Introdução à linguagem C/C++

AULA 2 - Recursos básicos
Introdução à linguagem C/C++

- Estrutura Básica de Programas C e C++
- Tipos de Dados
- Variáveis
- Entrada e Saída de Dados no C e C++
- Condições e Escolhas
- Repetições e Laços

C++ - Superconjunto do C

- O C++ aceita a sintaxe C, acrescentando melhoramentos, ampliando o escopo de aplicações, e possibilitando o desenvolvimento de programas baseados no paradigma da Orientação a Objetos;
- O C++ mantém à característica do C de ser uma linguagem de pequeno tamanho, o C++ possui apenas 62 palavras reservadas (32 delas comuns ao C);
- A diferença maior se dá na unidade de programação: no C são as funções enquanto que no C++ são as classes (que instanciam objetos e contém funções)

Palavras reservadas

C/C++

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

C++

asm	bool	catch	class	const_cast
delete	dynamic_cast	explicit	false	friend
inline	mutable	namespace	new	operator
private	protected	public	reinterpret_cast	static_cast
template	this	throw	true	try
typeid	typename	using	virtual	wchar_t

Programa Modelo

```
/* PROGBAS.CPP- Programa Modelo em C++ */
// Diretivas do Processador
   #include <iostream>
   #include <stdio>

// Variáveis Globais
   int contador = 0;

// Declaração de Funções
   int funcao_1 (int operando);
   void funcao_2 (void);
```

```
using namespace  std;
 int main () {    // Variaveis Exclusivas do Programa Principal
        int vezes, numero, x;
    /* Corpo de Instruções do Programa Principal */
        cout << "Entre o numero de vezes";
        cin >> vezes;
        for (int i=0; i<vezes; i++) {
            cout << "Entre com o " << i+1 << "numero : ";
            cin >> numero;
            x = funcao_1(numero);
            printf ("O quadrado de %d e' %d\n",numero, x);
        }
        funcao_2();
        Return 0;
}
```

```
/* Definição das Funções */
 int funcao_1 (int operando) {
     int resultado;
     contador += 1;
     resultado = operando*operando;
     return resultado;
  }
  void funcao_2 () {
     contador = contador + 1;
     cout << "Funcoes usadas" << contador << "
       vezes";
     cout << "\n FIM DO PROGRAMA";
}
```

Meu Primeiro Programa (I)

Compilador C++

```
// primeiro.cpp
#include <iostream.h>

int main() {
    cout << “Meu Primeiro Programa em
   C++”;
    return 0;
}
```

Meu Primeiro Programa (II)

Compilador C++ - ISO98

```
// primeiro.cpp
#include <iostream.h>

int main() {
  std::cout << “Programa em C++”;
  return 0;
}
```

Meu Primeiro Programa (III)

Compilador C++ - ISO98

```
// primeiro.cpp
#include <iostream>
using namespace std;

int main() {
     cout << “Meu Primeiro Programa em C++”;
     return 0;
}
```

Namespace – faz parte do que se chama ***Ambiente de Nomes*** no C++, que é um mecanismo para expressar agrupamentos lógicos.

Linguagens

- Interpretador (ou tradutor)
  - cada instrução da linguagem é traduzida a medida que seja necessário;
- Compilador
  - todo o código é traduzido de uma única vez;
- Linker
  - vários módulos compilados
  - Módulos e bibliotecas da linguagem são unidos para formar o programa executável.

Comentários

- Documentam programas
- Melhoram a leitura de programas
- São Ignorados pelo compilador

Diretivas de pre-processamento

- Começam com #
- Processadas por pré-processador antes da compilação

```
1   // Meu primeiro programa
2   // A first program in C++.
3   #include <iostream>
4    using namespace std

5   // função main inicia execução do programa
6   int main()
7   {
8     std::cout << "Welcome to C++!\n";
9
10    return 0;   // indica que o programa finalizou com sucesso
11
12  } // fim
```

```
Welcome to C++!
```

Tipos do C++

Tipo	Nome	Tamanho
Lógico	bool	1
Caracter	char, unsigned char	1
Inteiro	Int, unsigned (unsigned int), short (short int), long (long int), unsigned long	2 a 4
Real	Float, double, long double	4 a 10
-	void	-
	string	

Tipos do C

Tipo	Nome	Bits	Bytes	Faixa
caracter	char	8	1	-128 a 127
positivo	unsigned char	8	1	0 a 255
inteiro	int (signed)	16/32	2/4	
positivo	unsigned int (*unsigned*)	16/32	2/4	
pequeno	short int (*short*)	16	2	-32768 a 32767
grande	long int (*long*)	32	4	$-2.14*10^{9}$ a $2.14*10^{9}$
grande positivo	unsigned long	32	4	0 a $4.29*10^{9}$
real	float	32	4	$3.4\ e^{-38}$ a $3.4\ e^{38}$
	double	64	8	$1.7\ e^{-308}$ a $1.7\ e^{308}$
grande	long double	80	10	$3.4\ e^{-4932}$ a $1.1\ e^{4932}$

VARIÁVEIS

- Declaração de Variável
- Locais de Declaração de Variável
- Escopo de Variáveis
- Operador de Resolução de Escopo C++
- Classes de Armazenamento

