Introdução à Programação em C/C++

Afrânio Melo
afranio@peq.coppe.ufrj.br

Laboratório de Modelagem, Simulação e Controle de Processos
Programa de Engenharia Química, COPPE
Universidade Federal do Rio de Janeiro

06/2015

COPPE UFRJ

O que é programação?

Definições

- *Computadores* são máquinas projetadas para realizar operações aritméticas e lógicas seguindo uma série de instruções pré-definidas.
- Um conjunto de instruções destinadas a serem seguidas por um computador é conhecido como *programa*.
- Portanto, a *programação* pode ser definida como *a arte de se escrever códigos que contém instruções para determinadas tarefas a serem realizadas por um computador*.

Figura 1: Computador.

Programando

Comunicação com o computador

- Computadores só entendem *linguagem binária*. Ou seja, todas as instruções que desejarmos passar a um computador devem estar expressas em termos de 0 ou 1.
- Exemplo de uma instrução entendida por um computador: 000101010101.
- O problema é que seres humanos (normais) não entendem binário! Como efetuar a comunicação entre o homem e o computador?
- Com o objetivo de resolver este problema, foram criadas as *linguagens de programação*.

Programando

Linguagens de programação

- As primeiras linguagens de programação que surgiram foram as de *baixo nível de abstração* (ou apenas de *baixo nível*). Estas apresentam sintaxes relacionadas diretamente às instruções que devem ser seguidas por um processador (apresentando pouco comprometimento em relação à legibilidade por humanos). Um exemplo é a Assembly.
- Mais tarde, foram aparecendo linguagens de *alto nível de abstração* (ou apenas *de alto nível*), que têm mais relação com os objetivos gerais do programador do que com as tarefas específicas do processador (sendo muito mais práticas, porém menos eficientes). Exemplos são C, Fortran, Python ou Matlab.

Rodando o programa

Linguagens compiladas X Linguagens interpretadas

- Imagine que você escreveu um programa em alguma linguagem de alto nível. Como fazer para rodá-lo?
- Em primeiro lugar, é preciso traduzir o código da *linguagem de programação* (que *você* entende) para a *linguagem binária* (que a *máquina* entende). Isto pode ser feito de duas maneiras:
  - *Compilação*: neste caso, traduz-se todo o programa de uma só vez para o código binário, gerando um arquivo chamado *executável* (no Windows, um arquivo com extensão *.exe*). Esse executável então pode ser rodado. Exemplos de linguagens compiladas: C/C++, Fortran.
  - *Interpretação*: aqui, o programa é traduzido e executado linha por linha por algum software que age como *interpretador*. O desempenho é menor em relação aos programas compilados. Exemplos de linguagens interpretadas: Matlab, Octave, Python.

Figura 2: Ilustração da necessidade da compilação ou interpretação.

Erros de programação

Os famosos bugs!

- É preciso ter em mente que existem três tipos de erros de programação:
  - *Erro de sintaxe*: este erro ocorre quando escrevemos algo que *não está definido* na linguagem, ou seja, que o interpretador não entende. A *execução do programa é interrompida e uma mensagem de erro emitida*.
  - *Erro de lógica*: este erro aparece quando o programa funciona, mas não do jeito planejado, fornecendo *resultados espúrios* (talvez por alguma fórmula errada, etc) . É o mais difícil de ser encontrado.
  - *Erro de estilo*: é o menos grave de todos, no sentido de que não prejudica a execução do programa. Muitos nem consideram este como um tipo de erro. Ocorre quando o programador desenvolve o seu código de maneira bagunçada e ilegível, o que dificulta sua *manutenção, desenvolvimento* e *extensão*.

Figura 3: Fuja destes insetos!

COPPE UFRJ

A linguagem C

Figura 4: Dennis Ritchie.

A Linguagem

- O C é uma linguagem compilada, de médio a alto nível e de uso geral, inicialmente desenvolvida por Dennis Ritchie no AT&T Bell Labs no começo da década de 70.
- É uma das linguagens mais influentes e bem sucedidas da história da computação, tendo servido como base para o desenvolvimento de outras linguagens importantes, como C++ e C#.

Hello World!

O programa Hello World!

```c
#include <stdio.h>

int main ()
{
    printf ("Hello World!\n");
    return 0;
}
```

Nosso primeiro programa em C

Vamos começar analisando linha por linha o exemplo clássico do programa *Hello World!*

Compile o código ao lado e o rode. Qual o *efeito* que ele produz?

COPPE UFRJ

Hello World!

O programa Hello World!

```c
#include <stdio.h>

int main ()
{
    printf ("Hello World!\n");
    return 0;
}
```

#include <stdio.h>

Linhas iniciadas com # são *diretrizes para o pré-processador*.

Neste estágio preliminar, você pode entender as diretrizes como tarefas preliminares que o compilador precisa cumprir antes da compilação em si.

Neste caso, a diretriz *#include* inclui no nosso programa a biblioteca *stdio.h*, que contém as definições básicas para realizar entrada e saída de dados em C.

Hello World!

O programa Hello World!

```
#include <stdio.h>

int main ()
{
    printf ("Hello World!\n");
    return 0;
}
```

int main ()

Esta linha é o início da função *main*. Todo código em C deve ter uma função main; ela marca o ponto em que o programa começa sua execução.

Uma *função* em C pode aceitar uma certa quantidade de parâmetros e retornar um valor. Neste caso, a função main retorna um valor (do tipo inteiro, *int*) e não aceita nenhum parâmetro (devido ao parênteses vazio).

Dentro das chaves está o bloco que corresponde ao corpo da função.

Hello World!

O programa Hello World!

```c
#include <stdio.h>

int main ()
{
   printf ("Hello World!\n");
   return 0;
}
```

O corpo da função *main*

No corpo da main há dois *comandos* (ordens que passamos ao computador). Um comando é uma expressão, simples ou composta, que pode produzir algum *efeito*.

Todos os comandos em C necessariamente terminam em ponto-e-vírgula!

Hello World!

O programa Hello World!

```c
#include <stdio.h>

int main ()
{
   printf ("Hello World!\n");
   return 0;
}
```

printf("Hello World!");

O comando printf – que está definido na biblioteca *stdio.h*, incluída no início do nosso programa – serve para imprimir dados em tela.

No nosso caso, o comando imprime na tela o conjunto de caracteres *Hello World!*, e pula uma linha (devido ao caractere de escape $\backslash n$).

Hello World!

O programa Hello World!

```c
#include <stdio.h>

int main ()
{
   printf ("Hello World!\n");
   return 0;
}
```

return 0;

O comando *return* indica o fim da função (no caso da main, o fim do programa) e nele é especificado o valor que esta deve retornar.

O valor 0, retornado pela função main, avisa ao sistema operacional que o programa terminou normalmente.

Bibliotecas

Ferramentas alem da linguagem em si

- Uma biblioteca em C é um conjunto de funções e declarações definidas para serem utilizadas por outros programas.
- A *biblioteca padrão* contem as ferramentas mais usadas pelo programador no dia a dia.
- Para usá-las, precisamos incluir – por meio de uma diretriz do tipo *include* – seus cabeçalhos (*headers*) explicitamente no começo do programa. Ex: *#include<stdio.h>* ou *#include<math.h>*
- Muitas vezes precisamos, além da inclusão no corpo do código, informar ao compilador as bibliotecas utilizadas no programa.
- Uma lista com todos os cabeçalhos da *biblioteca padrão* de C pode ser encontrada em http://www.cplusplus.com/reference/

O que são variáveis?

Variáveis

- Variáveis são *porções da memória que armazenam algum valor*.
- Toda variável em C precisa ter um *identificador* (nome que a distingue das outras variáveis) e um *tipo* (que indica sua natureza).

O que são variáveis?

Tipos de dados

- Exemplos de alguns tipos de dados: *int* (número inteiro), *float* (número decimal), *double* (decimal com precisão dupla), *char* (caractere simples), etc.
- Cada tipo de dado ocupa um determinado espaço na memória. Por exemplo, uma variável *int* ocupa 4 bytes, geralmente.
- A linguagem C também permite a criação de tipos definidos pelo usuário.

Tipos de dados básicos

Name	Description	Size*	Range*
`char`	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
`short int (short)`	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
`int`	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`long int (long)`	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`bool`	Boolean value. It can take one of two values: true or false.	1byte	true or false
`float`	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
`double`	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
`long double`	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
`wchar_t`	Wide character.	2 *or* 4 bytes	1 wide character

Tabela 1: Tipos de dados básicos em C.

Declaração X Definição

Declaração de variáveis

- Antes da utilização de uma variável pelo programa, precisamos dizer ao compilador que ela existe. Isto é feito através da *declaração* desta variável.
- A sintaxe para se declarar uma variável em C é: *tipo_da_variavel identificador;*
- Exemplos: *int a; float nome; double minha_variavel;*

Declaração X Definição

Definição de variáveis

- Quando declaramos uma variável, é como se estivéssemos apenas dizendo ao compilador: "*há uma variável com este tipo, e que tem este nome*". Mas ela não armazena nenhum valor útil (apenas lixo de memória).
- Para atribuir um valor à uma variável, podemos *defini-la*.
- A definição é feita através do *operador de atribuição* (=).
- Ex: *a = 3; nome = 5.7; minha_variavel = 10.89;*
- Podemos definir uma variável ao mesmo tempo em que a declaramos: *int a = 4;*
- Erro comum: tentar definir uma variável sem tê-la declarado.

