**Universidade de Brasília**
Departamento de Ciência da Computação

# Linguagem C

Professora: Eliza Gomes

E-mail: eliza.gomes@unb.br

# Introdução

- C é uma linguagem de programação de uso geral, procedural e imperativa desenvolvida em 1972 por Dennis M. Ritchie nos *Laboratórios Bell Telephone* para desenvolver o sistema operacional UNIX;

- É uma linguagem de nível intermediário e pode ser usada para interagir mais diretamente com o hardware e a memória do computador;

- É uma linguagem de programação estruturada e pode-se usar as habilidades aprendidas em C para aprender outras linguagens de programação;

- A linguagem C pode ser usada para desenvolver projetos robustos, por ser potente e flexível.

# Características da Linguagem C

- **Rapidez** – Consegue obter performances semelhantes às obtidas pelo Assembly, através de instruções de alto nível, mesmo para usar mecanismos de mais baixo nível, como o endereçamento de memória ou a manipulação de bits.

- **Simples** – A sua sintaxe é extremamente simples, e o número de palavras reservadas, de tipos de dados básicos e de operadores é diminuto, reduzindo assim a quantidade de tempo e esforço necessários à aprendizagem da linguagem.

- **Portável** – Existe um padrão (ANSI) que define as características de qualquer compilador. Desse modo, o código escrito numa máquina pode ser transportado para outra máquina e compilado sem qualquer alteração (ou com um número reduzido de alterações).

# Características da Linguagem C

- Popular – É internacionalmente conhecida e utilizada. Está muito bem documentada em livros, revistas especializadas, manuais. Existem compiladores para todo tipo de arquiteturas e computadores.

- Modular – Permite o desenvolvimento modular de aplicações, facilitando a separação de projetos em módulos distintos e independentes, recorrendo à utilização de funções específicas dentro de cada módulo.

- Bibliotecas Poderosas – O fato de C possuir um número reduzido de palavras-chave indica que as capacidades de C são muito limitadas. A maior parte das funcionalidades da linguagem C é adicionada pela utilização de funções que existem em bibliotecas adicionais e realizam todo tipo de tarefas, desde a escrita de um caractere na tela até o processamento de *strings*.

- Macros – Tem a possibilidade de utilização de Macros no desenvolvimento de aplicações, reduzindo assim a necessidade de escrita de funções distintas para a realização do mesmo processamento para tipos de dados diferentes. As Macros permitem aumentar a velocidade de execução sem ter que aumentar a complexidade de escrita do código.

# Características da Linguagem C

- **Foco** – Permite ao programador escrever o código como bem quiser. Um programa pode ser todo escrito numa única linha ou dividido por inúmeras linhas.

- **Evolução** – C é uma linguagem particularmente estável. A evolução das linguagens fez com que também C evoluísse no sentido das Linguagens Orientadas a Objetos, dando origem a uma nova linguagem: C++, a qual mantém a sintaxe da linguagem C e permite um conjunto adicional de características (Encapsulamento, Hereditariedade, Polimorfismo, sobrecarga, etc.).

- Atualmente, uma nova linguagem – **Java** – apresenta-se como nova base expandida de trabalho para os programadores. Também essa linguagem se baseia em C e C++.

# Desvantagens da Linguagem C

- **Gerenciamento manual de memória** - A linguagem C precisa de gerenciamento manual de memória, onde um desenvolvedor precisa cuidar da alocação e desalocação de memória explicitamente.
- **Nenhum recurso orientado a objetos** – A maioria das linguagens de programação suporta os recursos POOs. Mas a linguagem C não suporta.
- ***No Garbage Collection*** – A linguagem C não suporta o conceito de *Garbage collection*. Um desenvolvedor precisa alocar e desalocar memória manualmente e isso pode ser propenso a erros e levar a vazamentos de memória ou uso ineficiente de memória.
- **Sem Tratamento de Exceções** – A linguagem C não fornece nenhuma biblioteca para tratamento de exceções. Um desenvolvedor precisa escrever código para lidar com todos os tipos de expectativas.

# Linguagem Compilada

- Linguagens compiladas fornecem desempenho de execução mais rápido em comparação a linguagens interpretadas;
- Linguagens de programação de alto nível, como C, C++, Java, etc., consistem em palavras-chave que são mais próximas de linguagens humanas, como o inglês;
- Um programa escrito em C (ou qualquer outra linguagem de alto nível) precisa ser convertido para seu código de máquina equivalente;
- Código de máquina é uma sequência de instruções binárias consistindo de 1 e 0 bits;
- O processo de compilação tem 4 etapas diferentes:
  - ✓ Pré-processamento
  - ✓ Compilação
  - ✓ Montagem
  - ✓ Linking

# Etapas do Processo de Compilação em C

- Etapa 1: Pré-processamento:
  - ✓ Remove todos os comentários nos arquivos de origem;
  - ✓ Inclui o código do(s) arquivo(s) de cabeçalho (arquivo com extensão **.h** que contém declarações de função C) e definições de macro;
  - ✓ Substitui todas as macros (fragmentos de código que receberam um nome) pelos seus valores;

# Etapas do Processo de Compilação em C

- Etapa 2: Compilação:
  - ✓ O compilador gera o código IR (*Intermediate Representation*) do arquivo pré-processado;
  - ✓ Esse processo gera um arquivo **.s**;
  - ✓ Outros compiladores podem produzir código *assembly* nesta etapa da compilação.

