Pontifícia Universidade Católica do Rio de Janeiro
Departamento de Informática - Coordenação Central de Extensão

APOSTILA de L3GO

Linguagem de 3ª Geração Orientada a Objetos - C++

Aulas Teóricas

Adelailson Peixoto

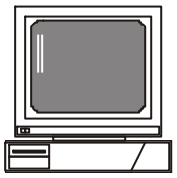
1-INTRODUÇÃO À LINGUAGEM C

1.1-INTRODUÇÃO

Linguagens de Programação:

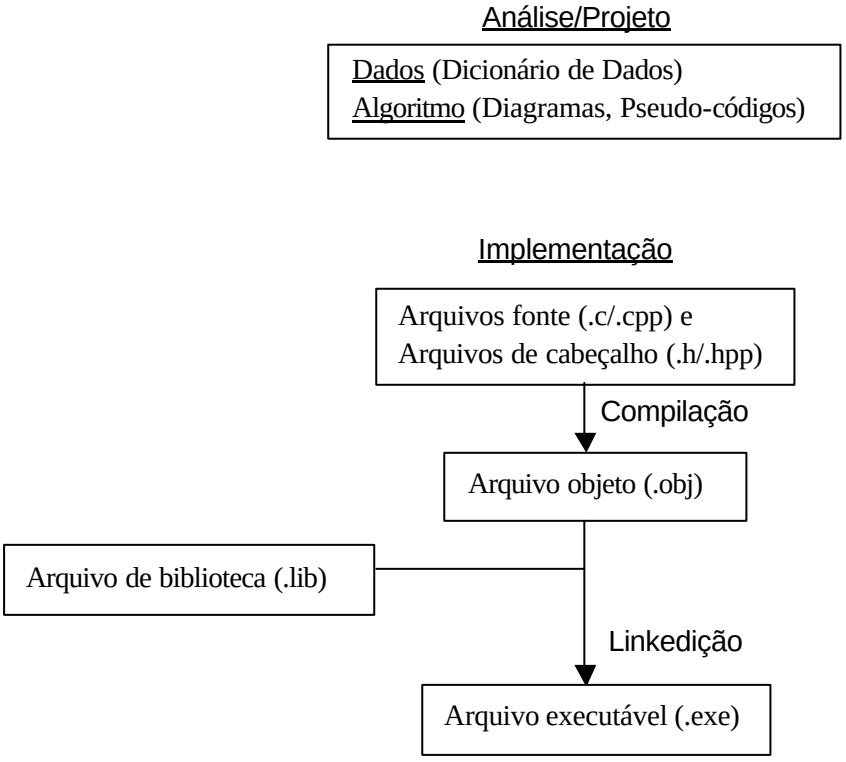
Alto Nível. Ex.: Pascal, COBOL, FORTRAN, BASIC, etc.
Médio Nível. Ex.: C/C++
Baixo Nível. Ex.: Assembler

C é uma linguagem de médio nível, pois combina elementos de linguagens de alto nível com a funcionalidade de linguagens de baixo nível.



Máquina (Baixo Nível) ← Programador (Alto Nível)

1.2-PROCESSO DE CONSTRUÇÃO DE UM PROGRAMA



- Os arquivos fonte e de cabeçalho são arquivos texto e são desenvolvidos pelo programador de acordo com o conjunto de regras (sintaxe) da linguagem de programação.
- Durante a compilação, o *compilador* verifica a sintaxe dos arquivos texto. Se a sintaxe estiver correta, o compilador gera um arquivo objeto para cada arquivo fonte.
- Arquivos objeto são arquivos binários, que contêm as informações dos arquivos texto. Contêm também informações de alocação de memória (alocação estática). Só são gerados se o compilador não encontrar erros nos arquivos texto correspondentes.
- Arquivos de biblioteca são arquivos binários gerados a partir de arquivos objeto Os arquivos de biblioteca contêm assim códigos já compilados que podem ser posteriormente utilizados em programas executáveis, evitando reimplementação de código.
- Na linkedição, as informações de todos os arquivos binários (.obj e .lib) são juntas para gerar um único arquivo binário que contenha todo o código do programa. O arquivo binário gerado é o executável. Na linkedição o código de cada função é associado ao arquivo objeto para gerar o executável.
- Ao iniciar um novo código, o programador deve especificar se se trata de uma biblioteca (.lib) ou de uma aplicação (programa executável - .exe).

1.3- SÍMBOLOS

Palavras Reservadas – são palavras que compõem a Linguagem de Programação e já possuem uso específico. Ex.: for, while, integer, etc.

Literais – conjunto de valores que podem ser utilizados. Ex.:
números inteiros: 0, 1, 300;
números reais: 1.42, 3.68, 0.01, 1.00.
caracteres: ‘a’, ‘1’, ‘\n’.

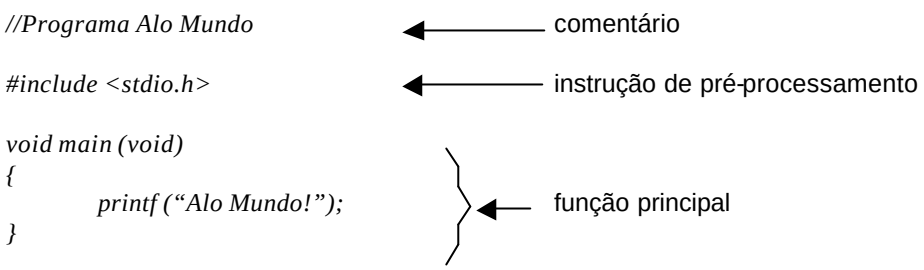
Separadores – são símbolos que separam outros símbolos. Ex.: caracter branco (‘ ’), tabulação (‘\t’), pula linha(‘\n’), etc.

Operadores – são símbolos que especificam operações entre outros símbolos. Ex.:
operadores aritméticos: soma (+), vezes (*), menos (-), divisão (/)
operadores relacionais: maior do que (>), igual (==), diferente (!=).
operadores lógicos: e (&&), ou (| |), negação (!).

Identificadores – palavras criadas pelo programador p/ denotar objetos. Para tal, é permitido o uso de letras, sublinhados e dígitos. Ex.: Maria, x, x10, x_10.

1.4 – ORGANIZAÇÃO BÁSICA DE UM PROGRAMA

Exemplo de um programa em C:



Um programa em C possui a seguinte organização básica:

- Comentários - podem ser usados para identificação do programa.
- Instruções de Pré-processamento. operações realizadas em tempo de compilação.
- Declarações Globais - nomes globais podem ser nomes de variáveis e funções. São visíveis a partir de sua declaração, até o fim do arquivo onde estão declarados.
- Definição da Função Principal - a função *main* é obrigatória para a geração de aplicações. A execução do programa começa sempre pela *main*. A definição da *main* é auto-declarada.
- Definição das Demais Funções – uma função pode ser definida antes ou depois da *main*. Funções definidas antes da main são auto-declaradas. As funções definidas após a *main* e as funções pré-definidas em biblioteca devem ser declaradas anteriormente.

Exemplos.:

```
//Função definida ANTES da main
#include <stdio.h>

void ImprimeNumero (int x) //auto-declaracao
{
        printf ("%d", x);
}
void main (void)
{       int x;

        scanf("%d",&x);
        ImprimeNumero (x);
}
```

```
//Função definida APOS a main
#include <stdio.h>

void ImprimeNumero (int x); //declaracao
void main (void)
{       int x;
       scanf("%d",&x)
        ImprimeNumero (x);
}
void ImprimeNumero (int x)//definicao
{
        printf ("%d", x);
}
```

2 – ELEMENTOS BÁSICOS DE UM PROGRAMA

2.1 – ESCOPO DE NOMES

Escopo é o local do programa onde um nome é visível e pode ser utilizado. O escopo começa na declaração do nome e termina no fim da região do programa onde o nome foi declarado.

Nome Local – é declarado dentro de um bloco (comando estruturado ou função). O nome somente pode ser referenciado de dentro do bloco.

Nome Global – é declarado fora das funções. A referência ao nome começa na sua declaração e termina no final do arquivo.

2.2 – TIPOS SIMPLES DE DADOS

char – caracter (8 bits).
int – inteiro (o tamanho pode variar de acordo com a máquina. Ex.:16 bits).
float – número real com precisão simples (idem. Ex.:32 bits).
double – número real com precisão dupla (idem. Ex.:64 bits).
void – conjunto vazio.

2.3 – VARIÁVEIS

São nomes (identificadores) que indicam regiões na memória, reservadas para armazenar um determinado tipo.

Declaração de uma variável: <tipo> <nomeVar> [=<valor>];

<tipo> é o nome de um tipo de dados,
<nomeVar> é o nome ou identificador da variável,
= é o operador de atribuição e
<valor> é um valor que pode ser atribuído à variável, durante a declaração.

Ex.:
```
int x;
char maiúscula, minúscula = 'a';
float a = 4.8;
int soma = 0, dif = -10;
```

2.4 – OPERADORES

- Alguns operadores são símbolos (*operadores gráficos*), outros são palavras (*operadores verbosos*).
- Quanto ao número de operando, os operadores podem ser:

Unários	**Binários**	**Ternários**
x++; !x; *x; sizeof tabela	x/y; a && b; x=z;	x.y?a:b

- Quanto à posição do operando em relação ao operador.

Pré-fixados	**Pós-fixados**	**In-fixados**

++x;
!x;
*x;
new int;
delete x;
	x++;
	x();
	x--;
		x+y;
		a*b;
		x == y;
		x=y;
		b-=10;
		a+=x;

Alguns operadores são sobrecarregados, ou seja, possuem mais de uma finalidade. Ex.:
a*b - * é o operador aritmético de multiplicação.
int *x - * é o operador especificando que x é um ponteiro.
*x - * é o operador que o acessa o valor contido na posição p/ onde x aponta.

Existem dois aspectos importantes sobre o uso de operadores:

- Prioridade (precedência da execução)
  Ex.: x=y++; equivale a x = y; y = y+1;
  x=++y; equivale a y = y+1; x = y;

- Associatividade (sentido da execução)
  Ex.: x=y=z; é executado da direita para a esquerda (←).
  x==y==z; é executado da esquerda para a direita (→).