Exemplo (certo)

```c
#include <stdio.h>

// este eh um comentario. comentarios sao ignorados pelo compilador

int main ()
{
    // declarando variaveis
    int a;
    double b;
    int c;
    double d, e;

    // definindo variaveis ao mesmo tempo em que declaramos
    double f = 1.5; float g = 0.5;

    // definindo as variaveis declaradas no inicio do codigo
    c = 4;
    d = e = 2.5;

    // operando com as variaveis
    a = 3*(c+2);                // "a" foi antes declarada, mas nao definida!
    b = f*(g*(d+e));            // "b" foi antes declarada, mas nao definida!
    c = c+a;                    // redefinindo "c"

    // mostrando resultados na tela
    printf("\nOs resultados sao: %d, %f, %d \n\n", a, b, c);
}
```

Exemplo (errado)

```c
#include <stdio.h>

/* este eh um bloco de comentarios
   quaisquer linhas que sejam escritas aqui
   serao consideradas comentarios */

int main ()
{
   // declarando variaveis
   float a, b;

   // calculando
   a = 2;
   float c = a/b;

   // mostrando resultado na tela
   printf("\nO resultado eh: %f \n\n", c);
 }
```

Conversão de tipos

Type casting

- A conversão de tipos, mais conhecida pelo nome em inglês *type casting*, permite que convertamos dados de um tipo para outro. Ela pode ser *explícita* ou *implícita*.
- A conversão *explícita* é feita com a sintaxe:

  (type_name) expression;

  Note que o parênteses no comando acima não é opcional.
- A conversão *implícita* é realizada automaticamente pelo compilador quando algumas operações com tipos de dados diferentes são realizadas.

Conversão de tipos

Conversões implícitas

- Cuidado com as conversões implícitas! Muitas vezes, seu resultado pode não ser o esperado ou o mais intuitivo.
- Regras gerais:
  - *float* para *int* causa truncamento (remoção da parte decimal);
  - *double* para *float* causa arredondamento (remoção de alguns dígitos);
  - *long* to *int* causa remoção de bits em excesso.
- Cuidado também com a *divisão de inteiros*!

Hirarquia das conversões implícitas

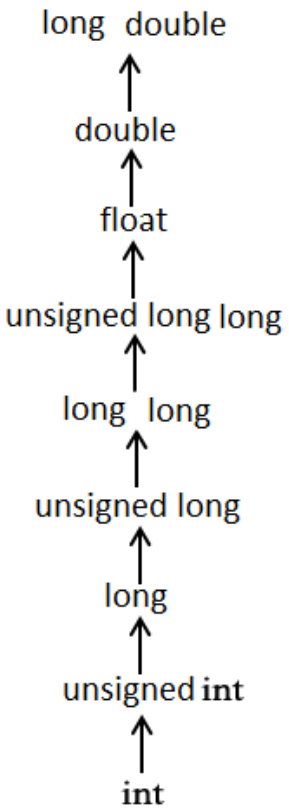

Figura 5: Quando, em uma operação, os operandos têm diferentes tipos, todos são convertidos para o tipo que aparece mais acima nessa hierarquia.

Exemplo – conversão explícita

```c
#include <stdio.h>

int main()
{
  int a, b;

  printf("Entre com uma fracao (numerador e denominador): ");
  scanf("%d %d", &a, &b);

  printf("A fracao em decimal eh  %f\n", (float) a / b);
}
```

Exemplo – conversão implícita

```c
#include <stdio.h>

int main()
{
    int  i = 17;
    char c = 'c'; /* ascii value is 99 */
    float sum;

    sum = i + c;
    printf("Value of sum : %f\n", sum );

}
```

Exemplo – divisão de inteiros

```c
#include<stdio.h>

int main ()
{
  int a, b, c;
  float d, e;

  a = 1; b = 2;

  c = a/b;
  d = a/b;
  e = (float) a/b;

  printf("c = %d\n",c);
  printf("d = %f\n",d);
  printf("e = %f\n",e);
}
```

Constantes

Não se pode mudar o valor

- As constantes são expressoes que têm valor fixo.
- Ja fomos apresentados a algumas constantes há pouco: em um comando de atribuição do tipo a = 2, o valor 2 do lado direito da equação é uma constante.
- Constantes podem também não serem numéricas. Ex: 'z', "oi, tudo bom?". O primeiro exemplo é um caractere constante (denotado com aspas simples), e o segundo é um conjunto de caracteres (ou *string*) constante (denotado com aspas duplas).
- Constantes também podem ser do tipo *bool* (booleano), podendo assumir o valor de *true* ou *false*. Atenção: o tipo *bool* eh exclusivo do C++. Na definição original da linguagem C, usa-se o inteiro 0 para valores falsos e 1 para verdadeiros.
- Constantes podem ser declaradas de maneira idêntica às variáveis, sendo a declaração precedida da palavra-chave *const*. Ex: const int a = 200; const char = 'a';

Operadores

Operando com as variáveis e constantes

- Um *operador* é um símbolo que faz com que o programa execute manipulações matemáticas ou lógicas específicas.
- Os operadores mais comuns da linguagem C podem ser classificados em:
  - aritméticos;
  - relacionais;
  - lógicos;
  - de atribuição.
- Cuidado com a precedência dos operadores! Na dúvida, sempre use parênteses.

Operadores aritméticos

Following table shows all the arithmetic operators supported by C language. Assume variable **A** holds 10 and variable **B** holds 20, then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increments operator increases integer value by one	A++ will give 11
--	Decrements operator decreases integer value by one	A-- will give 9

Tabela 2: Fonte: http://www.tutorialspoint.com/cprogramming

Operadores relacionais

Following table shows all the relational operators supported by C language. Assume variable **A** holds 10 and variable **B** holds 20 then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes tru	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Tabela 3: Fonte: http://www.tutorialspoint.com/cprogramming

Operadores lógicos

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Tabela 4: Fonte: http://www.tutorialspoint.com/cprogramming

Operadores de atribuição

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A

Tabela 5: Fonte: http://www.tutorialspoint.com/cprogramming

Entrada e saída de dados

Comandos

- Os comandos mais simples para entrada e saída de dados em C, respectivamente, são *scanf* e *printf* (este último, já apresentado).
- Em ambos os comandos, dois recursos são muito utilizados: os *especificadores de formato* e os *caracteres de escape*.
- Como ilustração, podemos analisar a seguinte linha de um dos exemplos dados anteriormente:
  printf("\nOs resultados são: %d, %f, %d \n \n", a, b, c);
- No caso, %d e %f são os *especificadores de formato* para variáveis inteiras e de ponto flutuante, respectivamente, e \n é o *caractere de escape*, que serve para pular uma linha.
- O *scanf* é utilizado de maneira análoga ao *printf*.

Especificadores de formato

specifier	Output	Example
d *or* i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	
%	A % followed by another % character will write a single % to the stream.	%

Tabela 6: Especificadores de formato em C.

COPPE UFRJ

Caracteres de escape

Escape sequence	Hex value in ASCII	Character represented
\a	07	Alarm (Beep, Bell)
\b	08	Backspace
\f	0C	Formfeed
\n	0A	Newline (Line Feed); see notes below
\r	0D	Carriage Return
\t	09	Horizontal Tab
\v	0B	Vertical Tab
\\	5C	Backslash
\'	27	Single quotation mark
\"	22	Double quotation mark
\?	3F	Question mark
nnn	any	The character whose numerical value is given by *nnn* interpreted as an octal number
\x*hh*	any	The character whose numerical value is given by *hh* interpreted as a hexadecimal number

Tabela 7: Caracteres de escape em C.

Exemplo - Input/output

```c
#include<stdio.h>

int main()
{
  int myvariable;

  printf("Enter a number:");
  scanf("%d",&myvariable);
  printf("%d",myvariable);

  return 0;
}
```

Estruturas de controle

Importância

- O coração da arte de se programar encontra-se nas *estruturas de controle*.
- Digitar uma linha para cada operação feita ou decisão tomada pelo computador seria cansativo, concorda?
- Através das estruturas de controle, podemos com poucas linhas de código ordenar ao computador o que ele realmente foi designado para fazer, ou seja, *tarefas repetitivas e automáticas*.
- Dividem-se em dois tipos.
  - Estruturas condicionais;
  - Estruturas iterativas.

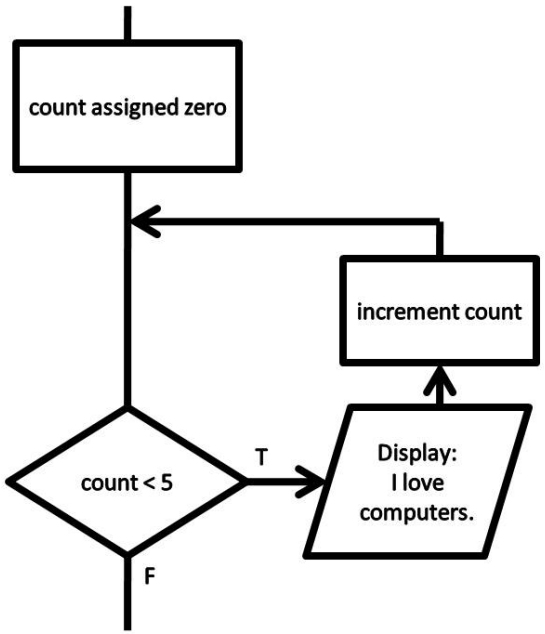

Figura 6: Exemplo de um algoritmo que utiliza estruturas condicionais e iterativas.

Estruturas Condicionais

Expressões lógicas

- As estruturas condicionais avaliam *expressões lógicas* para decidir se executam ou não algum comando ou conjunto de comandos.
- Expressões lógicas, construidas com operadores condicionais e logicos, só podem resultar em dois valores: *verdadeiro* ou *falso*.
- A ideia é simples: se o resultado da expressão for verdadeiro, o(s) comando(s) são executado(s). Se for falso, não são executado(s).
- As expressões lógicas são construídas de acordo com a *álgebra booleana*.

