Curso de Programação

Orientada a Objetos com

Borland C++

Sumário

1. Introdução

A linguagem C++ é, atualmente, uma das principais ferramentas para desenvolvimento profissional. Sua popularidade deve-se à compatibilidade com a linguagem C, simplicidade estrutural, portabilidade entre plataformas e à incorporação de ferramentas que possibilitam a programação orientada a objetos (POO ou OOP).

Além da linguagem, outros dois aspectos têm importância fundamental no sucesso dos pacotes de C / C++. São eles os ambientes de desenvolvimento integrados (**ADI**) de programação e as bibliotecas de funções.

Desde sua concepção ocorreram ampliações na linguagem C++ que ainda não são totalmente suportado pelos compiladores,. Entre as novas características, presentes na última padronização do "American National Standard Institute" (ANSI), estão a criação de classes paramétricas ("templates") e o tratamento de exceções. O Microsoft Visual C++, por exemplo, lida bastante bem com exceções, porém a criação de classes paramétricas não foi implementado. Já o Symantec C++ e o Borland C++ apresentam ambas as características.

A linguagem C++ propicia o desenvolvimento de bibliotecas devido à sua estrutura orientada a objetos. Com isso é possível minimizar bastante o trabalho do programador através da reutilização de código. É por este motivo que faz tanto sucesso o conjunto de funções para construção de interface como o **Turbo Vision**, da Borland, para DOS, a Object Windows Library (**OWL da Borland**) e a Microsoft Foundation Class Library (**MFC**), ambas para Windows. Há também várias bibliotecas destinadas ao controle de estruturas, banco de dados, etc. Uma dica: não fique reinventando a roda, verifique as bibliotecas disponíveis no mercado.

2. Novos Recursos de Compiladores C++ e Múltiplos Arquivos

2.1. Objetos de I/O

As streams C++ correspondentes de **stdin, stdout** e **stderr** são **cin, cout** e **cerr.** Se a include **"iostream.h"** estiver presente, estas três streams são abertas automaticamente quando o programa começa a execução e tornam-se a interface entre o programa e o usuário. A stream **cin** está associada ao teclado (por default) e as streams **cout, cerr** e **clog** estão associadas ao monitor de vídeo (por default).

Os operadores extração (>>) e inserção (<<) .

Em C+ + a chave para a biblioteca stream é o operador de saída (<<) ou entrada (>>), que substitui a chamada de função **printf** e **scanf** respectivamente.

Para mostrar uma mensagem na tela de vídeo usando o padrão C:

```
printf(" %d\n",  valor);
```

Em C++ para mostrar uma mensagem na tela do vídeo, usamos a forma mais concisa:

```
cout << valor << endl;
```

Note que não foi necessário incluir uma especificação de formato, já que o operador de saída (<<) determina isso para você. Com a biblioteca stream C++ é incluído somente o código necessário para uma conversão especifica. Em outras palavras, o código C++ para operações de I/O é menor e mais rápido do que seu correspondente em C.

C++ fornece também um operador de entrada (>>) que converterá o exemplo C a seguir:

```
scanf("%d", &int_valor);
```

em

```
cin >> int_valor;
```

Assumindo que int_valor tenha sido definido como **int**, o operador >> executa automaticamente a conversão necessária. O operador de entrada substitui a função padrão **scanf.** No entanto, não é uma substituição completa, porque o operador de entrada faz apenas a conversão de dados. A função **scanf** é um recurso de entrada completo e formatado que permite

descrever o formato da linha de entrada. O operador de entrada stream tem as mesmas vantagens que a função de saída; o código resultante é menor, mais específico e eficiente.

As streams **cerr** e **clog** tem a mesma função de **cout,** mas são usados para indicar com mais clareza no programa a emissão de uma mensagem de erro.

Saída de Caractere

Na nova biblioteca I/O, o operador de inserção << foi sobrecarregado para manusear caracteres. A instrução abaixo:

```
cout << letra;
```

daria como resultado a própria letra, mas é possível obter o valor ASCII da letra com a instrução:

```
cout << (int) letra;
```

Conversões de Base

Podemos em C++ apresentar um valor em várias bases diferentes com o seguinte comando:

```
cout << hex << valor;
```

Formatação Numérica

Dados numéricos podem ser formatados facilmente com ajuste à direita ou à esquerda, vários níveis de precisão, vários formatos (ponto flutuante ou científico), zeros anteriores ou posteriores e sinais. O trecho a seguir mostra o valor de *pi* alinhado à esquerda num campo de tamanho 20, com zeros posteriores:

```
cout .width(20);
cout.setf(ios::left);
cout .fill("0");
cout << pi;
```

Exemplos:

```
cout << " Isto e uma linha.\n";
cout << 1001 << "\n" << 1.2345;
cout << "Valor decimal de 0xFF: " << dec << 0xFF;
cout << endl << "Valor Octal de 10: " << oct << 10;
cout << "\nValor Hexadecimal de 255: " << hex << 255;
cerr << "Erro na abertura do arquivo!" << endl;
cout << setiosflags(ios::scientific) << value << endl;
```

Para saber mais:
Adendo B

Manual do Usuário Borland C++ 3.0 - Cap 16 “Using C++ streams”
Manual do Usuário Borland C++ 3.0 - Cap 4 “Object-oriented programming with C++”

2.2. Operador de Resolução de Escopo

Escopo é determinado pelo local onde a variável foi declarada. A variável definida dentro de uma definição de função é local por default - ela pode somente ser acessada pelo código dentro da função. Para maio clareza análise o código abaixo.

```
// INTRO26.CPP -- Example from Chapter 3, “An Introduction to C++”
#include <iostream.h>

void showval(void);

int main( ) {
  int mainvar = 100;
  showval( );
  cout << funccar  << ‘\n’;          // ERRO: não é escopo de funcvar
  return 0;
}

void showval(void) {
  int funcvar = 10;
  cout << funcvar ‘\n’;
  cout << mainvar << ‘\n’;           //ERRO: não é escopo de mainvar
}
```

O operador de resolução de escopo (**::**) permite o acesso a uma variável global que é ocultada por uma variável local com o mesmo nome.

```
int valor;                  // variável global
...
void func(void);
{
  int valor = 0;            // variável local valor oculta variável global valor
  valor = 3;                // este é o valor local
  :: valor = 4;             // este é o valor global
  cout << valor;            // imprime 3
}
```

2.3. Passagem por Referência

Em geral, pode-se passar argumentos para funções de duas maneiras.

