem impressos após o ponto decimal (por padrão, é 6); **Para especificadores g e G:** Este é o número máximo de dígitos significativos a serem impressos; **Para s:** este é o número máximo de caracteres a serem impressos. Por padrão, todos os caracteres são impressos até que o caractere nulo final seja encontrado
.*	A precisão não é especificada na string de formato, mas como um argumento de valor inteiro adicional que precede o argumento que deve ser formatado

comprimento	d, i	u, o, x e X	f, F, e, E, g, G, a , A	c	s	p	n
nenhum	int	unsigned int	double	int	char*	void *	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	char_t*		long int*
L			long double				

Quando o compilador compila uma instrução do tipo printf, ele simplesmente abre espaço no programa objeto para uma chamada à função da biblioteca, mas o compilador não sabe onde as funções da biblioteca se encontram, isso é função do *linker*. Assim, ao ser executado, o *linker* localiza as funções da biblioteca e insere as chamadas adequadas a elas no programa objeto. O programa objeto está completo e pronto para ser executado.

Uma variável é uma posição na memória onde um valor pode ser armazenado para ser utilizado por um programa. Todas as variáveis devem ser declaradas com um nome e um tipo de dado imediatamente após a chave esquerda que inicia o corpo de main antes que possam ser usadas em um programa. As declarações devem ser colocadas depois da chave esquerda e antes de qualquer instrução executável.

Um nome de variável em C é qualquer identificador válido. Um identificador é uma série de caracteres que consistem em letras, dígitos e sublinhados que não começa com um dígito. Um

identificador pode ter qualquer comprimento, mas somente os 31 primeiros caracteres serão reconhecidos pelos compiladores C, de acordo com o padrão ANSI C. Ele faz distinção entre letras maiúsculas e minúsculas.

Nomes de variáveis como *inteiro1* e *inteiro2* correspondem realmente a locais na memória do computador. Todas as variáveis possuem um nome, um tipo e um valor.

6.A função scanf()

A função scanf recebe uma string indicando quais os tipos das variáveis que receberão os valores indicados na entrada padrão, as variáveis que tiverem seus tipos indicados na string. Então, retorna **EOF** (***end of file***), caso tenha atingido o fim do arquivo, ou então retorna a quantidade de leituras realizadas.

Quando a função **s**canf é executada, o valor digitado pelo usuário é colocado no local da memória ao qual o nome da variável foi atribuído. Sempre que um valor é colocado em um local da memória, o novo valor invalida o anterior naquele local. Como as instruções anteriores são destruídas, o processo de ler as informações para um local da memória é chamado de leitura destrutiva.

int scanf (const char * format, …); // função definida na biblioteca
scanf (“%”, &); // utilização em um programa

O e-comercial (¨&), quando combinado com o nome da variável, diz a scanf o local na memória onde a variável está armazenada. O computador então armazena o valor da variável naquele local.

Caracteres em branco: A função irá ler e ignorar quaisquer caracteres de espaço em branco encontrados antes do próximo caractere que não seja de espaço em branco

Caracteres não em branco, exceto o especificador (%): Qualquer caractere que não seja um caractere de espaço em branco (em branco, nova linha ou tabulação) ou parte de um especificador de formato faz com que a função leia o próximo caractere do fluxo, compare-o com esse caractere que não seja de espaço em branco e, se corresponder, será descartado e a função continua com o próximo caractere do formato.

Especificador de formato (%): Uma sequência formada por um sinal de porcentagem inicial (%) indica um especificador de formato, que é usado para especificar o tipo e formato dos dados a serem recuperados do fluxo e armazenados nos locais apontados pelos argumentos adicionais.