Estruturas Condicionais - o bloco if

o bloco if

- Executa um ou mais comandos sob determinada condição, representada por uma expressão lógica. A sintaxe é:

```
if (condicao)
{
        lista de comandos;
}
else if (outra condicao)
{
        outra lista de comandos;
}
else
{
        ultima lista de comandos;
}
```

Exemplo - if

Exemplo - if

- Escreva um programa que peça ao usuário um valor de temperatura, em °C, e indique em qual estado físico, ao nível do mar, se encontra a água.

Exemplo - if

```c
#include <stdio.h>

int main ()
{

    double T;

    printf("Insira a temperatura, em graus Celsius \n");
    scanf("%lf", &T);

    if (T<-273.15)
    {
        printf ("Temperatura abaixo do zero absoluto! \n");
    }
    else if (T>-273.15 & T<0)
    {
        printf("Agua no estado solido \n");
    }
    else if (T==0)
    {
        printf("Agua no equilibrio S-L \n");
    }
```

Exemplo - if (cont.)

```
    else if (T>0 & T<100)
    {
        printf("Agua no estado liquido \n");
    }
    else if (T==100)
    {
        printf("Agua no equilibrio L-V \n");
    }
    else
    {
        printf("Agua no estado vapor \n");
    }
}
```

Estruturas Iterativas - o bloco while

o bloco while

- Executa, enquanto uma dada *condição* permanece satisfeita, um ou mais comandos. A sintaxe é:

```
while (condicao)
{
        lista de comandos;
}
```

Exemplo - while

Exemplo - while

- Mostre a lista dos números naturais em tela, desde o 5000, em ordem decrescente.

Exemplo - while

```c
#include <stdio.h>

int main ()
{

  int i = 1000;

  while(i>0)
  {
    printf("%d ",i);
    i = i-1;
  }

  printf("acabou!");

}
```

Estruturas Iterativas - o bloco for

o bloco for

- Executa, enquanto uma dada *condição* permanece satisfeita, um ou mais comandos enquanto um *contador* percorre valores em *passos* definidos. A sintaxe é:

```
for (inicializacao; condicao; passo)
{
        lista de comandos;
}
```

Exemplo - for

Exemplo - for

- Calcule a soma:

$$\sum_{i=1}^{6} 3^{i+1}$$

Exemplo - for

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

int main ()
{

  int i;                    // contador
  int soma = 0;         // valor da soma

  for(i=1;i<=6;i++)          // novo operador (++): i++ significa i = i+1
  {
    soma = soma + pow(3,(i+1));
  }

  printf("\nO resultado eh: %d\n\n", soma);

}
```

Funções

O que é uma função?

- Uma função *é um grupo de comandos que é executado quando chamado de algum ponto do programa*. A sintaxe é:

```
tipo_retornado nome_da_funcao (parametro1, parametro2,...)
{
        lista de comandos;
}
```

- Funções são extremamente úteis no sentido em que podemos organizar nosso programa de uma maneira *modular*, aproveitando-nos assim de todo o potencial que o *paradigma de programação estruturada* do C pode nos oferecer.

Funções

Chamando as funções

- Para chamar as funções no código, devemos usar seu nome, seguido de parênteses contendo os argumentos que serão passados. Ex: soma (3,5);
- Se a função não tiver argumentos, o parênteses deve ser mantido, mesmo que vazio! Ex: soma();

Funções

Exemplos

- 1) Escreva um programa que contenha uma função que compute a soma de dois números inteiros. Demonstre o uso da função.
- 2) Escreva um programa que resolva a equação $x - cos(x) = 0$ pelo método da bisseção.

Exemplo 1 - funções

```c
#include <stdio.h>

int adicao (int a, int b)
{
  int r;
  r = a+b;
  return r;
}

int main ()
{
  int z = adicao (2,3);
  printf("\nO resultado eh: %d\n\n", z);
}
```

Exemplo 2 - funções

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

// prototipo das funcoes:
// as funcoes 'f' e 'bissecao' sao declaradas como prototipos; suas definicoes
// so virao no final do codigo
double f (double x);        // equacao a ser resolvida
double bissecao (double a, double b, double tol, int it_max);  // metodo

int main ()
{
  double z = bissecao (0, 10, 1e-4, 50);
  printf("\n0 resultado eh: %e\n\n", z);
}

double f(double x)
{
  double y = x - cos(x);
  return y;
}
```

Exemplo 2 - funções (cont.)

```
double bissecao (double a, double b, double tol, int it_max)
{
  double c;    // usado na iteracao
  int it;      // contador

  printf("\nResolvendo equacao transcedental pelo metodo da Bissecao:\n\n");

  if(f(a)*f(b)>0)
  {
    printf("\nAtencao! A raiz nao encontra-se no intervalo dado.\n\n
    **** ERRO FATAL **** INTERROMPENDO ****");
    exit(-1);
  }

  else
  {
    printf("it \t a \t b \t c \t f(c) \t \n");
```

Exemplo 2 - funções (cont.)

```
    for (it=1;it<=it_max;it++)
    {
      c=0.5*(a+b);

      if (f(a)*f(c)<0)
         b=c;
      else
         a=c;

      printf("%d \t %e \t %e \t %e \t %e\n\n", it, a, b,c, f(c));

      if (fabs(f(c))<=tol)
      {
        return c;
        break;
      }

      if (it>=it_max)
      {
        printf("\n METODO DIVERGIU\n\n *****
        ERRO FATAL ***** INTERROMPENDO *****");
        exit(-1);
       }
    }
  }
}
```

Escopo

De onde as variáveis podem ser vistas?

- Variáveis definidas dentro do corpo de uma função ou bloco só são válidas, e portanto, só podem ser utilizadas, dentro da função ou bloco.
- A porção do código de onde a variável pode ser enxergada é chamada de *escopo* da variável.
- Variáveis definidas dentro de uma função ou bloco, portanto, *têm como escopo a própria função ou bloco*.

Escopo

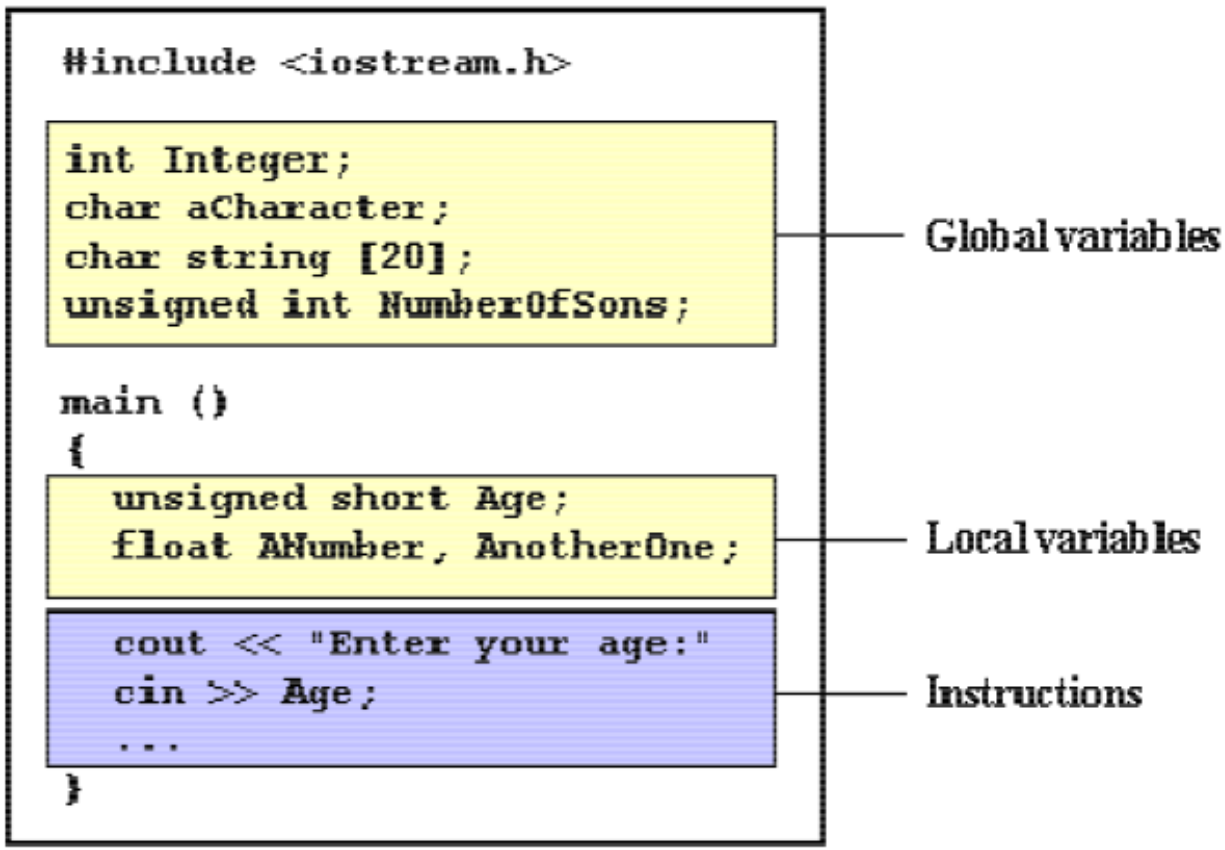

Figura 7: Variáveis locais e globais em C.

Exemplo de função recursiva

```c
// factorial calculator

#include<stdio.h>

long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return 1;
}

int main ()
{
  long number = 4;
  printf("\n%d! = %d\n\n",number,factorial(number));
  return 0;
}
```

Ponteiros

Um ponteiro aponta!

