



Notas de Aula de Algoritmos e Estruturas de Dados **(INF0435)**

Professor:
Rodrigo Augusto Barros Pereira Dias

Versão 1.8
2007-2

SUMÁRIO

1. Resumo de C++	3
1.1. Tipos primitivos	3
1.2. Declaração de variáveis	3
1.3. Constantes	3
1.4. Operadores	3
1.5. Comandos	4
1.6. Saída e entrada de dados	6
2. Tutorial de Visual C++	7
2.1. Instalação	7
2.2. Criação de projetos	9
2.3. Exibindo numeração das linhas	12
2.4. Configurar para o Visual C++ abrir com o padrão do C	13
3. Modularização	14
3.1. Fundamentos da modularização de código	14
3.2. Escopo de variáveis	15
4. Introdução às estruturas de dados em C++	16
4.1. Vetores	16
4.2. Vetores como parâmetros de funções	17
4.3. Structs	18
4.4. Structs como parâmetros de funções	20
5. Busca	22
5.1. Seqüencial	22
5.2. Binária	23
5.3. Tabela Hash	23
6. Ordenação	26
6.1. Inserção	26
6.2. Seleção direta	27
6.3. Bolha (bubblesort)	28
6.4. Mergesort	29
6.5. Quicksort	30
6.6. Shellsort	31
7. Listas	32
7.1. Pilha	32
7.2. Fila	35
8. Recursividade	38
8.1. Definição	38
8.2. Exemplos de algoritmos recursivos	38
9. Bibliografia	40

1 – Resumo de C++

1.1. Tipos primitivos

1.1.1. int : para dados inteiros

1.1.2. float ou double : para valores reais. Note que o double permite armazenar uma gama maior de reais positivos ou negativos.

1.1.3. char : permite armazenar um e apenas um caracter (um dígito ou uma letra ou sinal de acentuação ou sinal de pontuação, etc ...)

1.1.4. bool : para valores lógicos. Constantes deste tipo: **true** ou **false**

1.1.5. void : sem valor (pode ser tudo ou nada)

1.2. Declaração de variáveis

Regra geral: tipo <nome da variável>;

tipo <nome da variável 1>, <nome da variável 2>;

Exemplos:

```
int i;
```

```
float x, y;
```

```
char c, f;
```

1.3. Constantes

Regra geral: const <nome da constante> = <valor> ;

Exemplo : const MAX = 100;

1.4. Operadores

1.4.1. Atribuição: =

1.4.2. Aritméticos:

Operador	Operação realizada	Exemplos
+	Adição	
-	Menos unário / subtração	-3 int x = 10 - 4;
*	Multiplicação	
/	Divisão inteira	int X; X = 5/2; <i>Note : X valerá 2</i>
/	Divisão real	float f; f = 5.0/2; ou f = 5/2.0; ou f = 5.0/2.0; fornecerá 2.5
%	Resto da divisão entre inteiros	int y; y = 5%2; y valerá 1
++	Incremento	
--	Decremento	

Note que o incremento, assim como o decremento, podem ser pré ou pós-fixados. Caso não haja atribuição, é indiferente que haja incremento/decremento antes ou após a variável. Caso haja atribuição com ++ ou -- , será importante identificar se os operadores estão antes (pré) ou após as variáveis (pós).

Notas de Aula de Algoritmos e Estruturas de Dados

Considerando que haja atribuição:

- Incremento/decremento pré-fixado : 1º) incrementa ou decrementa 2º) atribui
- Incremento/decremento pós-fixado : 1º) atribui 2º) incrementa ou decrementa

1.4.2.1. Operadores Aritméticos de Atribuição

Regra geral:

<variável>	OAA	<expressão>;
	equivale	
<variável> = <variável>	OP (<expressão>;

Sejam:

- OAA (Operadores Aritméticos de Atribuição): +=, -=, *=, /= ou %=
- OP (OPeradores aritméticos): +, -, *, / ou %.

OBS: Os parênteses que envolvem a expressão são fundamentais.

Exemplo:

Sejam int x, y, z;

x += y;	equivale a	x = x + y;
x -= y + 10;	equivale a	x = x - (y + 10);
z /= x - y;	equivale a	z = z / (x - y);

1.4.3. Relacionais

Operador	Operação
>	Maior
>=	Maior ou igual
<	Menor
<=	Menor ou igual
==	Igual
!=	Diferente

1.4.4. Lógicos

Operador	Operação
!	Não
&&	E
	Ou

1.5. Comandos

1.5.1. Condicional/Seleção (se)

a) Simplex

```
if ( CONDIÇÃO) {
    bloco de instruções
}
```

b) Composto

```

if ( CONDIÇÃO ) {
    bloco de instruções
} else
    bloco de instruções
}

```

Exemplo:

```
int x;
cin >> x; // leitura de x
if (x > 0) {
    cout << "Positivo";
} else {
    cout << "Negativo ou nulo";
}
```

c) Múltiplos casos (**caso**)

```
switch (EXPRESSÃO){
    case opção1: [bloco de instruções]; break;
    case opção2: [bloco de instruções]; break;
    default: [bloco de instruções]; break;
}
```

O switch testa igualdades. Ou seja, testa se expressão é igual a opção1 ou se é igual a opção2 ou se é igual a opçãoN, etc. O comando break desvia a execução para fora do switch. A opção default é opcional. Ela deve ser usada para o caso de nenhuma opção ser igual a expressão. Em cada case é possível ter mais de uma instrução.

1.5.2. Repetição

a) while (**enquanto**)

```
while ( CONDIÇÃO ){
    Bloco de instruções;
}
```

Se o teste lógico for verdadeiro, o bloco de instruções dentro do while é executado. Caso contrário, segue-se para as instruções fora do corpo do while.

b) for (**para**)

```
for ( INICIALIZAÇÃO; TESTE LÓGICO; INCREMENTO/DECREMENTO ){
    Bloco de instruções;
}
```

A inicialização é feita apenas na 1ª vez. Se o teste_lógico for verdadeiro executa-se o corpo do for, caso contrário segue-se para as instruções fora do for. Após se executar as instruções dentro do for, segue-se para a parte dos incrementos/decrementos, para então em seguida, ir para o teste_lógico, que se verdadeiro fará com que o bloco de instruções dentro do for seja executado. Após a execução do bloco de instruções dentro do for, sempre se seguirá para a parte dos incremento/decremento e depois para a parte do teste lógico.

c) do ... while (**faça ... enquanto**)

```
do {
    Bloco de instruções;
}while ( CONDIÇÃO );
```

Este comando faz para depois testar. Se o teste lógico for verdadeiro, a repetição é realizada. Caso contrário, a execução segue para a 1ª instrução após o do ... while.

Todo bloco de instrução (conjunto com 2 ou mais instruções) deve estar entre chaves. Isso ocorrerá no if, no if/else, no while, no for, no do ... while.

1.6. Saída e entrada de dados

Não se esqueça de dar `#include <iostream>` e dê logo em seguida:
`using namespace std;`

1.6.1. Saída de dados – Exemplos:

```
cout << "Mensagem";  
cout << "Mensagem \n";  
cout << variável;  
cout << variável << "\n";  
cout << "Quociente : " << x/y ;
```

1.6.2. Entrada de dados

```
int idade;  
cin >> idade;
```

O código especial `\n` que faz com que uma linha seja pulada. Existem outros códigos especiais, como por exemplo, o `\t` que dá um espaço equivalente a tecla TAB.

A linguagem C++ é case sensitive, ou seja, faz distinção entre maiúsculas e minúsculas.

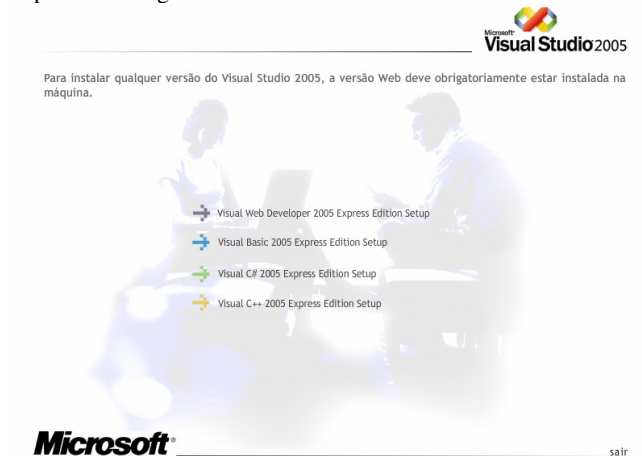
Os comentários devem estar entre `/*` e `*/` ou após `//` . Este último permite comentário de uma linha, ao passo que o primeiro permite alongar o comentário por mais linhas.