# Etapas do Processo de Compilação em C

- Etapa 3: Montagem
  - ✓ O *Assembler* pega o código IR e o transforma em código objeto, que é código em linguagem de máquina (ou seja, binário);
  - ✓ Isso produzirá um arquivo terminando em **.o**;
  - ✓ O arquivo **.o** não é um arquivo de texto, portanto seu conteúdo não será legível quando você abrir este arquivo com um editor de texto.

# Etapas do Processo de Compilação em C

- Etapa 4: Linking
  - ✓ O Linker cria o executável final, em binário;
  - ✓ Ele reúne os códigos de objeto de todos os arquivos de origem;
  - ✓ O Linker sabe onde procurar as definições de função nas **bibliotecas estáticas** ou nas **bibliotecas dinâmicas**;
  - ✓ Bibliotecas estáticas são o resultado do vinculador fazendo uma cópia de todas as funções de biblioteca usadas para o arquivo executável;
  - ✓ O código em bibliotecas dinâmicas não é copiado inteiramente, apenas o nome da biblioteca é colocado no arquivo binário;

# Começando com Programação em C

# Palavras-chave

- Palavras-chave são aquelas palavras predefinidas que têm significado especial no compilador e não podem ser usadas para nenhuma outra finalidade;
- As palavras-chave são sensíveis à capitalização;
- Das 44 palavras-chave:
  - ✓ 32 são originais da linguagem
  - ✓ A atualização C99 adicionou 5;
  - ✓ A atualização C11 adicionou 7;
  - ✓ A maioria das palavras-chave novas começa com um sublinhado.

**Tabela 3-1 Palavras-chave de Linguagem C**

| | | | |
|---|---|---|---|
| _Alignas | break | float | signed |
| _Alignof | case | for | sizeof |
| _Atomic | char | goto | static |
| _Bool | const | if | struct |
| _Complex | continue | inline | switch |
| _Generic | default | int | typedef |
| _Imaginary | do | long | union |
| _Noreturn | double | register | unsigned |
| _Static_assert | else | restrict | void |
| _Thread_local | enum | return | volatile |
| auto | extern | short | while |

# Arquivos de Cabeçalho

- Arquivos de cabeçalho são chamados pelo uso da declaração `#include` no início do código e terminam em **.h**;
- Estes arquivos não possuem os códigos completos das funções. Eles só contêm protótipos de funções;
- O compilador lê estes protótipos e, baseado nas informações lá contidas, gera o código correto;
- O corpo das funções cujos protótipos estão no arquivo-cabeçalho, no caso das funções do próprio C, já estão compiladas e normalmente são incluídas no programa no instante da *linkagem;*
- Este é o instante em que todas as referências a funções cujos códigos não estão nos nossos arquivos fontes são resolvidas, buscando este código nos arquivos de bibliotecas.

# Arquivos de Cabeçalho

- Os arquivos de cabeçalho são encontrados no diretório `/usr/include`;
- Os arquivos de biblioteca estão no diretório `/usr/lib`;
- Arquivos de cabeçalho são texto simples e arquivos de biblioteca são dados;
- Em um sistema Windows, os arquivos são mantidos com o compilador, normalmente, nas pastas `include` e `lib` relativas à localização do compilador;

# Arquivos de Cabeçalho

- Alguns arquivos de cabeçalho populares são:
  - ✓ **stdio.h** – Fornece funções de entrada e saída como `printf` e `scanf;`
  - ✓ **stdlib.h** – Contém funções envolvendo alocação de memória, função `rand` e outras funções utilitárias;
  - ✓ **math.h** – Inclui funções matemáticas como `sqrt` , `sin` , `cos`;
  - ✓ **string.h** – Inclui funções para manipular strings, como `strcpy`, `strlen`;
  - ✓ **ctype.h** – Funções para testar e mapear caracteres, como `isalpha`, `isdigit`;
  - ✓ **stdbool.h** – Define o tipo de dados booleano e os valores `true` e `false`;
  - ✓ **time.h** – Contém funções para trabalhar com data e hora;
  - ✓ **limits.h** – Define vários limites específicos de implementação em tipos inteiros;

# Regras de Nomenclatura de identificadores

- Palavras-chave não podem ser usadas como identificadores, pois são predefinidas;
- Apenas o alfabeto (maiúsculas e minúsculas) e o símbolo de sublinhado (_) são permitidos no identificador;
- O identificador deve começar com um alfabeto (maiúsculo ou minúsculo) ou um sublinhado. Isso significa que um dígito não pode ser o primeiro caractere do identificador;
- Os caracteres subsequentes podem ser letras, dígitos ou um sublinhado;
- O mesmo identificador não pode ser usado como nome de duas entidades. Um identificador só pode ser usado apenas uma vez no escopo atual;