- Um ponteiro é uma variável que *guarda o endereço de outra variável*.
- Ponteiros são declarados da mesma maneira que as variáveis comuns, porém com um asterisco (*) precedendo seu nome. Ex: int * ptr;
- No caso, *ptr* é um ponteiro que aponta para uma variável do tipo int, ou seja, *ptr* é uma variável que armazena o endereço de uma outra variável do tipo int.

Operador referência

Obtendo o endereço de uma variável

- Para obter o endereço de uma variável, podemos usar o operador referência, &.
- Por exemplo, se definirmos uma variável do tipo int:

  int var;

  e um ponteiro que aponta para uma variável do tipo int:

  int * ptr;

  podemos fazer com que o ponteiro *ptr* armazene o endereço da variável *var* com o comando:

  ptr = &var;
- O operador referência pode ser lido como "o endereço de".

Exemplo - operador referência

```c
/* Example to demonstrate use of reference operator in C programming. */

#include <stdio.h>

int main()
{
  int var=5;
  printf("Value: %d\n",var);
  printf("Address: %d",&var);  //Notice, the ampersand(&) before var.
  return 0;
}
```

Operador derreferência

Obtendo a variável de um endereço

- O operador derreferência * serve para acessar o valor da variável para a qual um ponteiro aponta.
- Por exemplo, se temos um ponteiro *ptr*, que aponta para uma variável *var*, como no caso do slide anterior, o comando:

  var2 = *ptr;

  associa à variável *var2* o valor apontado pelo ponteiro *ptr*, ou seja, o valor da variável *var*.
- O operador derreferência pode ser lido como "o valor apontado por".

Exemplo 1 - ponteiros

```c
/* Source code to demonstrate, handling of pointers in C program */

#include <stdio.h>

int main()
{
    int* pc;
    int c;
    c=22;
    printf("Address of c:%d\n",&c);
    printf("Value of c:%d\n\n",c);
    pc=&c;
    printf("Address of pointer pc:%d\n",pc);
    printf("Content of pointer pc:%d\n\n",*pc);
    c=11;
    printf("Address of pointer pc:%d\n",pc);
    printf("Content of pointer pc:%d\n\n",*pc);
    *pc=2;
    printf("Address of c:%d\n",&c);
    printf("Value of c:%d\n\n",c);
    return 0;
}
```

Exemplo 2 - ponteiros

```c
#include<stdio.h>

int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  printf("firstvalue is %d\n", firstvalue);
  printf("secondvalue is %d\n", secondvalue);
  return 0;
}
```

Exemplo 3 - ponteiros

```c
#include<stdio.h>

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;         // value pointed to by p2 = value pointed by p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed by p1 = 20

  printf("firstvalue is %d\n", firstvalue);
  printf("secondvalue is %d\n", secondvalue);

  return 0;
}
```

Aritmética de ponteiros

Contas com ponteiros

- As únicas operações aritméticas válidas para ponteiros são a adição e a subtração.
- Como os ponteiros representam posições na memória, sua aritmética respeita o espaço que cada tipo de dado ocupa. Se declaramos os ponteiros:

  char *mychar;
  short *myshort;
  long *mylong;

  e escrevemos:

  ++ mychar;
  ++ myshor;
  ++ mylong;

  as contas são feitas como ilustrado ao lado.

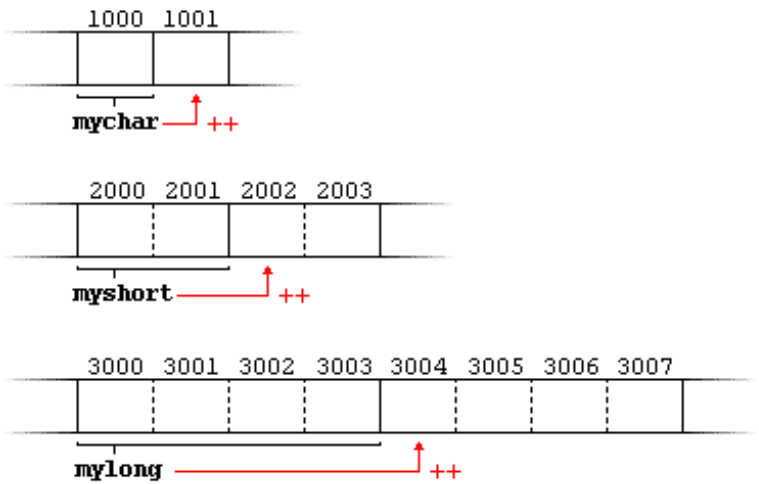

Figura 8: Ilustração da aritmética de ponteiros.

Ponteiros constantes

Ponteiros que não podem mudar o valor apontado

- Para se declarar um ponteiro que nao tem capacidade de mudar o valor da variável para a qual aponta, devemos preceder sua declaração pela palavra-chave *const*. Ex:

```
int x;
int y = 10;
const int * p = &y;
x = *p;     // ok: reading p
*p = x;     // error: modifying p, which is const-qualified
```

Ponteiros genéricos

Ponteiros que podem apontar para qualquer lugar

- Um ponteiro genérico é um ponteiro sem tipo, ou seja, que pode apontar para qualquer endereço da memória, independente do tipo de dado ali armazenado.
- Declaramos um ponteiro genérico da mesma maneira que os tradicionais, com a palavra-chave *void* no lugar do tipo:

  ```
  void * ponteiro;
  ```

- Ao se desreferenciar um ponteiro genérico (ou seja, obter o valor por ele apontado), é obrigatório fazer um *type casting* explícito.

Exemplo - ponteiros genéricos

```c
#include <stdio.h>

int main()
{
  int valor = 20;
  float valor2 = 5.23;

  void *ponteiro; // ponteiro generico

  ponteiro = &valor; // aponta para um inteiro
  printf("%d\n", *(int *)ponteiro);

  ponteiro = &valor2; // aponta para um float
  printf("%0.2f\n", *(float *)ponteiro);

  return 0;
}
```

Ponteiros indeterminados

Ponteiros que não sabemos para onde apontam

- Ponteiros podem apontar para qualquer lugar. Por exemplo:

```
int * p; // uninitialized pointer (local variable)
int myarray[10];
int * q = myarray+20; // element out of bounds
```

- Nesse caso, p e q apontam para locais específicos da memória, ainda que seus conteúdos sejam indefinidos. Não podemos prever o que vai acontecer ao se tentar acessar esses endereços, já que não sabemos o que está armazenado neles.

Ponteiros nulos

Ponteiros que não apontam para lugar algum

- Às vezes é necessário que se defina explicitamente que um ponteiro não aponta para lugar algum. Quando isso acontece, temos um *ponteiro nulo*.
- A sintaxe para se declarar um ponteiro nulo é:

  int * p = 0;
- É importante não confundir os conceitos de ponteiros *constantes*, *genéricos*, *indeterminados* e *nulos*!!

Passagem de argumentos por valor ou referência

Como passamos os argumentos para as funções?

- Até agora, em todas as funções que vimos, os argumentos foram fornecidos às funções *por valor*.
- Isso significa que, ao chamar as funções, o que passamos a elas foram *meras cópias de seus valores*, mas nunca *as variáveis em si*.
- Mas há casos em que precisamos manipular, de dentro de uma função, o valor de uma variável externa. Para isso passamos os argumentos *por referência.*

Passagem de argumentos por valor ou referência

Passagem de argumentos por referência

- Quando passamos os argumentos por referência, informamos para a função, através de um ponteiro, o *endereço da variável*, e não uma cópia de seu valor.
- Sendo assim, a variável para a qual o ponteiro passado como argumento aponta pode ser manipulada diretamente de dentro da função (obs: isso significa uma vantagem ou desvantagem? Por quê?)
- Além do mais, em relação a velocidade e espaço na memória, o que você considera mais eficaz, passar uma variável como argumento por valor ou por referência?

Exemplo 1 - Passagem de argumentos por referência

```c
#include <stdio.h>

void duplicate (int *a, int *b, int *c)
{
  *a*=2;
  *b*=2;
  *c*=2;
}


int main()
{
  int x=1, y=3, z=7;
  duplicate (&x,&y,&z);
  printf("%d, %d, %d",x,y,z);
  return 0;
}
```

Exemplo 2 - Passagem de argumentos por referência

```
#include <stdio.h>

void increment_all (int* start, int* stop)
{
  int * current = start;
  while (current != stop)
  {
    ++(*current);  // increment value pointed
    ++current;     // increment pointer
  }
}

void print_all (const int* start, const int* stop)
{
  const int * current = start;
  while (current != stop)
  {
    printf("%d \n",*current);
    ++current;     // increment pointer
  }
}

int main ()
{
  int numbers[] = {10,20,30};
  increment_all (numbers,numbers+3);
  print_all (numbers,numbers+3);
  return 0;
}
```

Arrays

Guardando vários valores sob um nome só

- Um *array* é uma série de elementos do mesmo tipo, dispostos em posições adjacentes da memória e que podem ser referenciados individualmente, adicionando-se um índice a um único identificador.
- A sintaxe para a declaração de um *array* é:

  tipo identificador [no. de elementos];

  Ex: int billy [5];

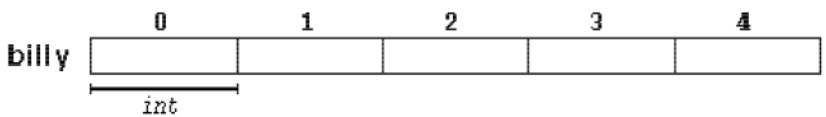

Figura 9: *Array* do exemplo ao lado.

Arrays

Referindo-se a cada valor individual

- Podemos nos referir a cada valor individual do *array* utilizando um índice em colchetes. Por exemplo, o comando:

  billy[3] = 77;

  atribui o valor 77 ao quarto (!) elemento do *array* billy. O comando:

  int var = billy[3];

  atribui o valor armazenado na quarta posição do *array* billy (no caso, 77) à variável recém-criada var.
- Atente às duas funções diferentes dos colchetes:
  - especificar o tamanho do *array*, no momento da declaração;
  - especificar o índice de um *array* que já existe.
- Atente-se também à peculiar numeração dos elementos de *arrays* em C, que começa do zero!