3 – FUNÇÕES DE E/S DA BIBLIOTECA PADRÃO

3.1 – LEITURA E ESCRITA DE CARACTERES

-int getchar (void);
Retorna o caracter lido do teclado, como um valor inteiro. A tecla pressionada é automaticamente mostrada na tela. O caracter só é armazenado depois que a tecla ENTER é pressionada.

-int getch (void);
Retorna o caracter lido do teclado. Não é necessário apertar a tecla ENTER. O caracter não é exibido na tela.

-int getche (void);
Retorna o caracter lido do teclado. Não é necessário apertar a tecla ENTER. O caracter é exibido na tela.

- int putchar (int c);
Escreve um caracter na tela. Retorna ou o caracter escrito ou EOF, se ocorrer erro.

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

void main (void)
{
   int ch;

   ch = getchar();
   if(islower(ch))
         ch = toupper(ch);
  putchar(ch);
}
```

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>

void main (void)
{
   int ch;

   ch = getch();
   if(islower(ch))
         ch = toupper(ch);
  putchar(ch);
}
```

3.2 – LEITURA E ESCRITA DE STRINGS (CADEIAS DE CARACTERES)

*-char * gets (char *str);*
Lê uma string a partir do teclado e a armazena em *str*. A string só é lido quando a tecla ENTER é pressionada.

*-int puts (char *str);*
Escreve a string *str* na tela. Retorna EOF, se ocorrer erro.

*-int printf (char *str,lista_de_arqgumentos);*
Imprime uma string formatada na tela. Retorna o número de caracteres escritos ou um valor negativo, se ocorrer erro. A string impressa pode ser composta por valores de vários tipos de dados.

*-int scanf (char *str,lista_de_argumentos);*
Lê uma string formatada da tela. Retorna o número de caracteres lidos ou um valor negativo, se ocorrer erro. A string lida pode ser composta por valores de vários tipos de dados.

Exemplo *gets*/*puts*.

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

void main (void)
{
   char str[80];
   gets(str);
   puts("A string lida foi:");
   puts(str);
}
```

Exemplo *scanf*/*printf*

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

void main (void)
{   int x;
    char str[80];
    float y;
    scanf("%d,%f\n",&x,&y);
    scanf("%s\n",str);
   printf("Os valores lidos foram %d,%f e %s",x,y,str);
}
```

Especificadores de Formato:

caracter - **%c**	inteiro decimal - **%d** ou **%i**	ponto flutuante - **%f** ou **%e** ou **%g**
inteiro octal - **%o**	string - **%s**	inteiro hexadecimal - **%x**
ponteiro - **%p**	inteiro sem sinal - **%u**	

4 – ESTRUTURAS DE CONTROLE E EXECUÇÃO

4.1 – ESTRUTURAS DE SELEÇÃO OU CONDICIONAIS

- if (<expressão>)
  <bloco1>;
  [else
  <bloco2>;
  ]

<expressao> deve retornar um valor verdadeiro (diferente de 0) ou um valor falso (igual a 0). Caso o valor seja verdadeiro, apenas o bloco1 será executado. Caso o valor retornado seja falso, apenas o bloco2 será executado (se a estrutura contiver o else). Exemplos:

```
//Programa if simples

#include <stdio.h>

void main (void)
{
  int x;
  scanf ("%d", &x);
  if ((x%2)==0)
     printf ("%d e par", x);
}
```

```
//Programa if composto

#include <stdio.h>

void main (void)
{
  int x;
  scanf ("%d", &x);
  if ((x%2)==0)
     printf ("%d e par", x);
  else printf ("%d e impar", x);
}
```

Obs.: Se <bloco1> ou <bloco2> for composto por mais de um comando, deve estar entre chaves. No exemplo abaixo, como há três comandos condicionados ao *if*, então devem estar entre chaves.

```
...
 if ((x%2)==0)
 {
       printf ("%d e par", x);   //comando 1
      x = x+10;                  //comando 2
     y+=x;                       //comando 3
 }
...
```

- switch (<expressão>)
  {
  case <constante1>:
  <bloco1>;
  case <constante2>:
  <bloco2>;
  . . .
  [default:
  <bloco_default>;]
  }

- o switch testa a igualdade entre a expressão e as constantes.
- quando expressão é igual a constante de um bloco, todos os blocos, a partir deste, são executados.
- se <expressão> não for igual a nenhuma constante, o bloco default é executado.
- o default é opcional.

Outra forma de usar o *switch*, é incluindo o comando *break*, dentro de cada *case*:

```
switch (<expressão>)
    {
    case <constante1>:
        <bloco1>;
        break;
    case <constante2>:
        <bloco2>;
        break;
        . . .
    [default:
        <bloco_default>;]
    }
```

- O uso do *break* faz com que só o bloco do *case*, onde a expressão é igual a constante seja executado.
- Isto por que quando o break é executado o comando switch deixa de ser executado.

Exemplo:

```
//Programa switch

#include <stdio.h>
#define BRASIL    0
#define EUA       1
#define FRANCA 2

void main (void)
{   int x;
   scanf ("%d",&x);
   switch(x)
   {
        case BRASIL:
          printf ("Brasil\n");
         break;
        case EUA:
           printf ("Estados Unidos\n");
           break;
        case FRANCA:
           printf ("Franca\n");
           break;
        default:
          printf ("Pais Invalido\n");

   } //fim do switch
}//fim da  main
```

4.2 – ESTRUTURAS DE REPETIÇÃO OU ITERAÇÃO

- while (<expressão>)
  {
  <bloco>;
  }

- expressão retorna um valor que pode ser verdadeiro (diferente de 0) ou falso (igual a 0).
- enquanto a expressão for verdadeira o bloco será executado.
- a execução só deixa o *while* quando expressão for falsa.
- é possível que o bloco não seja executado nenhuma vez.

- do {
  <bloco>;
  }
  while (<expressão>);

- expressao retorna um valor que pode ser verdadeiro (diferente de 0) ou falso (igual a 0).
- enquanto a expressao for verdadeira o bloco será executado.
- a execução só sai do laço quando expressao for falsa.
- O bloco é executado pelo menos uma vez.

Nas duas estruturas acima, a condição de saída do laço (ou seja a expressão atingir o valor falso) deve ser controlada dentro do bloco

- for ([expre1]; [expre2]; [expre3])
  {
  <bloco>;
  }

expre1 - faz todas as iniciações do laço *for*.
expre2 - é a condição testada. Se for verdadeira o bloco continua a ser executado. Se for falsa a execução sai do laço.
expre3 – é usada para atualizar dados, de modo que a condição de saída da expre2 seja atingida.

Obs.: O comando *break* também pode ser utilizado dentro das 3 estruturas acima (além do switch), como mais uma condição de saída do laço.

Ex.: while

```
void main (void)
{
  int soma=0, i=0,lim_sup;
  scanf ("%d", &lim_sup);
  while (i<10){
    soma+=i;
     i++;
    if(soma>lim_sup) break;
  }
}
```

Ex.: do while

```
void main (void)
{
  int soma=0, i=0,lim_sup;
  scanf ("%d", &lim_sup);
  do{
     soma+=i;
     i++;
     if(soma>lim_sup) break;
  } while (i<10)
}
```

Ex.: for

```
void main (void)
{
  int soma=0, i,lim_sup;
  scanf ("%d", &lim_sup);
 for(i=0;i<10; i++){
     soma+=i;
     if(soma>lim_sup) break;
  }
}
```

5 – MODULARIZAÇÃO EM SUBPROGRAMAS

5.1 – FUNÇÕES

A forma geral de uma função é:
tipo-retorno nome-funcao (lista-de-parametros)
{
corpo da função
}

tipo-retorno é o tipo do valor que o comando *return* devolve (caso não seja *void*).
nome-funcao é o identificador da função.
lista-de-parametros é o conjunto de parâmetros passados à função. Cada parâmetro é separado por vírgula e seu tipo deve ser especificado.
Exs.:

```
void Par_ou_Impa (int x)
{
 if ((x%2)==0) printf ("numero %d é par", x);
 else     printf ("numero %d é impar", x);
}
```

```
int LerMaior (int x, int y)
{
 if ( x>y) return x;
 else     return y;
}
```

Obs.: -Se o tipo-retorno não for especificado, o compilador assume que será retornado um inteiro. Ex.; A declaração *int função(void);* equivale à declaração *função(void);*

-Se tipo-retorno for void, a função não retorna nada (procedimento).
-Para indicar que a função não recebe parâmetros, deve-se passar apenas o void ou não passar nada. Ex.: A declaração *int nome_função ();* equivale a *int função (void);*

5.2 – DECLARAÇÃO, DEFINIÇÃO E CHAMADAS DE FUNÇÕES

Declaração – durante a declaração de uma função, são resolvidos os endereços de seus parâmetros e de seu valor de retorno. Uma função só é reconhecida, pelo compilador, após sua declaração. Os parâmetros definidos são chamados de *Parâmetros Formais*.
Definição - uma função só é definida (implementada) após sua declaração. Se no momento de sua definição a função ainda não foi declarada, então ela é declarada na própria definição (auto-declarada).
Chamada – pode ser feita em qualquer lugar do programa, desde que a função tenha sido declarada. Os parâmetros passados a uma função, durante sua chamada, são chamados de *Parâmetros Reais*.

Obs.: Os parâmetros formais e as variáveis declaradas durante a definição da função são variáveis locais. Assim seu escopo está definido apenas dentro da função.

5.3 – PASSAGEM DE PARÂMETROS

Em C, só há passagem de parâmetros por valor. Quando há chamada de função, os parâmetros reais são copiados para os parâmetros formais.

No exemplo 1, abaixo, a variável *x* é global. A função *ImprimeAteDobro*, chamada de dentro da *main*, altera o valor de *x*. Quando a função *ImprimeAteTriplo* é chamada de dentro da *main*, ela acessa a variável *x* com o valor já alterado (*x* foi alterada em *ImprimeAteDobro*). No exemplo 2, não há a varável global *x*. Em vez disso, as funções *ImprimeAteDobro* e *ImprimeAteTriplo* possuem um parâmetro formal, que receberá o valor real, durante sua chamada de dentro da *main*.