# Regras de Nomenclatura de identificadores

| | | | |
|---|---|---|---|
| age | ✓ | address1 | ✓ |
| $age | ⊘ | 1mg | ⊘ |
| Age | ✓ | #phone | ⊘ |
| AGE | ✓ | phone_no_personal | ✓ |
| average_age | ✓ | my name | ⊘ |
| Average-age | ⊘ | _my_name | ✓ |
| __temp | ✓ | phy+maths | ⊘ |

# Declarações e Estrutura

- Uma declaração é uma ação ou uma orientação que o programa dá ao hardware;
- Todas as declarações da linguagem C terminam com ponto e vírgula (;);
- Declarações em C são executadas uma após a outra, começando pelo topo do código-fonte e seguindo até o final;
- A sintaxe em nível de parágrafo para a linguagem C envolve o uso de chaves;
- O compilador C ignora o espaço em branco, procurando por pontos e vírgulas e chaves;

```
#include <stdio.h>
int main(){printf("Hello, world!");return 0;}
```

```
#include <stdio.h>

int main() {
    printf("Hello, world!");
    return 0;
}
```

# Comentários

- Comentários tradicionais começam com os caracteres **/*** e terminam com os caracteres ***/**;
- Um outro estilo de comentário utiliza os caracteres de barra dupla (**//**). Este tipo de comentário afeta o texto em uma linha.

```
/* Autor: Dan Gookin
Este programa exibe texto na tela */

#include <stdio.h>        /* Necessário para puts() */

int main(){
   printf("Hello, world!");     // Exibe texto
   return 0;
}
```

# Função main()

- É a primeira função que vai executar quando o programa começar;
- O protocolo adequado necessita que, quando um programa fecha, ele forneça um valor ao sistema operacional;
- Esse valor é um inteiro, normalmente zero (0), mas, às vezes, outros valores são utilizados, dependendo do que o programa faz e o que o sistema operacional espera;
- Tradicionalmente, um valor de retorno `0` é utilizado para indicar que um programa completou seu trabalho com sucesso.

# Função main()

- Os valores de retorno `1` ou maiores, frequentemente, indicam algum tipo de erro ou talvez eles indiquem os resultados de uma operação;
- A palavra-chave `return` pode ser utilizada em uma declaração com ou sem parênteses.

```
/* Autor: Dan Gookin
Este programa exibe texto na tela */

#include <stdio.h>        /* Necessário para puts() */

int main(){
   printf("Hello, world!");     // Exibe texto
   return 0;
}
```

# Caractere Especial \

- O símbolo \ é utilizado para retirar o significado especial que um caractere apresenta;
- No caso do caractere aspas (“), retira-lhe o significado de delimitador, passando a ser considerado simplesmente como o caractere aspas;
- No caso do **\n** (*New Line*), serve para representar um caractere que, de outro modo, seria difícil ou quase impossível de representar;
- A lista dos caracteres que podem ter que ser representados, precedidos do caractere especial \, é:

# Caractere Especial \

| | |
|---|---|
| \7 | Bell (sinal sonoro do computador) |
| \a | Bell (sinal sonoro do computador) |
| \b | BackSpace |
| \n | New Line (mudança de linha) |
| \r | Carriage Return (início da linha) |
| \t | Tabulação Horizontal |
| \v | Tabulação Vertical |

| | |
|---|---|
| \\ | Caractere \ (forma de representar o caractere \) |
| \' | Caractere ‘ (aspas simples) |
| \" | Caractere “ (aspas duplas) |
| \? | Caractere ? (ponto de interrogação) |
| \000 | Caractere cujo código *ASCII* em Octal é 000 |
| $\x_{nn}$ | Caractere cujo código *ASCII* em Hexadecimal é $_{nn}$ |
| %% | Caractere % |

# Vamos praticar?

1. Escreva um programa em C que apresente duas linhas com a *string* "Aqui vai um Apito", ouvindo-se ao final de cada *string* um sinal sonoro.

1. Escreva um programa em C que apresente a seguinte saída:

1 - Clientes

2 - Fornecedores

3 - Faturas

0 - Sair

# Resposta

- Exercício 1

```
#include <stdio.h>

int main() {
   printf("Aqui vai um Apito\a\n");
   printf("Aqui vai um Apito\7\n");

   return 0;
}
```

- Exercício 2

```
#include <stdio.h>

int main(){
   printf("1 -\tClientes\n");
   printf("2 -\tFornecedores\n");
   printf("3 -\tFaturas\n\n");
   printf("0 -\tSair\n");

   return 0;
}
```