Declaração de Variável

- A sintaxe de declaração de variável em C e C++.
  - tipo_de_dado Nome(s)_da(s)_Variável(eis);
- Exemplos :

  int x, y, z;

  float f;
- A declaração combinada com atribuição (inicialização) também pode ser feita
- Exemplos :

  float g = 9.81, Pi = 3.141592;

Declaração de Variáveis

- Regras para nomes de variáveis :
  - São reconhecidas pelo C os 32 primeiros caracteres de uma variável;
  - Só se pode iniciar um nome com um caracter do alfabeto ou pelo caracter underscore '_';
  - Os demais caracteres podem ser alfabeto, underscore ou caracteres numéricos;
  - O C considera diferentes nomes maiúsculos e minúsculos, portanto as seguintes variáveis são diferentes :

```
int maior, MAIOR, Maior, mAiOr;
```

Locais de Declaração de Variável

- Em C, só se pode declarar uma variável dentro de função, antes das instruções ou entre funções (sendo, neste caso, global a partir de sua declaração).
- Em C++, além destas formas, pode-se declarar uma variável em qualquer ponto de um programa - entre instruções ou mesmo dentro de intruções.

```
int x;

float f;
...
for (int i = 0; i < 20; i++) {...}
```

Escopo de Variáveis

- As regras de escopo referem-se à visibilidade da variável. Há 3 escopos possíveis para uma variável: local, arquivo e classe.
  - locais são acessadas única e exclusivamente pelo bloco no qual foi declarada. Um bloco é definido por {....}
  - globais são acessíveis em todo o arquivo.

```
float glob;
main() {
      int x, y;
      ...
     for (int i=0;...) {
        double d;
        ... }
     }
double doub;
```

Escopo de Variáveis

- Em C, blocos com variáveis locais homônimas a variáveis globais perdem o acesso às últimas.

```
int j = 0;
exemplo() {
     int x, j;
     ...
     for (int i=0;...)
          j = 5; // sempre para j local
        ...
}
```

Operador de Resolução de Escopo (::) INTERMEDIARIO?

- O C++ diferencia variáveis locais e globais homônimas com o **Operador de Resolução de Escopo** :: (também chamado Qualificação de Escopo), que colocado em frente à variável, faz com que o programa acesse a global.
- Um erro será apontado, caso não exista a variável global.

```
#include <iostream>
using namespace std;
int x = 2;

int dob (int v) {
  return 2*v;
}

main () {
  int x = 0;
  cout << "local " << dob( x );
  cout <<"global"<< dob(::x);
}

Resultado :
  local  = 0
  global = 4
```

Classe de armazenamento **static**

- Variáveis declaradas como especificador **static** possuem tempo de vida global, ou seja, permanecem alocadas em memória guardando seu último valor mesmo após o encerramento de seu escopo

Classe de armazenamento static

```
int dobro (int x) {
  static int cont=0;
    cont++;
    return 2*x;
}
```

- A declaração static faz com que a variável permaneça existindo em memória enquanto o programa estiver rodando.
- A abrangência da variável não se altera com a declaração **static**. Por exemplo, cont continua sendo uma variável local da função dobro()..
- ExemploCPP

Cuidados de um programador C

- O C permite atribuição entre tipos diferentes de dados
- Necessidade de alta disciplina ao programar - a legibilidade é muito importante para o entendimento do programa
- "***A maior parte das linguagens de programação pressupõe que o programador nunca sabe o que está fazendo. O C, ao contrário, pressupõe que ele sabe exatamente o que está fazendo, por isso ocorrem muito mais facilmente erros de lógica.***"

Entrada e saída de dados no C++

Bibliotecas Stream

- entrada de dados pelo teclado
  cin (scanf no C)
- saída de dados para tela
  cout (printf no C)
- é necessária a inclusão da biblioteca iostream
  #include <iostream> // no C stdio.h
- para outras entradas (arquivos, porta serial, etc.) e outras saídas (arquivos, impressora, etc.) utiliza-se sintaxe semelhante.

cout

- Está associada à saída padrão - tela.
  cout << Expressão
- Exemplos :

```
int x = 25;
double dob = 8.1;
char ch = 'F';
//      C++                          C
cout << "x = " << x;          printf("x=%d",x);
cout << dob << "e " << ch;    printf("%Lf e %c",dob, ch);
```

- A formatação da saída dos dados é feita automaticamente.

cin

- Está associada à entrada padrão - teclado.
  cin >> Variável

Exemplos :

Variáveis	*C++*	*C*
int x;	cin >> x;	scanf("%d",&x);
float f,g;	cin >> f >> g;	scanf("%f%f",&f,&g);

- A formatação dos dados, explícita em C, é automática em C++.

Macro ou Diretiva #define

- Sintaxe :
  #define frase1 frase2
- Durante a compilação ao se encontrar a ocorrência de frase1, o compilador substituirá por frase2.
  - ex :

```
#define PI 3.141596
...
area = PI * raio*raio; (no código)

area = 3.141596 *raio*raio; (código compilado)
```

Especificador const

- Surgiu como alternativa ao #define na primeira versão do C++ e mais tarde foi incorporado ao padrão ANSI C;
- Para se declarar um valor constante, usa-se a palavra const seguida do tipo e do valor da constante.
- Em C++, é possível tornar constantes: valores, ponteiros, conteúdo de ponteiros e parâmetros de função

Especificador const

Sintaxe :

const tipo tVar = valor;

Depois de definido o valor não se pode mais alterá-lo.