Exemplo 1:

```
// Programa Imprimir Dobro/Triplo

#include <stdio.h>
#include <conio.h>

void ImprimeAteDobro(void);
void ImprimeAteTriplo(void);
int x;

void main( void )
{
   clrscr( );
   printf("Entre com um inteiro:\n" );
   scanf( "%d\n", &x);
   ImprimeAteDobro();
   ImprimeAteTriplo();
}
void ImprimeAteDobro(void)
{
   int i=2*x;
   for ( ; x<=i; x++)
         printf("%d\n",x );
}
void ImprimeAteTriplo(void)
{
   int i=3*x;
   for ( ; x<=i; x++)
         printf("%d\n",x );
}
```

Exemplo 2:

```
// Programa Imprimir Dobro/Triplo

#include <stdio.h>
#include <conio.h>

void ImprimeAteDobro (int x);
void ImprimeAteTriplo (int x);

void main( void )
{
   int x;
   clrscr( );
   printf("Entre com um inteiro:\n" );
   scanf( "%d\n", &x);
   ImprimeAteDobro (x);
   ImprimeAteTriplo (x);
}
void ImprimeAteDobro (int x)
{
   int i=2*x;
   for ( ;x<=i; x++)
         printf("%d\n",x );
}
void ImprimeAteTriplo (int x)
{
   int i=3*x;
   for ( ; x<=i; x++)
         printf("%d\n",x );
}
```

5.4 – INSTRUÇÕES DE PRÉ-PROCESSAMENTO

São realizadas em tempo de compilação. Exemplos de instruções:

#include - inclui o código de um arquivo no programa fonte.
#define - define nomes no programa, que podem ser *constantes simbólicas* ou *macros*.

Constantes Simbólicas – são nomes que simbolizam valor.

Obs.1: pode-se definir uma variável constante no programa, usando *const*, antes de sua declaração. Ex.: *const int x=10*. A variável *x* só será usada para leitura e seu valor será sempre 10.

Obs.2: quando se uma constante com #define não há veificação de tipo.

Macros – são nomes que simbolizam ação.

Obs.1: a definição de uma macro admite passagem de parâmetros.

Obs.2: para garantir prioridades, deve-se usar parênteses.

```
//Programa Constantes Simbolicas

#include <stdio.h>
#define MAX 10
#define NOVA_LINHA '\n'

void main (void)
{
  int i, soma =0;

  for (i=0;i< MAX; i++)
   printf ("%d NOVA_LINHA", i);
}
```

```
//Programa Macro

#include <stdio.h>
#define SOMA (a,b) a+b

void main (void)
{
   int x;

  x= SOMA(2,3);
  printf ("%d \n",x);
}
```

6 – VETORES

Vetor - coleção de variáveis de mesmo tipo que é referenciada por um nome comum. Um elemento específico em um vetor é acessado através de um índice.

6.1 – PONTEIROS

Um ponteiro é uma variável que armazena endereços de outras variáveis. Um ponteiro tem normalmente o tamanho de uma variável inteira.

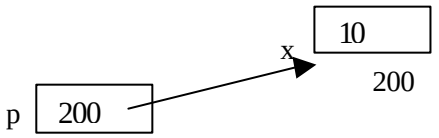
Declaração: <tipo> <*nome_ponteiro>;

Exemplo: *float x,*p*;
-*x* é uma variável que armazena um *float.*
- *p* é uma variável que armazena endereços de variáveis *float.*

Operadores * e &

O operador * tem dois significados:
-Na declaração de uma variável indica que a variável é um ponteiro (Ex.: float *p;)
-Após a declaração indica o acesso à variável para a qual o ponteiro aponta (indireção). Exemplo: printf("%d", *p);
O operador & retorna o endereço de uma variável.

```
float x=10,*p;
p=&x;
printf("%d",  *p);
printf("%d",  p);
```



No exemplo acima, *p=&x* faz com que *p* receba o endereço de *x* (aponte para *x*), ou seja o valor de p passa a ser 200, que corresponde à posição de memória 200, onde se encontra a variável *x*. No trecho *printf("%d", *p),* *p indica o valor da variável para a qual *p* aponta (variável *x*) e será impresso o valor 10. Na segunda impressão, *printf("%d", p),* será impresso o valor de *p* (endereço de *x*), ou seja 200.

6.2 – DECLARAÇÃO DE VETORES

<nome_tipo> <nome_var> [[num_elementos]] [= <lista-valores>];

-O nome da variável representa o endereço do primeiro elemento.
-O nome da variável é declarado internamente como um ponteiro constante.
-O nome da variável não sofre operação de incremento ou decremento.

Exemplo : *float x[6];*

-O endereço de cada elemento é seqüencial
-Somar i a x equivale a obter o endereço do elemento da posição i.
-A notação *(x+i) equivale a x[i].

6.3 – PASSAGEM POR REFERÊNCIA E VETOR COMO PASSAGEM DE PARÂMETRO

A passagem de parâmetros por referência é feita através do uso de ponteiros. O exemplo abaixo mostra uma função CalculaDobro que recebe uma variável x (por valor) e uma variável dobro (por referência) que retornará com o dobro de x. A função principal mostra uma chamada desta função com os seus respectivos parâmetros reais

```
//Programa calcula dobro por referencia
#include <stdio.h>

void  CalculaDobro (int x, int *dobro);

void main (void)
{
        int a,b;
        scanf("%d",&a);
        CalculaDobro(a, &b);
        printf(" O dobro de %d = %d", a,b);
}

void  CalculaDobro (int x, int *dobro)
{
  * dobro= 2*x;
}
```

Na passagem de um parâmetro por referência deve-se observar que:
-O parâmetro formal é um ponteiro (que apontará para a variável que se deseja alterar). Por exemplo, a variável *dobro* passada na declaração da função *CalculaDobro.*
-Na implementação da função o acesso ao parâmetro formal deve utilizar indireção (*) para se alterar a variável apontada, de fato. Exemplo: **dobro=2*x;*
-Na chamada à função, deve-se passar como parâmetro real o endereço da variável que se deseja alterar. No exemplo *CalculaDobro(a, &b)* é passado o endereço de *b (&b).*

Vetor como Passagem de Parâmetros

Não se pode passar um vetor completo como parâmetro real em uma chamada de função. O que é passado é apenas o endereço do vetor (ponteiro para o primeiro elemento).

As três declarações abaixo são equivalentes:
a) *void funcao (int *x);* */*ponteiro para inteiro */*
b) *void função(int x[10]);* */* vetor dimensionado */*
b) *void função(int x[ ]);* */* vetor adimensional */*

As três declarações produzem resultados idênticos durante a chamada da função: recebem um ponteiro como parâmetro.

É importante observar que a passagem de vetores como parâmetros descrita acima, permite que os elementos do vetor sejam alterados de dentro da função (passagem por referência). Para evitar esta possibilidade deve-se utilizar o modificador *const* antes de parâmetro formal (vetor). O uso do *const* permite que o vetor seja apenas lido de dentro da função (passagem por valor).

A alocação de um vetor em memória é resolvida estaticamente (em tempo de compilação).
Ex.:

```
int MAX=12;
float y[MAX];       /*Errado, pois MAX não é conhecido em  tempo de compilação*/

#define MAX 12
float y[MAX];    /*Está Correto */
```

6.4 - STRINGS

Uma string é um vetor de caracteres.
Exemplo: *char str[10]; /*string de 10 elementos */*

Algumas funções declaradas em *string.h* são:

-strcpy
*char *strcpy (char *str_dest, const char *str_orig);*

Copia o conteúdo de str_orig para str_dest. A string str_orig deve ser terminada com '\0'. Retorna um ponteiro para str_dest.

-strcat

*char *strcat (char *str_dest, const char *str_orig);*

Concatena uma cópia de str_orig para str_dest e finaliza str_dest com o caracter nulo '\0'. O caracter nulo, que originalmente finalizava str_dest, é sobreposto pelo primeiro caracter de str_orig. Retorna um ponteiro para str_dest.

-strcmp

*int strcmp (const char *str1, const char *str2);*

Compara lexicograficamente as strings str1 e str2 r devolve um inteiro baseado no resultado:

Valor	significado
<0	str1 é menor do que str2
=0	str1 é igual a str2
>0	str1 é maior do que str2

-strlen

*size_t strlen (const char *str);*

Devolve o comprimento (número de caracteres) de str, terminada pelo caracter nulo '\0'. O nulo não é contado.