Arrays

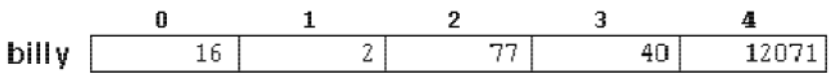

Figura 10: *Array* do exemplo ao lado.

Definir no momento de declarar

- Podemos definir um *array* diretamente no momento da declaração. Por exemplo, poderíamos ter definido o *array* billy com o comando:

  int billy [ ] = { 16, 2, 77, 40, 12071 };

- Os elementos devem vir dentro de chaves, após um operador de atribuição (=). Note que esta forma de definição do *array* sé é válida no momento da declaração.

Exemplo - *arrays*

```
// arrays example

#include <stdio.h>

int foo [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; ++n )
  {
    result += foo[n];
  }
  printf("%d", result);
  return 0;
}
```

Arrays multidimensionais

Construindo matrizes

- Um *array* multidimensional é construído adicionando-se mais índices no momento de sua definição. Por exemplo, o seguinte comando:

  int jimmy [3][5];

  cria uma matriz do tipo int 3x5, ou seja, separa 15 espaços consecutivos na memória para elementos do tipo int, identificados com o nome comum *jimmy*.
- Acessamos os elementos individuais da mesma maneira que para o caso unidimensional:

  jimmy [1][3] = 4;

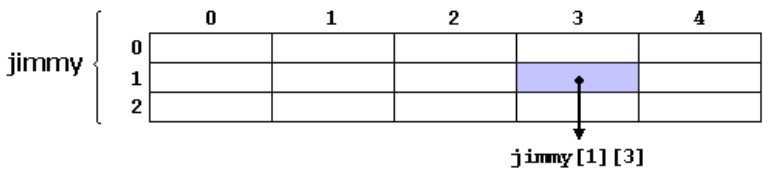

Figura 11: *Array* multidimensional do exemplo ao lado.

Arrays de caracteres

Armazenando caracteres

- *Arrays* também podem armazenar elementos do tipo char (caracteres).
- O comando:

  char foo [20];

  cria um *array* (ainda não definido) de 20 caracteres, ou seja, separa 20 espaços consecutivos na memória para elementos do tipo char, identificados com o nome comum *foo*.

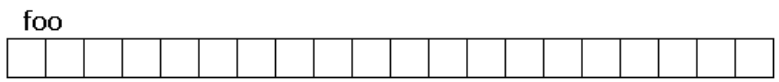

Figura 12: *Array* de caracteres foo no momento de sua declaração.

Strings

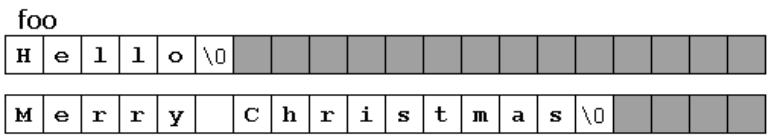

Figura 13: *Array* foo armazenando os strings "Hello!" e "Merry Christmas!", respectivamente.

Palavras!

- Um *string* é um *array* de caracteres cujo último elemento definido é o caractere de escape nulo \0.
- Podemos armazenar um *string* em um *array* de caracteres, desde que aquele tenha no máximo o mesmo número de elementos que este. O compilador reconhece o fim do *string* através do caractere de escape nulo, \0. Ex:

  foo[1] = 'H'; foo[2] = 'e';
  foo[3] = 'l'; foo[4] = 'l';
  foo[5] = 'o'; foo[6] = '\0';

Strings

Definindo no momento da declaração

- Podemos definir um *string*, no momento da declaração, à maneira tradicional dos *arrays*:

  char foo [ ] = { 'H', 'e', 'l', 'l', 'o', '\0' };

  ou usando aspas duplas:

  char foo [ ] = "Hello";

- Nesse último caso, não há a necessidade de adicionar o caractere nulo '\0' (as aspas duplas já sinalizam ao compilador que trata-se de um *string* e o '\0' é adicionado automaticamente).

Relação entre ponteiros e *arrays*

Um *array* é um tipo especial de ponteiro

- Ponteiros e *arrays* em C estão intimamente relacionados. Na verdade, um *array* pode ser encarado como um *ponteiro que sempre aponta para seu primeiro elemento*.
- Diferença para ponteiros tradicionais: uma vez declarados, os *arrays* não podem mais ser apontados para outro endereco na memória.

Relação entre ponteiros e *arrays*

Um array é um tipo especial de ponteiro

- Sendo equivalentes os conceitos de *arrays* e ponteiros, os seguintes comandos são válidos:

  ```
  int myarray [20];
  int * mypointer;
  mypointer = myarray;
  ```

  Note que, ao longo do código, outro endereço de memória pode ser atribuido a mypointer, enquanto myarray sempre vai apontar para o mesmo lugar.

- O comando

  ```
  myarray = mypointer;
  ```

  não seria válido nesse caso. Por quê?

Relação entre ponteiros e *arrays*

Operador offset

- O operador [ ], que foi apresentado como sendo um símbolo que especifica a posição de um elemento em um *array*, na verdade é um tipo de operador derreferência, chamado de *operador offset*.
- O que ele faz é simples: derreferencia a variável que o precede, assim como o operador *, mas adicionando o número entre as chaves ao endereço da variável derreferenciada.
- Por exemplo, os comandos:

```
a[5] = 0;       // a [offset of 5] = 0
*(a+5) = 0;     // pointed by (a+5) = 0
```

são equivalentes.

Exemplo - relação entre ponteiros e *arrays*

```c
#include <stdio.h>

int main ()
{
  int numbers[5];
  int * p;
  int n;

  p = numbers; *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;

  for (n=0; n<5; n++)
    printf("%d, ", numbers[n]);

  return 0;
}
```

Exemplo 2 - relação entre ponteiros e *arrays*

```c
#include<stdio.h>

int main()
{
    // Declaring/Initializing three characters pointers
    char *ptr1 = "Himanshu";
    char *ptr2 = "Arora";
    char *ptr3 = "TheGeekStuff";

    //Declaring an array of 3 char pointers
    char* arr[3];

    // Initializing the array with values
    arr[0] = ptr1;
    arr[1] = ptr2;
    arr[2] = ptr3;

    //Printing the values stored in array
    printf("\n [%s]\n", arr[0]);
    printf("\n [%s]\n", arr[1]);
    printf("\n [%s]\n", arr[2]);

    return 0;
}
```

Indo além dos limites

Olha o perigo

- Em C, é sintaticamente correto exceder o tamanho de um *array*. Ou seja, se criarmos um *array* de 5 elementos e no meio do código tentarmos acessar o décimo elemento, o compilador não irá reclamar e o executável será gerado normalmente.
- Ou seja, exceder os limites de um *array* em C *não é um erro de sintaxe e sim de lógica*.

Indo além dos limites - exemplo

```c
#include<stdio.h>

unsigned int count = 1;

int main(void)
{
    int b = 10;
    int a[3];
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;

    printf("\n b = %d \n",b);
    a[3] = 12;
    printf("\n b = %d \n",b);

    return 0;
}
```

Memória dinâmica

E quando não sabemos de quanto espaço precisamos?

- Muitas vezes, não sabemos a priori (a tempo de compilação) o tamanho da memória que precisamos para dada operação. Por exemplo, quando temos um array cujo número de elementos é um input a ser dado pelo usuário.
- Nesses casos, são utilizadas técnicas de *alocação dinâmica de memória*.
- Ao utilizarmos tais técnicas, alocamos a memória necessária para dada operação a *tempo de execução*.
- Na definição da linguagem em si, não ha nenhuma técnica do tipo, mas na biblioteca padrão (cabecalho *stdlib.h*) temos quatro funções implementadas com esse fim: *malloc, calloc, free* e *realloc*.

Memória dinâmica

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	dellocate the previously allocated space
realloc()	Change the size of previously allocated space

Tabela 8: Funções disponíveis na biblioteca padrão de C para alocação dinâmica de memória.

Função malloc

malloc

- A função malloc aloca um bloco de bytes consecutivos na memória do computador e retorna o endereço desse bloco, na forma de um ponteiro genérico. O número de bytes alocados é especificado no argumento da função. A sintaxe é:

  ptr = malloc (tamanho-em-bytes);

- Caso queiramos que o ponteiro ptr não seja genérico e sim tenha um tipo definido, podemos fazer um *type casting* explícito no momento da declaração:

  ptr = (tipo*) malloc (tamanho-em-bytes);

- Exemplo:

  ptr = (int*) malloc (100*sizeof(int));

  Nesse caso, a função malloc aloca um espaço na memória correspondente a 100 variáveis do tipo int e retorna um ponteiro que aponta para o endereço do primeiro byte.

- Note o uso da função sizeof, que garante a portabilidade do código.

COPPE UFRJ

Função malloc - exemplo

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

int main()
{
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc

    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements of array: ");

    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }

    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

Função calloc

calloc

- A função calloc é parecida com a malloc. A diferença é que esta aloca um bloco simples de memória, enquanto aquela aloca blocos contíguos, todos de mesmo tamanho, e os inicializa como zero. A sintaxe é:

  ptr = (tipo*) calloc (no-de-elementos, tamanho-em-bytes-por-elemento);

- Exemplo:

  ptr = (float*) calloc (25, sizeof(float));

Função calloc - exemplo

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

int main()
{
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));

    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements of array: ");

    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }

    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

Função free

free

- Para liberar o espaço alocado pelas funções malloc ou calloc, precisamos usar a função free, cuja sintaxe é bem simples:

  free (ptr);

  sendo ptr o endereço da memória que se deseja desalocar.