Segue-se o seguinte protótipo: **%[*][largura][comprimento]especificador**

Especificador	Descrição	Caracteres extraídos
i	Inteiro	Qualquer número de dígitos, precedido opcionalmente pelo sinal (+,-). Números decimais são o *default*, porém, um prefixo 0 indica octais e 0x indica hexadecimais
d(signed) ou u (unsigned)	Inteiro decimal	Qualquer número decimal (0-9)
o	Inteiro octal	Qualquer número octal (0-7). É unsigned
x	Inteiro hexadecimal	Qualquer número de dígitos hexadecimais (0-9, a-f, A-F). É unsigned
f, e, g	Números de ponto flutuante	Série de dígitos decimais, opcionalmente contendo um ponto decimal, opcionalmente precedido por sinal (+,-) e opcionalmente seguido do caractere *e* ou *E*
c	Caractere	O próximo caractere. Se uma largura diferente de 1 for especificada, a função lê exatamente os caracteres de largura e os armazena nos locais sucessivos do array passado como argumento. Nenhum caractere NULL é adicionado no fim
s	String de caracteres	Qualquer número de caracteres que não sejam espaços em branco, parando no primeiro caractere de espaço em branco encontrado. Um caractere NULL de terminação é adicionado automaticamente no final da sequência armazenada
p	Endereço de ponteiro	Uma sequência de caracteres que representa um ponteiro. O formato específico usado depende do sistema e da implementação da biblioteca
[caracteres]	Scanset	Qualquer número de caracteres especificados entre colchetes
[^caracteres]	Scanset negado	Qualquer número de caracteres, nenhum deles especificado como caracteres entre colchetes
n	Count	Nenhuma entrada é consumida
%	%	Um % seguido de outro % corresponde a um único %.

O **especificador** também pode conter **sub-especificadores**

Sub-especificador	Descrição
*	Um asterisco inicial opcional indica arrays a serem ignorados a partir dos dados a serem lidos da entrada
largura	Especifica o número máximo de caracteres a serem lidos da entrada (opcional)
comprimento	Um dos caracteres da tabela abaixo (opcional). Isso altera o tipo a ser armazenado pelo ponteiro no atual argumento usado

comprimento	d e i	u, o, x e X	f, F, e, E, g, G, a e A	c	s	p	n
nenhum	int	unsigned int	double	int	char*	void *	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	char_t *		long int*
L			long double				

Dependendo do formato da ***string***, a função pode esperar uma sequência de argumentos adicionais, cada um contendo um ponteiro para armazenamento alocado onde a interpretação dos caracteres extraídos é armazenada com o tipo apropriado. Estes argumentos são esperados como ponteiros. Para armazenar o resultado de uma operação da função em uma variável regular, o seu nome deve ser precedido pelo operador de referência &.

7.Tipos de variáveis, aritmética e operadores na linguagem C

Na linguagem de programação C, temos os seguintes tipos básicos de variáveis:

Tipo da variável	Modificador	Valor em bits|bytes	Tamanho total
char (caractere)	%c	1 byte | 8 bits	-128 até 127
int (inteiro)	%d	4 bytes | 32 bits	-2.147.483.648 até 2.147.483.647
float (ponto flutuante)	%f	4 bytes | 32 bits	$(+,-)10^{-38}$ até $(+,-)10^{38}$
double (número real com 8 pontos de precisão)	%lf	8 bytes | 64 bits	$(+,-)10^{-308}$ até $(+,-)10^{308}$

Também é possível acrescentar algumas variações do tipo:

- **unsigned:** caractere sem sinal; apenas números positivos
- **long:** variável com domínios estendidos
- **short:** variável com domínios reduzidos

Exemplos das combinações são:

Tipo da variável	Modificador	Valor em bits|bytes	Tamanho total
unsigned char	%c	1 byte | 8 bits	0 até 255
unsigned int	%u	4 bytes | 32 bits	4.294.967.295
long int	%li	4 bytes | 32 bits	-2.147.483.648 até 2.147.483.647
unsigned long int	%lu	4 bytes | 32 bits	0 até 4.297.967.295
short int	%hi	2 bytes | 16 bits	-32.768 até 32.767
unsigned short int	%hu	2 bytes | 16 bits	0 até 65.535
long double	%lf	12 bytes | 96 bits	Muito grande

A maioria dos programas em C realiza cálculos aritméticos. **As expressões aritméticas em C** devem ser escritas no formato linear para facilitar a digitação de programas no computador. Assim, expressões como *a dividido por b* devem ser escritas como *a/b* de forma que todos os operadores e operandos apareçam em uma única linha. O C calcula as expressões aritméticas em uma sequência exata determinada pelas seguintes regras de precedência de operadores, que geralmente são as mesmas utilizadas em álgebra.

1. As expressões ou partes de expressões localizadas entre pares de parênteses são calculadas em primeiro lugar. Desta forma, os parênteses podem ser usados para impor a ordem dos cálculos segundo uma sequência desejada pelo programador.
2. As operações de multiplicação, divisão e resto são calculadas em seguida. Se uma expressão possuir várias operações de multiplicação, divisão e resto, o cálculo é realizado da esquerda para a direita.
3. As operações de adição e subtração são calculadas por último. Se uma expressão possuir várias operações de adição e subtração, os cálculos são realizados da esquerda para a direita. Adição e subtração também estão no mesmo nível de precedência.