Exemplo:

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

void main (void)
{
        char s1[80], s2[80];

        gets (s1);  /* ler uma string da tela e armazena em s1*/
        gets (s2);  /* ler uma string da tela e armazena em s2*/

        printf("Comprimentos: %d %d\n",strlen(s1), strlen(s2));

        if (!strcmp(s1,s2))
                printf ("As strings sao iguais\n");

        strcat (s1,s2);
        printf ("%s\n",s1);

        strcpy(s1,"Isto é um teste.\n");
        printf ("%s\n", s1);
}
```

7 – CONSTRUTORES DE TIPOS

7.1 – CRIAÇÃO DE NOMES DE TIPOS

Cria-se um nome de tipo através do comando *typedef.*
typedef <tipo> <nome _novo_tipo>;

Ex.: *typedef float real_ps;*
typedef double real_pd;
typedef int inteiro;

-O nome do tipo possui escopo global quando é criado for a das funções e possui escopo local quando é criado dentro de alguma função. Ex.:

```
typedef int inteiro;                //definicao de tipo com escopo global
float funcao (inteiro x);
void main (void)
{
   typedef float real;              //definicao de tipo com escopo local
   real a;
   inteiro b;
   scanf ("%d", &b);
   a = funcao(b);
}

float função (inteiro x)
{ return (x*0.25);
}
```

É interessante observar que um tipo também pode ser definido com #define. Por exemplo *#define inteiro int*, pode ser usado em vez de *typedef int inteiro*. Porém, é importante lembrar que qualquer símbolo definido com #define possui escopo global.

7.2 ENUMERADOS (enum)

São construtores de tipos (tipo de dados simples), definido pelo usuário, que consiste de uma lista de elementos com nomes constantes.

- O valor do primeiro elemento, por default, é zero.
- O valor de um elemento qualquer é igual ao valor de seu antecessor mais 1.
- O valor de cada elemento é tratado como um inteiro.

Definição do tipo enum

- enum [nome_tipo] {<lista_elementos> [=<valores>]} [nome_var];
- typedef enum {<lista_elementos> [=<valores>]} <nome_tipo>;

Declaração de variáveis

• Se o tipo foi criado SEM typedef: enum <nome_tipo> <nome_var>; (em C) <nome_tipo> <nome_var>; (em C++)	• Se o tipo foi criado COM typedef: <nome_tipo> <nome_var>;

Exemplos:

```
//Program enum

enum {a,b,c} x;  //x é uma variave global
enum t_x {i,j,k}; //t_x é um tipo de dados

t_x  y,z; //y e z sao variaveis gobais do tipo t_x

void main (void)
{
   x=a;
   y=j;
   z = k;
}
```

```
//Program enum

enum t_dia {dom=1,seg,ter,qua,qui,sex,sab};
typedef  enum {inf=-10, med=0, sup=10 } t_lim;

void main (void)
{
   t_dia  dia; //dia é uma variavel do tipo t_dia
  t_lim  lim; //lim é uma variavel do tipo t_lim

   dia = seg;
   lim = sup;
}
```

7.3 ESTRUTURAS (struct)

São construtores que definem tipos e/ou variáveis compostos por um conjunto de variáveis (tipo de dados composto). Para cada elemento do conjunto é alocado um espaço exclusivo na memória. Cada elemento é chamado de campo.

Definição do tipo enum

- struct [nome_tipo] {<lista_elementos>} [nome_var];
- typedef struct {<lista_elementos>} <nome_tipo>;

Declaração de variáveis

- Se o tipo foi criado SEM typedef:
  struct <nome_tipo> <nome_var>; (em C)
  <nome_tipo> <nome_var>; (em C++)

- Se o tipo foi criado COM typedef:
  <nome_tipo> <nome_var>;

Exemplos:

```
//Program struct

#include <stdio.h>

struct { int a, char b, float c} x;  //x é variavel

void main (void)
{
   x.a=10;
   x.b = 'm';
   x.c = 0.56;
   printf ("%d, %c, %f",x.a, x.b, x.c);
}
```

```
//Program struct

struct t_xyz  {float x, int y, int z };
typedef  struct {int l, char m, int n} t_lmn;

void main (void)
{
   t_xyz  xyz; //xyz é uma variavel do tipo  t_xyz
   t_lmn  lmn; //lmn é uma variavel do tipo t_lmn

   xyz.y =10;
   lmn.l = xyz.y;
}
```

7.4 UNIÕES (union)

Tipo e/ou variável definido pelo usuário que pode conter valores de diferentes tipos (tipo composto), em momentos diferentes. Semelhante à struct, exceto que a alocação de memória é feita apenas para o maior campo.

Definição do tipo enum

- union [nome_tipo] {<lista_elementos>} [nome_var];
- typedef union {<lista_elementos>} <nome_tipo>;

Declaração de variáveis

- Se o tipo foi criado SEM typedef:
  union <nome_tipo> <nome_var>; (em C)
  <nome_tipo> <nome_var>; (em C++)

- Se o tipo foi criado COM typedef:
  <nome_tipo> <nome_var>;

Exemplos:

```
//Program struct

#include <stdio.h>

union { int a, char b, float c} x;  //x é variavel

void main (void)
{
   x.a=10;
   x.b = 'm';
   x.c = 0.56;
   printf ("%d, %c, %f",x.a, x.b, x.c);
}
```

```
//Program struct

union t_xyz  {float x, int y, int z };
typedef  union {int l, char m, int n} t_lmn;

void main (void)
{
   t_xyz  xyz; //xyz é uma variavel do tipo  t_xyz
   t_lmn  lmn; //lmn é uma variavel do tipo t_lmn

   xyz.y =10;
   lmn.l = xyz.y;
}
```

8-ALOCAÇÃO DE MEMÓRIA DINÂMICA

Um programa C compilado cria e usa quatro regiões, logicamente distintas na memória, que possuem funções específicas: área do código do programa, área de dados, área de pilha e área de heap.

8.1-ÁREA DO CÓDIGO DO PROGRAMA

Contém o código do programa armazenado.

8.2-ÁREA DE DADOS

-Área onde as variáveis globais são armazenadas.
-Os endereços das variáveis são resolvidos pelo compilador, a partir das suas declarações.
-As variáveis possuem **nome, que é usado para referenciá-la** no programa.
-A área reservada para cada variável é alocada no início e só é destruída no final da execução do programa.

8.3-ÁREA DE PILHA

-Área que possui os endereços das variáveis locais, endereços de retorno das chamadas de funções e endereços dos parâmetros formais.
-Cada vez que uma função é chamada é reservada uma área, a partir da qual são resolvidos os endereços das variáveis locais, do valor de retorno e dos parâmetros.
-As variáveis locais possuem **nome, que é usado para referenciá-las** no programa.
-O endereço de cada variável é alocado no início da execução da função, onde a variável está declarada e só é liberado no fim da execução da função.

Exemplo:

```
void main(void)
{
  int x;
  funcao1(x);
  funcao2( );
}
```

```
void funcao1 (int x)
{
   printf ("%d",x);
}
```

```
void funcao2(void)
{
   int x,y;
   y=funcao3(x);
}
```

```
 int funcao3(int x)
{
return (2*3);
}
```

8.4-ÁREA DE HEAP

-Região de memória livre, que o programa pode usar via funções de alocação dinâmica. Nesta área são alocadas as *variáveis dinâmicas.*
-O endereço da variável é resolvido pela rotina de alocação dinâmica, chamada explicitamente pelo próprio programa.
-As variáveis dinâmicas **não possuem nome (por isso são também chamadas de variáveis anônimas) e são referenciadas unicamente por ponteiros.**
-A alocação e liberação de endereços são feitas em quaisquer pontos do programa, de forma desestruturada. Exemplo:

```
//Prog Exemplo
#include <stdio.h>
int *a, b=5;                    //variaveis globais
void main (void)
{
        int y=10;               //variavel local
        float *z, w=20.0;
        a = new int;            //variaveis dinâmicas
        *a = b;
        z = new float;
        *z =  w;
        printf ("%d, %f",*a,*z);
        delete a;
        delete z;
}
```

8.5 – PONTEIROS

-Possibilita a criação e uso de variáveis dinâmicas (anônimas).
-Armazena o endereço de outra variável.
-É inicializada com o operador *new*.
-Quando não estão sendo usados devem conter o valor nulo NULL.

Operadores de Ponteiros

-Operador new
Tenta alocar dinamicamente (em tempo de execução) uma variável ou um vetor. Retorna o endereço do objeto alocado. Se não conseguir alocar, retorna NULL. Ex.:

```
float * x;
x = new float;
```

```
int * y;
y = new int[5];
```

-Operador delete

Libera uma area de memória previamente alocada com *new*. Ex.:

```
delete x;     /* usado para um único objeto*/
delete []y;   /* usado para vetores */
```

Obs.: eliminar um vetor com a sintaxe delete simples ou eliminar um objeto com a sintaxe delete[] tem efeito indefinido.

-Operador *

Possui duas finalidades:

a)Em uma declaração de variável, especifica que a variável é ponteiro

b)Devolve o valor da variável (conteúdo) localizada no endereço armazenado. Ex.:

```
int *p;
p = new int;
*p = 38;
```

-Operador &

Devolve o endereço de memória do seu operando. Ex.:

```
int *p, m;          |     int *p, **q;
p = &m;             |     q = &p;
```

-Operador ++ e –

Incrementam ou decrementam uma posição de memória de acordo com o tipo de dado da variável alocada nesta posição. Ex.:

```
int *p;
float *q;
```

Variáveis Estáticas X Variáveis Dinâmicas

Exemplo var. estática

```
void main (void)
{
        float x, *y;
        x = 15.8;
        printf ("%d\n",x);
        y = &x;
        printf ("%d  %d\n", x, *y);
}
```

Exemplo var. dinâmica

```
void main (void)
        {
                float  *y;
                y = new float[3];
                *y = 15.8;
                 printf ("%d \n",  *y);
                 delete[ ]y;
        }
```

Problemas com Ponteiros

-Lixo: uma variável dinâmica dinâmica para a qual ninguém aponta. Ex.:

```
int * x,* y;        |   * y=20;
* x=10;             |    y=x;
x = new int;        |
y = new int;
```

-***Referência Solta:*** uma variável anônima é destruída antes que todos os ponteiros p/ essa variável sejam destruídos. Ex.:

```
int *x, *y;
x = new int;
*x=10 ;
```

```
y = x;
delete x;
```

Exemplos de Erros

```
int x, *p;
x=10 ;
*p = x;
```

```
int x, *p;
x = 10;
p=x;
```

```
char s[80], y[80];
char *p1, *p2;
p1=s;   p2=y;
if (p1 < p2) ……..
```

Algumas Considerações sobre o Uso de Ponteiros

-Deve-se sempre perguntar se há espaço suficiente ao se tentar alocar memória dinamicamente. Ex.:

```
#define MAX 10
void main (void)
{
 int *x, i;
 x = new int [MAX];
 for (i=0; i< MAX; i++)
    printf("%d\n", x[i] = i);
 delete[ ]x;
}
```

```
#define MAX 10
void main (void)
        {
          int *x, i;
          x = new int [MAX];
          if (x==NULL)
                  printf ("Memória Insuficiente\n");
          else
          {
                  for (i=0; i< MAX; i++)
                          printf("%d\n", x[i] = i);
                  delete[ ]x;
          }
        }
```

-Pode-se percorrer um vetor alocado dinamicamente tanto usando o índice dos elementos quanto usando aritmética de endereço.

Percorrer pelo índice