ANOTAÇÕES:

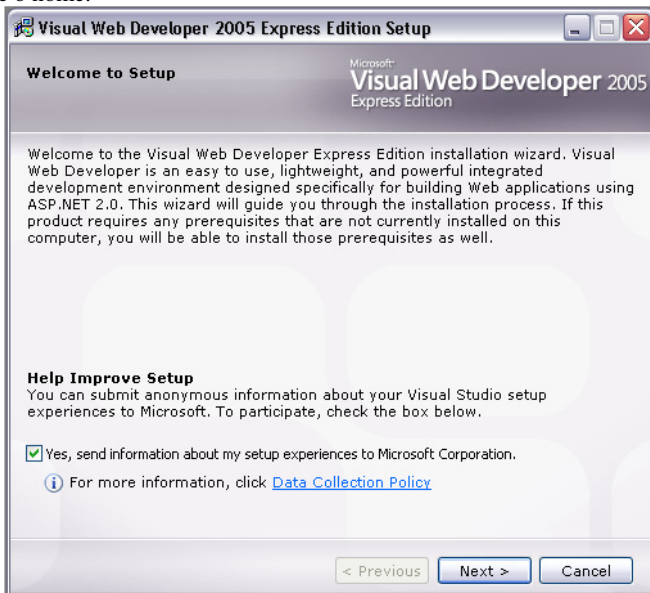
2 – Tutorial de Visual C++

2.1. Instalação:

2.1.1. Insira o disco do Visual Studio 2005 no drive de CD-ROM do seu computador. Aparecerá a seguinte tela:

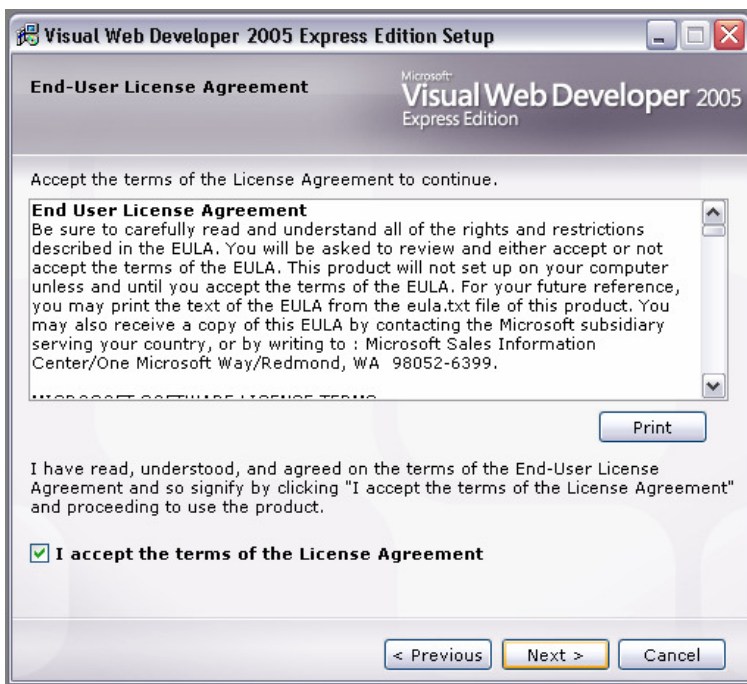


2.1.2. Primeiro deve instalar o Visual Web Developer 2005 Express Edition. Clique sobre o nome.



2.1.3. Nesta tela marque a opção do “Yes, ...” e clique em “Next >”.

2.1.4. Agora marque a opção “I accept ...” e clique em “**Next >**”.



2.1.5. Depois é só escolher todas as opções, clicar em “Next >” e esperar a instalação.

2.1.6. Volte para a tela inicial e agora podemos instalar o Visual C++ 2005 Express Edition. Clique sobre o nome.

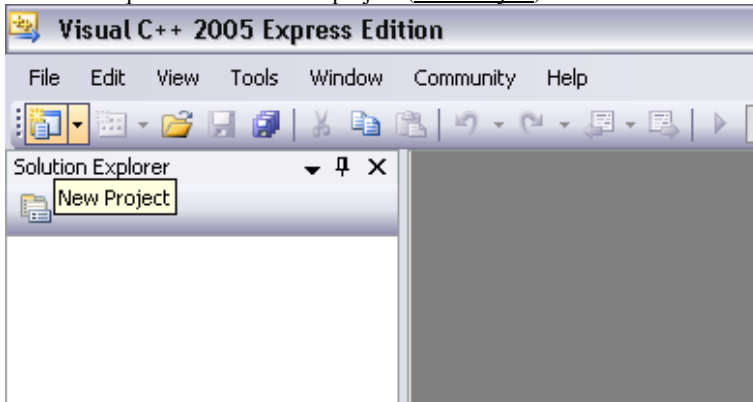
2.1.7. Proceda de maneira similar a instalação do Visual Web Developer 2005 Express Edition. Basta aceitar as licenças e clicar em “**Next >**” até o fim da instalação.

ATENÇÃO: Você deve instalar o Visual Web Developer 2005 Express Edition antes de instalar o Visual C++ 2005 Express Edition.

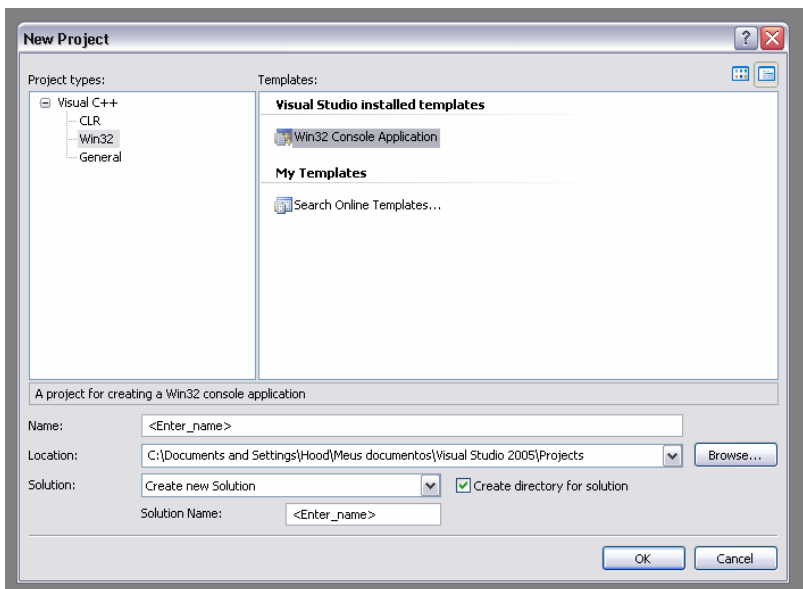
Depois de instalado o pacote ocupa cerca de 670Mb no disco rígido.

2.2. Criação de Projeto:

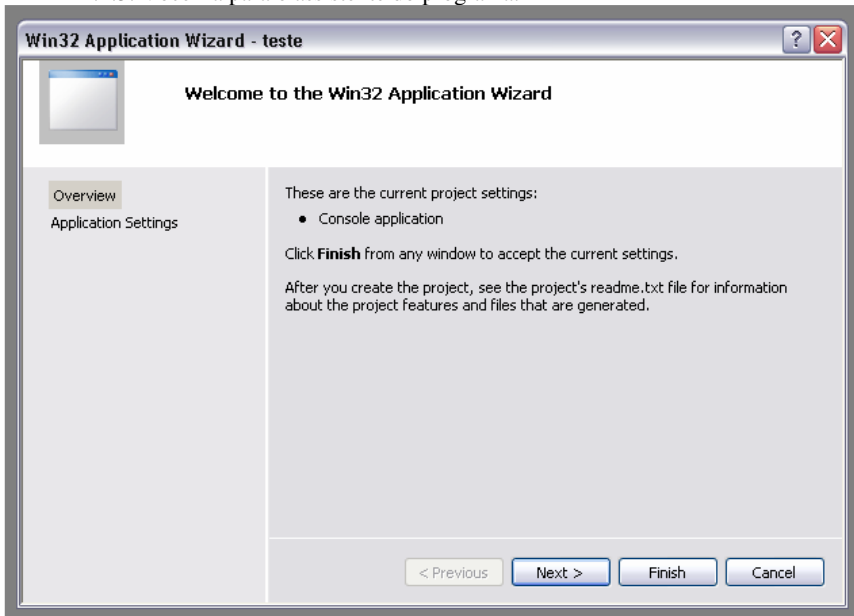
2.2.1. Clique no botão de novo projeto (New Project).



2.2.2. Escolha a opção Win32 Console Application (dentro do item Win32).

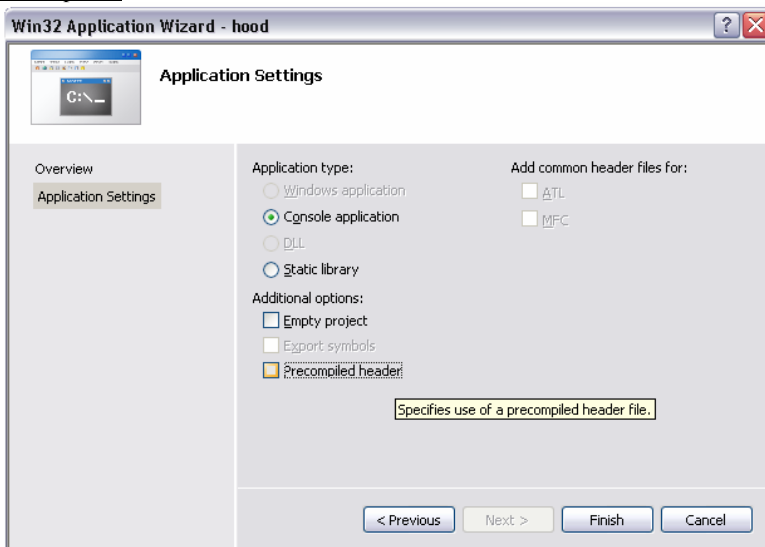


ATENÇÃO: Dê um nome para o projeto (onde está escrito **<Enter_name>**).

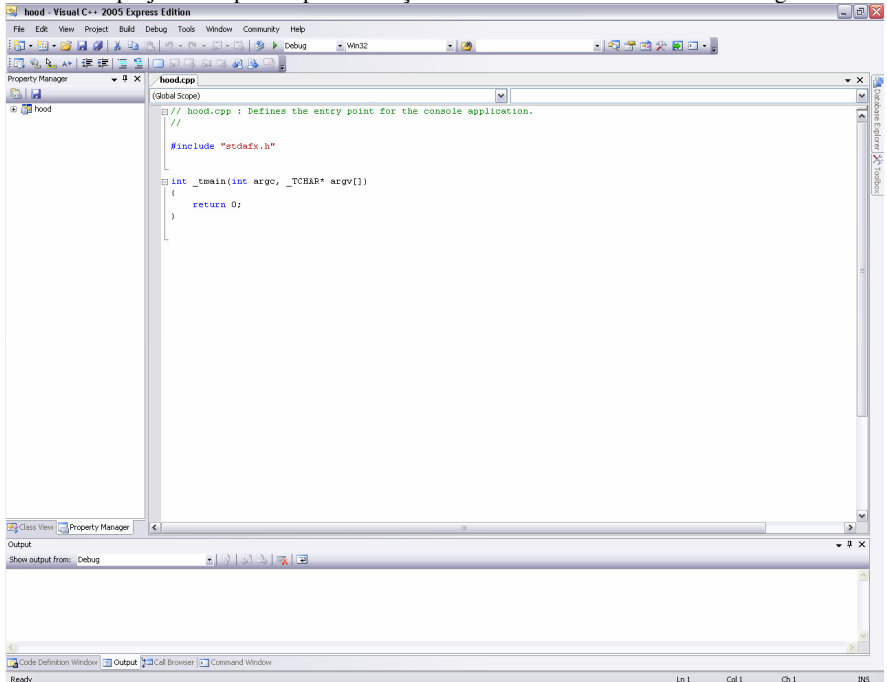


Nesta tela deve-se clicar no botão “**Next >**”. Indo para a tela seguinte.