A primeira é chamada **passagem por valor**. Este método copia o valor de uma argumento no parâmetro formal da função. Assim, alterações feitas nos parâmetros da função não têm nenhum efeito nas variáveis usadas para chamá-las.

A segunda forma é a **passagem por referência**. Este método copia, o endereço de um argumento real usado na chamada. Isso significa que as alterações feitas no parâmetro afetam a

variável usada para chamar a função. É representado pelo símbolo “&” precedendo o nome do parâmetro na declaração.

Pode-se também utilizar a **passagem por referência por ponteiros**, permitindo assim que mais de uma variável (onde uma foi passada como argumento) seja alterada dentro da função.

```
#include <iostream.h>
#include <iomanip.h>

void call_by_value(int a, int b, int c)
{
  a = 3; b = 2; c = 1;
}

void call_by_pointer_reference(int *a, int *b, int *c)
{
  *a = 3; *b = 2; *c = 1;
}

void call_by_reference(int& a, int& b, int& c)
{
   a = 1; b = 2; c = 3;
}

void main(void)
{
  int a = 1, b = 2, c = 3;
  int& a_alias = a;          // a_alias é um pseudônimo (apelido) para a
  int& b_alias = b;          // b_alias é um pseudônimo (apelido) para b
  int& c_alias = c;          // c_alias é um pseudônimo (apelido) para c

  call_by_value(a, b, c);
  cout << "By value: " << a << b << c << '\n';

  call_by_pointer_reference(&a, &b, &c);
  cout << "By pointer: " << a << b << c << '\n';

  call_by_reference(a_alias, b_alias, c_alias);
  cout << "By reference: " << a << b << c << '\n';
}
```

2.4. Operadores new e delete

Os operadores **new** e **delete** permitem alocação e desalocação dinâmica de memória, similar, porém superior, as funções de biblioteca padrão, **malloc( )** e **free( )**.

A sintaxe simplificada é:

ponteiro_para_classe = **new** classe <tipo_inicializador>;

delete ponteiro_para_nome;

O operador **new** tenta criar um objeto do tipo classe, alocando (se possível) **sizeof(**classe**) bytes** na memória livre (também chamada de heap). O tempo de duração de um objeto novo (variável nova) é do ponto de criação até o operador **delete** liberar o espaço reservado, ou até o fim do programa.

Em caso de sucesso, **new** retorna um ponteiro para o novo objeto. Um ponteiro **NULL** é retornado em caso de falha(tal como insuficiência ou memória livre fragmentada)

Exemplo do uso de **new** e **delete:**

```
void main(void)
{
  char *array = new char[256];
  char *target, *destination;

  int i;

  target = new char[256];
  for (i = 0; i < 256; i++)
  {
    array[i] = 'A';
    target[i] = 'B';
   }

  delete array;

  destination = new char[256];
  for (i = 0; i < 256; i++)
   {
    destination[i] = target[i];
    cout << destination[i] << ' ';
   }

  delete target;
  delete destination;
}
```

2.5. Argumentos default

Para as declarações de funções no padrão ANSI C, C++ adiciona argumentos default que são definidos dentro da declaração da função. Pode-se definir valores por default para alguns argumentos de uma função. Neste caso, pode-se omitir estes argumentos quando a função é chamada. Considere o seguinte protótipo:

```
unsigned func(int a, int b = 0);
```

Esta função, **func( )**, recebe dois argumentos inteiros onde o segundo argumento, se não for passado nenhum valor atribui 0 por default. Então, dado o protótipo acima, as três chamadas seguintes são equivalentes.

```
int  i, x = 2, y = 0;
i = func(x, 0);
i = func(x, y);
i = func(x);
```

Mais de um argumento pode receber valores default na mesma função., mas eles devem ser atribuídos da direita para esquerda. A seguinte declaração não é válida:

```
unsigned func(int a = 0, int b);          //ERRO!
```

.

Os argumentos com valores por default devem sempre ser os últimos argumentos da função.

```
unsigned char valor( unsigned char c = 0, char bitIni = 0, char bitFim = 0){
  c <<= 7 - bitFim;
  c >>= 7 - bitFim + bitIni;
  return c;
}


void main(){
  unsigned char i = 77, k;   // i recebe  77       = binário 01001101
  k = valor(i, 2, 5);        // k recebe 3         = binário 00000011
  k = valor(i, 2);           // k recebe 19        = binário 00010011
  k = valor(i);              // k recebe 77        = binário 01001101
  k = valor( );              // k recebe 0         = binário 00000000
}
```

2.6. Funções inline

Em C++ uma função pode ser declarada como **inline**. A definição de uma função como **inline** faz com que cada vez que a função é invocada, ela seja expandida no local da chamada.

Resumindo, esta palavra chave sugere ao compilador substituir cada chamada da função pela introdução do seu código no local da chamada.