Função realloc

realloc

- Se o tamanho da memória alocada for insuficiente, podemos usar o comando realloc para realocar a memória. A sintaxe é bem simples:

  ptr = realloc (ptr,newsize);

structs

Estruturas

- Uma estrutura de dados, ou mais simplesmente *struct*, é um tipo de dado, criado pelo usuário, que contém vários elementos agrupados sob um mesmo nome. Os elementos, chamados de *membros*, podem ter diferentes tipos ou comprimentos. A sintaxe para a declaração é:

```
struct nome
{
    tipo_do_membro1   nome_do_membro1;
    tipo_do_membro2   nome_do_membro2;
    tipo_do_membro3   nome_do_membro3;
    .
    .
    .
} nomes_dos_objetos;
```

structs

Estruturas

- A utilidade das estruturas fica clara ao analisarmos um exemplo concreto:

```
struct product
{
    int weight;
    double price;
} apple, banana, melon;
```

- Os três objetos também poderiam ser declarados após a declaração do struct, em qualquer ponto do código, à maneira tradicional de declaração de variáveis:

```
struct product apple;
struct product banana, melon;
```

structs

Estruturas

- É importante diferenciarmos entre um struct (que representa um tipo de dado, definido pelo usuário) e um *objeto* dessa struct (ou seja, uma variável que tem como tipo a struct).
- No caso anterior, o novo tipo criado foi a struct *product* e os três objetos (variáveis cujos tipos são struct) foram *apple*, *banana* e *melon*. Note que podemos criar quantos objetos quisermos do tipo *product*.

structs

Estruturas

- Após criada a struct e seus objetos, podemos operar com os membros de cada objeto utilizando o operador ponto. Ex:

  apple.weight
  apple.price
  banana.weight
  banana.price
  melon.weight
  melon.price

- Note que cada objeto tem seu próprio conjunto de membros.

structs - exemplo

```c
struct database {
  int id_number;
  int age;
  float salary;
};

int main()
{
  struct database employee;  /* There is now an employee variable that has
                                modifiable variables inside it.*/
  employee.age = 22;
  employee.id_number = 1;
  employee.salary = 12000.21;
}
```

structs - exemplo 2

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

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

/* function declaration */
void printBook( struct Books book );

int main( )
{
   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;
```

structs - exemplo 2 (cont.)

```
   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info */
   printBook( Book1 );

   /* Print Book2 info */
   printBook( Book2 );

   return 0;
}

void printBook( struct Books book )
{
   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n\n", book.book_id);
}
```

structs - exemplo 3

```c
// array of structures

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

struct movies_t {
  char title [20];
  int year;
};

void printmovie (struct movies_t movie);

int main ()
{
  int i,n;

  struct movies_t * films;

  printf("Enter the number of movies: ");
  scanf("%d", &n);

  films = (struct movies_t *) malloc (n*sizeof(struct movies_t));
```

structs - exemplo 3 (cont.)

```
  for (i=0; i<n; i++)
  {
    printf("\nEnter title: ");
    scanf("%s",films[i].title);
    printf("Enter year: ");
    scanf("%d",&films[i].year);
  }

  printf("\nYou have entered these movies:\n");

  for (i=0; i<n; i++)
    printmovie (films[i]);

  return 0;
}

void printmovie (struct movies_t movie)
{
  printf("%s", movie.title);
  printf(" (%d)\n",movie.year);
}
```

structs

Operador seta ->

- Podemos ter ponteiros que apontem para variáveis (objetos) cujos tipos são structs.
- Se temos um ponteiro que aponta para algum objeto, podemos acessar os membros desse objeto através do ponteiro. Para isso utilizamos um tipo especial de operador derreferência, chamado *operador seta* (->). Ex:

```
struct product orange;
struct product * ptr;
ptr = &orange;
ptr->price = 5;
```

- Acessamos o membro *price* do objeto *orange*, indiretamente, através do ponteiro *ptr*.

Operador seta - exemplo

```c
#include <stdio.h>

struct student
{
      int id;
      char name[30];
      float percentage;
};

int main()
{
      int i;
      struct student record1 = {1, "Raju", 90.5};
      struct student *ptr;

      ptr = &record1;

      printf("Records of STUDENT1: \n");
      printf("  Id is: %d \n", ptr->id);
      printf("  Name is: %s \n", ptr->name);
      printf("  Percentage is: %f \n\n", ptr->percentage);

      return 0;
}
```

structs

Expression	What is evaluated	Equivalent
`a.b`	Member `b` of object `a`	
`a->b`	Member `b` of object pointed to by `a`	`(*a).b`
`*a.b`	Value pointed to by member `b` of object `a`	`*(a.b)`

Tabela 9: Combinações entre operadores de ponteiros e de membros de estruturas.

unions

Um único endereço com vários tipos

- *Unions* permitem que uma única porção da memória tenha ao mesmo tempo vários tipos.
- A sintaxe para criar um union é semelhante a do struct:

```
union nome
{
    tipo_do_membro1   nome_do_membro1;
    tipo_do_membro2   nome_do_membro2;
    tipo_do_membro3   nome_do_membro3;
    .
    .
    .
} nomes_dos_objetos;
```

unions

Um único endereço com vários tipos

- Exemplo:

```
union mix_t
{
    int l;
    struct
    {
       short hi;
       short lo;
    } s;
    char c[4]
} mix;
```

unions

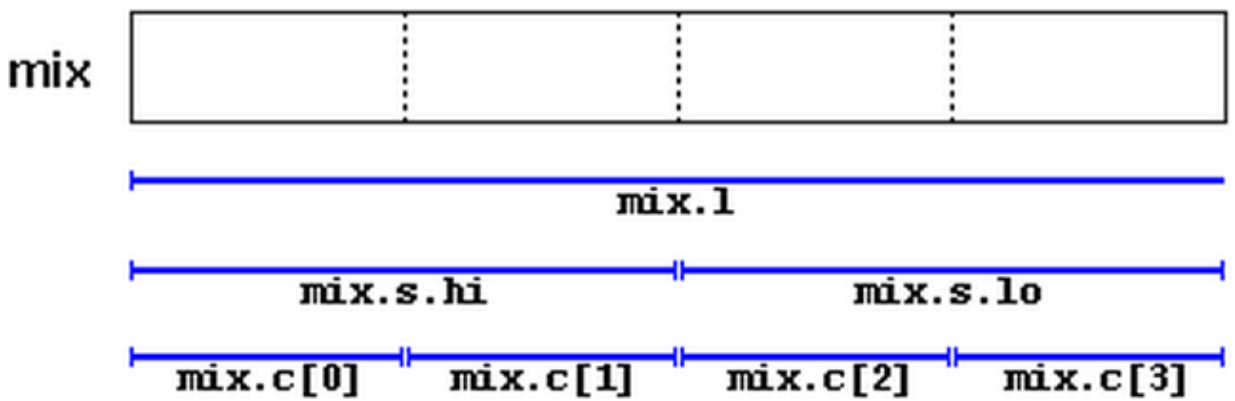

Figura 14: Memória reservada pelo union do slide anterior.

unions - exemplo 1

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

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   printf( "Memory size occupied by data : %d\n", sizeof(data));

   return 0;
}
```

unions - exemplo 2

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

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

unions - exemplo 3

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

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

enumerations

Constantes nomeadas

- Um *enumeration* é um tipo que consiste de um conjunto de constantes inteiras, cada uma com um nome.
- A sintaxe para criar um *enumeration* é familiar:

```
enum nome
{
    nome_do_membro1;
    nome_do_membro2;
    nome_do_membro3;
    .
    .
    .
} nomes_dos_objetos;
```

enumerations

Constantes nomeadas

- Note que, na declaração de um enumeration, não usamos nenhum outro tipo pré-definido. Portanto, um enumeration pode ser visto como um tipo criado do zero, totalmente novo.
- A cada membro de um enumeration está associado um inteiro constante. Por default, ao primeiro membro está associado 0, ao segundo membro 1, e assim sucessivamente. Mas o programador pode mudar essa configuração. Ex:

```
enum clubes
{
    vasco=0;
    flamengo=10;
};
```

enumerations - exemplo

```c
#include <stdio.h>

enum week{sunday, monday, tuesday, wednesday, thursday, friday, saturday};

int main()
{
    enum week today;
    today=wednesday;
    printf("%d day\n",today+1);
    return 0;
}
```

typedefs

Atalhos para tipos

- Um *typedef* é simplesmente um nome diferente pelo qual um tipo pode ser identificado.
- A sintaxe para criar um *typedef* é simples:

  typedef tipo_existente novo_nome ;
- Por exemplo, após definirmos o typedef:

  typedef char C;

  podemos criar uma variável mychar, do tipo char, com o comando:

  C mychar;

A linguagem C++

Figura 15: Bjarne Stroustrup.

A Linguagem

- O C++ é uma extensão da linguagem C, desenvolvida por Bjarne Stroustrup com o objetivo de incorporar nesta o paradigma de programação orientada a objetos.
- O sucesso da linguagem no mundo da computação pode ser verificado com os vários softwares nela escritos, como Windows, Mac OS X, Mozilla Firefox, Microsoft Office, Adobe Acrobat, etc.