- **ex :**

const float PI = 3.141596;

...

area = PI * raio*raio; (no código)

area = PI * raio*raio; (código compilado)

Definição de Tipos

- Podem-se definir tipos próprios de variáveis, combinando tipos já existentes;
- Para isto utiliza-se a palavra reservada :

  typedef
- ex:

  typedef unsigned char uchar

  typedef int MEU_INT;

Operadores Especiais

- Linguagens Convencionais :

  total = total * 7; cont = cont +1;
- C e C++ - atribuição e aritmética combinadas

  total *= 7; cont += 1;
- Incremento e Decremento :

  cont ++; ou ++ cont;
- ex :

  cont = 7; total = 3;

  total *= cont - -;

  total *= - - cont;

Exemplo

```
1   // Mais um exemplo
2   // Preincremento e pos-incremento.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   // início do programa
9   int main()
10  {
11    int c;              // variável do exemplo
12
13    // demonstração de pós-incremento
14    c = 5;                    // atribui 5 a c
15    cout << c << endl;        // imprime 5
16    cout << c++ << endl;      // imprime c com valor 5 e depois incrementa
17    cout << c << endl << endl; // imprime 6
18
19    // demonstração de pré-incremento
20    c = 5;                    // atribui 5 a c
21    cout << c << endl;        // imprime 5
22    cout << ++c << endl;      // incrementa e depois imprime c com valor  6
23    cout << c << endl;        // imprime  6
```

continuação

```
24
25    return 0;  // indica final feliz
26
27  } // fim do programa
```

```
5
5
6

5
6
6
```

Operadores Aritméticos e Lógicos

operadores		associatividade	exemplo
() []			
++	incremento		x++ ou ++x
--	decremento		x—ou –x
+	positivo		+x
-	negativo		-x
tipo()	cast		float(x)
sizeof	tamanho		sizeof(x)
!	negação lógica		!x
*	multiplicação		x*y
/	divisão		x/y
%	módulo		x%y
+	adição		x+y
-	subtração		x-y

Operadores Aritméticos e Lógicos

>	maior que		x>y
>=	maior ou igual		x>=y
<	menor que		x<y
<=	menor ou igual		x<=y
==	igual		x==y
!=	diferente		x!=y
&&	E lógico		exp1&&exp2
||	OU lógico		exp1 || exp2
?:	operador condicional		e?r:o
=	atribuição		x = 3
+=			x += 10
-=			x -= 4
*=			x *= 6
/=			x /=2
%=			x %=7
,	virgula		

Operadores Bit a Bit

- **Programas Aplicativos manipulam BYTEs;**
- **Compiladores, Sistemas Operacionais, Periféricos : trabalham no nível de BITs**

&	E (AND)
|	OU (OR)
^	XOR (ou exclusivo)
>> x	desloca os Bits 'x' vezes a Direita
<< x	desloca os Bits 'x' vezes a Esquerda
~	NEGAÇÃO (Complemento)

- **Não Confundir os Operadores lógicos && e \|\| com os operadores BIT a Bit & e |**

enum

- tipo de dado aplicado normalmente a valores que seguem determinada ordem
- Em C++, o nome da enumeração é também um tipo de dado, com isso, palavra enum é desnecessária em declaração.

```
enum  dia_util {
    segunda = 2,        // 2
    terca,              // 3
    quarta,             // 4
    quinta,             // 5
    sexta }             // 6
dia_util dia;   // variáveis enumeradas
```

COMANDOS

- Comandos Condicionais
  - Comando if...else
  - Comando switch
  - As Palavras Chave break e continue
- Laços e Repetições
  - Comando while
  - Comando do...while
  - Comando for

Comandos Condicionais - if/else

```
if (expressão)
    { bloco_de_comandos-T; }
[else]
    { bloco_de_comandos-F; };
```

- expressão identifica uma expressão condicional a ser avaliada. Toda expressão que produza um valor não-nulo é considerada verdadeira.
- pode-se usar if alinhados, isto é, um if dentro de um bloco de comandos de outro if
- deve-se cuidar para que não cause erros de lógica

Comandos condicionais - switch/case

- O comando switch() seleciona determinado bloco de comandos baseando-se no valor de uma expressão inteira (char ou variações).

```
switch (escolha) {
   case valor_inteiro_1 : {bloco_1};  break;
    (...)
   case valor_inteiro_n : {bloco_n};  break;
   [default : {bloco_default};              break;]
}
```

  - escolha é qualquer expressão que produza um inteiro;
  - o label case provoca a execução do bloco;
  - default é executado quando nenhum case ocorreu.

Laços e Repetições – while e do/while

```
while (expressão)
    { bloco_de_comandos; };
```

- Enquanto expressão for verdadeira, o bloco de comandos dentro do while é executado.
- o conteúdo (bloco de comandos) não é executado se a condição (expressão) for falsa

```
do {
    bloco de comandos
} while (expressão);
```

- No comando do...while ao menos uma vez, o bloco de comandos será executado, já que a avaliação da expressão condicional está no fim.