Para definir uma função como inline basta usar a palavra reservada **inline**.

```
inline  void  ptrTroca(int *x, int *y){
  int aux = *x;
  *x = *y;
  *y = aux;
}

inline  void refTroca(int & x, int & y)          {
  int aux = x;
  x = y;
  y = aux;
}

void main( ){
  int i = 15, j = 51;
```

```
  cout << "\ n i = " << i <<  "    j = " << j ;
  ptrTroca(& i, & j);
  cout << "\ n i = " << i <<  "    j = " << j ;
  refTroca(i, j);
  cout << "\ n i = " << i <<  "    j = " << j ;
}
```

saída do programa:

i = 15 j = 51

i = 51 j = 15

i = 15 j = 51

Os projetistas do C++ preferiram criar uma forma de substituir **#define**, tal como

```
#define square(x)  x*x
```

Este tipo de **#define** gerava erros para algumas declarações, devido a precedência de operações, tais como

```
int y = square(2+5);          // resultado esperado 49 e o obtido 17 (2+5*2+5)
```

ou

```
int x = 2;
int y = square(x++);          // resultado esperado x = 3 e y = 9
                              // obtido x = 4 e y = 6 (2*3)
```

Estruturas de controle(tais como **if**, **for**, etc.) não são permitidas dentro de funções **inline**. Estes tipos de funções devem ser usadas para criar funções pequenas. Uma sugestão é que funções com no máximo 4 ou 5 linhas sejam criadas como **inline**.

2.7. Projeto

O **ADI** (Ambiente de Desenvolvimento Integrado) coloca todas as informações necessárias para trabalharmos com **múltiplos** arquivos dentro de um arquivo binário de projeto, um arquivo com extensão **PRJ**. Arquivos projetos contém as configurações para:

- opções do compilador, linker, maker e bibliotecas;
- diretórios de pesquisa (includes, bibliotecas, arquivos, etc);
- a lista de todos os arquivos necessários para execução do projeto e
- tradutores especiais (tal como Turbo Assembler).

Arquivos projeto podem ser carregados de três formas:

1. Quando iniciar o Turbo C+ 3.0, fornecer o nome do projeto com extensão **PRJ** após o comando **TC**.

   tc meuproj.prj

   Obs.: Deve colocar a extensão PRJ para diferenciar dos arquivos fontes (CPP).
2. Se existir somente um arquivo **PRJ** no diretório corrente, o ADI assume que este diretório é dedicado para este projeto e automaticamente o carrega.
3. Para carregar um projeto de dentro do ADI, escolha a opção **Project | Open** e selecione um arquivo projeto. Se um nome não existente for dado o ADI cria um novo projeto.

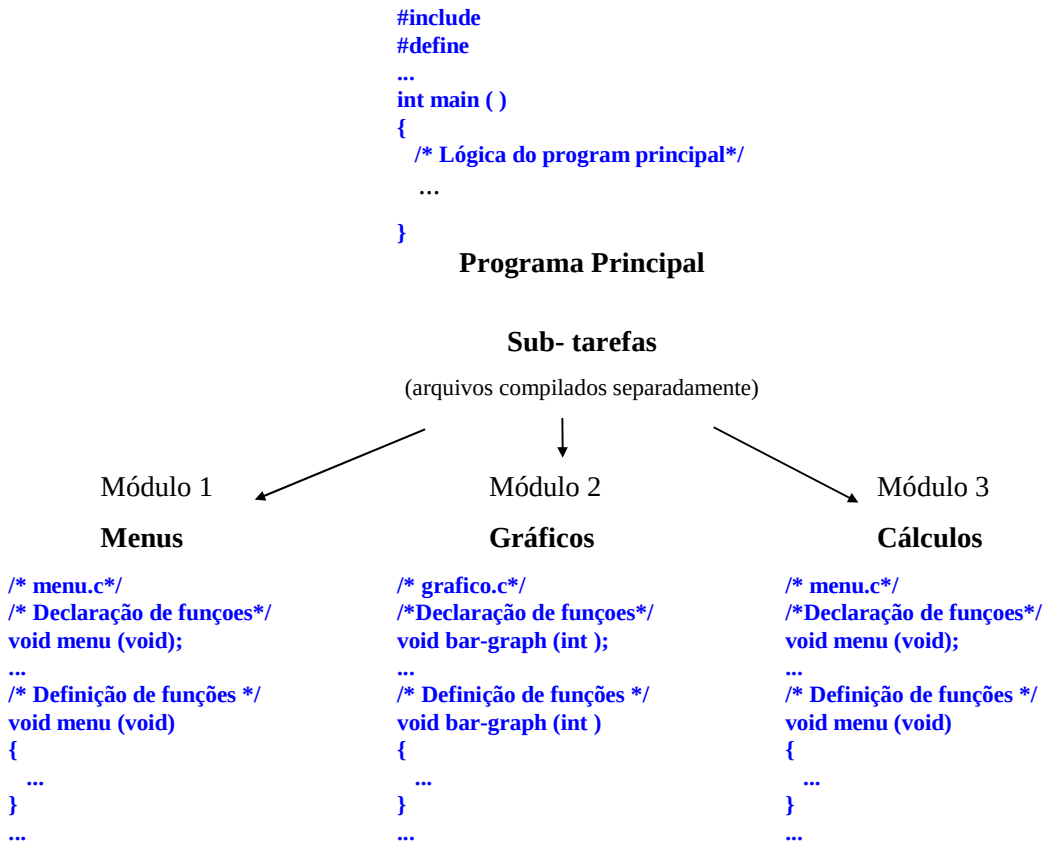
Headers (.h ou .hpp), Funções(.c ou .cpp) e Bibliotecas (.lib)

Em um programa pequeno, provavelmente declarações e definições de funções serão colocadas no mesmo arquivo, juntas com a função principal (**main**). Como a estrutura mostrada a seguir