2.2.4. Certifique-se de desmarcar a opção Precompiled header dentro do item Additional options:



O projeto está pronto para começar. Você verá seu ambiente como a seguir:



Na área principal da janela você verá o seguinte código padrão:

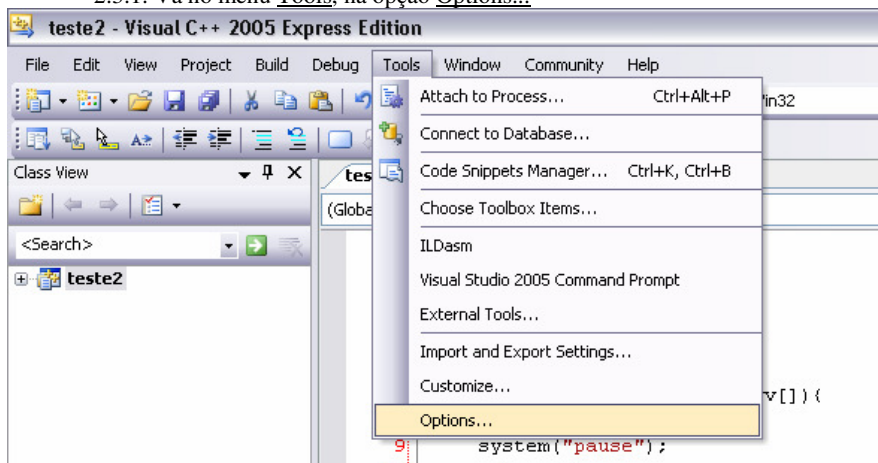
```
// Novo_projeto.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

Você pode substituir todo esse código pelo seguinte:

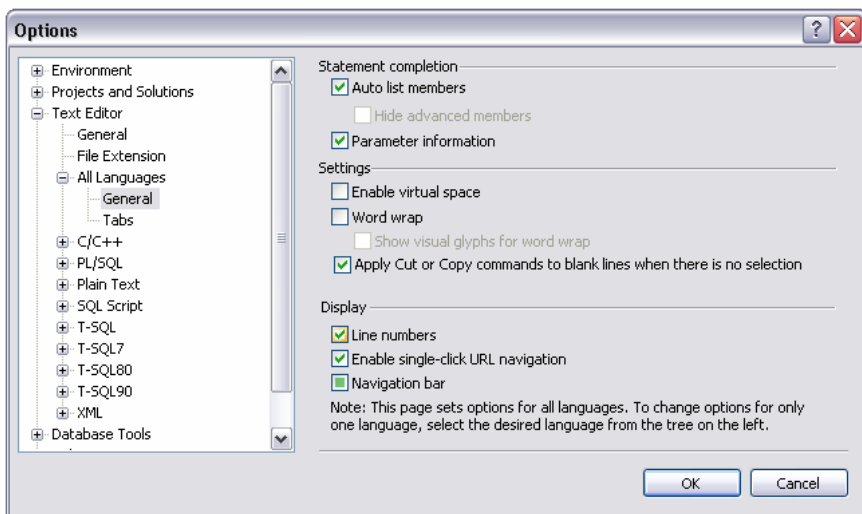
```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(int argc, char* argv[]){
    /* Área de definição de variáveis */
    /* Área de entrada de dados */
    /* Área de processamento dos dados */
    /* Área de saída de dados */
    /* Pausa e retorno no fim do programa */
    cout << "\n";
    system("PAUSE");
    return 0;
}
```

2.3. Exibir numeração das linhas:

2.3.1. Vá no menu Tools, na opção Options...



2.3.2. Irá abrir uma janela de configuração, na área da esquerda, vá no item Text Editor, sub item All Languages, opção General. Na área da direita é só marcar a opção Line numbers dentro do item Display.



2.4. Configurar para o Visual C++ abrir com o padrão do C:

Se você desejar pode alterar um arquivo de configuração do Visual C++ para não ter de copiar e colar o padrão do C toda vez que iniciar um novo projeto.

2.4.1. Localize no seu disco rígido a pasta:

C:\Arquivos de programas\Microsoft Visual Studio 8\VC\VCWizards\AppWiz\Generic\Application\templates\I033

Note que está sendo considerado que a pasta inicial de instalação do programa não foi alterada. Se você optou por alterar a pasta na instalação substitua C:\Arquivos de programas\Microsoft Visual Studio 8\ pela pasta em que ficou sua instalação.

2.4.2. Lá dentro você encontrará o arquivo root.cpp.

2.4.3. Faça uma cópia de segurança deste arquivo para evitar problemas.

2.4.4. Abra o arquivo em um editor de texto (Bloco de Notas ou Word Pad).

2.4.5. Localize e apague a linha:

```
#include "stdafx.h"
```

2.4.6. Localize a linha:

```
#![if CONSOLE_APP]
```

2.4.7. Depois dessa linha você verá o código padrão que costuma aparecer:

```
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

Depois deste trecho de código deve haver ainda mais uma pequena parte:

```
#![endif]
#endif]
```

2.4.8. Troque o código pelo padrão do C listado no quadro abaixo, mas lembre-se de deixar os dois `#![endif]` que se encontram depois.

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char* argv[]){
    /* Coloque seu código aqui */
    cout << "\n";
    system("PAUSE");
    return 0;
}
```

3 – Modularização

3.1. Fundamentos da modularização de código

Um programa em C/C++ tipicamente tem uma estrutura seqüencial, nesta unidade veremos as vantagens de um código modularizado com a utilização de funções, que são estruturas que permitem aos usuários separar a estrutura dos seus programas em blocos, permitindo dividir programas grandes em vários blocos menores facilitando a compreensão e a manutenção.

Uma função em C/C++ tem a seguinte estrutura:

```
tipo_do_retorno nome_da_função (parâmetros_de_entrada) {
    /* aqui fica o corpo da função */
}
```

O tipo do retorno é o tipo de dado do valor a ser retornado pela função em questão. O nome da função não pode começar com letra maiúscula, número ou caractere simbólico e não deve conter espaços. Os parâmetros de entrada são na verdade uma lista contendo o tipo para cada um dos valores que serão recebidos pela função, seguido de um nome para uma variável que alocará o mesmo valor em memória.

Toda função termina depois de realizar os comandos encontrados em seu corpo, ou quando encontrar o comando *return* (o que acontecer primeiro). O comando *return* deve vir acompanhado de um valor se a função retornar um tipo diferente de *void*.

As funções podem ser declaradas antes da função principal *main*, onde devem ser descritas por inteiro; ou no depois do *main*, onde além de conteúdo da função se faz necessário a apresentação de um protótipo da mesma antes do *main* para que a mesma possa ser interpretada pelo compilador. Um protótipo de função tem o seguinte formato:

```
tipo_do_retorno nome_da_função (parâmetros_de_entrada);
```

3.1.1. Funções predefinidas: Na linguagem C/C++ existem uma série de funções pré-definidas que possuem sua interface conhecida (nome da função com lista de parâmetros necessária bem como seu tipo de retorno) e que já são de domínio da maioria dos alunos, só para citar exemplos: *sqrt* (raiz quadrada), *pow* (eleva uma potência), *cin* (entrada de dados), *cout* (saída de dados), etc.

3.1.2. Declaração e definição de funções: um usuário também pode e deve produzir suas próprias funções. Como por exemplo a função a seguir que recebe um número e retorna o número ao quadrado:

```
#include <iostream>
using namespace std;
void quadrado (int n){      return (n * n);      }
int main (int argc, char* argv[]) {
    int num;
    cout << "Digite um número: ";
    cin >> num;
    cout << "\nO quadrado vale: " << quadrado(num);
    return 0;
}
```

3.2. Escopo de variáveis

Toda variável tem escopo local, ou seja, a visibilidade e validade de seu nome e conteúdo são válidos apenas dentro da função que a define. A exceção a esta regra se dá quando a variável é definida globalmente, fora do escopo da função main e de qualquer outra variável. No exemplo a seguir a variável global está em vermelho e as locais em verde. Note que podem haver variáveis locais com nomes repetidos, desde que em escopos diferentes, e que analogamente, uma variável global (cujo escopo é todo o programa) não pode ter seu nome repetido.