# Tipos de Dados

# Tipos de Dados Básicos

- São 4 os tipos de dados básicos em C:

| Tipo Básico | Tipo Variado | Tamanho | Intervalo de Valores |
|---|---|---|---|
| char | char | 1 byte | -128 até 127 ou 0 até 255 |
| | unsigned char | 1 byte | 0 até 255 |
| | signed char | 1 byte | -128 até 127 |
| float | float | 4 bytes | 1.2E-38 até 3.4E+38 |
| double | double | 8 bytes | 2.3E-308 até 1.7E+308 |
| | long double | 10 bytes | 3.4E-4932 até 1.1E+4932 |

# Tipos de Dados Básicos

- São 4 os tipos de dados básicos em C:

| Tipo Básico | Tipo Variado | Tamanho | Intervalo de Valores |
|---|---|---|---|
| int | int | 2 ou 4 bytes | -32.768 até 32.767 ou<br>-2.147.483.648 até 2.147.483.647 |
| | unsigned int | 2 ou 4 bytes | 0 até 65.535 ou<br>0 até 4.294.967.295 |
| | short | 2 bytes | -32.768 até 32.767 |
| | unsigned short | 2 bytes | 0 até 65.535 |
| long | long | 8 bytes | -9.223.372.036.854.775.808 até<br>9.223.372.036.854.775.807 |
| | unsigned long | 8 bytes | 0 até 18.446.744.073.709.551.615 |

# Declaração de Variáveis

- Uma variável deve ser sempre definida antes de ser usada;
- A definição de uma variável indica ao compilador qual o tipo de dado que fica atribuído ao nome que indicarmos para essa variável;
- A definição de variáveis é feita utilizando a seguinte sintaxe:

```
Tipo var1 [, var2, var3, ..., varn];
```

# Atribuição de Valores para Variável

- Sempre que uma variável é declarada, é solicitado ao compilador que reserve espaço em memória para armazená-la;
- Esse espaço passará a ser referenciado através do nome da variável;
- Quando uma variável é declarada fica sempre com um valor, o qual resulta do estado aleatório dos *bits* que a constituem;
- Desse modo, uma variável poderá ser iniciada com um valor através de uma operação de atribuição;

```
int num = 17;
```

# Tipo de Dados Vetor

- Um vetor é uma coleção de múltiplos valores do mesmo tipo de dados armazenados em localizações de memória consecutivas;
- O tamanho do vetor é mencionado entre colchetes [].

```
int num[5];
```

- Os vetores podem ser inicializados no momento da declaração;
- Os valores a serem atribuídos são colocados entre chaves.

```
int num[] = {50, 56, 76, 67, 43};
```

# Tipo de Dados Matriz

- C suporta vetores multidimensionais.

```
int marks[5][2];

//O valor 5 representa a quantidade de linhas.

// O valor 2 representa a quantidade de colunas.


float marks2[5][2] = {{7.5, 6.8}, {6.5, 6.3},
                      {5.7, 8.6}, {4.5, 5.8},
                      {3.6, 7.6}};
```

# Tipo de Dados *Struct*

- Permitem colocar, em uma única entidade, elementos de tipos diferentes;
- É um conjunto de uma ou mais variáveis agrupadas sob um único nome, de forma a facilitar a sua referência;
- Podem conter elementos com qualquer tipo de dados válidos em C;
  - ✓ Tipos básicos
  - ✓ Vetores
  - ✓ Strings
  - ✓ Ponteiros
  - ✓ Struct

# Declaração e Atribuição de *Struct*

- A definição da ***struct Data*** indica que, a partir daquele momento o compilador passa a conhecer um outro tipo, chamado ***struct Data***, que é composto por dois inteiros e um vetor com 12 caracteres;
- A atribuição é feita colocando entre chaves os valores dos membros da estrutura, pela ordem em que esses foram escritos na sua definição.

```
typedef struct Data {
   int Dia;
   char Mês[12];
   int Ano;
}data;

data = dt_nasc;
```

```
struct Data {
   int Dia;
   char Mês[12];
   int Ano;
};

struct Data = dt_nasc;
```

```
dt_nasc = {05, "Janeiro", 2004};
```

# Conversão de Tipo Explícita

- Quando é preciso converter um tipo de dado com tamanho de *byte* maior para outro tipo de dado com tamanho de *byte* menor, é preciso informar especificamente ao compilador;
- Isso é chamado de conversão de tipo explícita;
- C fornece um operador *typecast*.
- Basta colocar o tipo de dado entre parênteses antes do operando a ser convertido.

```
type2 var2 = (type1) var1;
```

# Conversão de Tipo Explícita

- Qual a saída desse código?

```
#include<stdio.h>

int main(){
   int x = 10, y = 4;
   float z = x/y;
   printf("%f", z);
   return 0;
}
```

- Out: 2.000000

# Conversão de Tipo Explícita

- Isso ocorre porque ambos os operandos na expressão de divisão são do tipo *int*;
- Em C, o resultado de uma operação de divisão será sempre do tipo de dados com maior comprimento de *byte*;
- É preciso converter um dos operandos inteiros para *float*.

```
#include<stdio.h>

int main(){
   int x = 10, y = 4;
   float z = (float)x/y;
   printf("%f", z);
   return 0;
}
```