Laços e Repetições - for

- O laço for é usado quando uma ou mais instruções devem ser executadas um determinado número de vezes.

```
for (inicializações;[expressão];[modificações ])
    { bloco_de_comandos };
```

  - inicializações - ajusta-se (uma só vez) os valores iniciais de uma ou mais variáveis separadas por vírgula.
  - expressão - controla o laço.
  - modificações - alterações em variáveis

Controle do laço - break

- **break** interrompe o laço não executando os comandos que estiverem abaixo dele.

```
// executa as funcoes ate' que se mande terminar
do {
   prepara_dados();
   calcula();
   cout << "Continua ? (S/N)";
   if (toupper(cin.get()!= 'N')
     break;
   }  while(1);
```

NÃO FAÇAM NUNCA ESTE TIPO DE PROGRAMAÇÃO ! EXISTE SEMPRE UMA FORMA MAIS CLARA

Controle do laço - continue

- **continue** : salta os comandos que o seguem, voltando à expressão que controla o laço.

```
// imprime os pares entre 0 e 10
int cont = 0;
while (cont++ < 11) {
   if (cont % 2 != 0)
            continue;
   cout << cont << "/n";
}
```

- TAMBÉM GERA CONFUSÃO !

Escolha de laços apropriados

- while
  - o conteúdo nunca é executado quando a condição é falsa;
  - utilizado quando deve-se verificar antes de entrar no laço, se uma condição externa é verdadeira.
- do .. while
  - conteúdo do laço deve ser executado pelo menos uma vez; (SIMILAR AO REPEAT … UNTIL DO PASCAL)
- for
  - quando já se sabe o número de vezes que o laço será executado

Formatação de Saída :

- Manipuladores (incluir IOMANIP.H)
- São "Funções" usadas junto com o cout:
  cout << manipulador;

manipulador		exemplo
dec	sair inteiro como decimal	cout << dec << i;
hex	sair inteiro como hexadecimal	cout << hex << i;
oct	sair inteiro como octal	cout << oct << i;
endl	insere linha ("\n")	cout << endl;
ends	insere término de string ("\0")	
setw(int n)	ajusta o tamanho do campo para n caracteres	cout << setw(20) << '*';
setfill(char n)	preenche o campo com n	cout << setfill('#');
flush	libera o buffer	
setprecision (int n)	real com n casas decimais depois do ponto	cout << setprecision(2) << f;

- FUNÇÕES
  - PASSAGEM DE PARÂMETROS
- FUNÇÕES INLINE
- FUNÇÕES RECURSIVAS
- VETORES
- PONTEIROS
- OPERADORES DE ENDEREÇO

Funções

- Protótipos
- Valor de Retorno
- Parâmetros
- Defaults
- Funções Inline
- Sobrecarga de Funções

CONCEITO

- Dividir para conquistar
  - Construa um programa à partir de pequenos pedaços ou componentes
  - Cada componente é mais "gerenciável" que um grande bloco.

Componentes em C++

- Módulos: funções e classes
- Programas usam módulos novos e "pre-empacotados"
  - Novos: definidos pelo usuário (programador)
  - Pre-empacotados: da "standard library"
- Funções invocadas por chamadas de funções
  - Nome da função e argumentos (se necessário)
- Definição de funções
  - Escritas somente uma vez

Funções matemáticas da biblioteca

- Realizam cálculos matemáticos comuns
  - Incluir o header **`<cmath>`**
- A chamada ocorre assim:
  - Nomefunção (argumento);

  ou
  - Nomefunção(argumento1, argumento2, …);
- Exemplo

  `cout << sqrt( 900.0 );`
  - sqrt (square root)

Todas as funções da "math library" retornam `double`

Funções de biblioteca (cont)

- Argumentos de funções podem ser:
  - Constantes
    - `sqrt( 4 );`
  - Variáveis
    - `sqrt( x );`
  - Expressões
    - `sqrt( sqrt( x ) ) ;`
    - `sqrt( 3 - 6x );`

Protótipo	Descrição	Exemplo
`ceil( x )`	Arredonda *x* para o menor inteiro não menor que *x*	`ceil( 9.2 )` é `10.0` `ceil( -9.8 )` é `-9.0`
`cos( x )`	Cosseno de *x* (*x* in radians)	`cos( 0.0 )` é `1.0`
`exp( x )`	exponencial	`exp( 1.0 )` é `2.71828` `exp( 2.0 )` é `7.38906`

`fabs( x )`	Valor absoluto de *x*	`fabs( 5.1 )` é `5.1`
		`fabs( 0.0 )` é `0.0`
		`fabs( -8.76 )` é `8.76`

`floor( x )`	arredonda *x* para o maior inteiro menor que *x*	`floor( 9.2 )` é `9.0` `floor( -9.8 )` é `-10.0`
`fmod( x, y )`	Resto de *x*/*y* em real	`fmod( 13.657, 2.333 )` é `1.992`
`log( x )`	Locaritmo natural de *x* (base *e*)	`log( 2.718282 )` é `1.0` `log( 7.389056 )` é `2.0`
`log10( x )`	Logaritimo de *x* (base 10)	`log10( 10.0 )` é `1.0` `log10( 100.0 )` é `2.0`
`pow( x, y )`	x elevado a y	`pow( 2, 7 )` é `128` `pow( 9, .5 )` é `3`
`sin( x )`	Seno de *x* (*x* em radianos)	`sin( 0.0 )` é `0`
`sqrt( x )`	Raiz quadrada de *x*	`sqrt( 900.0 )` é `30.0` `sqrt( 9.0 )` é `3.0`
`tan( x )`	Tangente trigonométrica de *x* (*x* em radianos)	`tan( 0.0 )` é `0`