```
#include <iostream>
#include <cstdlib>
using namespace std;
g = 10;
int junta (int b){
    int a;
    cout << "\nValor de G: " << g;
    a = b + g;
    g = g + a;
    return a;
}
int main(int argc, char* argv[]){
    int a = 5, b;
    cout << "\nValor de A: " << a;
    b = junta(a);
    cout << "\nValor de B: " << b;
    cout << "\nValor de G: " << g;
    cout << "\n\t"; system("PAUSE");
    return 0;
}
```

Como exercício faça o chinês da função acima e verifique qual a sua impressão na tela para cada um das linhas de “cout”.

ANOTAÇÕES:

4 – Introdução às estruturas de dados em C++

4.1. Vetores

Vetores são estruturas homogêneas que possibilitam o armazenamento de um conjunto de valores em um espaço contínuo da memória (ou seja, as posições de memória onde estão armazenados os elementos de um mesmo vetor estão em sequência). Um vetor é uma estrutura de dados homogênea porque todos os seus elementos devem ser do mesmo tipo de dado.

Declaração:	tipo nome_do_vetor[tamanho];
-------------	------------------------------

Onde tipo indica o tipo do dado dos valores a serem armazenados no vetor, nome_do_vetor indica o nome da variável vetor e tamanho indica a quantidade de elemento que o vetor pode armazenar.

Exemplo:	int vetA[5];
----------	--------------

Cada elemento do vetor é acessado por um índice, que representa a posição relativa do elemento em relação ao vetor. Em C/C++ o primeiro elemento tem índice 0 (zero). É importante observar que o terceiro elemento do vetor, terá índice 4 (já que o primeiro tem índice 0).

Observação importante: Um vetor é uma variável estática e portanto, seu tamanho deve ser definido antes da compilação. Ou seja, não é possível declarar o vetor na forma: `int vetA[n];` onde o valor de `n` só será conhecido em tempo de execução. Porém, alguns compiladores aceitam esta forma de declaração e, durante a compilação, geram o código para alocação dinâmica (que será visto posteriormente) do espaço de memória para armazenamento dos elementos do vetor.

4.1.1. Acesso a um elemento: é realizado através do nome do vetor, seguido do seu índice entre colchetes: nome [índice];

Um vetor pode ser inicializado na declaração.

<code>int vetB[3] = { 10, 20, 30};</code>

Nem todas as posições precisam ser inicializadas:

<code>int vetC[4] = { 10, 20};</code>

Quando isto ocorrer as demais posições ficam inicializadas com zero (no caso de ser um vetor de inteiros). Assim, quando é desejado inicializar um vetor com 0 é comum usar: `int vetC[5] = {0};`

Observação: Nos compiladores mais recentes do C++, não há necessidade do comando acima, pois todo vetor de inteiros declarado já é inicializado com 0.

4.1.2. Strings: Vetores de caracteres representam strings (variáveis em que normalmente são armazenados nomes de pessoas, endereços, enfim, textos). C/C++ não possui o tipo primitivo string (apesar de que o C++ possui a classe String – que não é o foco deste curso, já que ainda não se está usando orientação a objetos). Quando um vetor de caracteres é declarado, deve-se sempre defini-lo com uma posição a mais da maior string a ser armazenada no vetor: Assim, se o vetor irá armazenar no máximo 30 caracteres, ele deve ser declarado com o tamanho 31. Pois o C/C++ reserva a posição após os caracteres válidos, para o caracter `\0` (nulo) – caracter terminador da string.

<code>char nomePessoa [10];</code>

Pode-se inicializar uma string na sua declaração, como mostram os exemplos na página a seguir:

```
char nomePessoa [10]="Paulo";
ou
char nomePessoa [10]='P','a','u','l','o','\0';
```

Porém a atribuição de um valor a um vetor de caracteres não é possível através do operador = (atribuição).

```
char string1, string2={"Ruy"};    ← OK
string1= string2                  ← ERRADO!
```

4.2. Vetores como parâmetros de funções

Vetores quando declarados já são ponteiros por definição, de maneira que sempre podemos trata-los como passagem de valores por referência (quando se usa ponteiros) ao invés de por valor (quando se passa uma cópia do valor da variável para a função).

Um exemplo simples de passagem de valores por referência pode ser vista no programa a seguir que troca os valores de duas variáveis inteira através da função *troca*:

```
#include <iostream>
#include <cstdlib>
using namespace std;
void troca (int *x, int *y) {
    int temp; temp = *x; *x = *y; *y = temp;
}
int fim(){
    cout << "\n"; system("PAUSE"); return 0;
}
int main(int argc, char* argv[]){
    int a = 3, b = 4;
    cout << "\nTroca com referencia:";
    cout << "\nPassando A = 3 e B = 4:";
    troca (&a, &b);
    cout << "A = " << a << " e B = " << b << "\n";
    return(fim());
}
```

No programa anterior o & (e comercial) na frente da variável serve para indicar que estamos passando como parâmetro o endereço dela e não seu valor. Na função *troca*, note que o nome do parâmetro recebido é precedido de * (asterísco), que indica que se recebe um ponteiro para o tipo e não uma variável do tipo.

A vantagem do ponteiro é permitir uma manipulação como se fosse global mantendo a segurança da declaração local da variável. É uma ferramenta muito versátil.

No programa a seguir trabalharemos com o mesmo conceito aplicado a vetores, desta vez o programa deverá ler dois vetores de inteiros e depois comparar a soma dos elementos de ambos os vetores para que possa declarar qual dos dois possui a maior soma.

A função acessória *somaVet* recebe um ponteiro para o vetor de inteiros e retorna para o usuário a soma dos elementos deste vetor.

A função acessória *compara* recebe ponteiros para dois vetores de inteiros, chama a função *somaVet* para cada um, guardando seus retornos em variáveis temporárias *sa* e *sb*, para depois a partir de uma comparação dos valores retornar -1 (se *sa* > *sb*), 1 (se *sb* > *sa*) ou 0 caso ambos empatem em suas somas.

Por fim, a função *saida* faz um tratamento no retorno da função *compara* para dar ao usuário uma mensagem mais significativa sobre o resultado do programa.

```

#include <iostream>
#include <cstdlib>
#define TAM 5
using namespace std;
int somaVet (int *vet){
    int i, r = 0;
    for (i = 0; i < TAM; i++){ r = r + vet[i]; }
    return r;
}
int compara (int *la, int *lb){
    int sa = somaVet(la), sb = somaVet(lb);
    if (sa > sb){ return -1; }
    else{      if (sb > sa){ return 1; }
              else{      return 0; }
    }
}
void saida (int r){
    if (r == -1){ cout << "\nO primeiro tem maior soma..."; }
    if (r == 0){cout << "\nAs somas dao o mesmo valor...";}
    if (r == 1){cout << "\nO segundo tem maior soma...";}
}
int fim(){ cout << "\n"; system("PAUSE"); return 0; }
int main(int argc, char* argv[]){
    int la[TAM], lb[TAM], i;
    for (i = 0; i < TAM; i++){
        cout << "Digite o elemento " << i+1 << " da lista A: ";
        cin >> la[i];
    }for (i = 0; i < TAM; i++){
        cout << "Digite o elemento " << i+1 << " da lista B: ";
        cin >> lb[i];
    }cout << "\nComecando a comparacao . . . \n";
    cout << "\nSoma do Vetor A: " << somaVet(la);
    cout << "\nSoma do Vetor B: " << somaVet(lb);
    saida(compara(la, lb));
    return(fim());
}

```

4.3. Structs

Ou estrutura, é uma coleção de variáveis referenciadas através de um nome definido pelo programador. É semelhante ao record da linguagem Pascal. Na realidade a idéia com o uso de struct é definir um *registro de dados*.

Definição de uma struct:

```

struct nome_da_struct{
    tipo1 var1;
    tipo2 var2;
    ...
    tipon varn;
} variáveis;

```

Exemplo:

```
struct Data {
    int dia; int mes; int ano;
};
```

Data é o nome_da_struct. É oportuno observar que um novo tipo de dado foi definido, chamado Data, mas nenhuma variável deste tipo foi declarada. Tipo de dado não deve ser confundido com variável. int é um tipo de dado e, com as linhas de comando do exemplo acima, o tipo Data passa a existir. Observe ainda que o tipo de dado struct Data é composto de 3 elementos: dia, mês e ano, chamados de campos (ou membros) da struct Data.

4.3.1. Declarando-se uma variável struct: para se declarar uma variável de um tipo struct deve-se usar:

```
struct nome_da_struct nome_da_variável;
```

Exemplo:

```
struct Data dt;
int x;
```

Os tipos de dados são: int e struct Data, enquanto as variáveis são: x e dt, respectivamente. Ou ainda, a variável dt poderia ter sido definida no momento da declaração da struct, note assim ela seria uma variável global:

```
struct Data {
    int dia; int mes; int ano;
}dt;
```

Um exemplo de struct composta por campos cujos tipos dos dados são diferentes:

```
struct Pessoa {
    int idade;
    char nome[51]; // string, ou seja vector de 51 caracteres
};
```

4.3.2. Acessando elementos da struct: depois de ser declarada uma variável dt do tipo struct Data, pode-se acessar os diversos elementos através do operador . (ponto):

```
dt.dia
dt.mes
dt.ano
```

Programa completo para imprimir o conteúdo de uma struct Pessoa:

```
#include <iostream>
#include <cstdlib>
using namespace std;
struct Pessoa {
    int idade; char nome[51];
};
int main(int argc, char* argv[]) {
    struct Pessoa pes={27,"Rodrigo Dias"};
    cout << "Idade: " << pes.idade << "\n";
    cout << "Nome: " << pes.nome << "\n";
    system("PAUSE");
    return 0;
}
```

Notas de Aula de Algoritmos e Estruturas de Dados

4.3.3. Manipulação entre variáveis de um tipo struct: deve-se ter alguns cuidados na manipulação entre variáveis do tipo struct. Para exemplificar a struct Pessoa será utilizada. Supondo que foram declaradas 3 variáveis do tipo struct Pessoa: pes1, pes2, pes3:

```
...
// declarando as variáveis pes1, pes2 e pes3
struct Pessoa pes1, pes2, pes3;
...
cout << "Digite a idade da primeira pessoa: ";
cin >> pes1.idade;
cout << "Digite o nome da primeira pessoa: ";
cin.get(); cin.get(pes1.nome,50); // leitura de string
...
```

É possível a atribuição de uma variável de um tipo struct em outra do mesmo tipo struct (o que é muito interessante):

```
pes2 = pes1;
```

Na linha de comandos acima, são copiados ambos os elementos (o nome e a idade) de pes1 para pes2.

É possível ainda a comparação (igualdade e desigualdade) entre duas variáveis do tipo struct:

```
if (pes1 == pes2)
```

No comando if acima, o resultado só será verdadeiro de ambos os elementos de cada uma das variáveis struct forem iguais, ou seja, a idade e o nome em pes1 deve ser igual a idade e o nome em pes2 respectivamente.

4.4. Structs como parâmetros de funções

Como qualquer outro tipo de variável uma struct pode ser passada como parâmetro por referência ou por valor.

No programa a seguir a função *checkDate* recebe a struct *dma* (que guarda o dia o mês e o ano de uma data) por valor, pois só precisa dos valores contidos nela para uma tomada de decisão. Já a função *leitura* recebe a struct por referência, pois ela fará a leitura dos valores que devem popular a struct definida no escopo do *main*.