# Constantes

- Uma constante em C é um nome atribuído pelo usuário a um local na memória, cujo valor não pode ser modificado depois de declarado;
- É possível declarar uma constante em um programa C de duas maneiras:
  - ✓ Usando a palavra-chave **const**
  - ✓ Usando a diretiva **#define**

```
//const type NAME = value;
const float PI = 3.14159265359;

//#define name = value
#define PI 3.14159265359
```

# Operadores

# Operadores Aritméticos

- Suponha que a variável A contém 10 e a variável B contém 20;

| Operador | Descrição | Exemplo |
|---|---|---|
| + | Soma dois operandos | A + B = 30 |
| - | Subtrai o segundo operando do primeiro | A - B = -10 |
| * | Multiplica ambos os operandos | A * B = 200 |
| / | Divide o numerador pelo denominador | B / A = 2 |
| % | Operador de módulo e resto após uma divisão inteira | B % A = 0 |
| ++ | O operador de incremento aumenta o valor inteiro em um | A++ = 11 |
| -- | O operador de decremento diminui o valor inteiro em um | A-- = 9 |

# Operadores Relacionais

| Operador | Descrição | Exemplo |
|---|---|---|
| == | Verifica se os valores de dois operandos são iguais ou não. Se sim, então a condição se torna verdadeira. | (A == B) não é verdade. |
| != | Verifica se os valores de dois operandos são iguais ou não. Se os valores não forem iguais, então a condição se torna verdadeira. | (A != B) é verdadeiro. |
| > | Verifica se o valor do operando esquerdo é maior que o valor do operando direito. Se sim, então a condição se torna verdadeira. | (A > B) não é verdade. |
| < | Verifica se o valor do operando esquerdo é menor que o valor do operando direito. Se sim, então a condição se torna verdadeira. | (A < B) é verdadeiro. |
| >= | Verifica se o valor do operando esquerdo é maior ou igual ao valor do operando direito. Se sim, então a condição se torna verdadeira. | (A >= B) não é verdade. |
| <= | Verifica se o valor do operando esquerdo é menor ou igual ao valor do operando direito. Se sim, então a condição se torna verdadeira. | (A <= B) é verdadeiro. |

# Operadores Lógicos

| Operador | Descrição | Exemplo |
|---|---|---|
| && | Chamado operador lógico **AND**. Se ambos os operandos forem diferentes de zero, então a condição se torna verdadeira. | (A && B) é falso. |
| \|\| | Chamado de Operador Lógico **OU**. Se qualquer um dos dois operandos for diferente de zero, então a condição se torna verdadeira. | (A \|\| B) é verdadeiro. |
| ! | Chamado de Operador Lógico **NOT**. É usado para reverter o estado lógico de seu operando. Se uma condição for verdadeira, então o operador Lógico **NOT** a tornará falsa. | !(A && B) é verdadeiro. |

# Operadores Bitwise

- Operadores *bitwise* permitem manipulação dados armazenados na memória do computador;
- Esses operadores são usados para executar operações de nível de bit nos operandos;
- Os operadores *bitwise* mais comuns são AND (&), OR (|), XOR (^), NOT (~), *left shift* (<<) e *right shift* (>>).

# Operadores Bitwise

- Suponha que A = 60 e B = 13 em formato binário, eles serão os seguintes:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

# Operadores Bitwise

| Operador | Descrição | Exemplo |
|---|---|---|
| & | O operador binário AND copia um bit para o resultado se ele existir em ambos os operandos. | (A e B) = 12, ou seja, 0000 1100 |
| \| | O operador binário OR copia um bit se ele existir em qualquer operando. | (A \| B) = 61, ou seja, 0011 1101 |
| ^ | O operador binário XOR copia o bit se ele estiver definido em um operando, mas não em ambos. | (A ^ B) = 49, ou seja, 0011 0001 |
| ~ | O operador de complemento binário de um é unário e tem o efeito de "inverter" bits. | (~A ) = ~(60), ou seja, 11000011 |
| << | Operador de deslocamento binário para a esquerda. O valor dos operandos da esquerda é movido para a esquerda pelo número de bits especificado pelo operando da direita. | A << 2 = 240 ou seja, 1111 0000 |
| >> | Operador de deslocamento binário para a direita. O valor dos operandos da esquerda é movido para a direita pelo número de bits especificado pelo operando da direita. | A >> 2 = 15 ou seja, 0000 1111 |

# Escrita e Leitura dos Dados

# Escrita de *Strings: printf()*

- Recebe como formato uma *string*;
- Permite a impressão de variáveis de diversos tipos e a formatação personalizada de *strings*.

```
printf(”Mensagem”);
```

```
printf("Formato", variáveis);
```

```
int idade = 25;
float altura = 1.75;
printf("Idade: %d anos, Altura: %.2f metros\n", idade, altura);
```

