system(“cls”) limpa a tela

system(“pause”) para congelar a tela

getchar() para congelar a tela

**Formatação da escrita**

cout.fill (‘caractere’) - Preenche os espaços com o caractere escolhido - se definido uma vez vale para todo o programa a seguir;

cout.fixed - Instruímos o programa a exibir valores reais usando a notação de ponto fixo;

cout.scientific - instruímos o programa a utilizar a notação científica;

cout.precision(x) - define o número de caracteres depois da vírgula

cout. ceil(num) - arredonda o número para cima

cout. floor(num) - arredonda o número para baixo

cout.round(num) - Arredonda o número com base na primeira casa decimal

cout.width(x) - define o número mínimo de caracteres a serem exibidos na tela

Junto com o cout.width podemos alinhar o texto

cout.width(10)

cout.fill(‘\*’)

cout << left << -48.6; (podemos usar rigth e internal)

exibe -48.6\*\*\*\*\*;

cout.fabs(num) - retorna o número sem considerar o sinal

**Funções matemáticas**

Biblioteca cmath é usada para a execução de várias operações matemáticas;

* pow: calcula a potência de um número elevado a outro.

sintaxe: pow(x,y)

pow(3,2)

retorna = 9

* sqrt: calcula a raiz quadrada de um número.

sqrt(x)

sqrt(4)

retorna = 2

* fabs: valor absoluto (desconsidera o sinal).
* ceil: arredondamento de um número float para cima, e retorna um número inteiro.

ceil(-4,75)

retorna = - 4

* floor: arredondamento de um número float para baixo.

floor(-4,75)

retorna = - 5

* round: arredondamento de um número float conforme a primeira casa decimal. Se for maior que 5 para cima, se for menor para baixo.

**quando é usado um int em uma função para arredondar um número ele retorna um inteiro**

**++a == incrementa antes**

**a++ == incrementa depois**

**String**

**getline(cin, nome)**; para ler uma string;

**Função gets(vetor);**

é utilizada para a leitura de uma string, funciona como o cin, mas adiciona um finalizador \0 para saber onde essa string termina

**Método getline**

**cin.getline(vetor\_char, tamanho)** - lê somente o número de caracteres especificado;

**sizeof(vetor)** - encontra o tamanho de um vetor

**Funcção puts()** - funciona como o cout, porem não tem muitos recursos;

**Função strcpy()** - copia o conteúdo de uma string.

Sintaxe: strcpy() - (destino, origem);

Os vetores devem ser do mesmo tamanho ou o vetor destino tem que ser maior;

**Função strcat()** - unir 2 strings e colocar em um só lugar - concatena as informações, diferente do strcpy() ele não apaga o conteúdo do vetor anterior apenas concatena.

o vetor de destino deve ter espaço para guardar a informação anterior e a que sera adicionada;

**Função strcmp()** - compara duas strings se forem iguais ele retorna 0;

**Função strlen(string)** - retorna o tamanho da string

Considere s como uma variável qualquer:

string <nome-da-string> - cria uma string vazia

string s1

string s2(s1) - cria uma string como cópia da outra string

string s2(x, ‘b’) - cria uma string com x cópias do caractere entre aspas

s.empty() retorna verdadeiro se uma string é vazia;

s.size() retorna o tamanho em caracteres da string;

s[n] acessa um determinado elemento da string;

s1 = s2 armazena o conteúdo de s2 em s1

s1 = s1 + “ ” + s2 armazena o conteúdo de s1 e s2 em s1 ;

* concatena

== é usado para comparação

toupper(x) Transforma o caractere para maiúsculo

tolower(x) Transforma o caractere para minúsculo

isalnum(x)Verifica se o caractere é um número ou uma letra e retorna verdadeiro(1)

isalpha(x) Retorna verdadeiro(1) caso o caractere for uma letra

isdigit(x) retorna se o caractere é um número

islower(x) retorna verdadeiro se o caractere for minúsculo

**isupper(x)** retorna verdadeiro se o caractere for maiúsculo

**isspace(x)** retorna verdadeiro caso x for um espaço

**atoi(vetor char)** - função que converte um vetor char em inteiro.

* os espaços em branco no início da string são ignorados
* se a string é composta por conjuntos de caracteres separados por espaços em branco, apenas o primeiro conjunto de caracteres é considerados na conversão
* Se a string é um conjunto vazio ou possui caracteres não numéricos, a função retorna um valor 0;
* geralmente utilizado para transformar valores lidos em arquivos texto;

**atof(vetor char)**

Função que converte um vetor de char em um valor float seguindo as mesmas características do atoi

**Funções**

Funções ou sub-rotinas, são utilizadas para realizar quebras de código em partes menores que serão utilizados novamente dentro do programa;

As funções são definidas após as variáveis e antes do programa principal;

Para defini-la devemos colocar o tipo de retorno (float, int, bool.. etc). Se ela não tem um retorno é utilizado void. O nome da função. E depois parênteses onde serão passados os parâmetros. Por fim abre chaves e fecha chaves e dentro deles vai o código.