```
#include <iostream>
#include <cstdlib>
using namespace std;
struct dma {
    int dia;
    int mes;
    int ano;
};
void leitura (struct dma *d){
    cout << "Digite o dia: ";
    cin >> x->dia;
    cout << "Digite o mes: ";
    cin >> x->mes;
    cout << "Digite o ano: ";
    cin >> x->ano;
}
```

```

void checkDate (struct dma d){
    if ((d.ano > 2010) || (d.ano < 0)){ cout << "Ano invalido...\n"; return; }
    else {
        if ((d.mes < 1) || (d.mes > 12)){ cout << "Mes invalido...\n"; return; }
        else{
            if ((d.dia < 1) || (d.dia > 31)){ cout << "Dia invalido...\n"; return; }
            if ((d.mes == 4) || (d.mes == 6) || (d.mes == 9) || (d.mes == 11)){
                if (d.dia == 31){cout<<"O mes " << d.mes << " tem 30 dias...\n";}
                return;
            }if (d.mes == 2){
                if ((d.ano % 4) == 0){
                    if (d.dia > 29){cout<<"O mes 2 tem 29 dias neste ano...\n"; return;}
                } else {
                    if (d.dia > 28){ cout<<"O mes 2 tem 28 dias neste ano...\n"; return;}
                }
            }
        }
    }
    cout << "Data OK: " << d.dia << "/" << d.mes << "/" << d.ano << "\n";
}

int main(int argc, char* argv[]){
    struct dma x;
    leitura(&x);
    cout << "Resposta: \n"; checkDate(x);
    cout << "\n\n\t"; system("PAUSE");
    return 0;
}

```

Note que na passagem por referência de uma struct você passa a manipular seus elementos utilizando o separador -> (menos e maior) ao invés do . (ponto), isso se dá para mostrar uma simbologia de utilização por referência da variável (o ponteiro).

ANOTAÇÕES:

5 – Busca

5.1. Sequencial

Este tipo de busca caracteriza-se por um conjunto de dados que não está ordenado e a única maneira de localizar um elemento é percorrer sequencialmente toda coleção atrás do elemento desejado até encontra-lo. Note que se o elemento que deseja-se localizar pode não estar entre os da coleção e neste caso a busca obrigatoriamente deverá varrer toda a coleção para poder constatar este fato.

Por exemplo no programa a seguir lemos um vetor (*ler*) depois procuramos em todas suas posições até achar um certo elemento (*achaPos*) e por fim imprimimos como resultado, a posição em que o elemento se encontrava ou uma mensagem de erro caso não exista no vetor lido (*resp*):

```
#include <iostream>
#include <cstdlib>
#define TAM 10
using namespace std;
void ler (int l[TAM]){
    int i;
    for (i = 0; i < TAM; i++){
        cout << "Digite o valor da posicao " << i << " :";
        cin >> l[i];
    }
}
int achaPos (int elem, int *l){
    int i;
    for (i = 0; i < TAM; i++){
        if (l[i] == elem) return i;
    }
    return -1;
}
void resp (int r){
    if (r == -1) cout << "Elemento nao encontrado no vetor\n";
    else cout << "Elemento encontrado na posicao " << r << "\n";
}
int fim(){ cout << "\n"; system("PAUSE"); return 0; }
int main(int argc, char* argv[]){
    int lista[TAM], elem, pos;
    ler(lista);
    cout << "Que elemento procurar: ";
    cin >> elem;
    pos = achaPos(elem, lista);
    resp(pos);
    return(fim());
}
```

Note que o vetor sempre é passado por referência (isto é implícito), mesmo quando não mencionamos (como na função *ler*).

5.2. Binária

Para que esta busca seja possível é implícita à natureza do proble que a coleção de dados a ser utilizada esteja ordenada. A estratégia é bem simples. Deve-se manter o controle de três posições ao longo da busca: o início da coleção, o fim da coleção e o meio da coleção. A busca começa na posição média da coleção (o meio, claro!) e verifica-se se esta é a posição procurada, caso positivo algoritmo encerrado, caso negativo deve-se verificar se o valor procurado é maior ou menor que o valor presente na posição média. Dependendo do caso a algoritmo passa a dedicar a busca na metade da coleção condizente com a opção, abandonando a outra metade (na primeira metade se for menor e na segunda se for maior). Esses passos prosseguem até que se ache o elemento a ser encontrado ou até o marcador de início encontrar o de fim (sinal que percorremos toda a coleção).

Veja o exemplo seguinte o programa tenta “adivinhar” um número entre 1 e 1024 imaginado pelo usuário:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(int argc, char* argv[]){
    int cont = 0, op = 0, ini = 1, fim = 1024, metade;
    cout << "Escolha um numero entre 1 e 1024\n\n\t"; system("PAUSE");
    while (op != 1){
        cont++;
        if (op == 2 ) {fim = metade - 1;}
        if (op == 3 ) {ini = metade + 1;}
        metade = (ini + fim) / 2;
        cout << "\nO numero e:\t" << metade;
        cout << "\n1 - Sim";
        cout << "\n2 - Nao, e menor";
        cout << "\n3 - Nao, e maior";
        cout << "\nDigite sua opcao: ";
        cin >> op;
    }cout << "\nAcertei em " << cont << " tentativas. \n";
    system("PAUSE");
    return 0;
}
```

Faça o chinês para este programa e verifique como ele nunca erra (na verdade ele vai acertar sempre me até 10 tentativas!) se o usuário não roubar (como imaginar um número fora da faixa 1-1024 ou ficar trocando de número...).

5.3. Tabela Hash

Também conhecida por tabela de espalhamento, é uma estrutura de dados especial, que associa chaves de pesquisa (hash) a valores. Seu objetivo é, a partir de uma chave simples, fazer uma busca rápida e obter o valor desejado.

Tabelas hash são tipicamente utilizadas para implementar vetores associativos, sets e caches. São tipicamente usadas para indexação de grandes volumes de informação (como bancos de dados). A implementação típica busca uma função hash em que se chegue a qualquer registro sempre com o mesmo número de passos fixos (de preferência o menor possível), não importando o número de registros na tabela (desconsiderando colisões). O ganho com relação a outras estruturas associativas (como um vetor simples) passa a ser maior conforme a quantidade de dados aumenta.

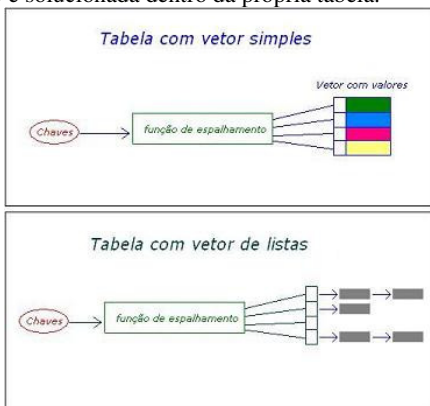
Notas de Aula de Algoritmos e Estruturas de Dados

Outros exemplos de uso das tabelas hash são as tabelas de transposição em jogos de xadrez para computador até mesmo em serviços de DHCP.

5.3.1. A função de espalhamento: Ou função hash, é a responsável por gerar um índice a partir de determinada chave. Caso a função seja mal escolhida, toda a tabela terá um mau desempenho. O ideal para a função de espalhamento é que sejam sempre fornecidos índices únicos para as chaves de entrada. A função perfeita seria a que, para quaisquer entradas A e B, sendo A diferente de B, fornecesse saídas diferentes. Quando as entradas A e B são diferentes e, passando pela função de espalhamento, geram a mesma saída, acontece o que chamamos de colisão.

Na prática, funções de espalhamento perfeitas ou quase perfeitas são encontradas apenas onde a colisão é intolerável (por exemplo, nas funções hash da criptografia), ou quando conhecemos previamente o conteúdo da tabela armazenada). Nas tabelas hash comuns a colisão é apenas indesejável, diminuindo o desempenho do sistema. Muitos programas funcionam sem que seu responsável suspeite que a função de espalhamento seja ruim e esteja atrapalhando o desempenho.

Por causa das colisões, muitas tabelas hash são aliadas com alguma outra estrutura de dados, tal como uma lista encadeada ou até mesmo com árvores balanceadas. Em outras oportunidades a colisão é solucionada dentro da própria tabela.