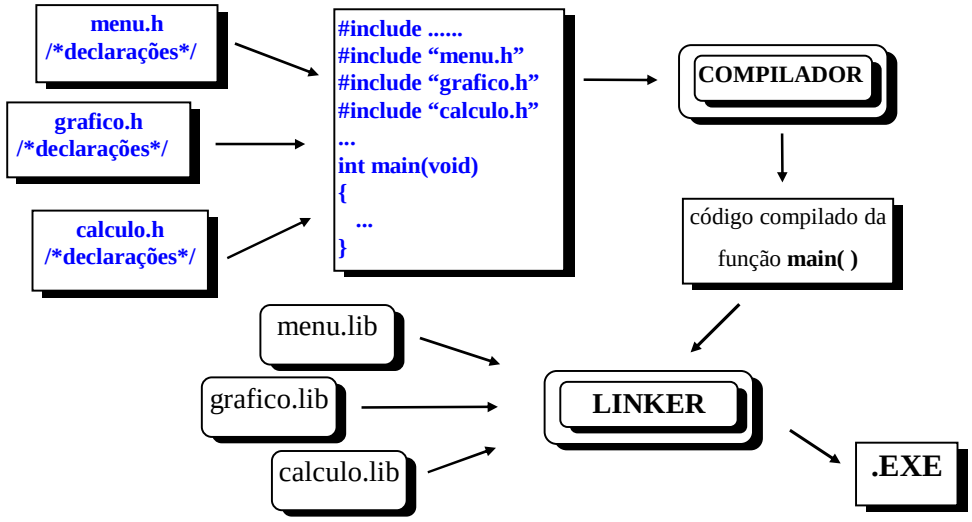
```
# include ...
# define ...

/* Declaração de funções */
void desenha_menu(void);
...
/* Dados globais */
...
main( )
{
  /* Lógica do programa principal , */
  /* incluindo as chamadas de funções */
}
/* Definição de funções (código)*/
void desenha_menu(void)
{
  /* código para desenha_menu( ) */
  ...
}
/* Definição de outras funções */
...
```

Quando os programas tornam-se maiores, agrupar definições de **funções relacionadas** em arquivos separados aumenta a clareza e diminui o tempo perdido na compilação de programas. Por exemplo, as funções relacionadas a interface com o usuário ficariam em arquivo (**menu.c**), as funções relacionadas com processamento de dados em um segundo arquivo (**calculo.c)** e as funções relacionadas com apresentações gráficas em um terceiro arquivo

(**grafico.c**). Turbo C++ pode compilar os três arquivos juntos para criar um programa final executável. Este tipo de estrutura é apresentado a seguir:



Algumas partes do programa são estáveis, ou seja, não são modificadas, pode-se compilar grupos de funções e torná-las bibliotecas. As declarações para as funções em cada uma das bibliotecas podem ser colocadas em arquivos cabeçalhos (**headers**) da mesma forma que acessamos as bibliotecas pré-compiladas do Turbo C++ (ex: **stdio.h**). Seu programa principal inclui as **headers**, as quais inserem as declarações no texto do programa (como a inserção de um editor de texto). Após a compilação, o **linker** une (“linka”) as bibliotecas em seu **programa objeto**. Este processo e´ apresentado na figura a seguir:



Para saber mais:

Manual do Usuário Borland C++ 3.0 - Cap 7 "Managing multi-file projects"

3. Introdução a Programação Orientada a Objetos

3.1. Conceitos

Muito se fala a respeito da Programação Orientada a Objetos (**POO**), mas defini-la é um tanto complicado. Para começar, não existe concordância sobre os elementos que devem existir numa linguagem para que esta seja especificada como orientada a objetos. Pior, muitos autores não têm a mínima idéia da teoria das linguagens de programação e não têm vergonha em escrever sobre o que não sabem.

A orientação a objetos pressupõe a colocação de **código** (instruções) e **dados** no mesmo **nível de igualdade**. As definições, porém, costumam englobar mais do que isso. A mais simples diz que deve-se ter essa igualdade e **encapsulamento** ou a possibilidade de esconder elementos uns dos outros.

A idéia é incluir uma construção (ou reduzir tudo o que já existe a uma única construção) chamada **classe**, similar aos **tipos** (ex.: **int**) das linguagens tradicionais, que admite **instâncias** ou realizações, similares as **variáveis** (ex.: int **x**, **y;**). As instâncias são comumente chamadas de objetos e podem conter tanto dados quanto códigos. Em teoria, somente o código de um objeto pode manipular os dados contido nele.

Uma segunda definição, muito popular, engloba a anterior e acrescenta o mecanismo de **herança**. Este é a possibilidade de se aproveitar classes já existentes para a criação de novas, especificando somente as diferenças.

A POO traz um conceito há muito sonhado pelos programadores profissionais: Circuitos Integrados de Software (**CIS**), ou seja, elementos fechados que têm uma determinada função com uma interface muito bem definida. Com larga disponibilidade destes componentes, um programador poderia trabalhar como um engenheiro ou técnico eletrônico o faz, conectando componentes.

3.1.1. Classe

Classes são os veículos primários para implementar **encapsulamento de dados**, **abstração de dados** e **herança -** as características que fazem o C++ uma linguagem de POO.

Muitos objetos no mundo real têm características similares e realizam operações similares. POO é um **método de programação** que procura “imitar“ a forma que modelamos o

mundo real. Para trabalhar com as complexidades da vida, a linguagem de POO desenvolveu uma maravilhosa capacidade de **generalizar**, **classificar** e **produzir** abstrações. Quase todos os substantivos em nosso vocabulário representam uma **classe de objetos** compartilhando alguns conjuntos de atributos e comportamentos semelhantes.

3.1.2. Objetos ou Instâncias

Um **objeto** ou **instância** é um componente do mundo real que é mapeado no domínio do software. No contexto de um sistema baseado em computador, um objeto é tipicamente um **produtor** ou **consumidor** de informação ou itens de informação. Exemplos: máquinas, comandos, arquivos, displays, chaves, sinais, caracteres alfanuméricos, ou qualquer outra pessoa, lugar ou coisa. Quando um objeto é mapeado no domínio do software, ele consiste de uma estrutura de dados e processos (chamadas operações ou métodos) que podem legitimamente transformar a estrutura de dados.