```
for (i=0; i<MAX; i++)
        x[i] = i;
```

Aritmética não-destrutiva

```
for (i=0;i<MAX;i++)
 *(x+i) = i;
```

Aritmética destrutiva

```
for (i=0; i<MAX; i++)
{   *x=i;    x++;  }
```

9– MODIFICADORES DE TIPOS

Na declaração de uma variável temos:

[modificador] <tipo> <nome_var> [=valor];

Neste caso, modificador pode alterar os valores literais, o acesso e o armazenamento da variável nome_var. Os modificador aqui estudados serão:

-Modificador dos Tipos Básicos,
-Modificador de Armazenamento

9.1 – MODIFICADORES DE TIPOS BÁSICOS

Modificam os valores literais que os tipos básicos (*int, char, float* e *double*) podem receber. Os modificadores de tipos básicos podem ser:

a) modificadores de sinal: *signed* e *unsigned*
b) modificadores de tamanho: *short* e *long.*

Os modificadores de sinal *signed* e *unsigned* indicam que o conjunto de valores literais que uma variável pode assumir podem respectivamente conter sinal e não conter sinal. São aplicados aos tipos *char* e *int*. Vejamos os exemplos:

int x; //x ocupa 16 bits e pode assumir 65.536 valores distintos: de –32.767 a 32.767
unsigned int x; //x ocupa 16 bits e pode assumir 65.536 valores distintos: de 0 a 65.535
signed int x; //o mesmo que *int* x.
char y; //y ocupa 8 bits e pode assumir 256 valores distintos: de -127 a 127
unsigned char y; //y ocupa 8 bits e pode assumir 256 valores distintos: de 0 a 255
signed char y; //o mesmo que *char* y. Observe que *signed* é redundante

Os modificadores de tamanho *short* e *long* alteram o espaço necessário para armazenar as variáveis. O *short* pode ser aplicado a *int* e o *long* pode ser aplicado a *int* e a *double*. Ex.:

int x; //ocupa 16 bits e pode assumir 65.536 valores distintos: de –32.767 a 32.767
long int x; //ocupa 32 bits (4.294.967.296 valores distintos): de -2.147.483.647 a 2.147.483.647
short int x; //o mesmo que *int* x.
unsigned long int x; //ocupa 32 bits (4.294.967.296 valores distintos): de 0 a 4.294.967.295
double z; //ocupa 64 bits, com dez dígitos de precisão
long double z; // ocupa 128 bits, com dez dígitos de precisão

9.2- MODIFICADORES DE ARMAZENAMENTO

Indicam ao compilador como as variáveis devem ser armazenadas na memória. Os modificadores de armazenamento (também chamados de classe de armazenamento de dados) são: *register*, *automatic*, *static* e *extern.*

register

Declarar uma variável como *register* significa solicitar ao compilador que, caso seja possível, armazene a variável diretamente em um dos registradores da CPU. Isto otimiza o acesso à variável. Apenas variáveis locais (incluindo parâmetros formais) podem ser declaradas como *register.* Ex.:

```
register int i;              //i é acessada muitas vezes no laço, por isto, se for declarada como
for (i=0; i<MAX; i++)  // register seu acesso será otimizado.
{   ….  }
```

static

A classe *static* pode ser aplicada a variáveis locais e globais. Em cada um dos dois caso, a interpretação do uso de *static* é diferente.

Quando utilizada em variáveis locais, a classe *static* indica que o valor da variável é mantido entre as chamadas da função onde ela está declarada. Isto acontece porque variáveis locais *static* são armazenadas na área de dados e não na área de pilha. Sua alocação é feita na primeira vez que for feita uma chamada à função onde ela está declarada. Sua desalocação só é feita ao final da execução do programa. Apesar de ser alocada na área de dados, seu escopo continua sendo local, ou seja, só pode ser acessada de dentro da função onde está declarada. Ex.:

```
void funcaoX (int x);

void main (void)
{
  int i;

 for (i=0; i<5; i++)
        funcaoX(i);
}
```

```
void funcaoX(int x)
{
   static int total=0, i;

   for (i=0; i<x; i++, total++)
         printf("%d\n", total);
}
```

Neste exemplo, como a variável *total* foi declarada como *static*, seu valor será mantido cada vez que for feita uma chamada à *funcaoX* dentro da main.
Quando utilizada em variáveis globais, a classe static indica que as variáveis só podem ser utilizadas no aquivo onde forma declaradas. Para uma melhor assimilação da idéia da classe static, podemos dizer que, quando aplicada a variáveis locais, as variáveis são alocadas na área de dados e só são acessíveis de dentro da função onde estão declaras. Quando aplicadas a variáveis globais, as variáveis são alocadas na área de dados e só são acessíveis de dentro do arquivo onde foram declaradas.

extern

Este modificador indica que o espaço de armazenamento (e possivelmente o valor inicial) de uma variável já foi resolvido em outro arquivo. Uma variável só pode ser declarada como *extern* se ela não foi definida como *static* em outro arquivo.

O *extern* é aviso ao compilador de que ele não preciso resolver a alocação de uma variável (ou seja definir a variável), pois isto já foi decidido em outro arquivo. O uso do *extern* indica apenas uma declaração da variável e não uma definição da mesma (alocação de memória).

Analisemos o exemplo:

arquivo1.c	arquivo2.c
int a=1;	*int a;*
int b =1;	*extern double b;*
extern int c;	*extern int c;*

Neste exemplo há erros no uso das 3 variáveis:
- a é definida 2 vezes, ou seja, nos 2 arquivo o compilador tenta resolver a alocação de a;
- b é declarada duas vezes com tipos diferentes.
- c é declarada duas vezes, mas o compilador não sabe onde c foi definida (alocada).

Obs1.: Se for encontrada a declaração de uma variável em um único arquivo e o extern for utilizado, o compilador, apenas neste caso, resolve o endereçamento e alocação da variável.
Obs2.: É importante observar que uma declaração seguida de atribuição significa uma tentativa de alocar espaço para a variável. Portanto, como o uso extern implica em não alocação de espaço, não é permitido declarar uma variável com extern e atribuir a ela um valor. Exceto se a variável não existir em nenhum outro arquivo, conforme obs1.

Ex.:

arquivo1.c	arquivo2.c
int a=1;	*extern int a; //correto pois a só é alocada no arquivo 1*
int b;	*extern int b=3; //errado, pois está tentando alocar b nos dois arquivos*
extern int c=1;	*//correto, pois c só está declarada no arquivo1, logo será alocada .*

Observações:

O uso de *static* e *extern* podem ser estendidos a funções. Quando uma função é definida como static, indica que só é possível fazer chamadas a ela no arquivo onde ela foi definida. O exemplo abaixo mostra a declaração de uma função com static.

Ex.: arquivo1.c

```
static void funcao(void);
```

Uma função definida em um arquivo1 pode ser chamada em um arquivo2, bastando que no arquivo2 ela seja declarada como *extern*. Ex.:

arquivo1.c

```
int a=1;
extern int func1(void)  ;

void main (void)
{
  int b = func1();
  printf ("%d\n", b);
}
```

arquivo2.c

```
extern int a;

int func1(void)
{
   return 2*a;
}
```

Neste exemplo a função func1 é definida no arquivo2. A declaração desta função no arquivo1 é apenas um aviso ao compilador de que não é necessário resolver a alocação para func1, pois isto já foi resolvido em outro arquivo. A variável a foi alocada apenas no arquivo1.

10– ARQUIVOS (TEXTO)

O objetivo da utilização de arquivos é manter a persistência dos dados manipulados em um programa. Durante a execução de um programa as variáveis são alocadas na memória RAM: na região global, local e na área dinâmica. Porém, conforme visto no capítulo 8, até a finalização da execução do programa estas variáveis são desalocadas (de acordo com seu escopo) e os valores armazenados são todos perdidos. O uso de arquivos permite que os valores dos dados sejam armazenados no disco rígido da máquina de modo que possam ser acessados e recuperados durante processamentos posteriores do programa.

10.1 – O TIPO FILE

O tipo FILE é uma estrutura que permite que um arquivo armazenado no disco rígido possa ser acessado por uma variável da memória RAM, durante a execução do programa. Para tal é necessário que seja declarado um ponteiro da seguinte forma:

*FILE *arq;*

10.2 – ABERTURA DE UM ARQUIVO

Abertura de um arquivo residente no disco rígido é feita através da função fopen:

FILE *fopen(const char *****nome_arquivo**, const char *****modo *****);***

A variável nome_arquivo indica o nome do arquivo, que deve incluir a pasta onde o aqruivo se encontra. Ex.: "C:\\AulasPuc\L3GO\\Aulas.txt". Se o nome da pasta não for especificado, pressupõem-se que o arquivo esteja no diretório corrente, de onde o programa está sendo executado.

A variável modo indica o modo de abertura e pode assumir os seguintes valores:

- **"r"**, indica que o arquivo está sendo aberto para apenas para leitura.
- **"w"**, indica que o arquivo está sendo aberto para gravação. Se o arquivo não existir , um novo arquivo é criado. Se já existir, as informações são anteriores à abertura são perdidas e as novas informações são gravadas
- **"a"**, indica que o arquivo está sendo aberto para gravação. Se o arquivo não existir , um novo arquivo é criado. Se já existir, as novas informações são gravadas ao final das informações existentes antes da abertura, ou seja, mantém as informações antigas.

Retorna o endereço do arquivo aberto. Se não conseguir abrir o arquivo, retorna NULL.Exemplo:

```
FILE *arq;
arq=fopen ("C:\\AulasPuc\L3GO\\Aulas.txt","r");
if(arq==NULL)
 printf("Nao foi possivel abrir o arquivo");
```

10.3 – LEITURA ESCRITA DE CARACTERES EM ARQUIVOS

int getc(FILE *arq);

Lê um caracter do arquivo apontado por arq. O caracter é retornado como um inteiro. Exemplo:

```
 FILE *arq;
int    c;
arq=fopen (“C:\\AulasPuc\L3GO\\Aulas.txt”,”r”);
if(arq==NULL){
 printf(“Nao foi possivel abrir o arquivo”);
}
c=getc(arq);
```

int putc(int c, FILE *stream);

Grava um caracter no arquivo apontado por arq. O caracter é retornado como um inteiro. Exemplo:

```
FILE *arq;
int    c;
arq=fopen (“C:\\AulasPuc\L3GO\\Aulas.txt”,”w”);
if(arq==NULL){
 printf(“Nao foi possivel abrir o arquivo”);
}
scanf(“%c”,&c); //lê o caracter do teclado
putc(c, arq);      //grava o caracter no arquivo
```

10.4 – LEITURA E ESCRITA DE STRINGS EM ARQUIVOS

char *fgets(char *str, int n, FILE *arq);

Lê uma string do arquivo apontado por *arq* e armazena em *str*. O número *n* indica o tamanho máximo da string lida. Retorna a string lida. Se não conseguir ler, retorna NULL. Exemplo:

```
FILE *arq;
char  str[100];
arq=fopen (“C:\\AulasPuc\L3GO\\Aulas.txt”,”r”);
if(arq==NULL){
   printf(“Nao foi possivel abrir o arquivo”);
   return;
}
fgets(str,100,arq);
```

int fscanf(FILE *arq, const char *str [, argument]...);

Lê uma string formatada do arquivo apontado por *arq*. Além da string *str*, deve receber um conjunto de variáveis para os quais os valores lidos serão copiados. Retorna a quantidade de valores lidos corretamente. Exemplo:

```
FILE *arq;
int     x,y;
float   z;

arq=fopen ("C:\\AulasPuc\L3GO\\Aulas.txt","r");
if(arq==NULL){
  printf("Nao foi possivel abrir o arquivo");
  return;
}
fscanf(arq," %d,%d,%f",&x,&y,&z);
```

int fputs(const char *str, FILE *arq);

Grava a string str em arq. Retorna um número não negativo se executar a gravação corretamente. Exemplo:

```
FILE *arq;
char  str[100]="Exemplo de gravacao";
arq=fopen ("C:\\AulasPuc\L3GO\\Aulas.txt","w");
if(arq==NULL){
  printf("Nao foi possivel abrir o arquivo");
  return;
}
fputs(str,arq);
```

int fprintf(FILE *arq, const char *str [, argument]...);

Grava a string formatada *str* para o arquivo arq. Retorna o número de bytes gravados. Exemplo:

```
FILE *arq;
char  str[100]="Exemplo de gravacao";
int   x=10;
arq=fopen ("C:\\AulasPuc\L3GO\\Aulas.txt","w");
if(arq==NULL){
  printf("Nao foi possivel abrir o arquivo");
  return;
}
fprintf(arq,"%s: %d",str,x);
```

10.4 – FECHANDO UM ARQUIVO

Após a leitura/gravação o arquivo deve ser fechado, através da função fclose:

int fclose(FILE *****arq ***);

Retorna 0 se conseguiu fechar o arquivo arq.

10.5 – OUTRAS FUNÇÕES

int feof(FILE *****arq ***);

Retorna verdadeiro (diferente de 0) se a posição atual do arquivo (posição, dentro do arquivo, onde o mesmo está sendo acessado) indicar o final do arquivo.

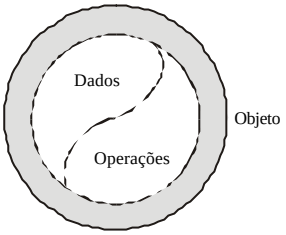
int fseek(FILE *****arq, long** *dist*, **int** *origem* **);**

Move a posição atual do arquivo para uma nova posição que esteja distante *dist* bytes a partir da posição indicada por *origem*.

11 – PROGRAMAÇÃO ORIENTADA A OBJETOS

11.1 – INTRODUÇÃO

- Orientação a objetos (OO) é um paradigma de programação que oferece um nível mais alto de abstração. Promove modelos mais próximos do mundo real.
- Cada ***objeto*** forma uma unidade que contém informações da ***estrutura*** e do ***comportamento*** do objeto.
- A estutura está relacionada aos dados e o comportamento está relacionado às operações.



Exemplos de Objetos: Aluno e Carro.

Aluno

Dados	Operações
Matrícula, Nome, Endereço, Data Nascimento, Curso, Notas,	IncluirDados Pessoais, Matricular, Alterar Endereço, Incluir Notas, Emitir Histórico

Carro

Dados	Operações
Modelo, Fabricante, Placa, Cor, Ano,	Ligar, Acelerar, Mudar a marcha, Frear, Virar à esquerda, Virar à direita,

Em C++ :

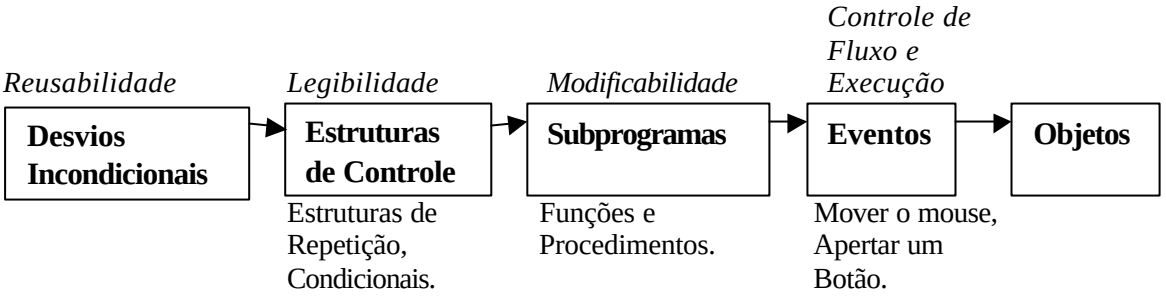
```
class aluno{
    private:
        int     matricula;
        char    nome[100];
        char    endereco[100];
        char    data_nasc[8];
        int     curso;
        float   notas[10];

    public:
        void IncluirDadosPessoais (int mat, char *nom, char *end, char *nas);
        void Matricular (int mat, int curs);
        void AlterarEndereco (char *end);
        void IncluirNotas (float *not);
        void EmitirHistorico (void);
};
```

A estrutura é implementada como o conjunto de dados (variáveis) e o comportamento é implementado como o conjunto de operações (funções) que atuam sobre o objeto.

Comparação entre Objeto e Registro

Um objeto sabe o que deve fazer e como se comportar, um registro não. Isto se deve ao fato de um registro conter basicamente um conjunto de dados, enquanto um objeto contém o conjunto de dados e o conjunto de operações que podem atuar sobre estes dados.

Evolução dos Paradigmas de Programação



Programação Orientada a Objetos (POO)

É um estilo de programação que aplica os conceitos e princípios de Orientação a Objetos.

Linguagens de Programação Orientada a Objetos (LPOO)

Linguagens de programação que suportam Orientação a Objetos, oferecendo facilidades para Programação Orientada a Objetos.

LPOO's Puras. Ex.: Smalltalk, Java, etc.
LPOO's Híbridas. Ex.: C++, Pascal Objeto, etc.

11.2 – CONCEITOS BÁSICOS

Objeto- É uma entidade que possui estrutura de dados e comportamento *encapsulados* juntos em uma unidade. Em um programa, um objeto é uma variável.

Método- É cada uma das operações (subprogramas) aplicadas ao objeto. Em um programa, um método é a implementação de uma função.

Comportamento- É o conjunto de métodos do objeto.

Estado- É o conjunto de valores associados aos dados de um objeto, em um determinada instante.

Mensagem- É a solicitação de um serviço (método) para um objeto. As mensagens alteram o estado dos objetos. Em um programa, uma mensagem é uma chamada de função.

Classe- É um conjunto de objetos com as mesmas características (dados e operações). Em um programa, uma classe é um tipo de dados. Em uma classe, os dados são chamados de *atributos* e as operações são chamadas de *serviços*. Cada objeto criado é chamado de *instância* da classe.

Método Construtor- Método utilizado para iniciar automaticamente os dados das instâncias de uma classe. Possui o mesmo nome da classe. Não possui retorno e é chamado no momento que a instância é criada.
Método Destrutor- Método utilizado para destruir automaticamente os dados das instâncias de uma classe. Possui o mesmo nome da classe, precedido de ~. Não possui retorno e é chamado no momento que a instância é liberada.

```
class professor{
  private:
        char nome[100];
        char endereco[100];
        float salario;

  public:
        professor (void);
        void AtualizarDadosPessoais (const char *n, const char *end);
        void AtualizarSalario (float s);
        void LerDadosPessoais (char *n, char *end);
        float LerSalario (void);
};//fim da definicao da classe
professor::professor (void)
{
        strcpy (nome, "Novo Professor");
        strcpy (endereco,  "Rua X");
        salario = 0.0;
}
void professor::AtualizarDadosPessoais (cons char *n, const char *end)
{
        strcpy (nome, n);
        strcpy (endereco, end);
}
void professor::AtualizarSalario (float s)
{
        salario = s;
}
void professor::LerDadosPessoais (char *n, char *end)
{
        strcpy  (n, nome);
        strcpy  (end, endereco);
}
float professor::LerSalario (void)
{
 return salario;
}
```