A linguagem C++

Diferenças menores para o C

- Além da orientação a objetos, existem algumas diferenças menores entre o C++ e o C:
  - uma biblioteca padrão mais poderosa, a *STL* (*Standard Template Library*), que contém por exemplo uma classe nova para input/output de dados (*iostream*);
  - novos operadores para gerenciamento de memória dinâmica (*new* e *delete*);
  - presença de um novo tipo de dado (*bool*) para operações lógicas;
  - suporte a *tratamento de exceções*, uma ferramenta útil para debug,

  entre outras.
- Para mais diferenças, consulte: http://www.cprogramming.com/tutorial/c-vs-c++.html

A linguagem C++

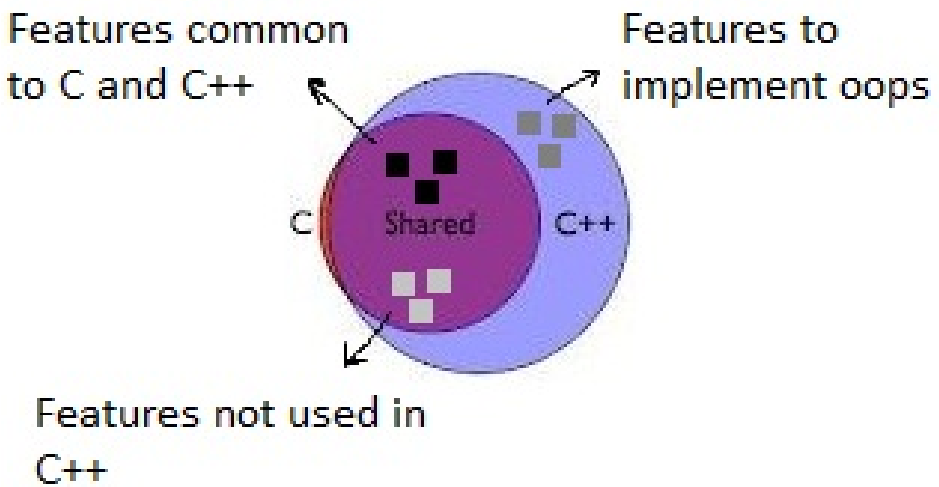

Figura 16: Diagrama mostrando a relação entre o C e o C++.

Classes

Estendendo a ideia de estruturas

- A ideia inicial da orientação a objetos é o conceito de *classes*.
- Uma classe é um tipo de dado personalizado, parecido com o struct, só que com duas diferenças:
  - seus membros podem ser funções;
  - há mecanismos de controle de acesso aos membros.

Classes

Estendendo a ideia de estruturas

- A sintaxe para a declaração de uma classe é:

```
class nome
{
    especificador_de_acesso:
      membro1;
    especificador_de_acesso:
      membro2;
    ...
} nomes_dos_objetos;
```

- Note a presença dos *especificadores de acesso*, palavras-chave que controlam o acesso aos membros por entidades de fora da classe.

Classes

Especificadores de acesso

- São três os especificadores de acesso que podemos usar:
  - *private*: membros privados são acessíveis apenas por membros da mesma classe ou por seus *amigos*.
  - *protected*: membros protegidos são acessíveis por membros da mesma classe, por seus amigos ou por membros de classes *derivadas*.
  - *public*: membros públicos são acessíveis de qualquer lugar do código em que o objeto seja visível.
- Quando não especificado, o acesso default dos membros é o privado.

Classes

Exemplo

- Observe o seguinte exemplo:

```
class Rectangle
{
      int x,y;
   public:
      void set_values (int,int);
      int area (void);
} rect;
```

- Declaramos uma classe chamada Rectangle, com dois membros privados (x e y) e dois membros públicos (set_values e area), e criamos um objeto (rect) dessa classe.
- O processo de criar um objeto (variável) de uma classe (tipo) chama-se *instanciar* a classe.

Classes - exemplo 1

```
// classes example

#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area() {return width*height;}
};

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

int main () {

  Rectangle rect1, rect2;

  rect1.set_values (3,4);
  rect2.set_values (5,8);

  cout << "area 1: " << rect1.area() << endl;
  cout << "area 2: " << rect2.area() << endl;

}
```

Classes

Operador de escopo

- No exemplo anterior, ao definirmos a função membro set_values fora da classe a qual ela pertencia (Rectangle), foi preciso preceder seu nome pelo nome da classe e o operador :: (chamado *operador de escopo*). Ou seja, foi preciso escrever Retangle::set_values no lugar onde estaria normalmente apenas set_values, o nome da função.
- Isso sempre é necessário quando definimos membros de classe fora do corpo da classe em si.

Classes - exemplo 2

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

class temp
{
    private:
        int data1;
        float data2;
    public:
       void int_data(int d)
       {
          data1=d;
          cout<<"Number: "<<data1;
       }
       float float_data()
       {
           cout<<"\nEnter data: ";
           cin>>data2;
           return data2;
       }
};

int main()
{
      temp obj1, obj2;

      obj1.int_data(12);
      cout<<"You entered "<<obj2.float_data();
}
```

Classes - exemplo 2

obj1		obj2	
data1	data2	data1	data2
12	Random value	Random value	12.43

Figura 17: Objetos do exemplo (instâncias da classe temp) e seus membros.

Classes

Construtor

- No nosso primeiro exemplo, criamos uma função chamada set_values que servia para inicializar os membros de um dado objeto.
- Na verdade, o C++ possui uma função especial, o *construtor*, que tem justamente esse objetivo: inicializar os membros de um objeto. O construtor é chamado automaticamente toda vez que um novo objeto de uma classe é declarado.
- O construtor é uma função cujo nome é o mesmo nome da classe, que não retorna nenhum tipo (nem mesmo void) e pode aceitar argumentos normalmente.

Classes - exemplo 3 (construtor)

```
// example: class constructor

#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle (int,int);
    int area () {return (width*height);}
};

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {

  Rectangle rect1 (3,4);
  Rectangle rect2 (5,8);

  cout << "area 1: " << rect1.area() << endl;
  cout << "area 2: " << rect2.area() << endl;
  return 0;
}
```

Classes

Construtor

- Quando definimos um construtor para uma classe, todos os objetos dessa classe têm que ser declarados de acordo com a sintaxe do construtor (ou seja, com o mesmo número de argumentos especificado no construtor).
- Se não definimos um construtor para uma classe, o compilador assume que a classe possui um construtor padrão (sem argumentos). Sendo assim, podemos declarar os objetos dessa classe da mesma maneira que definimos variáveis comuns (sem argumentos).
- Também podemos nos prevenir para o caso do objeto ser declarado sem argumentos, mesmo quando seu construtor os exige: definimos para isso, no parênteses do próprio construtor, valores-padrão para as variáveis a serem inicializadas.

Classes - exemplo 4

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

class Box
{
   public:
      // Constructor definition
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
      }
      double Volume()
      {
         return length * breadth * height;
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
```

Classes - exemplo 4 (cont)

```
int main(void)
{
    Box Box1(3.3, 1.2, 1.5);     // Declare box1
    Box Box2(8.5, 6.0, 2.0);     // Declare box2
    Box Box3;                    // Declare box3

    Box *ptrBox;                 // Declare pointer to a class.

    // Save the address of first object
    ptrBox = &Box1;
    // Now try to access a member using member access operator
    cout << "Volume of Box1: " << ptrBox->Volume() << endl;

    // Save the address of second object
    ptrBox = &Box2;
    // Now try to access a member using member access operator
    cout << "Volume of Box2: " << ptrBox->Volume() << endl;

    // Try to access directly a member using member access operator
    cout << "Volume of Box3: " << Box3.Volume() << endl;

    return 0;
}
```

Classes

Inicialização de membros no construtor

- Quando um construtor é usado para inicializar membros, estes podem ser inicializados sem a necessidade de escrevermos comandos no corpo do construtor. Isso é feito incluindo-se dois pontos, :, antes do corpo do construtor, seguidos de uma lista das inicializações pretendidas. Por exemplo, o construtor:

  Rectangle::Rectangle (int x, int y) { width=x; height=y; }

  poderia ser escrito como:

  Rectangle::Rectangle (int x, int y) : width(x), height(y) { }
- Note como o corpo do construtor está vazio.

Classes - exemplo 5

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

class Circle {
    double radius;
  public:
    Circle(double r) : radius(r) { }
    double area() {return radius*radius*3.14159265;}
};

class Cylinder {
    Circle base;
    double height;
  public:
    Cylinder(double r, double h) : base (r), height(h) {}
    double volume() {return base.area() * height;}
};

int main () {
  Cylinder foo (10,20);

  cout << "foo's volume: " << foo.volume() << '\n';
  return 0;
}
```

Classes

Filosofia da orientação a objetos

- Depois de vários exemplos, podemos finalmente entender a filosofia da orientação a objetos: *dados (variáveis) e métodos que lidam com essas variáveis (funções) são encapsulados em uma só entidade chamada classe*.
- Na OO, não mais criamos conjuntos de variáveis globais que passamos de uma função para outra como parâmetros, mas sim *manipulamos objetos que têm seu próprio conjunto de dados e funções*.
- Esse conceito cria poderosas ferramentas de abstração, em especial a *composição*, a *herança* e o *polimorfismo*.

Classes

Composição

- A relação entre classes conhecida como *composição*, ou relação *has-a*, ocorre quando uma instância de uma dada classe é membro de outra.
- Já vimos isso no nosso último exemplo: uma instância (objeto) da classe Circle era membro da classe Cylinder.

Amizade

Classes e funções também amam

- A princípio, membros privados ou protegidos de uma classe não podem ser acessados de fora da classe. Essa regra, no entanto, não se aplica aos *amigos*.
- Amigos são funções ou classes declaradas com a palavra-chave *friend*.
- Muito cuidado com o uso da amizade, pois ela quebra a ideia básica da orientação a objetos: o encapsulamento de dados!
- Via de regra, tente programar com o menor número de amigos possíveis. Se possível, não faça nenhum amigo.

Função amiga - exemplo