5.3.2. Exemplo de função de espalhamento e colisão: Imagine que seja necessário utilizar uma tabela hash para otimizar uma busca de nomes de uma lista telefônica (dado o nome, temos que obter o endereço e o telefone). Nesse caso, poderíamos armazenar toda a lista telefônica em um vetor e criar uma função de espalhamento que funcionasse de acordo com o seguinte critério:

Para cada nome começado com a letra A, devolver 0
Para cada nome começado com a letra B, devolver 1
... (e assim sucessivamente até que) ...
Para cada nome começado com a letra Z, devolver 25

O exemplo anterior poderia ser implementado, em C, da seguinte forma:

```
int hash(char *chave){  
    return (chave[0]-65);  
}
```

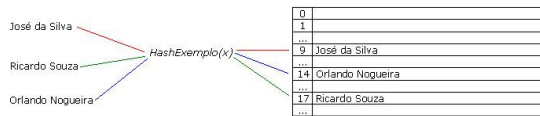
Agora vamos inserir alguns nomes em nossa lista telefônica:

José da Silva; Rua A, 35; Tel.: 1111-1111

Ricardo Souza; Rua B, 54; Tel.: 2222-2222

Orlando Nogueira; Rua C, 125; Tel.: 3333-3333

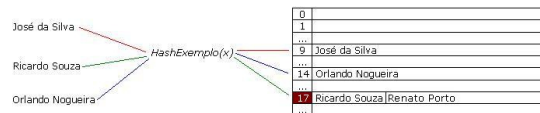
Nossa função distribuiria os nomes assim:



Agora se inserimos mais um nome:

Renato Porto; Rua D, 687; Tel.: 4444-4444

Teremos uma colisão:



Como se pode notar, a função de exemplo causaria muitas colisões. Se inserirmos um outro nome começado com a letra R, teremos uma outra colisão na letra R. Se inserirmos "João Siqueira", a entrada estaria em conflito com o "José da Silva". Abaixo um exemplo de programa para testar esta função de hash:

```
#include <iostream>
#include <cstdlib>
#define NOMES 28
using namespace std;
struct agenda{
    char nome[50]; enredeco[50]; tel[10];
};
int hash(char *chave){
    return (chave[0]-65);
}
int main(int argc, char* argv[]){
    struct agenda a[NOMES];
    char nome[50];
    int op=1, pos;
    while (op != 0){
        cout << "Digite um nome: ";
        cin >> nome;
        pos = hash(nome);
        cout << "Esse nome seria armazenado na posicao ";
        cout << pos << ".\n";
        cout << "Continuar? [0 - Nao / 1 - Sim] ";
        cin >> op;
    }system("PAUSE");
    return 0;
}
```

6 – Ordenação

6.1. Inserção

Primeiro método de ordenação criado, tem por objetivo manter uma coleção de elementos ordenados simulando a inserção dos mesmos na coleção sempre na sua posição apropriada considerando os elementos já inseridos. Pode ser utilizado para realizar inserção de novos elementos na coleção e assim manter a lista sempre organizada.

A idéia é bem simple, começa-se com uma coleção de dois elementos (tipicamente os dois primeiros da coleção), e verifica-se se os mesmos já estão em ordem (se não estiverem trocam de lugar). Depois vai-se inserindo sequencialmente os elementos subseqüentes da coleção sempre na posição apropriada. Veja o exemplo a seguir em C/C++:

```
#include <iostream>
#include <cstdlib>
#define TAM 10
using namespace std;
void insertSort(int v[], int tamanho) {
    int i, j, valor;
    for(i = 1; i < tamanho; i++) {
        valor = v[i];
        for (j=i-1; (j>=0) && (v[j]>valor); j--){ v[j + 1] = v[j]; }
        v[j+1] = valor;
    }
}
int main(int argc, char* argv[]){
    int a[50]={9,7,5,3,1,2,4,6,8,0}, i;
    for (i=0;i<TAM;i++){ cout << a[i] << " "; }
    cout << "\n\nChamando ordenacao...\n\n";
    insertSort(a,TAM);
    for (i=0;i<TAM;i++) { cout << a[i] << " "; }
    cout << "\n\n";
    system("PAUSE");
    return 0;
}
```

A função *insertSort* é que ordena o vetor *a* inicial (que tem 50 posições mas está em uso apenas 10). Note que a função recebe não só o vetor mas como também o número de elementos do vetor.

Exercícios:

1. Faça o chinês do programa acima para que o mesmo exiba o passo a passo da ordenação do vetor “*a*” inicializado como no exemplo.
2. Refaça a função *insertSort* para que a mesma ordene um vetor de struct de alunos como exemplo a seguir, levando como critério de ordenação as matrículas:

```
struct aluno {
    int matricula;
    char nome[50];
    float av1, av2, av3, media;
}
```

6.2. Seleção Direta

Outro método simples de ordenação é o método da seleção direta ou selection sort. Esse método pega um vetor e procura o seu menor elemento, assim que o esse elemento é encontrado ele é permutado com o primeiro elemento. Depois ele procura o menor elemento no subvetor que começa com o segundo elemento do vetor e permuta com o segundo elemento. O método faz isso até que o subvetor tenha somente 1 elemento, nesse caso o vetor já está ordenado.

Esse método também pode ser implementado procurando o maior elemento e trocando pelo último.

```
#include <iostream>
#include <cstdlib>
#define TAM 10
using namespace std;
void selectSort(int v[], int tamanho) {
    int i, j, menor;
    for(i = 0; i < (tamanho - 1); i++) {
        menor = v[i];
        for (j=i+1; j < tamanho; j++){
            if (v[j] < menor){
                v[i] = v[j];
                v[j] = menor;
                menor = v[i];
            }
        }
    }
}

int main(int argc, char* argv[]){
    int a[50]={9,7,5,3,1,2,4,6,8,0}, i;
    for (i=0;i<TAM;i++) { cout << a[i] << " "; }
    cout << "\n\nChamando ordenacao...\n\n";
    selectSort(a,TAM);
    for (i=0;i<TAM;i++) { cout << a[i] << " "; }
    cout << "\n\n";
    system("PAUSE");
    return 0;
}
```

A função *selectSort* ordenará o vetor lido com N posições (netse caso, $N = 10$), e assim como no exemplo anterior, a função recebe o vetor e o número de elementos do mesmo.

Exercícios:

3. Faça o chinês do programa acima para que o mesmo exiba o passo a passo da ordenação do vetor “a” inicializado como no exemplo.
4. Refaça a função *selectSort* para que a mesma ordene o vetor a partindo ao mesmo tempo das extremidades para o centro do vetor. Note que nas extremidades da esquerda (início do vetor) devem ficar os menores elementos e na da direita (fim do mesmo) devem ficar os maiores.

6.3. Bolha (Bubblesort)

O bubble sort, ou ordenação "por bolha", é um algoritmo de ordenação dos mais simples. A ideia é percorrer o vetor diversas vezes, a cada passagem fazendo flutuar para o início o menor elemento da sequência. Essa movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, daí o nome do algoritmo.

```
#include <iostream>
#include <cstdlib>
#define TAM 10
using namespace std;
void bubbleSort(int v[], int tamanho){
    int i, trocou, aux;
    do {
        tamanho--;
        trocou = 0;
        for(i = 0; i < tamanho; i++){
            if(v[i] > v[i + 1]){
                cout << "v[" << i << "]=" << v[i] << " v[" << i+1 << "]=" << v[i+1] << " - ";
                aux = v[i];
                v[i] = v[i + 1];
                v[i + 1] = aux;
                cout << "v[" << i << "]=" << v[i] << " v[" << i+1 << "]=" << v[i+1] << "\n";
                trocou = 1;
            }
        }
    } while(trocou);
}

int main(int argc, char* argv[]){
    int a[50]={9,7,5,3,1,2,4,6,8,0}, i;
    for (i=0;i<TAM;i++) { cout << a[i] << " "; }
    cout << "\n\nChamando ordenacao...\n\n";
    selectSort(a,TAM);
    for (i=0;i<TAM;i++) { cout << a[i] << " "; }
    cout << "\n\n";
    system("PAUSE");
    return 0;
}
```

As linhas em vermelho são apenas para auxiliar na visualização do algoritmo na hora em que o programa estiver rodando, elas vão mostrar passo a passo a ordenação do vetor de entrada.

Este método oferece uma grande vantagem sobre os dois anteriores pois ele não fica muito mais do que uma iteração completa para verificar que não é preciso ordenar o vetor. Esta vantagem vem principalmente do fato de o loop principal ser controlado por uma estrutura de *while*, ao invés de um *for* como nos outros. Com este advento o programa para de ficar tentando ordenar o vetor assim que percebe que não é necessário (não houveram mais trocas, ou seja *trocou* = 0).

Exercícios:

- Compare o algoritmo do *bubblesort* contra o *selectsort* e o *insertsort*, verificando para cada um quantas atribuições e quantos testes são feitos em cada um. Utilize o vetor a, dado como parâmetro nos 3 exemplos para sua métrica.

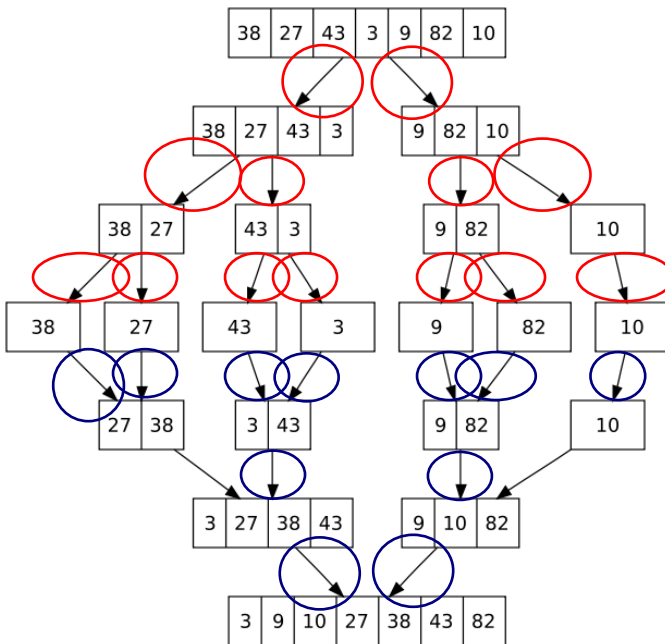
6.4. Mergesort

O merge sort, ou ordenação por mistura, é um exemplo de algoritmo de ordenação do tipo divisão e conquista. Sua idéia básica é que é muito fácil criar uma sequência ordenada a partir de duas outras também ordenadas. Para isso, ele divide a sequência original em pares de dados, ordena-as; depois as agrupa em sequências de quatro elementos, e assim por diante, até ter toda a sequência dividida em apenas duas partes.