3.1.3. Dados ou Estados

A parte **privada** de um objeto são sua **estrutura de dados** e o conjunto de operações (funções) para esta estrutura de dados.

Dados ou estados, então, são as informações relacionadas (variáveis) a um determinado objeto que alteram a forma que o objeto responder a um tipo particular de mensagem **ou** armazenam detalhes que descrevem o objeto.

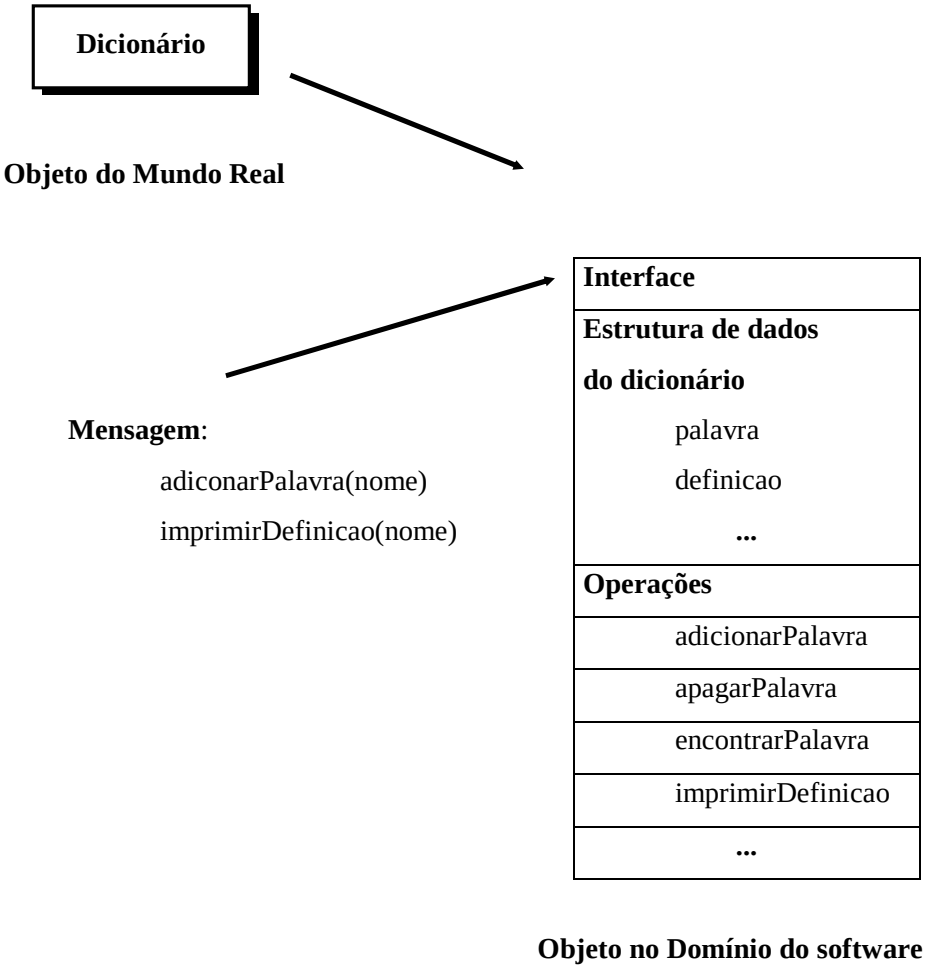
3.1.4. Métodos e Protocolos (Mensagens)

Um objeto também têm uma parte compartilhada que são sua **interface**. **Mensagens** "atravessam" a interface e especificam **qual** operação no objeto é desejada, mas **não como** é executada. O objeto que recebe a **mensagem** determina qual a **operação** requisitada **(ou método**) será implementada.

3.1.5. Exemplo

Observando a figura a seguir, um **objeto** do mundo real (um dicionário) é mapeado no domínio do software para um sistema baseado em computador. Neste domínio para dicionário existem estrutura de dados privados e operações relacionadas aos mesmos. Os **dados** do dicionário são compostos de palavras, um guia de pronuncia, e um ou mais definições. Uma figura ou um diagrama pode também estar contida nos dados. O objeto dicionário também

contém um conjunto de operações (ex.: adicionarPalavra, encontrarPalavra) - **métodos** - que podem processar os elementos da estrutura de dados descrita acima. Estes métodos são executados através de **mensagens** passadas ao objeto.



3.2. Propriedades

3.2.1. Encapsulamento de dados

Encapsulamento de dados ocorre quando combinamos uma **estrutura de dados** com **funções** (ações ou métodos) dedicadas a **manipular** estes dados. Com a união de **código** e **dados** em um objeto de uma classe a POO muda a forma de trabalhar com os dados. O acesso aos dados de um objeto será permitido somente por meio de **métodos** do próprio objeto.

Encapsulamento de dados, então por definição, temos quando um objeto com parte privada tem mensagens particulares para invocar processamentos adequados, ou seja, detalhes de implementação estão escondidos de todos os elementos do programa que são externos ao objeto. Objetos e suas operações estão fortemente conectados, ou seja, elementos de software (**dados e processos**) são agrupados juntos com um mecanismo de interface bem definido (**mensagens**).

3.2.2. Herança

Herança, como já mencionado, é a possibilidade de se aproveitar classes já existentes para a criação de novas, especificando somente as diferenças. Uma classe pode ser baseada em classes já existentes, herdando assim seus dados e funções.

Existem duas formas de se implementar herança: a **simples**, onde uma classe aproveita o que foi definido somente de uma classe, e a **múltipla**, onde uma classe pode reunir e aproveitar as definições de várias classes ao mesmo tempo.

A idéia de herança dá um grande impulso à POO, pois fornece de uma só vez recursos para **reutilizar** o que já foi feito e de modelar o mundo exterior: cada **classe** representa **grupos** de elementos e os **objetos** representam **elementos** que têm existência física individual.