É possível chamar a função dentro da própria função.

A chamada de uma função é feita com **nomedafunção();**

int NomeDaFunção( int parâmetro){

código

}

**Parâmetros**

**Passar parâmetro por valor:** Não altera o valor passado, é feita uma cópia do valor e passado para a função;

**Passar parâmetro por referência:** Neste tipo as variáveis ocupam o mesmo local de memória, isso faz com que se o valor seja alterado dentro da função a variável passada será alterada também:

ex: int Nomedafunção(int &a, int &b){ };

**Parâmetros com o valor default:**

**S**e um parâmetro foi declarado com um valor default todos os valores devem ser declarados como default, ou podem ser colocados antes do primeiro valor default porque os que vierem a seguir terão que ser default

É automaticamente usado na função caso não seja passado nenhum novo valor para a mesma.

ex: void calcular(int x = 0, int i = 10){ código }

**return :**

encerra a execução da função e retorna um valor ou não;

O return tem que ser compatível com o tipo da função;

Pode haver diversos return dentro da função, mas deve ser cuidado quando forem usados em excesso por poderem causar erros no código;

pode ser feita uma expressão na hora do retorno

ex: return n1+n2;

**Vetor com função**

Para manipular um array em uma função devemos criar um parâmetro na função que seja um vetor ( não precisa ser declarado seu tamanho ) e então será passada a posição na memória deste vetor e tudo que acontecer com ele na função será mudada em sua memória

**ex:**

void função (float vetor [ ] ) { código}

**Matriz com função**

Para manipular uma matriz com função é da mesma forma que um vetor, porém deve ser informado o valor de colunas de uma matriz;

**ex:**

void função (int matriz [ ][ informar] )

**Matriz quadrada é uma matriz n x n;**

A ordem de uma matriz é quantos elementos podem ser inseridos em uma linha;

DP = diagonal principal= Diagonal principal o índice da linha é igual ao índice da coluna;

**Elementos acima da diagonal principal** o índice da linha < coluna;

**Elementos abaixo da diagonal principal** o índice da linha > coluna;

DS = diagonal secundária = linha + coluna = ordem - 1 ;

**matriz transposta =** cada linha da matriz se transforma na coluna

**Matriz simétrica =** os valores acima e abaixo da diagonal principal tem que ser iguais

**matriz[linha][coluna] = matriz[coluna][linha]**

**matriz triangular inferior =** porque os valores válidos estão abaixo da diagonal principal

**matriz triangular superior =** os valores validos estão acima da diagonal principal

**matriz triangular superior unitária =** valores que estão acima da diaginal principal são válidos e a diagonal principal é tudo = 1

**Recursividade**

Recursividade é o mecanismo de programação no qual uma definição de função ou de outro objeto refere-se ao próprio objeto sendo definido. Assim, função recursiva **é uma função que é definida em termos de si mesma**. Recursividade é o mecanismo básico para repetições nas linguagens funcionais.

Ocorre quando uma função chama a si mesma.

Ao criar uma função recursiva a primeira providência é identificar um ponto de parada na recursividade, evitando que a função entre em loop infinito.

A função começa a execução do seu primeiro comando cada vez que é chamada;

Novas e distintas cópias do parâmetros passados por valor vão ser criados a cada execução da função

As novas chamadas da função devem ser semelhantes à chamada original.

A recursividade é usada quando se observa um padrão, uma rotina que se repete.

**Struct**

capacidade de criar um novo tipo de dado que é heterogêneo e possui vários campos ou variáveis

pode ou não ter um nome

**sem nome:**

struct aluno {

int mat;

float nota;

vetor[];

} x, y; // quando não tem nome devemos atribuir as variáveis junto com o struct

**com nome**

aluno turma;

cria uma variável turma com a estrutura de aluno

# Ponteiro

Ponteiro é o endereço de memória de onde esta a variável

Ponteiros são utilizados

**Por que usar?**

* manipular elementos de array
* receber argumentos em funções que necessitam modificar o valor original
* manipular estruturas de dados complexas, como listas encadeadas, árvores grafo, em que um item de conter referência de outro
* alocar memória dinamicamente
* manipular string de uma função para outra
* manipular referências de endereços de memória entre funções