Os três passos úteis dos algoritmos divisão e conquista (divide and conquer), que se aplicam ao merge sort são:

- Dividir: Dividir os dados em subsequências pequenas;
- Conquistar: Classificar as duas metades recursivamente aplicando o merge sort;
- Combinar: Juntar as duas metades em um único conjunto já classificado.

Veja a seguir um exemplo de vetor sendo ordenado pela estratégia do merge sort:



As setas emarcadas em vermelho são as divisões (sempre tentando visar o meio do vetor) e as em azul mostram o resultado da combinação de duas listas já ordenadas.

Exercícios:

* Nos itens a seguir (incluindo este), não haverão exercícios aparentes, pois os métodos serão temas de trabalho para compor a nota da AV2.

6.5. Quicksort

O quicksort (ou ordenação rápida) é outro algoritmo que adota a estratégia de divisão e conquista. Seus passos são:

- Escolha um elemento da lista, denominado *pivô*;
- Rearranje a lista de forma que todos os elementos anteriores ao pivô sejam menores ou iguais a ele, e todos os elementos posteriores ao pivô sejam maiores ou iguais a ele. Ao fim do processo o pivô estará em sua posição final (esta operação é denominada partição);
- Utilizando o próprio quicksort ordene a sublista dos elementos menores e a sublista dos elementos maiores;
- A condição de parada básica é que as listas de tamanho zero ou um, estão sempre ordenadas. O processo é finito pois a cada iteração pelo menos um elemento é posto em sua posição final e não será mais manipulado na iteração seguinte.

```
#include <iostream>
#include <cstdlib>
#define TAM 10
int divide(int x[], int inf, int sup){
    int a = x[inf], dir = sup, esq = inf, temp;
    while (esq < dir) {
        while (x[esq] <= a && esq<sup) esq++;
        while (x[dir] > a) dir--;
        if (esq < dir) {
            temp = x[esq];
            x[esq] = x[dir];
            x[dir] = temp;
        }
    }
    x[inf] = x[dir];
    x[dir] = a;
    return dir;
}
void quickSort (int x[], int inf, int sup){
    int j;
    if (inf >= sup) return;
    j= divide(x,inf,sup);
    quickSort(x,inf,j-1);
    quickSort(x,j+1,sup);
}
int main(int argc, char* argv[]){
    int a[50]={ 9,7,5,3,1,2,4,6,8,0}, i;
    for (i=0;i<TAM;i++) { cout << a[i] << " "; }
    cout << "\n\nChamando ordenacao...\n\n";
    selectSort(a,TAM);
    for (i=0;i<TAM;i++) { cout << a[i] << " "; }
    cout << "\n\n";
    system("PAUSE");
    return 0;
}
```


6.6. Shellsort

Criado por Donald Shell em 1959 (daí o seu nome, não tem nada a ver com concha, como muitos pensam), Shell sort é o mais eficiente algoritmo de classificação dentre os de complexidade quadrática. Basicamente o algoritmo passa várias vezes pela lista dividindo a em grupos menores. Nos grupos menores é aplicado outro método de ordenação, tipicamente o de ordenação por inserção (*insertsort*).

Por ser uma generalização do insertsort ele se aproveita de duas observações:

- O algoritmo de inserção é eficiente se a entrada está “quase” ordenada, e
- O algoritmo de inserção é ineficiente porque muda a penas um valor de posição por vez

```
#include <iostream>
#include <cstdlib>
#define TAM 10
using namespace std;
void shellSort(int v[], int tamanho){
    int i, j, valor, gap = 1;
    do {
        gap = 3 * gap + 1;
    } while(gap < tamanho);
    do {
        gap = gap / 3;
        for (i=gap; i < tamanho; i++){
            valor = v[i];
            j = i - gap;
            while ((j >= 0) && (valor < v[j])){
                v[j+gap] = v[j];
                j = j - gap;
            } v[j+gap] = valor;
        }
    } while(gap > 1);
}
int main(int argc, char* argv[]){
    int a[50]={9,7,5,3,1,2,4,6,8,0};
    int i;
    for (i=0;i<TAM;i++){
        cout << a[i] << " ";
    }cout << "\n\nChamando ordenacao...\n\n";
    shellSort(a,TAM);
    for (i=0;i<TAM;i++) cout << a[i] << " ";
    cout << "\n\n";
    system("PAUSE");
    return 0;
}
```

7 – Listas

Estruturas abstratas são o tema mais ricos em estruturas de dados, pois além da escolha dos tipos para representá-las na memória existe uma “política” que está diretamente ligada ao novo tipo abstrato. Isso quer dizer que além dos artifícios para garantir o armazenamento dos dados também se aplicam a eles métodos e/ou regras para ditar a maneira como será o acesso a elas. Veremos os principais deles.

7.1. Pilha

Tem esse nome pois sua filosofia se assemelha à de uma pilha de pratos. Em que elementos do mesmo tipo vão sendo empilhados uns sobre os outros e o acesso aos mesmos sempre são feitos pelo topo da pilha. É o que se diz de uma estrutura verticalizada.

A política estipulada para uma pilha é dita LIFO (last in, first out) ou seja, o último a entrar será o primeiro a sair, de maneira que só importa manter o controle sobre o elemento na posição mais alta da pilha.

Precisaremos impor algumas convenções para estudar a pilha, uma delas é a forma de representação que utilizaremos. Veja o exemplo a seguir:

```
struct pilha {
    int a[TAM];
    int topo;
    int tamanho;
};
```

Foi criado uma estrutura com uso do *struct* para que se possa representar a pilha. O vetor *a*, se destina a realizar o armazenamento dos dados da pilha. Neste exemplo uma pilha de inteiros (int). A variável *topo* vai guardar sempre a posição da área de armazenamento que está sendo acupada pelo elemento *mais alto* da pilha. E a variável *tamanho* vai guardar o tamanho atual da pilha, ou seja, quantos elementos ela está armazenando.

Além disso, criaremos também funções que realizaram as tarefas ligadas à manipulação da pilha. Note bem que a pilha só será manipulada através da utilização destas funções, não faremos acesso direto aos dados da pilha. São elas:

- void iniciaPilha (struct pilha *p){...} → Esta função fará a inicialização de uma nova pilha para que a mesma possa ser utilizada.
- bool pilhaVazia (struct pilha *p){...} → Esta função testará se uma pilha *p* encontra-se vazia, ou seja não pode se retirar mais nenhum elemento.
- bool pilhaCheia (struct pilha *p){...} → Esta função testará se uma pilha *p* encontra-se cheia, ou seja não pode receber mais nenhum elemento.
- bool push (struct pilha *p, int n){...} → Nome clássico da função destinada a empilhar um novo elemento *n* em uma pilha *p*.
- bool pop (struct pilha *p, int *r){...} → Nome clássico da função destinada a desempilhar um elemento *r* de uma pilha *p*.

Nas páginas seguintes há um exemplo de um programa que realiza as operações citadas sobre uma pilha e dá a opção de imprimir o conteúdo da mesma através da função *exibePilha*. Veja como ela foi programada:

```

#include <iostream>
#include <cstdlib>
#define TAM 200 // tamanho máximo para a área de armazenamento
using namespace std;
struct pilha {
    int a[TAM];
    int topo;
    int tamanho;
};
void iniciaPilha (struct pilha *p){
    p->topo = -1;
    p->tamanho = TAM - 1;
}
bool pilhaVazia (struct pilha *p){
    if (p->topo == -1) return true;
    else return false;
}
bool pilhaCheia (struct pilha *p){
    if (p->topo == p->tamanho) return true;
    else return false;
}
bool push (struct pilha *p, int n){
    if (pilhaCheia(p)){ return false; }
    else {
        p->topo++;
        p->a[p->topo] = n;
        return true;
    }
}
bool pop (struct pilha *p, int *r){
    if (pilhaVazia(p)){ return false; }
    else {
        *r = p->a[p->topo];
        p->topo--;
        return true;
    }
}
void exibePilha (struct pilha *p){
    int n; struct pilha t;
    iniciaPilha(&t);
    // Inverte a pilha para a pilha temporária t
    while (!pilhaVazia(p)){ pop(p,&n); push(&t,n); }
    cout << "\n\n";
    while (!pilhaVazia(&t)){
        pop(&t,&n);
        cout << n << " ";
        push(p,n);
    } // Inverte a pilha temporária t de volta para p exibindo na tela
}

```

```
int main(int argc, char* argv[]){
    struct pilha p;
    int x, op;
    iniciaPilha(&p);
    do {
        cout << "\n1 - Testa vazia";
        cout << "\n2 - Testa cheia";
        cout << "\n3 - Empilha";
        cout << "\n4 - Desempilha";
        cout << "\n5 - Exibir pilha";
        cout << "\n6 - Sair";
        cout << "\nDigite sua opcao: ";
        cin >> op;
        switch (op){
            case 1: if (pilhaVazia(&p)){
                    cout << "Sim\n\n";
                } else {
                    cout << "Nao\n\n";
                }break;
            case 2: if (pilhaCheia(&p)){
                    cout << "Sim\n\n";
                } else {
                    cout << "Nao\n\n";
                }break;
            case 3: cout << "Digite um numero pra empilhar: ";
                    cin >> x;
                    if (push(&p,x)){
                        cout << "Ok\n\n";
                    } else {
                        cout << "Erro\n\n";
                    }break;
            case 4: if (pop(&p,&x)){
                    cout << "Desempilhado = " << x << "\n\n";
                } else {
                    cout << "Erro\n\n";
                }break;
            case 5: exhibePilha(&p); break;
            case 6: break;
            default: cout << "Opcao invalida . . . \n\n";
        }
    } while (op != 6);
    cout << "\n\n";
    system("PAUSE");
    return 0;
}
```

7.2. Fila

Tem esse nome pois sua filosofia se assemelha à de uma fila de banco. Em que elementos do mesmo tipo vão sendo enfileirados uns atrás dos outros e o acesso aos mesmos sempre são feitos a partir do que está no início da fila. É o que se diz de uma estrutura horizontalizada.