3.2.3. Polimorfismo

É a habilidade de diferentes objetos responderem diferentemente a mesma mensagem. Os objetos agem em resposta as mensagens que recebem. A mesma mensagem pode resultar em ações completamente diferentes quando recebidas por objetos diferentes.

Polimorfismo é a característica presente nas linguagens orientadas a objeto, que consiste na possibilidade de um mesmo nome (**protocolo**) ser usado para identificar procedimentos (**métodos**) diferentes.

4. Definição de Classes em C++

Classes em **C++** estão fortemente relacionadas com estruturas (**struct)** do **C**. Uma estrutura em C é uma ferramenta poderosa para modelar dados. Usando estruturas um programador pode desenvolver estrutura de dados (objetos) que ficam bem próximos dos objetos do "mundo real". Por exemplo, um programador C desenvolvendo uma estrutura de dados para correios poderia fazer algo como segue:

```
struct Endereco {
  unsigned numeroRua;
  char* nomeRua;
  char* nomeCidade;
  char estado[3];          /* um byte extra para o caracter NULL */
  char CEP[10];            /* idem */
};
```

Uma vez definido a estrutura, o programador deve então definir o conjunto de "funções de acesso", com as quais pode-se acessar objetos do tipo **Endereco**. As **declarações** desse conjunto de funções são apresentadas a seguir junto com uma pequena função **main( )** para visualização.

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

/* Definição da estrutura de dados */
struct Endereco {
  unsigned numeroRua;
  char* nomeRua;
  char* nomeCidade;
  char estado[3];          /* um byte extra para o caracter NULL */
  char Cep[10];            /* idem */
};
        /* Declaração das funções de acesso */

        /* Criar um endereço */
void iniciaEndereco (struct Endereco* ptrEndereco, unsigned numRua,
                                        char* nomRua, char* nomCidade,
                                        char* est, char* cep);
        /* Apagar um endereço */
void apagaEndereco(struct Endereco* ptrEndereco);
        /* Procurar pelo nome da rua de  um endereço e o numero */
void buscaNomeRua(struct Endereco* ptrEndereco, char* nomRua);
        /* Procurar pelo cep de um endereço, cidade e estado */
void buscaCep(struct Endereco* ptrEndereco, char* cep);
        /* Retornar cep como long*/
unsigned long cepComoNumero(struct Endereco* ptrEndereco);

void main( ){
        struct Endereco end;
        char buffer[80];
```

```
                iniciaEndereco(&end, 107, “Brasil”, “Santana”, “PE”, “33353000”);

                buscaNomeRua(&end, buffer);
                printf("%s\n",buffer);
                buscaCep(&end, buffer);
                printf("%s\n",buffer);

                printf("Eis o cep como long %ld\n", cepComoNumero(&end));
                apagaEndereco(&end);
        }

/* Definicao das funcoes de acesso */
        void iniciaEndereco (struct Endereco* ptrEndereco, unsigned numRua,
                                        char* nomRua, char* nomCidade,
                                        char* est, char* cep)
        {
          int tamanho;
          ptrEndereco->numeroRua = numRua;
          tamanho = strlen (nomRua) + 1;
          if (ptrEndereco->nomeRua = (char*) malloc (tamanho))
                strcpy (ptrEndereco->nomeRua, nomRua);
          tamanho = strlen (nomCidade) + 1;
          if (ptrEndereco->nomeCidade = (char*) malloc (tamanho))
                strcpy (ptrEndereco->nomeCidade, nomCidade);
          strncpy (ptrEndereco->estado, est, 2);
          ptrEndereco->estado[2] = ‘\0’;
          strncpy (ptrEndereco->Cep, cep, 9);
          ptrEndereco->Cep[9] = ‘\0’;
        }

        void apagaEndereco (struct Endereco* ptrEndereco)
        {
          if (ptrEndereco->nomeRua) free (ptrEndereco->nomeRua);
          if (ptrEndereco->nomeCidade) free (ptrEndereco->nomeCidade);
        }

        void buscaNomeRua (struct Endereco* ptrEndereco, char* nome)
        {
          *nome = ‘\0’;
          itoa (ptrEndereco->numeroRua, nome, 10);
          strcat (nome, " ");
          strcat (nome, ptrEndereco->nomeRua);
        }

        void buscaCep (struct Endereco* ptrEndereco, char* nome)
        {
          *nome = '\0';
          strcat (nome, ptrEndereco->nomeCidade);
          strcat (nome, " ");
          strcat (nome, ptrEndereco->estado);
          strcat (nome, "\nCEP ");
          strcat (nome, ptrEndereco->Cep);
        }

        unsigned long cepComoNumero (struct Endereco* ptrEndereco)
        {
          return atol (ptrEndereco->Cep);
        }
```

Uma vez que o programa esteja completo, se a **estrutura** (Endereco) mudar, o programador deverá saber a localização das funções que devem ser modificadas e corrigidas. E mais, desde que o **C** procedural não limita o acesso aos dados de um objeto do tipo Endereco, o programador não pode ter certeza se outros módulos não serão afetados também.

Outro detalhe em relação ao novo tipo criado é que não é possível "ensinar" a nenhuma das funções existentes ou operadores como operar neste novo tipo. Cada nova estrutura deve ter um conjunto de funções de acesso com nomes diferentes das já existentes.

Uma vez que uma estrutura como Endereco é construída em C, é frequentemente necessário "**derivar**" novas estruturas que são subconjuntos do original. Por exemplo, nós poderiamos construir um endereço de negócios que consistiria de duas partes: o endereço da compania mais a caixa postal ou o número do departamento da pessoa na compania. A nova estrutura poderia ser definida assim em C:

```
struct NegocioEndereco{
  unsigned cxPostal;
  unsigned depto;
  struct Endereco endCompania;
};
```

Esta estrutura requer um conjunto novo de funções de acesso com novos nomes (duas funções não podem ter o mesmo nome em C).

4.1. Declaração de Classes

Nesta seção será apresentado como o C++ arruma alguns dos "problemas" com estruturas do C.

As funções de acesso podem ser definidas como parte da estrutura criando assim o encapsulamento. Estas **funções membros** tornam-se parte da estrutura tal como os **dados membros**.

Não somente a associação entre a estrutura de dados e suas funções de acesso são mais explicítas, mas parte da estrutura só é **acessível** pelas funções membros. Ou seja, o acesso aos **membros** da estrutura é extritamente controlado.

Desde que o C++ não deve modificar as regras de criação de uma **struct**, sem perder a compatibilidade com C, C++ definiu uma nova estrutura chamada **class**. O formato geral é mostreado a seguir:

```
class NomeDaClasse {
  private:
        unsigned dadoPrivado;
  public:
        unsigned dadoPublico;
        unsigned lerDadoPrivado( );
};
```

A Borland sugere **iniciar** nome de **variáveis** com letras **miniscúla** e nomes de **classes** com letra **maiuscúla**, e #defines com todas as letras maiuscúlas. Ela também sugere para variáveis com múltiplas palavras que todas após a primeira tenham a primeira letra em maiúsculo (Ex: int dataDeNasc).

A **declaração** da função membro **lerDadoPrivado( )** é um **protocolo** da classe e a **definição** da mesma é um **método**.

Classes adiciona duas palavras-chave novas: **private** e **public**. Membros declarados após a palavra-chave **private** são ditos membros privados (ou particulares) e os declarados após **public** são membros públicos. **Membros públicos** são como os membros de uma struct, eles são acessíveis para qualquer função. **Membros privados** são acessíveis somente para as **funções membro**. A declaração de um **objeto** de classe (processo também conhecido como **"instanciar uma classe"**) é semelhante a declaração de um objeto de uma estrutura. Em C++ o uso da palavra-chave **struct** ou **class** não é mais necessário na declaração, mas ainda é permitida. Desta forma as duas declarações abaixo são possíveis:

```
class NomeDaClasse exClass;
ou
NomeDaClasse exClass;
```

O acesso aos **membros públicos** da classe ocorre da mesma forma que ocorria na **struct**:

```
unsigned valor;
NomeDaClasse *ptrExClass = &exClass;

exClass.DadoPublico = valor;              // correto para acessar membro público
ptrExClass->DadoPublico = valor + 2;      // idem
...
```

```
valor = exClass.lerDadoPrivado( );
valor = ptrExClass->lerDadoPrivado( ) + 3;
```

4.2. Funções Membros

As funções membro de uma classe são parte integrante tal como os seus dados membros. As funções membro têm várias características especiais em C++. Uma delas é que somente as funções membro têm acesso aos **membros privados** de uma classe.

Estas funções podem ser **definidas dentro** da classe e **fora** delas. Por exemplo a função membro **lerDadoPrivado( )** pode ser escrita como:

```
class NomeDaClasse {
  private:
        unsigned dadoPrivado;
  public:
        unsigned dadoPublico;
        unsigned lerDadoPrivado( ) { return dadoPrivado; }
};
```

A mesma função, em vez de ser definida dentro da classe, pode ser definida separadamente como segue:

```
unsigned NomeDaClasse :: lerDadoPrivado( ) {
  return dadoPrivado;
}
```

A função membro definida dentro da classe é assumida como uma função **inline**. A segunda forma é a mais comum, ou seja, não é assumida como **inline** (a não ser que a palavra-chave seja usada).

O operador escopo **::** na segunda definição informa ao compilador C++, a qual classe a função **lerDadoPrivado( )** tem acesso aos dados, ou seja, a qual classe a função pertence.

Uma função membro é chamada **sobre uma instância** que pode ser vista como um argumento suplementar passado à função membro:

```
Ponto umPonto;              // umPonto é uma instância da classe Ponto
...
umPonto . desenha(50, 50);
```

Existe dois tipos de funções membro que são especiais. A primeira com o mesmo nome da classe é denominada **construtora**. Ela é chamada cada vez que uma instância da classe é criada. As construtoras são opcionais e somente usadas quando é necessário disparar um processo no momento de criar o objeto. A segunda com o mesmo nome da classe, mas precedida pelo símbolo (~), é denominada **destrutora** e executada quando o objeto é eliminado ou não faz

mais sentido a sua existência (ex.: quando uma função é encerrada os **objetos criados no seu interior** devem ser destruídos ao final da função).

4.3. Um exemplo - a classe Endereco

O exemplo escrito em C procedural é reescrito com classes em C++.

```
#include <string.h>
#include <stdlib.h>
#include <iostream.h>

/* Definição da  classe */
class Endereco {
  private:
          unsigned numeroRua;
          char* nomeRua;
          char* nomeCidade;
          char estado[3];
          char Cep[10];
  public:
          void iniciaEndereco(unsigned numRua,
                              char* nomRua, char* nomCidade,
                              char* est, char* cep);
          void apagaEndereco( );
          void buscaNomeRua(char* nomRua);
          void buscaCep(char* cep);
          unsigned long cepComoNumero( ) {          // inline por default
          return atol (Cep);
          }
};

void main( ){
          Endereco end;
          char buffer[80];

          end . iniciaEndereco(107, "Brasil", "Santana", "PE", "33353000");
          end . buscaNomeRua( buffer);
          cout << buffer << "\n";
          end . buscaCep( buffer);
          cout << buffer << "\n";
          cout << "Eis o Cep como long " << end.cepComoNumero( );
          end . apagaEndereco( );
}