```
void main (void)
{
 professor jose, maria;
 char nome[100], end[100];

 jose.AtualizarDadosPessoais ("Jose Maria", "Rua da Paz, 15");
 jose.AtualizarSalario (50.0);
 maria.AtualizarDadosPessoais ("Maria Jose", "Rua do Sol, 20");
 maria.AtualizarSalario (45.0);
jose.LerDadosPessoais (nome, end);
 printf ("%s mora na %s e ganha %f reais\n", nome, end, jose.LerSalario());
 maria.LerDadosPessoais (nome, end);
 printf ("%s mora na %s e ganha %f reais\n", nome, end, maria.LerSalario());
}
```

Obs1.: Os métodos construtores declarados sem passagem de parâmetros formais, ou seja com void, são denominados de ***métodos construtores default***. Cada vez que um objeto é criado, o método construtor default é automaticamente chamado.

Obs2.: Os métodos cuja implementação ocorre dentro da definição da classe são chamados ***métodos inline***. Se o método é inline, então, durante a compilação, as chamadas aos métodos (mensagens enviadas aos objetos) são substituídas pela sua implementação e, portanto, durante a execução, não há uso da área de pilha para armazenar as variáveis locais destes métodos.

Os três princípios do paradigma da orientação a objeto são: encapsulamento, herança e polimorfismo. A seguir, cada um deles será estudado.

11.3- ENCAPSULAMENTO

É um dos princípios da O.O., no qual a estrutura de um objeto só é acessível através de mensagens enviadas ao objeto. Assim, encapsular significa esconder a estrutura de dados. O encapsulamento determina que uma classe consiste em duas partes:

a) Parte Invisível ou Privativa: formada pelas variáveis de instâncias e implementação dos métodos (em C++ a parte privada é especificada pela palavra reservada *private*).
b) Parte Visível ou Pública: formada pelas assinaturas dos métodos. (em C++ a parte publica é especificada pela palavra reservada *public*).

A partir do princípio do encapsulamento é definido *Tipo Abstrato de Dados*: tipo de dados que contém a representação e as operações em uma mesma unidade de encapsulamento. ***Uma classe é um Tipo Abstrato de Dados!***

11.4 – HERANÇA

É um dos princípios da O.O., no qual classes compartilham atributos e serviços de classes pré-existentes, evitando assim que seja projetado e implementado um código já existente. É um mecanismo para organização, construção e uso de classes reusáveis.

Superclasses – geram outras classes. Também são chamadas de classes bases ou ancestrais.
Subclasses – são herdadas a partir de outras classes. Também são chamadas de classes derivadas ou descendentes. Ex.: Estrutura de classes sem herança:

PROFESSOR

Dados	Operações
Nome	AtualizarEndereço
Endereço	IncluirDadosPessoais
Grau	AlterarGrau
Disciplinas	AlterarDisciplinas
Área	IncluirArea
Nasc	LerDadosPessoais
NumDisc	LerDadosAcademicos

ALUNO

Dados	Operações
Nome	AtualizarEndereço
Endereço	IncluirDadosPessoais
Matricula	Matricular
Curso	IncluirNotas
Disciplinas	LerDadosPessoais
Notas	LerCurso
Nasc	LerNotasDisc
NumDisc	LerMatricula

FUNCIONARIO

Dados	Operações
Nome	AtualizarEndereço
Endereço	IncluirDadosPessoais
Salario	AlterarSalario
Depto	IncluirDepto
Nasc	LerDadosPessoais
Cargo	LerSalário
	LerDepto
	LerCargo
	AlterarCargo

A mesma estrutura, utilizando herança, onde pessoa é uma superclasse e professor, aluno e funcionário são três subclasses de pessoa:

PESSOA

Dados	Operações
Nome	AtualizarEndereço
Endereço	IncluirDadosPessoais
Nasc	LerDadosPessoais

PROFESSOR

Dados	Operações
Grau	IncluirGrau
Disciplinas	IncluirDisciplinas
Área	IncluirArea
NumDisc	LerDadosAcademicos

ALUNO

Dados	Operações
Matricula	Matricular
Curso	IncluirNotas
Disciplinas	LerMatricula
Notas	LerCurso
NumDisc	LerNotasDisc

FUNCIONÁRIO

Dados	Operações
Salario	AlterarSalario
Depto	AlterarDepto
Cargo	LerSalário
	LerDepto
	LerCargo
	AlterarCargo

Implementação em C++: Na classe pessoa, o método *IncluirDadosPessoais* é construtor.

```
#include <string.h>
#define MAX_DISC 10
#define MAX_STR   100

//DEFINICAO DA SUPERCLASSE PESSOA
class pessoa{
   private:
        char nome[MAX_STR];
        char endereco[MAX_STR];
        char nasc[8];
   public:
        pessoa (const char *nom, const char *end, const char *d_nasc);
        void AtualizarEndereço ( const char *end) { strcpy (endereco, end);}
        void LerDadosPessoais (char *n, char *end, char *d_nasc);
};//fim da definicao da classe

pessoa::pessoa (const char *nom, const char *end, const char *d_nasc)
{       strcpy (nome ,nom);
        strcpy (endereco, end);
        strcpy (nasc, d_nasc);
}
void pessoa::LerDadosPessoais (char *n, char *end, char *d_nasc)
{       strcpy  (n, nome);
        strcpy  (end, endereco);
        strcpy  (d_nacs, nasc);
}

//DEFINICAO DA SUBCLASSE PROFESSOR
class professor: public pessoa{
   private:
        int grau;
        int disciplinas[MAX_DISC];
        int area;
        int n_disc;

   public:
        professor (const char *n, const char *e, const char *dn, int g,int a );
        void IncluirGrau (int g) {grau = g};
        void IncluirDisciplinas (int nd, const int *disc)
        {     n_disc=nd;
             for(int i=0; i<n_disc; i++) disciplinas[i]=disc[i];
        }
        void IncluirArea  (int a) {area=a;}
        void LerDadosAcademicos (int *g, int *disc, int *a);
};//fim da definicao da classe


professor::professor (const char *n, const char *e, const char *dn, int g,int a):pessoa(n,e,dn)
{       grau = g;
```

```
        area=a;
        n_disc=0;;
        for (int i=0; i<MAX_DISC; i++)   disciplinas[i]=0;
}
void professor:: LerDadosAcademicos (int *g, int *disc, int *a)
{
        *g=grau; *a= area;
        for (int i=0; i<n_disc; i++) disc[i]=disciplinas[i];
}

//DEFINICAO DA SUBCLASSE ALUNO
class aluno: public pessoa{
   private:
        int matricula;
        int curso;
        int disciplinas[MAX_DISC];
        int notas[MAX_DISC];
        int n_disc;

   public:
        aluno (const char *n, const char *e, const char *dn, int m,int c );
        void Matricular (int nd, const  int *d);
        void IncluirNotas (const int *n) { for (int i=0; i<n_disc; i++) notas[i]=n[i];}
        int LerMatricula (void) {return matricula;}
        int LerCurso (void) {return curso;}
        void LerNotasDisc (int *n, int *d);
}//fim da definicao da classe

aluno::aluno(const char *n, const char *e, const char *dn, int m,int c):pessoa(n,e,dn)
{       matricula = m;
        curso = c;
        n_disc=0;
        for (int i=0; i<MAX_DISC; i++)   { disciplinas[i]=0; notas[i] =0;}
}

void aluno:: Matricular (int nd, const  int *d);
{       n_disc = nd;
        for (int i=0; i<n_disc; i++) disciplinas[i]=disc[i];
}

void aluno::LerNotasDisc (int *n, int *d)
{   int i;
    for ( i=0; i<n_disc; i++)
   {
        n[i]=notas[i];
        d[i]= disciplinas[i];
   }
}

//DEFINICAO DA SUBCLASSE FUNCIONARIO
class funcionario: public pessoa{
```

```
   private:
        float  salario;
        int    depto;
        int    cargo;
   public:
        funcionario (const char *n, const char *e, const char *dn, float s,int d, int c );
        void AlterarSalario (float s) {salario =s;}
        void AlterarDepto (int d) {depto =d;}
        void AlterarCargo (int c) {cargo = c;}
        float LerSalario (void) {return salario;}
        int   LerDepto (void) {return depto;}
        int   LerCargo (void) {return cargo;}
}//fim da definicao da classe

funcionario::funcionario (const char *n, const char *e, const char *dn, , float s,int d, int c):
pessoa(n,e,dn)
{       salario = s;
        depto = d;
        cargo = c;
}
```

Acesso aos Campos de uma Classe

Em C++ há três formas de se acessar os campos (dados ou operações) definidos em uma classe: *private*, *protected* e *public*.

private: dados ou operações definidos como *private* só podem ser acessados dentro da própria classe, ou seja, por outros campos definidos dentro da própria classe..
protected: dados ou operações definidos como *protected* só podem ser acessados ou dentro da própria classe ou dentro das suas classes derivadas.
public: dados ou operações definidos como *public* podem ser acessados em qulauqe instrução, externa ou interna à classe. Ou seja campos *public* podem ser acessados de qualquer parte do programa.

De acordo com as definições acima, uma classe derivada D de uma classe base B não tem acesso aos seus campos privados, sendo necessárias, portanto, definições de métodos públicos em B, para que D acesse (indiretamente) seus dados privados.

No exemplo de herança, visto acima, as classes professor, aluno e funcionario foram definidas como derivadas de pessoa, de forma publica. O texto *:public pessoa*, escrito logo após o nome de cada classe derivada, diz ao C++ que as propriedades herdadas devem manter o seu status original, ou seja, o que era *public* na classe base deve permancer *public* na classe derivada e o que era *private* deve continuar *private*. Se o texto utilizado fosse *:private pessoa*, então as classes derivadas herdariam todos os campos da classe base de forma privada. Seja o exemplo abaixo:

```
class A{
 private:
    int a;
 public:
    void funcaoA(void);
};
```

```
class B: public A{
  private:
    int b;
  public:
    void funcaoB(void);
};
```

```
class C: private A{
   private:
         int c;
   public:
void funcaoC(void);
};
```

A variável *a* é privada da classe A, logo só pode ser acessada de dentro da classe A. A função *funcaoA* é pública, logo pode ser acessada fora da classe A. As classes B e C são derivadas da classe base A. Logo um objeto da classe B tem dois dados (*a* e *b*) e duas operações (*funcaoA* e *funcaoB*) e um objeto da classe C também tem dois dados (*a* e *c*) e duas operações (*funcaoA* e *funcaoC*). Lembre-se que o dado *a* , apesar de constar nos objetos das classes derivadas, não podem ser acessados de dentro destas.

Como B é derivada de A de forma pública, então o que é público em A continua público em B e o que é privado continua privado, ou seja B tem os dados *a* e *b* como privados e *funcaoA* e *funcaoB* como publicos. Como C é derivada de A de forma privada, então todas as informações de A passam a C de forma privada, ou seja, C tem *a*, *c* e *funcaoA* como privados e *funcaoC* como publico. Isto significa que , se agora definirmos as subclasses

```
class B1: public B{
 private:
    int b1;
 public:
    void funcaoB1(void);
};
```