# Escrita de *Strings: puts()*

- Imprime uma *string* no console, seguida automaticamente de uma nova linha (\n).

```
puts(”Olá, mundo”);
```

# Leitura de *Strings*: *scanf()*

- Permite realiza a leitura de *strings*;
- Converte a entrada em um tipo de dado desejado com especificadores de formato apropriados;
- A não ser que seja uma *string*, a variável é prefixada pelo operador **&;**

```
scanf(”%tipo_dado”, &var);


scanf(“%tipo_dado1 %tipo_dado2”, &var1, &var2);


scanf(“%s”, var);
```

# Formatos para Escrita e Leitura

| Especificador | Descrição |
|---|---|
| %c | Character |
| %d | Signed integer |
| %f | Float values |
| %i | Unsigned integer |
| %s | String |
| %l or %ld or %li | Long |
| %[^\n] | String com espaço |

| Especificador | Descrição |
|---|---|
| %lf | Double |
| %Lf | Long double |
| %lu | Unsigned int or unsigned long |
| %lli or %lld | Long long |
| %llu | Unsigned long long |
| %c | Character |

# Vamos Praticar?

- Escreva um código que você informe o nome e a data de nascimento e imprima a seguinte mensagem: “Fulano nasceu no dia 01/01/2001”.

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

int main(){
   char nome[10];
   int dia, mes, ano;
   scanf("%s%d%d%d", nome, &dia, &mes, &ano);
   printf("%s nasceu no dia %d/%d/%d \n",nome,dia,mes,ano);
   return 0;
}
```

# Comandos de Controle Condicional

# Comando *if*

- É utilizado sempre que é necessário escolher entre dois caminhos dentro do programa ou quando se deseja executar um ou mais comandos que estejam sujeitos ao resultado de um teste.

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

int main(){
   int num;
   printf("Digite um número: ");
   scanf("%d", &num);

   if(num == 10){
      printf("O número é igual a 10. \n");
   }else{
      printf("O número é diferente de 10. \n");
   }
   return 0;
}
```

# Comando *if* – Operador ?

- Também conhecido como operador ternário;
- Trata-se de uma simplificação do comando **if-else**;
- Sua forma geral é:

```
expressão condicional ? expressão1: expressão2;
```

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

int main(){
   int num;
   printf("Digite um número: ");
   scanf("%d", &num);

   (num == 10)? printf("O número é igual a 10. \n") :
           printf("O número é diferente de 10. \n");

    return 0;
}
```

# Comando *switch*

- É muito parecido com o aninhamento de comandos **if-else-if**;
- É mais limitado que o comando **if-else**;
- Só verifica se uma variável (do tipo **int** ou **char**) é ou não igual a certo valor constante;
- É indicado quando se deseja testar uma variável em relação a diversos valores preestabelecidos;
- O comando **default** é opcional.

```
#include<stdio.h>

#include<stdlib.h>


int main(){

   char ch;

   printf("Digite um símbolo de pontuação: ");

   ch = getchar;


   switch(ch){

       case '.': printf("Ponto. \n"); break;

       case ',': printf("Vírgula. \n"); break;

       case ':': printf("Dois pontos. \n"); break;

       case ';': printf("Ponto e vírgula. \n"); break;

       default: printf("Não é pontuação \n");

   }

    return 0;

}
```

# Vamos Praticar?

- Escreva um programa que, dada a idade de um nadador, classifique-o em uma das seguintes categoria:

| Categoria | Idade |
|---|---|
| Infantil A | 5 - 7 |
| Infantil B | 8 - 10 |
| Juvenil A | 11 - 13 |
| Juvenil B | 14 - 17 |
| Sênior | Maiores de 18 anos |

- Usando o comando switch, escreva um programa que leia um inteiro entre 1 e 7 e imprima o dia da semana correspondente a esse número. Isto é, domingo, se 1, segunda-feira, se 2, e assim por diante.

# Comandos de Repetição

# Comando *while*

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

int main(){
   int a, b;
   printf("Digite o primeiro número: ");
   scanf("%d", &a);
   printf("Digite o segundo número: ");
   scanf("%d", &b);

   while (a < b){
        printf("%d \n", a);
        a++;
    }
    return 0;
}
```

# Comando *for*

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

int main(){
   int a, b, c;
   printf("Digite o primeiro número: ");
   scanf("%d", &a);
   printf("Digite o segundo número: ");
   scanf("%d", &b);

   for (c = a; c <= b; c++){
     printf("%d \n", c);
   }
    return 0;
}
```

# Comando *do-while*

- Semelhante ao comando **while**;
- O comando **while** avalia a condição para depois executar uma sequência de comandos;
- O comando **do-while** executa uma sequência de comandos para depois testar a condição;
- É utilizado sempre que se desejar que a sequência de comandos seja executada pelo menos uma vez.