Protótipos de Função

- **Protótipos foram incorporados ao padrão ANSI C 1988, influenciado pelo C++.**

Protótipos de funções

- Protótipos contém
  - Nome da função
  - Parâmetros (número e tipo)
  - Tipo de Retorno (**`void`** se não retorna nada)

  Necessários se as chamadas ocorrerem antes da definição

  Devem ser compatíveis com a definição da função

Protótipos Função

TipoRetorno NomeFunção (**Tipos_parâmetros**);

TipoRetorno

- Pode ser qualquer tipo convencional da linguagem (**int, float**, etc) e tipos definidos pelo usuário. Quando a função não retornar nada, usa-se a palavra **void**.

NomeFunção

- O compilador reconhece os primeiros 32 caracteres.
- C e C++ diferenciam minúsculas de maiúsculas Por exemplo, as funções print() e Print() são diferentes para o compilador.

Parâmetros

- Uma função pode ter nenhum, um ou mais parâmetros, com o tipo especificado e separados por vírgula.

Função

- No C++ e ANSI C ausência de parâmetros significa o mesmo que utilizar void.

```
void menu(void);        // geral
void menu();      /* C++ e ANSI C */
```

Definição de uma Função

- Ela possui a mesma 'cara' do protótipo, acrescentando o nome da variável que está sendo passada como parâmetro e o corpo da função:

```
TipoRetorno NomeFunção (TipoP pNome) {
   // código referente as tarefas a serem
   cumpridas
}
```

pNome - possui as mesmas caracteristicas da declaração de uma variável e possuirá escopo local da função

Definição de uma Função - Exemplo

```
void area (float);
....
void area (float raio)  {
     float area = PI * raio * raio;
     cout    << "a área do círculo de raio : "
             << raio << ' é de : ' << area;
 }
```

- **O nome das variáveis parâmetros nos protótipos pode ser omitida, pois o compilador ignora estes nomes na declaração. Mas colocação de nomes pode identificar melhor a finalidade de determinadas funções.**

Tipo de Retorno

- **É o valor que a função retorna para o local onde houve a chamada;**
- **Faz com que a função possa ser tratada com se fosse uma variável do programa, isto é, pode-se fazer operações aritméticas e lógicas, etc.**
- **utiliza-se para retornar o valor a palavra reservada return, acrescida do valor a ser retornado :**

```
TipoRetorno NomeFunção (TipoP tNome) {
        return Valor_compatível_a_TipoRetorno;

}
```

Retorno de uma Função - exemplo

```
float valorGasto (int);
int main() {
    int iL;     cin >> iL;
    float totalImposto = valorGasto(iL) * 1.1;
    if (valorGasto(iL) > 20)
        cout   << "Você gastou "  << valorGasto(iL)
               << "e ganhou uma lavagem gratis";
    return 0;
}
 float valorGasto(int litros) {
        return litros * 1,53;
 }
```

Inicializadores '*Default*' para Funções

- **Em C++ pode-se definir argumentos *default* para as funções.**

```
retorno função ( tipo par = default, tipo2 = default2);
```

- **Caso a chamada da função omita algum parâmetro, a função pode usar o *default* previamente definido.**
- **Valores *default* DEVEM ESTAR na declaração da função (protótipo), NÃO devendo-se repeti-los na definição.**

Mais sobre Defaults

- Chamada de funções com omissão de parâmetros
  - Se não houver parâmetros o suficiente, o mais à direita assume valor default
  - Valores default
    - Podem ser constantes, variáveis globais ou mesmo chamadas de funções
- Exemplo:

```
int myFunction( int x = 1, int y = 2, int z = 3 );
```

  - **`myFunction(3)`**
    - **`x = 3`**, **`y`** e **`z`** assumem default (rightmost)
  - **`myFunction(3, 5)`**
    - **`x = 3`**, **`y = 5`** e **`z`** assume default

Inicializadores Default para Funções

```
#include <math.h>   // pow( , )
double elevado (int, int =2);
int main ()
   {
      int a=30;
      cout << elevado (7, 5);
      cout << elevado (a);
      return 0;
    }
double elevado (int x , int y)
   {
      return pow(x,y);                    // x^y
   }
```

Recursão

- Funções Recursivas
  - Funções que chamam a si próprias (ou…)
  - Resolvem o caso base
- Caso não seja o caso base
  - Quebrar o problema em sub-problema(s)
  - Executar nova cópia da função para resolver o sub-problema (recursive call/recursive step)
    - Deve convergir para o caso base
    - Função realiza auto-chamada dentro do return
  - Em algum momento o caso base é resolvido
    - Resposta é devolvida na pilha de chamada

Recursão

- Exemplo: factorial

  *n! = n * (n – 1) * (n – 2) * … * 1*

  – Relação recursiva: (n! = n * (n – 1)!)