A política estipulada para uma fila é dita FIFO (first in, first out) ou seja, o primeiro a entrar será o primeiro a sair, de maneira que agora nos importa manter o controle sobre o elemento na posição inicial e final da fila, bem como seu tamanho total.

Também precisaremos impor algumas convenções para estudar a fila, a forma de representação que utilizaremos, será parecida com a da pilha, veja:

```
struct fila {  
    int a[MAX];  
    int ini;  
    int fim;  
    int tam;  
};
```

Foi criado uma estrutura com uso do *struct* para que se possa representar a fila. O vetor *a*, se destina a realizar o armazenamento dos dados da fila. Neste exemplo uma fila de inteiros (int). A variável *ini*, vai guardar a posição do elemento que se encontra no início da fila, da mesma maneira, a variável *fim*, vai guardar a posição do elemento que se encontra no fim da fila. E a variável *tam* vai guardar o tamanho total da fila, ou seja, quantos elementos ela está armazenando.

Além disso, criaremos também funções que realizaram as tarefas ligadas à manipulação da fila. Note bem que a fila só será manipulada através da utilização destas funções, não faremos acesso direto aos dados da fila. São elas:

- void iniciaFila (struct fila *f){...} → Esta função fará a inicialização de uma nova fila para que a mesma possa ser utilizada.
- bool filaVazia (struct fila *f){...} → Esta função testará se uma fila *f* encontra-se vazia, ou seja não pode se retirar mais nenhum elemento.
- bool filaCheia (struct fila *f){...} → Esta função testará se uma fila *f* encontra-se cheia, ou seja não pode receber mais nenhum elemento.
- bool enfileira (struct fila *f, int n){...} → Esta função irá colocar um novo elemento *n* no fim de uma fila *f*.
- bool desinfileira (struct fila *f, int *r){...} → Esta função irá remover um elemento *r* do início de uma fila *f*.

Nas páginas seguintes há um exemplo de um programa que realiza as operações citadas sobre uma fila e dá a opção de imprimir o conteúdo da mesma através da função *exibeFila*.

Note bem que existe uma manobra que é feita na operação de enfileirar e desinfileirar um elemento, pois seria muito custoso ficar movendo os elementos na posição de memória sempre que alguém entrasse ou sáísse da fila...

Assim os índices de início e fim são sempre crescentes, e se acessam os elementos na posição exata em que estão armazenados pegando o resto da divisão desta posição (seja ela início ou fim) pelo tamanho máximo da fila.

Veja como isto foi programado:

```
#include <iostream>
#include <cstdlib>
#define MAX 200 // tamanho máximo para a área de armazenamento
using namespace std;
struct fila {
    int a[MAX];
    int ini;
    int fim;
    int tam;
};
void iniciaFila (struct fila *f){
    f->ini = 0;
    f->fim = 0;
    f->tam = 0;
}
bool filaVazia (struct fila *f){
    if (f->tam == 0) return true;
    else return false;
}
bool filaCheia (struct fila *f){
    if (f->tam == MAX) return true;
    else return false;
}
bool enqueue (struct fila *f, int n){
    if (filaCheia(f)){ return false; }
    else {
        f->a[f->fim % MAX] = n;
        f->fim++;
        f->tam++;
        return true;
    }
}
bool dequeue (struct fila *f, int *r){
    if (filaVazia(f)){ return false; }
    else {
        *r = f->a[f->ini % MAX];
        f->ini++;
        f->tam--;
        return true;
    }
}
void exibeFila (struct fila *f){
    int i,n;
    for (i=0; i< f->tam; i++){
        dequeue(f,&n);
        cout << n << " ";
        enqueue(f,n);
    }cout << "\n\n";
}
```

```

int main(int argc, char* argv[]){
    struct fila f;
    int x, op;
    iniciaFila(&f);
    do {
        cout << "\n1 - Testa vazia";
        cout << "\n2 - Testa cheia";
        cout << "\n3 - Enfileira";
        cout << "\n4 - Desenfileira";
        cout << "\n5 - Exibir fila";
        cout << "\n6 - Sair";
        cout << "\nDigite sua opcao: ";
        cin >> op;
        switch (op){
            case 1: if (filaVazia(&f)){
                    cout << "Sim\n\n";
                } else {
                    cout << "Nao\n\n";
                }break;
            case 2: if (filaCheia(&f)){
                    cout << "Sim\n\n";
                } else {
                    cout << "Nao\n\n";
                }break;
            case 3: cout << "Digite um numero pra enfileirar: ";
                    cin >> x;
                    if (enfileira(&f,x)){
                        cout << "Ok\n\n";
                    } else {
                        cout << "Erro\n\n";
                    }break;
            case 4: if (desenfileira(&f,&x)){
                    cout << "Desenfileirado = " << x << "\n\n";
                } else {
                    cout << "Erro\n\n";
                }break;
            case 5: exhibeFila(&f); break;
            case 6: break;
            default: cout << "Opcao invalida . . . \n\n";
        }
    } while (op != 6);
    cout << "\n\n";
    system("PAUSE");
    return 0;
}

```

Pergunta filosófica: Como seria a estrutura e política de um “*deque*” ?

8 – Recursividade

8.1. Definição

Um problema é dito recursivo quando usa parte de sua definição para a resolução do próprio problema. Muitos problemas possuem uma natureza recursiva. A própria numeração decimal que utilizamos é definida recursivamente.

Recursão é um método de programação no qual uma função pode chamar a si mesma. O termo é usado de maneira mais geral para descrever o processo de repetição de um objeto de um jeito similar ao que já fora mostrado. Um bom exemplo disso são as imagens repetidas que aparecem quando dois espelhos são apontados um para o outro.

Para todo problema recursivo além da definição ou chamada recursiva é necessário também um caso base, ou condição de parada, que é onde este problema começa, ou termina.

8.2. Exemplos de algoritmos recursivos

Quando se estuda programação alguns casos são clássicos:

8.2.1. Fatorial → Na matemática o fatorial (representado por !) de um número $N!$ é dado pela fórmula $N * (N - 1)!$ (definição recursiva) e sabe-se que $0!$ Vale 1 (condição de parada). Vejamos uma função recursiva para cálculo de fatorial:

```
#include <iostream>
#include <cstdlib>
using namespace std;

int fatorial(int n) {
    if (n < 0){
        return -1; // Controle de erro. Opcional...
    }
    if (n == 0){
        return 1;
    } else {
        return (n * fatorial(n-1));
    }
}

int main(int argc, char* argv[]){
    int f, r;
    cout << "Digite o numero: ";
    cin >> f;
    r = fatorial(f);
    cout << "\nO fatorial de " << f << " eh: " << r << "\n\n";
    system("PAUSE");
    return 0;
}
```


8.2.2. Sequência de Fibonacci → É uma sequência gerada a partir dos seguintes termos sabidos:

- O primeiro elemento é o 0
 - O segundo elemento é o 1
 - Os demais elementos podem ser obtidos somando-se os dois elementos anteriores
- Vejam uma função que retorna o n-ésimo elemento desta sequência:

```
#include <iostream>
#include <cstdlib>
using namespace std;

int fibo(int n) {
    if ( n < 1 ) return -1; // Controle de erro. Opcional...
    if ( n == 1 ) return 0;
    if ( n == 2 ) return 1;
    else return (fibo(n-1) + fibo(n-2));
}

int main(int argc, char* argv[]){
    int f, r, i;
    cout << "Digite a quantidade de elementos: "; cin >> f;
    for (i = 1; i <= f; i++){ r = fibo(i); cout << r << " "; } cout << "\n\n";
    system("PAUSE");
    return 0;
}
```

8.2.3. Busca binária → Vela conhecida nossa, também pode ser implementada recursivamente, observe como ficou bem mais simples:

```
#include <iostream>
#include <cstdlib>
using namespace std;

int buscaBinaria (int n, int v[], int ini, int fim){
    int meio = (fim + ini) / 2;
    // cout << "\n" << ini << " " << fim << " " << v[meio];
    if ( ini > fim ) return -1;
    if ( n == v[meio] ) return meio;
    if ( n < v[meio] ) return buscaBinaria (n,v,ini,meio-1);
    else return buscaBinaria (n,v,meio+1,fim);
}

int main(int argc, char* argv[]){
    int a[50]={0,1,2,3,4,5,6,7,8,9,10}, n, p;
    cout << "Escolha um numero entre 0 e 10: ";
    cin >> n;
    p = buscaBinaria(n, a, 0, 10);
    if (p == -1){ cout << "\n\n Numero nao existe na colecao \n\n"; }
    else{ cout << "\n\n Numero encontrado na posicao " << p << "\n\n"; }
    system("PAUSE");
    return 0;
}
```

9 – Bibliografia

Bibliografia Básica para o curso de Algoritmos e Estruturas de Dados :

- SZWARCFITER, Jayme Luiz; MARKENZON, Lilian. *Estrutura de dados e seus algoritmos*. Rio de Janeiro: LTC, 1994.
- GUIMARÃES, Ângelo de Moura; LAGES, Newton A. C. *Algoritmos e estruturas de dados*. Rio de Janeiro: LTC, 1994.
- DEITEL, Harvey M.; DEITEL, Paul J. *C++: como programar*. Porto Alegre: Bookman, 2005.

Bibliografia Complementar para o curso de Algoritmos e Estruturas de Dados:

- FARRER, Harry et al. *Algoritmos estruturados (3ª ed.)*. Rio de Janeiro: LTC, 1999.
- SEXTON, Conar. *Dominando a linguagem C++*. Rio de Janeiro: IBPI, 2001.

Bibliografia de Referência para estas Notas de Aula :

- Exemplos dados em sala de aula pelo professor Rodrigo Dias.