As regras de precedência de operadores são diretrizes que permitem ao C calcular expressões na ordem correta. Quando dissemos que os cálculos são realizados da esquerda para a direita, estamos referindo a associatividade de operadores.

Para produzir um cálculo em ponto flutuante com valores inteiros, devemos criar valores temporários que sejam números de ponto flutuante para o cálculo. A linguagem C fornece o operador unário de coerção para realizar essa tarefa. O compilador C só sabe calcular expressões nas quais os tipos de dados dos operandos são idênticos.

Para assegurar que os operadores sejam do mesmo tipo, o compilador realiza uma operação chamada **promoção** em operadores selecionados. Por exemplo, em uma expressão que contenha tipos de dados ***int*** e ***float***, o padrão ANSI específica que sejam feitas cópias dos operandos ***int*** e que elas sejam promovidas a ***float***. Os operadores de conversão estão disponíveis para qualquer tipo de dado. O operador de conversão é formado pela colocação de parênteses em torno do nome de um tipo de dado. O operador de conversão é um operador unário.

Operação em C	Operador aritmético	Expressão algébrica	Expressão em C
Adição	+	f+7	f+7
Subtração	-	p-c	p - c
Multiplicação	*	bm	b*m
Divisão	/	x/y	x/y
Resto	%	r mod s	r%s

Operador	Operação	Ordem de Cálculo
()	Parênteses	Calculado em primeiro lugar. Se houver parênteses aninhados, a expressão dentro do par de parênteses mais interno é calculada em primeiro lugar. No caso de vários pares de parênteses, eles são calculados da esquerda para a direita
*, / ou %	Multiplicação Divisão Resto	Calculado em segundo lugar. No caso de vários operadores, são calculados da esquerda para a direita
+ ou -	Adição ou Subtração	Calculados por último. No caso de vários operadores, eles são calculados da esquerda para a direita

Operador algébrico	Operador de igualdade/relacional	Exemplo de condição em C	Significado da condição
=	==	x == y	x é igual a y
≠	!=	x != y	x é diferente de y
>	>	x > y	x é maior que y

<	<	x < y	x é menor que y
≥	≥	x >= y	x é maior ou igual a y
≤	≤	x <= y	x é menor ou igual a y

Operadores lógicos binários	Operação
&&	AND (E)
||	OR (OU)
!	NOT (NÃO)

Operadores lógicos bit-a-bit	Operação
&	AND (E)
|	OR (OU)
^	XOR (OU EXCLUSIVO)
~	NOT (NÃO)
<<	deslocamento de bits à esquerda
>>	deslocamento de bits à direita

O operador de endereço (&) é usado para obter o endereço de uma variável em C, enquanto os ponteiros (*) são variáveis que armazenam esses endereços. Eles permitem que você acesse e altere o valor de uma variável através de seu endereço.

8.Operadores de atribuição

A linguagem C fornece vários operadores de atribuição para abreviar as expressões de atribuição.

Por exemplo, a instrução: **c = c + 3**;
pode ser abreviada com o operador de atribuição de adição += como: **c += 3**

Admita: c=3, **d=5**, **e=4**, **f=6**, **g=12**

Operador de atribuição	Exemplo de expressão	Explicação	Atribui
+=	c += 7	c = c + 7	10 a c
-=	d -= 4	d = d - 4	1 a d

*=	e *= 5	e = e * 5	20 a e
/=	f /= 3	f = f/3	2 a f
%=	g %= 9	g = g%9	3 a g
++	++a	Incrementa a de 1 e depois usa o novo valor de a na expressão	
++	a++	Usa o valor atual de a na expressão onde a se localizar e depois incrementa a em 1	
--	--b	Decrementa b de 1 e depois usa o novo valor de b na expressão	
--	b--	Usa o valor atual de b na expressão onde b se localiza e depois decrementa b de 1	

É importante destacar que quanto uma variável é incrementada ou decrementada em uma instrução isolada, as formas de incrementar e decrementar causam o mesmo efeito. Somente quando uma variável aparece no contexto de uma expressão maior é que os efeitos de pré-incrementar e pós-incrementar serão diferentes.