  *5! = 5 * 4!*

  *4! = 4 * 3!…*

  – Caso base (1! = 0! = 1)

- Exercício

Exemplo

```
1
2   // Recursive factorial function.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include <iomanip>
9
10  using std::setw;
11
12  unsigned long factorial( unsigned long ); // function prototype
13
14  int main()
15  {
16     // Loop 10 times. During each iteration, calculate
17     // factorial( i ) and display result.
18     for ( int i = 0; i <= 10; i++ )
19        cout << setw( 2 ) << i << "! = "
20             << factorial( i ) << endl;
21
22     return 0;  // indicates successful termination
23
24  } // end main
```

Tipo `unsigned long` pode armazenar um inteiro entre 0 e 4 bilhões.

```
25
26  // recursive definition of function factorial
27  unsigned long factorial( unsigned long number )
28  {
29     // base case
30     if ( number <= 1 )
31        return 1;
32
33     // recursive step
34     else
35        return number * factorial( number - 1 );
36
37  } // end function factorial
```

O caso base ocorre com `0!` ou `1!`. Todos os outros casos devem ser resolvidos pela recursão.

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Recursão x Iteração

- Repetição
  - Iteração: loops explícitos
  - Recursão: chamadas repetidas de funções
- Término
  - Iteração: Condição do loop falha
  - Recursão: caso base reconhecido
- Ambos podem ter infinitos loops
- Iterações têm melhor desempenho contra elegância e legibilidade da recursão

Recursão x Iteração

- Exercício:
  - Fatorial com iteração

Arrays

- Arrays
  - Vetores
  - Matrizes

Arrays

- Propriedades
  - Estruturas de dados de itens semelhantes
  - Podem ser estáticos (mesmo tamanho durante o programa)

Arrays - continuação

- Array
  - Áreas de memória consecutivas
  - Mesmo nome e tipo (**`int`**, **`char`**, etc.)
- Para se referir a um elemento
  - Especifique nome e posição (índice)
  - Formato: nome_array[posição]
  - Primeiro elemento na posição 0
- Array c com N elementos

  `c[ 0 ]`, `c[ 1 ]` ... `c[ n - 1 ]`
  - O último está na posição N-1

Arrays - continuação

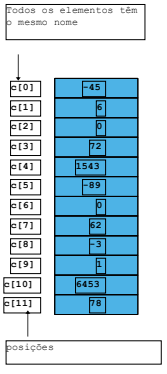
- Elementos de um array são similares a variáveis
  - Atribuição e escrita :

```
c[ 0 ] =  3;
cout << c[ 0 ];
```

- Operações pode ser executadas dentro do índice:

  `c[ 5 - 2 ]` é o mesmo que `c[3]`

4.2 Arrays



Vetores – 1 dimensão

Servem para guardar um conjunto de variáveis de um mesmo tipo;

Sintaxe :

tipo nomeVetor [tamanho];

Ex :

float f [5];

Cada elemento do vetor é independente. E o primeiro elemento tem o índice zero. Os demais elementos estarão colocados nos endereços seguintes de memória;

f[0] = 2.34;

	1006	100A	100E	1013
f[0]	f[1]	f[2]	f[3]	f[4]
2.34	45.878	46.0	67.66	1.111

Usando vetores

- Inicialização
  - Loop For
    - Inicializa elemento a elemento
  - Lista de inicialização
    - Especifica os valores na declaração

```
int n[ 5 ] = { 1, 2, 3, 4, 5 };
```

    - Se não houver inicializadores o suficiente, os mais á direita (rightmost) são 0
    - Se houver demais -> syntax error
  - Para inicializar todos com o mesmo valor

```
int n[ 5 ] = { 0 };
```

  - Se o tamanho for omitido, os inicializadores determinam o tamanho

```
int n[] = { 1, 2, 3, 4, 5 };
```

    - 5 inicializadores -> 5 elementos

Usando vetores

- Tamanho
  - Pode ser especificado com elementos constantes
    - `const int tamanho = 20;`
  - Constantes não se modificam
  - Constantes devem ser inicializadas quando são declaradas

Passando Arrays para Funções

- Arrays passagem por referência
  - Funções podem modificar dados originais
- Elementos individuais passados por valor
  - Como qualquer outra variável
  - `square( myArray[3] );`

Passando Arrays para Funções

- Funções que recebem arrays
  - Protótipo
    - `void modifyArray( int b[], int arraySize );`
    - `void modifyArray( int [], int );`
      - Nomes são opcionais nos protótpos
    - Ambos recebem um vetor de inteiros e um simples inteiro
  - Não é necessário especificar tamanho do array
    - Ignorado pelo compilador

```
1  // Fig. 4.14: fig04_14.cpp
2  // Passing arrays and individual array elements to functions.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setw;
11
12 void modifyArray( int [], int );  // appears strange
13 void modifyElement( int );
14
15 int main()
16 {
17    const int arraySize = 5;              // size of array a
18    int a[ arraySize ] = { 0, 1, 2, 3, 4 };  // initialize a
19
20    cout << "Effects of passing entire array by reference:"
21         << "\n\nThe values of the original array are:\n";
22
23    // output original array
24    for ( int i = 0; i < arraySize; i++ )
25       cout << setw( 3 ) << a[ i ];
```