```
#include<stdio.h>

#include<stdlib.h>


int main(){

   int i;

   do {

        printf("Escolha uma opção: \n");

        printf("(1) Opção 1 \n");

        printf("(2) Opção 2 \n");

        printf("(3) Opção 3 \n");

        scanf("%d", &i);

   } while ((i < 1) || (i > 3));

   printf("Você escolheu a Opção %d. \n", i);

   return 0;

}
```

# Vamos Praticar?

- Faça um programa que leia um número inteiro positivo N e imprima todos os números naturais de 0 até N em ordem crescente;

- Faça um algoritmo que leia um número positivo e imprima seus divisores. Exemplo: os divisores do número 66 são: 1, 2, 3, 6, 11, 22, 33 e 66.

# Ponteiros

# Ponteiros

- Toda informação que manipulamos dentro de um programa obrigatoriamente está armazenada na memória do computador;
- Quando criamos uma variável, o computador reserva um espaço de memória no qual podemos guardar o valor associado a ela;
- Ponteiros são um tipo especial de variáveis que permitem armazenar endereços de memória em vez de dados numéricos ou caracteres;
- Por meio dos ponteiros, podemos acessar o endereço de uma variável e manipular o valor que está armazenado lá dentro;

# Ponteiros

- **Variável**: é um espaço reservado de memória usado para guardar um valor que pode ser modificado pelo programa;
- **Ponteiro**: é um espaço reservado de memória usado para guardar um endereço de memória;
- Quando declaramos um ponteiro, informamos ao compilador para que tipo de variável poderemos apontá-lo;
- Um ponteiro do tipo int* só pode apontar para uma variável do tipo int (ou seja, esse ponteiro só poderá guardar o endereço de uma variável int).

```
tipo_do_ponteiro *nome_do_ponteiro;
```

# Ponteiros

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

int main(){
   int *p;
   float *x;
   char *y;
   int soma, *z;
   return 0;
}
```

# Ponteiros

- Quando um ponteiro é declarado, ele não possui um endereço associado;
- Os ponteiros devem ser inicializados (apontados para algum lugar conhecido) antes de serem usados;
- Um ponteiro pode ter um valor especial **NULL**, que é o endereço de nenhum lugar;
- Para apontar para um endereço válido, o ponteiro deve apontar para uma variável que já exista no programa;
- Para saber o endereço onde uma variável está guardada na memória, usa-se o operador **&** na frente do nome da variável.

# Ponteiros

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

int main(){
    int count = 10;
    int *p;
    p = &count;
    return 0;
}
```

| Memória | | |
|---|---|---|
| Endereço | Variável | Conteúdo |
| 119 | | |
| 120 | int *p | #122 |
| 121 | | |
| 122 | int count | 10 |
| 123 | | |

# Ponteiros

- Operador "*"
  - ✓ Declara um ponteiro: **int *x;**
  - ✓ Conteúdo para onde o ponteiro aponta: **int y = *x;**

- Operador "&"
  - ✓ Endereço onde uma variável está guardada na memória: **int z = &y;**

# Ponteiros

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

int main(){
   int count = 10;
   int *p;
   p = &count;
   printf("Conteúdo apontado por p: %d \n", *p);
   *p = 12;
   printf("Conteúdo apontado por p: %d \n", *p);
   printf("Conteúdo de count: %d \n", count);
   return 0;
}
```

# Ponteiros para Ponteiros

- Também ocupam um espaço na memória do computador e possuem o endereço desse espaço de memória associado ao seu nome;
- É possível criar um ponteiro que aponte para o endereço de outro ponteiro;
- A declaração de um ponteiro para ponteiro criado pelo programador segue esta forma geral:

```
tipo_do_ponteiro **nome_do_ponteiro;
```

# Ponteiros para Ponteiros

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

int main(){
    int count = 10;
    int *p = &count;
    int **p2 = &p;
    return 0;
}
```

| Memória | | |
|---|---|---|
| Endereço | Variável | Conteúdo |
| 119 | | |
| 120 | int *p | #122 |
| 121 | int **p2 | #120 |
| 122 | int count | 10 |
| 123 | | |

# Aplicações de Ponteiros

- Para acessar elementos em **array** e matriz
- Para alocar memória dinamicamente
- Para passar argumentos como referência
- Para passar um **array** para uma função
- Para retornar vários valores de uma função

# Função

# Funções

- É uma sequência de comandos que recebe um nome e pode ser chamada de qualquer parte do programa, quantas vezes forem necessárias, durante a sua execução;
- Duas são as principais razões para o uso de funções:
  - ✓ Estruturação dos programas
  - ✓ Reutilização de código