```
class C1: public C{
  private:
         int c1;
  public:
        void funcaoC1(void);
};
```

onde B1 é derivada de B e C1 é derivada de C, então B1 tem acesso direto a *funcaoB* e *funcaoA* (que são informações públicas de B) e C1 só tem acesso direto a *funcaoC* (única informação pública de C).

Substituição de Métodos nas Classes Derivadas

Métodos declarados na classe base podem ser substituídos nas classes derivadas, desde que sejam declarados de forma idêntica (mesmos parâmetros formais e retorno). Esta substituição é uma primeira tentativa de modificar o comportamento de uma subclasse. A classe drivada herda todos os métodos da classe base, exceto os métodos de substituição, uma vez , como o nome já diz, são substituídos. Ex.:

```
class empregado{
 private:
   float salario;
   char nome[100];
 public:
   empregado (void)
    {strcpy (nome,"Novo"); salario=100;}
   void CadastrarNome (const char *n)
    { strcpy (nome,n); }
  void AtualizarSalario(float s){salario=s;}
};

//Aplicacao

void main (void)
{ vendedor v;

 v.CadastrarNome ("José da Silva");
 v.AtualizarSalario (100.05);
}
```

```
class vendedor:public empregado{
 private:
        float comissao;
 public:
        vendedor(void):empregado()
        {comissao = 10;}
        void AtualizarSalario (float s)
        {
         s += s*comissao/100;
         empregado::AtualizarSalario(s);
        }
};
```

Nesta aplicação, o método *CadastrarNome* associado à instância *v* é definido na classe empregado. O método *AtualizarSalario* associado à instância *v* é definido na classe vendedor. O método *AtualizarSalario* é um método de substituição. É importante estar ciente de que quando um método é substituído em uma classe derivada, o método original não desaparece da classe base.

Ponteiros para Classes Derivadas

Armazenar instâncias (objetos) de classes na área de heap ajuda os programadores a utilizarem a memória eficientemente (isto vocês já sabiam!). Porém, a novidade é que um ponteiro pode endereçar tanto uma instância de uma classe base, quanto uma instância de *qualquer outra classe derivada desta*. Ex.: Seja a aplicação que utiliza as classes empregado e vendedor, definidas acima.

```
void main (void)
{  empregado *p;

   p = new empregado;   /*endereca uma instância da classe empregado*/
  if (p==NULL)
  {  printf ("Memoria Insuficiente\n");
     return;
  }
   p->CadastrarNome ("José da Silva");
   p->AtualizarSalario (100.05);
   delete p;

   p = new vendedor;   /*endereca uma instância da classe vendedor*/
   if (p==NULL)
  {  printf ("Memoria Insuficiente\n");
     return;
  }
   p->CadastrarNome ("Joao Lima");
   p->AtualizarSalario (100.02);
  delete p;
}
```

Neste caso, porém, p é um ponteiro para o tipo empregado e ambas as mensagens *p->AtualizarSalario* estão associadas ao método definido na dasse empregado. Já no exemplo abaixo p é um ponteiro para vendedor e a mensagem *p->AtualizarSalario* está associada à classe vendedor:

```
void main (void)
{ vendedor *p;

   p = new vendedor;   /*endereca uma instância da classe empregado*/
  if (p==NULL)
  {  printf ("Memoria Insuficiente\n");
     return;
  }
   p->CadastrarNome ("Joao Lima");
   p->AtualizarSalario (100.02);
  delete p;
}
```

Isto deve-se ao fato de os métodos serem associados às mensagens em TEMPO DE COMPILAÇÃO. Quando o compilador encontra um envio de uma mensagem (chamada de função), além dos parâmetros reais da chamada de função, ele acrescenta mais um parâmetro que é o endereço do objeto que está recebendo a mensagem. No caso de p ser declarado como *empregado *p*,quando o compila dor encontra a mensagem *p->AtualizarSalario()*, ele só sabe que p foi declarado como ponteiro de *empregado* e, portanto, associa esta mensagem ao método *AtualizarSalario* da classe *empregado* (mesmo que p esteja apontando para um objeto da classe *vendedor*). O uso do princípio do polimorfismo ajuda a resolver este problema.

11.5 – POLIMORFISMO

É a capacidade de instâncias múltiplas assumirem seus métodos em TEMPO DE EXECUÇÃO. Permite que mensagens iguais sejam enviadas a objetos que responderão diferentemente a estas mensagens. Exemplo: Este é o mesmo exemplo anterior, agora usando polimorfismo.

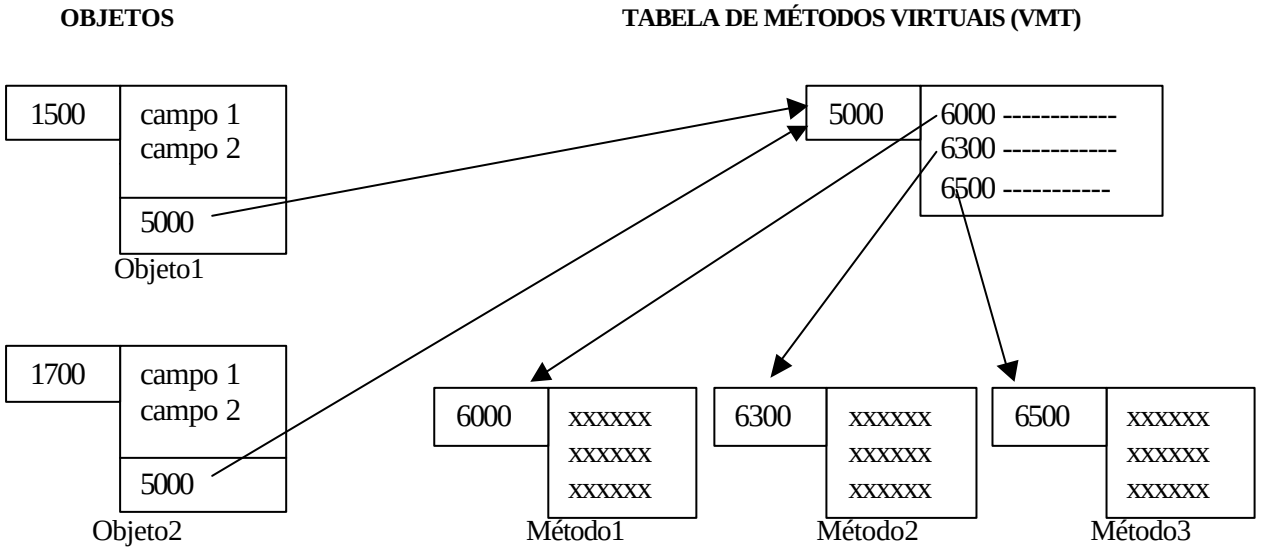
```
class empregado{
  private:
     float salario;
    char nome[100];

public:
    empregado (void)
      {strcpy (nome,"Novo"); salario=100;}

     void CadastrarNome (const char *n)
      { strcpy (nome,n); }

    virtual void AtualizarSalario(float s)
      {salario=s;}
};
```

```
class vendedor:public empregado{
  private:
        float comissao;

  public:
     vendedor(void):empregado()
        {comissao = 10;}

     virtual  void AtualizarSalario (float s)
              {
        s +=  s*comissao/100;
        empregado::AtualizarSalario(s);
        }
};
```

Um método declarado com *virtual* é um método polimórfico. Com isto, quando uma mensagem é enviada a uma instância, ela só é associada a um determinado método em TEMPO DE EXECUÇÃO. Para que isto seja possível, para cada classe definida existe uma tabela, chamada Tabela de Métodos Virtuais, contendo os endereços de todos os métodos virtuais da classe.

Tabela de Métodos Virtuais (VMT)

É uma estrutura alocada para cada classe que contenha algum método virtual. Esta estrutura contém vários campos, onde cada campo corresponde a um endereço de um de seus métodos virtuais. Ou seja a VMT é basicamente uma lista de ponteiros para métodos. Quando um objeto é alocado na memória (em tempo de execução), o endereço da VMT de sua classe é associado a ele. Através da VMT, chega-se aos métodos correspondentes às mensagens enviadas ao objeto.



Na figura acima existem três métodos virtuais e seus endereços estão guardados na tabela de métodos virtuais. Quando os objetos 1 e 2 são alocados , o endereço da tabela de métodos virtuais é associado a eles.

11.6- CONCEITOS AVANÇADOS

Método Abstrato –Método virtual que não está definido (implementado) . É apenas declarado. Para definir um método virtual em C++, basta atribuir zero em sua declaração.

virtual tipo-retorno nome_do_metodo (parâmetros formais) = 0;

A criação de um método abstrato em uma superclasse significa que a implementação deste método só será definida nas subclasses.

Classe Abstrata- Classe que contém pelo menos um método abstrato. Uma classe abstrata não pode conter instâncias diretas.

Atributo Classe – Atributo que contém informações globais da classe. Não pode ser referenciado diretamente a partir de nenhuma instância da classe, ou seja, é referenciado diretamente pela própria classe. Para definir um atributo classe, em C++, deve-se declará-lo com static.

Método Classe – Método que trata (acessa) os atributos classe. Em C++, um método classe deve ser declarado com static.

Exemplo de Métodos Abstrato e Classe Abstrata: Neste exemplo não mostra as implementações dos métodos.

```
class shape{
        public:
                virtual void Draw(void) =0;  //Método Abstrato, logo shape é classe abstrata!
};
//-----------------------------------------------
class circle: public shape{
        public:
                virtual void Draw(void) ;
};
//-----------------------------------------------
class square: public shape{
        public:
                virtual void Draw(void) ;
};
//-----------------------------------------------
class line: public shape{
        public:
                virtual void Draw(void) ;
};
//-----------------------------------------------

void main (void)
{
   shape *p[3];  //vetor de 3 elementos: cada elemento é um ponteiro

        p[0] = new circle;
        p[1] = new square;
        p[2] = new line;

        for ( int i=0; i<=2; i++)
                p[i]->Draw();
}
```

Perceba que nenhuma instância criada é do tipo shape. Isto não é possível, pois shape é classe abstrata.