Sintaxe para aceitar um array em uma lista de parâmetros

```
26
27    cout << endl;
28
29    // pass array a to modifyArray by refe
30    modifyArray( a, arraySize );
31
32    cout << "Os valores do array  modificado são:\n";
33
34    // output modified array
35    for ( int j = 0; j < arraySize; j++ )
36       cout << setw( 3 ) << a[ j ];
37
38    // output value of a[ 3 ]
39    cout << "\n\n\n"
40       << "Efeito de passar um elemento por valor:"
41       << "\n\nThe value of a[3] is " << a[
42
43    // pass array element a[ 3 ] by value
44    modifyElement( a[ 3 ] );
45
46    // output value of a[ 3 ]
47    cout << "O valor de a[3] é " << a[ 3 ] << endl;
48
49    return 0;  // indicates successful termination
50
51 } // end main
```

Passa o nome do array (`a`) e o tamanho para a função. Arrays são passados por referência (passed-by-reference).

Passa um elemento de um array (por valor); o original não pode ser modificado.

```
52
53 // in function modifyArray, "b" points to
54 // the original array "a" in memory
55 void modifyArray( int b[], int sizeOfArray )
56 {
57    // multiply each array element by 2
58    for ( int k = 0; k < sizeOfArray; k++ )
59       b[ k ] *= 2;
60
61 } // end function modifyArray
62
63 // in function modifyElement, "e" is a local copy of
64 // array element a[ 3 ] passed from main
65 void modifyElement( int e )
66 {
67    // multiply parameter by 2
68    cout << "Valor dentro de modifyElement é "
69       << ( e *= 2 ) << endl;
70
71 } // end function modifyElement
```

Apesar de ter o nome `b` (genérico), o array aponta para o array original `a`. Pode modificar `a`.

Passagem por valor.

Matrizes

- Extensão do conceito de vetores, onde os Arrays são Multi-dimensionais.

Ex :

char cMatriz[3][2];

- Resultado : uma planilha de 6 elementos char, organizados em forma de 3 linhas e 2 colunas;

1017 **cMatriz[0][0]**	1019 **cMatriz[0][1]**	**'A'**	**'E'**
101A **cMatriz[1][0]**	101C **cMatriz[1][1]**	**'I'**	**'O'**
101E **cMatriz[2][0]**	1020 **cMatriz[2][1]**	**'U'**	**'B'**

Matrizes

- Inicialização
  - Default é `0` – `(cuidado:quando inicializa)`
  - Inicializadores agrupados por linha (entre chaves)

```
int b[2][2] = { { 1, 2 }, { 3, 4 } };
```

1	2
3	4

```
int b[2][2] = { { 1 }, { 3, 4 } };
```

1	0
3	4

Matrizes

- Referência intuitiva

1	*0*
3	4

```
cout << b[ 0 ][ 1 ];
```

  - Não se referencia com vírgula

```
cout << b[ 0, 1 ];
```

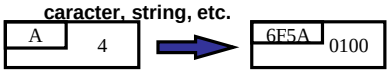
    - Erro de sintaxe
- Protótipo de funções
  - Deve-se especificar tamanhos
  - Primeiro índice não é necessário (assim como nos vetores)
  - `void printArray( int [][ 3 ] );`

PONTEIROS

- Introdução
- Sintaxe
- Operador &
- Operador *
- Ponteiros void
- Ponteiros const e Ponteiros para const
- Parâmetros por Referência em C++

Memória

- **RAM**
  - **local contínuo para estocagem de instruções e dados;**
  - **no PC cada locação armazena 8 bits (0 byte) = char**
- **Os dados que necessitam ter seu valor alterado, cada vez que o programa é executado, devem ter um local de memória para armazenar seu valor.**
- **Deve-se então criar variáveis para armazenar estes valores, sejam eles inteiros, real, caracter, string, etc.**



Ponteiros

- **É um Tipo de Variável que armazena um endereço de memória (RAM), correspondendo a localização do valor de uma varíavel do mesmo tipo.**
  - **O tamanho de uma variável ponteiro não é necessariamente igual ao do seu tipo;**
  - **No PC, normalmente o ponteiro tem o tamanho de 2 bytes;**
  - **Pode-se fazer operações aritméticas com ponteiro (++, - -, - ,+ ,*, etc...), e o comportamento destas operações dependerá do tipo ao qual o ponteiro foi definido.**

Ponteiro - Sintaxe

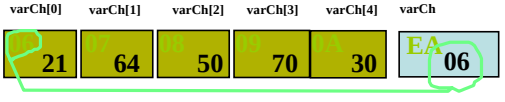
tipoDado * nomePonteiro;
ex :

```
int * piPonteiro;
float *pfPonteiro;
Estrutura * pEstPonteiro;
```

- **não se pode usar ponteiros para tipos diferentes do qual ele foi criado;**
- **o comportamento do ponteiro varia de acordo com o tipo para o qual foi definido**
- **é muito importante a inicialização de um ponteiro.**
- **O NULL é o zero (ou o falso) para ponteiros**

Ponteiros - Onde já usamos ?