```
tipo_retornado nome_função (lista_de_parâmetros)
{
    sequência_de_declarações;
    sequência_de_comandos;
}
```

# Funções

- Deve ser declarada antes de ser utilizada, antes da cláusula **main()**;
- Funcionamento de uma função:
  - ✓ O código do programa é executado até encontrar uma chamada de função.
  - ✓ O programa é então interrompido temporariamente, e o fluxo do programa passa para a função chamada.
  - ✓ Se houver parâmetros na função, os valores da chamada da função são copiados para os parâmetros no código da função.
  - ✓ Os comandos da função são executados.
  - ✓ Quando a função termina (seus comandos acabaram ou o comando **return** foi encontrado), o programa volta ao ponto em que foi interrompido para continuar sua execução normal.
  - ✓ Se houver um comando **return**, o valor dele será copiado para a variável que foi escolhida para receber o retorno da função.

# Funções

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


int Square (int a){
   return (a*a);
}

int main(){
   int n1, n2;
   printf("Entre com um numero: ");
   scanf("%d", &n1);
   n2 = Square(n1);
   printf("O seu quadrado vale: %d \n", n2);
   return 0;
}
```

# Parâmetros de uma Função

- São o que o programador utiliza para passar a informação de um trecho de código para dentro da função;
- São uma lista de variáveis, separadas por vírgula, em que são especificados o tipo e o nome de cada variável passada para a função;

```
tipo_retornado nome_função (tipo nome1, tipo nome2, ...) {
    sequência_de_declarações;
    sequência_de_comandos;
}
```

```
tipo_retornado nome_função () {
    sequência_de_declarações;
    sequência_de_comandos;
}
```

```
tipo_retornado nome_função (void) {
    sequência_de_declarações;
    sequência_de_comandos;
}
```

# Parâmetros de uma Função

- Existem dois tipos de passagem de parâmetro: **passagem por valor** e **por referência**;
- Passagem por valor:
  - ✓ Os argumentos para uma função são sempre passados por valor
  - ✓ Uma cópia do dado é feita e passada para a função
  - ✓ Quaisquer modificações que a função fizer nos parâmetros existem apenas dentro da própria função
- Passagem por referência:
  - ✓ Quando se quer que o valor da variável mude dentro da função e essa mudança se reflita fora da função
  - ✓ Não se passam para a função os valores das variáveis, mas os endereços das variáveis na memória
  - ✓ Para passar um parâmetro por referência, usa-se o operador "*" na frente do nome do parâmetro durante a declaração da função
  - ✓ Na chamada da função, é necessário utilizar o operador "&" na frente do nome da variável que será passada por referência para a função

# Parâmetros de uma Função

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

void soma_mais_um(int n){
   n = n + 1;
   printf("Dentro da função: x = %d \n", n);
}

int main(){
   int x = 5;
   printf("Antes da função: x = %d \n", x);
   soma_mais_um(x);
   n2 = Square(n1);
   printf("Depois da função: x = %d \n", x);
   return 0;
}
```

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

void soma_mais_um(int *n){
   *n = *n + 1;
   printf("Dentro da função: x = %d \n", *n);
}

int main(){
   int x = 5;
   printf("Antes da função: x = %d \n", x);
   soma_mais_um(&x);
   n2 = Square(n1);
   printf("Depois da função: x = %d \n", x);
   return 0;
}
```

# Corpo da Função

- É no corpo da função que as entradas (parâmetros) são processadas, as saídas são geradas ou outras ações são feitas;
- Evita-se fazer operações de leitura e escrita dentro de uma função;
- Operações de leitura e escrita não são proibidas dentro de uma função. Apenas não devem ser usadas se esse não for o foco da função.

# Corpo da Função

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

int main(){
   int x;
   printf("Digite um número inteiro positivo: ");
   scanf("%d", &x);
   int i, f = 1;
   for(i=1; i<=x; i++){
      f = f * i;
   }
   printf("O fatorial de %d é: %d \n", x,f);
   return 0;
}
```

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

int fatorial (int n){
   int i, f = 1;
   for(i=1; i<=x; i++){
      f = f * i;
   }
   return f
}
int main(){
   int x;
   printf("Digite um número inteiro positivo: ");
   scanf("%d", &x);
   int fat = fatorial(x);
   printf("O fatorial de %d é: %d \n", x,fat);
   return 0;
}
```

# Retorno da Função

- O retorno da função é a maneira como uma função devolve o resultado (se ele existir) da sua execução para quem a chamou;
- Uma função pode retornar qualquer tipo válido na linguagem C:
  - ✓ Tipos básicos predefinidos: **int**, **char**, **float**, **double**, **void** e **ponteiros**.
  - ✓ Tipos definidos pelo programador: **struct**, **array.**
- Quando se chega a um comando return, a função é encerrada imediatamente;
- O tipo **void** é conhecido como tipo vazio;
- Uma função declarada com o tipo **void** vai apenas executar um conjunto de comando e não devolverá nenhum valor para quem a chamar.

# Retorno de Função

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

void imprime (int n){
   int i;
   for(i=1; i<=x; i++){
      printf("Linha %d \n", i);
   }
}
int main(){
   imprime(5);
   return 0;
}
```

# Vamos Praticar?

- Escreva uma função que receba por parâmetro dois números e retorne o maior deles.
- Escreva uma função que receba o peso (quilos) e a altura (metros) de uma pessoa. Calcule e retorne o IMC (Índice de Massa Corporal) dessa pessoa:

IMC = peso / (altura * altura)