Seu valor indica onde uma variável está armazenada, e não o que está armazenando

Para manipular ponteiros utilizamos o operador indireto (\*) asterisco

**Como declarar :**

**string \*s1; /// ponteiro string**

**int \*p = &x ; // p recebe o endereço de x**

**Para manipular uma struct pode usar :**

**(\*px).nome**

**px -> nome**

**para acessar um campo da struct**

**Manipulação de array com ponteiro**

* Um vetor é uma variável que é capaz de armazenar n vezes um determinado tipo;
* Em se tratando de vetores, o nome do vetor representa o endereço de memória onde se inicia o armazenamento do vetor;
* Portanto, em C, quando se refere a um vetor, sem especificar se índice (somente com o nome da variável , se está referenciando o ENDEREÇO DE MEMÓRIA da 1º posição
* O ponteiro nunca vai ser um vetor porque ele indica ou usa apenas um endereço de cada vez
* Se for indicado o início ou o fim do vetor podemos percorrer o mesmo
* No acesso ao valor contido na posição apontada não usa o caractere \* precedendo a variável, pois o índice indica o conteúdo partindo de um deslocamento
* deslocamento \*(vet + i ) // pego a posição dentro do vetor

struct CAD

{

int N;

float x ;

}

main

{

CAD vet[5]

}

void LER ( CAD \*pvet, int p )

{

cin >> pvet[p].N

}

# Aritmética de ponteiros

char: 1 byte

int : 4 bytes

float : 4 bytes

double : 8 bytes

O operador sizeof() identifica o tamanho em bytes que uma variável ocupa ou um tipo de dado

—--------------------------------------------------------------------------------------------------------------------

* Outra observação importante na manipulação de ponteiros é a ordem de precedência dos operadores.

Ex.

int x = 234;

int \*px;

px = &x;

(\*px)++;

cout << \*px << endl;

Saída

235

—-------------------------------------------------------------------------------------------------------------

* Só é permitido a utilização de quatro operadores aritméticos com ponteiros:

▪ Incremento ( ++ )

▪ Decremento ( - - )

▪ Adição ( + )

▪ Subtração ( - )

**Só são consecutivos os endereços de memória de vetores e matrizes as variáveis isoladas não são consecutivas**

—-----------------------------------------------------------------------------------------------------------

* A manipulação do tipo string também é possível com o uso de aritmética de ponteiros, para isso basta observar a forma de endereçar o ponteiro.

string texto;

char \*ptexto;

ptexto = &texto[0];

# Ponteiro para ponteiro

Serve para passar valores de uma função para outra

o asterisco volta um nível

int \*p; // cria um ponteiro

int \*\*pv // cria um ponteiro que aponta para outro ponteiro

# Alocando memória dinamicamente

O espaço de memória é criado em tempo de execução

**//Definindo o ponteiro e alocando a memória**

int \*px;

px = new int;

**//Também pode ser declarado desta forma**

int \*px = new int;

**//Alocando ó espaço e inicializando com o valor 4**

int \*px = new int(4);

**//Agora com caracter**

char \*c = new char(‘A’);

O operador **delete** não apaga o ponteiro mas sim a memória para onde o ponteiro aponta. Na verdade, a memória não é de fato apagada, ela retorna ao estado de disponível como memória livre para ser alocada novamente quando for

necessário.

**▪ Exemplo espaço comum:**

delete px;

**▪ Exemplo array:**

delete [] pvet;

**//Alocando memória dinamicamente para um array**

int \*px = new int[5];

srand(time(NULL));

for(int x = 0; x < 5; x++)

{

px[x] = rand() % 50;

}

**//ou desta forma**

for(int i = 0; i < 5; i++)

{

\*(px + i) = rand() % 50;

}

**EXEMPLO ARRAY**

08/11/2022 08:01

**//ou usando aritmética de**

ponteiro

for(int i = 0; i < 5; i++)

{

\*pvet = rand() % 50;

pvet++;

}

**//criando um array de tamanho dinâmico.**

int x;

cout << "Informe o tamanho do vetor: ";

cin >> x;

fflush(stdin);

int \*px = new int[x];

srand(time(NULL));

for(int i = 0; i < x; i++)

{

px[i] = rand() % 50;

}

É possível manipular um array multidimensional (matriz) com o operador “new”, mas somente a primeira dimensão pode ser definida em tempo de execução, as demais devem ser constantes.

**Array de duas dimensões**

int n;

int (\*notas)[3]; //declarando o ponteiro

cout << "Informe o numero de linhas: ";

cin >> n;

fflush(stdin);

notas = new int[n][3];

for(int i = 0; i < n; i++)

{

for(int j = 0; j < 3; j++)

{

cin >> notas[i][j];

}

}

Para manipular a matriz em uma função:

void mostra(int (\*notas)[3], int n)

{

for(int i = 0; i < n; i++)

{

for(int j = 0; j < 3; j++)

{

cout << notas[i][j] << "\t";

}

cout << "\n";

}

}

Caso o desenvolvedor não tenha conhecimento prévio de nenhuma das dimensões da matriz, o que pode ser feito?

▪ Como já é de conhecimento que se tratando de array a memória é alocada sequencialmente, é possível utilizar um array de uma dimensão para trabalhar a

quantidade de elementos da matriz.