- o nome de um array é um ponteiro, e nele está armazenado o endereço do seu primeiro elemento, isto é, por exemplo, seja :
- char varCh[5] { 33, 100, 'P', 'p', '0'};



Operador de Endereçamento &
- **Fornece o endereço de uma variável**
  &varCh[0] = 06 = varCh;

Ponteiro - operador de localização *

- **retorna a váriavel apontada pelo ponteiro**

***varPonteiro** = valor armazenado no endereço guardado pelo ponteiro*

- **ex :**

```
int * iPtr;
int iVar = 99;
iPtr = &iVar; // inicializando o ponteiro
cout << *iPtr; // mostra '99'
*iPtr = 77;
cout << iVar // mostra '77'
```

Utilização de Ponteiros

- int main () {
  double dVetor[5];
  ler(dVetor , 5);
  imprime (dVetor , 5);
  return 0;
  }

Como Fazer para passar um vetor como parâmetro de uma função?

```
void ler ( double *vetor, int q) {
    for (int I = 0; I <  q; I++)   cin >> vetor[I];
  }

void imprime ( double * vetor, int q){
    for (int I = 0; I <  q; I++)   cout << vetor[I];
  }
```

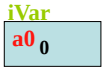
Exercício

- Implementar uma função que retorne o cubo de um inteiro:
  - Usando passagem por valor
  - Usando passagem por referência usando ponteiros

Parâmetros por Referência em C++

- Em C isto requer que a função tenha ponteiros como parâmetros. Em C++ isto é feito usando-se o símbolo '&' (para referência).

```
void zera(  int &  );
int main() {
        int iVar = 9999;
        zera(    iVar);
        cout << iVar;
        return 0;
}
void zera(  int & x  ){
        x = 0;
}
```



Parâmetros por Referência em C++

- Em C isto requer que a função tenha ponteiros como parâmetros. Em C++ isto é feito usando-se o símbolo '&'.
- Exemplo :
- Em C :

```
void zera (int * valor) {
    *valor = 0;
}
int x;
zera(&x);
```

Em C++ :

```
void zera (int& valor) {
    valor = 0;
}
int x;
zera(x);
```

Resumo

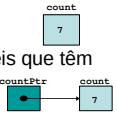
- Ponteiros são poderosos, mas difíceis de se dominar
- Simula passagem por referência
- Estreita relação com arrays e strings

Ponteiros - resumo

- Ponteiros variáveis
  - Contém endereço de memória como valores
  - Normalmente, variáveis contém valores específicos (referência direta)
  - Ponteiros contém endereços de variáveis que têm valores específicos (referência indireta)
- Indireção
  - Referencia valores através de ponteiros
- Declaração de ponteiros
  - `*` indica que a variável é um ponteiro

```
int *myPtr;
```

declara ponteiro para `int`



Ponteiros - resumo

- Pode-se declarar ponteiro para qualquer tipo de dado
- Inicialização
  - Initializa com `0`, `NULL`, ou endereço
    - `0` ou `NULL` aponta para nada

Operadores

- **&** (Operador de endereço)
  - Retorna endereço de memória do operando
  - Exemplo

```
int y = 5;
int *yPtr;
yPtr = &y;    // yPtr recebe
endereço de y
```

  - **yPtr** "aponta para" **y**

Struct

- Estruturas
  - Armazenam variáveis de diferentes tipos
  - Similar a classes (mas todos os membros são **public)**

Definição

- Definição

```
struct Funcionario {
   char nome[20];
   char sobrenome[20];
};
```

  - Palavra chave **struct**
  - Funcionario é o nome da estrutura
    - Usado para declarar variáveis deste tipo
  - Dados e funções declarados entre chaves
    - Nomes devem ser únicos
    - Estrutura não pode conter uma instancia dela mesma, apenas ponteiro
  - Definição não reserva memória
  - Definição termina com ;

Definição

- Declaração
  - Declarado como qualquer outra variável
    - **Funcionario alpha;**
  - Pode-se declarar variáveis quando da definição

```
struct funcionario {
    char nome[20];
    char sobrenome[30];
} alpha, beta;
```

Definição

- Operações
  - Atribuições a estruturas do mesmo tipo
  - Recebe endereço (**&**)
  - Acessa-se um elemento através do ponto (funcionario**.nome**)
  - Usando **sizeof**
    - estrutura pode não ter bytes de memória em posições consecutivas

Inicialização de estruturas

- Listas de Inicialização (como em arrays)
  - **Funcionario x = {"João","Silva" };**
  - Se membro não especificado, default é 0
- Atribuições
  - Uma estrutura para outra

```
Funcionario x = y;
```

  - Membros individualmente

```
Funcionario x;
   x.nome = "João";
   x.sobrenome = "Silva";
```

Usando estruturas c/ funções

- Duas formas de se passar estruturas p/ funções
  - Passar estrutura inteira
  - Passar membros individualmente
  - Ambas são por valor (call-by-value)
- Por referência (call-by-reference)
  - Passar endereço
  - Passar referência para a estrutura
- Passando arrays por valor (call-by-value)
  - Criar estrutura c/ array como membro
  - Passar a estrutura

typedef

- Palavra-chave **`typedef`**
  - Cria sinônimos para tipos já definidos
    - Não cria tipo, apenas sinônimo
  - Cria nomes mais curtos
- Exemplo
  - **`typedef funcionario *pfunc;`**
  - Define novo tipo pfunc como sinônimo para funcionario **`*`**
    - **`pfunc meu_ponteiro;`**
    - **`funcionario * meu_ponteiro;`**