```
// friend functions
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param)
{
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}

int main () {
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

Classe amiga - exemplo

```
// friend class
#include <iostream>
using namespace std;

class Square;

class Rectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (Square a);
};

class Square {
  friend class Rectangle;
  private:
    int side;
  public:
    Square (int a) : side(a) {}
};

void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}
```

Classe amiga - exemplo (cont)

```
int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

Herança

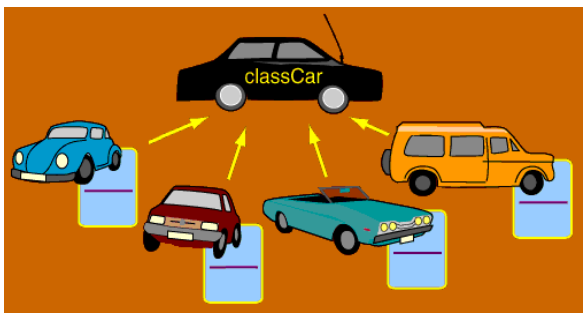

Figura 18: Ilustração do mecanismo da herança.

Derivando classes a partir de classes

- Na OO, classes novas podem ser criadas a partir de antigas, aproveitando todo o código nesta implementado e adicionando novas funcionalidades. Esse mecanismo é conhecido como *herança*, ou relação *is-a*.
- Classes representando modelos genéricos, portanto, podem ser implementadas, e delas serem derivados modelos específicos.
- Note a possibilidade de alto reaproveitamento de código já implementado!

Herança

Derivando classes a partir de classes

- A sintaxe para a declaração de classes derivadas é:

  class derived_class_name: public base_class_name
  { /*...*/ };

- O especificador de acesso limita o nível mais acessível para membros derivados; membros com nível mais acessível do que o especificado são herdados com o nível especificado.
- Via de regra, quando não desejamos modificar os níveis de acesso, usamos o especificador public.

Herança - exemplo

```
// derived classes
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
 };

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };
```

Herança - exemplo (cont.)

```
int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

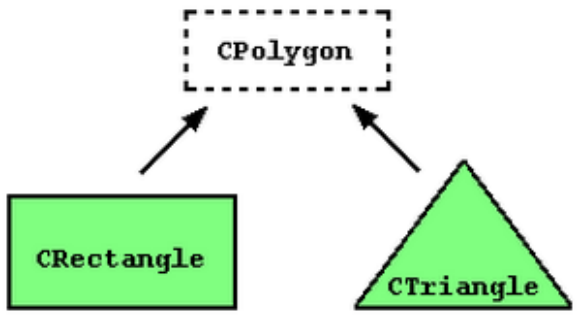

Figura 19: Mecanismo de herança no exemplo anterior.

Polimorfismo

Várias formas

- Uma das principais características da herança entre classes é que *um ponteiro para uma classe derivada é compatível em tipo com um ponteiro para a classe mãe*.
- O polimorfismo é a arte de tirar vantagem dessa situação.
- Com o polimorfismo, podemos programar utilizando ponteiros que apontam para classes base genéricas, que são substituídas, a nível apropriado, por classes derivadas específicas.

Figura 20: Ilustração do mecanismo do polimorfismo.

Polimorfismo - exemplo

```
// pointers to base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};

class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    int area()
      { return width*height/2; }
};
```

Polimorfismo - exemplo (cont.)

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

Polimorfismo

Membros virtuais

- Um *membro virtual* é uma função membro que pode ser redefinida nas classes derivadas, preservando as propriedades polimórficas (chamada da função através de ponteiros).
- Uma função virtual tem sua declaração precedida pela palavra-chave *virtual*.
- É importante ter em mente que funções membro não virtuais também podem ser redefinidas nas classes derivadas, porém não podem ser mais acessadas através de referências para a classe-mãe (ou seja, perdem suas propriedades polimórficas).
- Uma classe que declara ou herda uma função virtual é chamada de *classe polimórfica*.

Membros virtuais - exemplo

```
// virtual members
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return 0; }
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return (width * height / 2); }
};
```

Membros virtuais - exemplo (cont.)

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}
```

Polimorfismo

Classes abstratas

- *Classes abstratas* são classes que contém pelo menos uma *função puramente virtual*.
- *Uma função puramente virtual é uma função sem implementação*. A sintaxe para sua declaração é:

  virtual tipo nome () = 0;

  As implementações para as funções puramente virtuais vêm apenas nas classes derivadas.
- Classes abstratas não podem ser instanciadas. Não há objetos cujos tipos sejam classes abstratas. Elas apenas fornecem a base a partir da qual outras classes serão derivadas.
- As classes abstratas, apesar de não poderem ser instanciadas, não são inúteis! Podemos programar fazendo referências a elas, tirando proveito de todo o poder do polimorfismo.

Classes abstratas - exemplo

```
// abstract base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
};

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

class Triangle: public Polygon {
  public:
    int area (void)
      { return (width * height / 2); }
};
```

Classes abstratas - exemplo (cont.)

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  return 0;
}
```

Classes abstratas - exemplo 2

```
// pure virtual members can be called
// from the abstract base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area() =0;
    void printarea()
      { cout << this->area() << '\n'; } /* this: keyword representing a pointer
           to the object whose member function is being executed */
};

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

class Triangle: public Polygon {
  public:
    int area (void)
      { return (width * height / 2); }
};
```

Classes abstratas - exemplo 2 (cont.)

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  return 0;
}
```

Exemplo aplicado à engenharia química

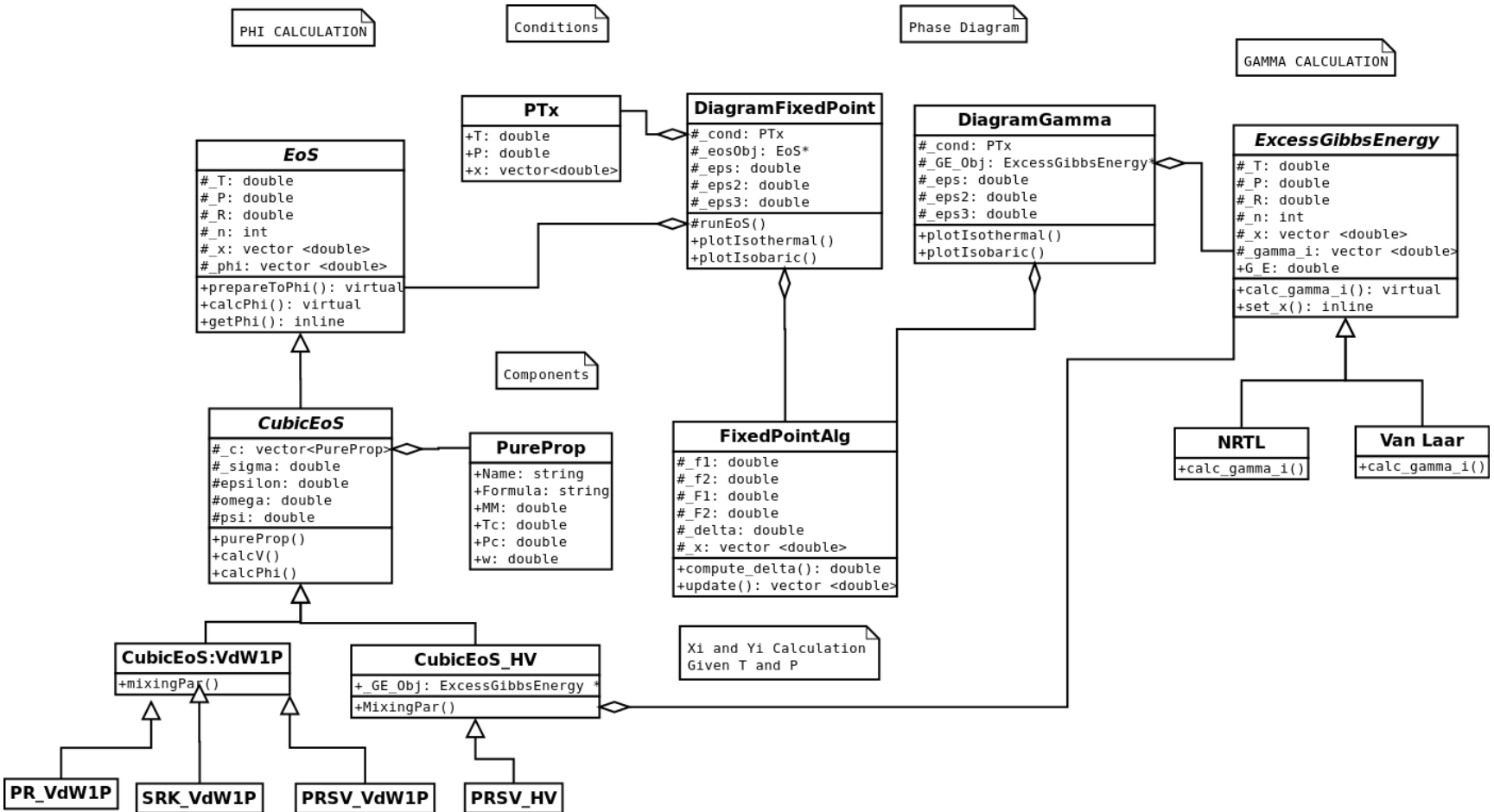

Figura 21: Diagrama UML de um programa básico para cálculo de equilíbrio de fases.

COPPE UFRJ

Bibliografia

- Stephen G. Kochan – Programming in C – Sams Pulishing;
- Herbert Schildt – C Completo e Total – Makron Books;
- Bartotz Milewski – C++ In Action – Industrial Strength Programming Techniques – RO Release;
- Daoqi Yang - C++ and Object Oriented Numeric Computing for Scientists and Engineers – Springer;
- Julian Soulié - C++ Language Tutorial - Disponível em www.cplusplus.com;
- www.cprogramming.com

Obrigado!