/*Definições da funções membros declaradas */
void Endereco :: iniciaEndereco (unsigned numRua,
                                 char* nomRua, char* nomCidade,
                                 char* est, char* cep)
{
  int tamanho;
  numeroRua = numRua;
  tamanho = strlen (nomRua) + 1;
  if (nomeRua = (char*) malloc (tamanho))
          strcpy (nomeRua, nomRua);
  tamanho = strlen (nomCidade) + 1;
  if (nomeCidade = (char*) malloc (tamanho))
          strcpy (nomeCidade, nomCidade);
  strncpy (estado, est, 2);
  estado[2] = '\0';
  strncpy (Cep, cep, 9);
  Cep[9] = '\0';
```

```
}
inline void Endereco :: apagaEndereco ( )                    //  inline através da diretiva
{
  if (nomeRua)  free (nomeRua);
  if (nomeCidade) free (nomeCidade);
}

void Endereco :: buscaNomeRua ( char* nome)
{
  *nome = '\0';
  itoa (numeroRua, nome, 10);
  strcat (nome, " ");
  strcat (nome, nomeRua);
}

void buscaCep (char* nome)
{
  *name = '\0';
  strcat (nome, nomeCidade);
  strcat (nome, " ");
  strcat (nome, estado);
  strcat (nome, "\nCEP ");
  strcat (nome, Cep);
}
```

Podemos observar que o **acesso aos membros** da classe pelas funções membro são feitos diretamente através do nome dos mesmos, ficando mais claro a compreensão.

4.4. Contrutores e Destrutores

4.4.1. O que são e quando são chamados?

Geralmente, um objeto deve ser inicializado antes de ser usado. **Variáveis globais** são automaticamente **inicializadas com 0**, mas este valor frequentemente não é aceitável. Em C a função que declara um objeto atribui um valor a cada um dos seus membros. Os valores são atribuídos individualmente ou todos ao mesmo tempo através de uma função.

Em C++, entretanto este processo é inadequado. Para obter o máximo benefício do encapsulamento, a maioria dos dados membros de uma classe deve ser privado. Uma função que declara um objeto de uma classe não têm acesso para atribuir valores iniciais aos seus membros privados.

C também permite que objetos sejam inicializados através de uma função, tal como **iniciaEndereco( )**, e então invoca a função depois da declaração do objeto e antes do seu uso. Algo similar pode ser feito em C++ desde que a função de inicialização seja uma função membro, pois esta têm acesso aos dados privados. Para facilitar este processo, C++ permite ao

programador definir **métodos especiais** chamados **construtores**. Um construtor é uma função membro especial que automaticamente é invocada sempre que um objeto de um tipo é criado.

Construtores recebem sempre o **mesmo nome da classe**, podem conter argumentos e **não têm valor de retorno** (nem mesmo void). Uma classse pode conter vários construtores com número ou tipo de argumentos diferentes.

Um contrutor para a **classe Endereco** pode ser declarado dentro da mesma como segue:

```
Endereco(unsigned numero, char* rua, char* cidade, char* estado, char* cep);
```

Os argumentos são passados para o construtor quando a classe é instanciada. Para invocar o construtor acima, o objeto **end** pode ser declarado como segue:

```
Endereco end (107, "Brasil", "Santana", "PE", "33353000");
```

C++ também permiti o programador definir uma função membro que é chamada quando o objeto é destruído. Este método, chamado **destrutor**, recebe também o **mesmo nome da classe precedido pelo símbolo** ~. Cada classe pode possuir somente um único **destrutor** que **não pode conter argumentos e nem valor de retorno** (nem mesmo void).

Um destrutor para a classe **Endereco** é declarado como segue:

```
~Endereco( );
```

O formato do construtor e destrutor são mais flexíveis que funções de inicialização e finalização que programadores geram em código C, pois eles são automaticamente invocados na criação e destruição de objetos. Objetos globais e objetos locais declarados como estáticos (**static)** são criados quando um programa começa a execução mas antes que a função **main( )** toma o controle. Estes objetos são destruídos quando o programa termina como resultado da chamada da função **exit()** ou retorno da **main( )**.

Construtores e destrutores podem ser definidos pelo programador, caso contrário o compilador C++ irá gerar construtores e destrutores default.

4.4.2. Construtor default

O construtor default não têm argumento (tipo **void**). Este construtor é usado quando um objeto é declarado sem valores de inicialização. Seja o exemplo a seguir:

```
#include <iostream.h>

class Identidade {
  private:
        char* sobreNome;
        char* primeiroNome;
```

```
  public:
        Identidade( ) {
          cout << "Inicializacao default" << endl;
          sobreNome = (char*) 0;
          primeiroNome = (char*) 0;
        }
};

Identidade instanciaGlobal;
void main( ) {
  cout << "Inicio da funcao main( )" << endl;
  Identidade instanciaLocal;
  cout << "Fim da funcao main( )" << endl;
}
```

A execução deste programa gera as seguintes mensagens:

Inicializacao default

Inicio da funcao main( )

Inicializacao defautl

Fim da funcao main( )

O construtor default é invocado antes da execução da **main( )** para inicialização de **instanciaGlobal** e dentro da função **main( )** para a de **instanciaLocal**.

4.4.3. Construtor por cópia (construtor CI)

Um construtor pode ter quase qualquer tipo de argumentos. Um construtor pode até mesmo ter um objeto do mesmo tipo. Este construtor específico têm a seguinte declaração:

```
MinhaClasse :: MinhaClasse (MinhaClasse& instancia);
```

Este costrutor cria um novo objeto copiando os dados membros da instância passada. Usado para copiar ou inicializar um objeto através dos dados de outro objeto, este construtor é chamado de **construtor por cópia.**

O construtor por cópia para a classe **Identidade**, poderia ser definido (considerando que estivesse declarado) da seguinte forma:

```
Identidade :: Identidade (Identidade& fonte) {
  cout << "Copiando " << fonte.primeiroNome << "  " << fonte.sobreNome << endl;
  sobreNome = fonte.sobreNome;
  primeiroNome = fonte.primeiroNome;
}
```

Então para o programa a seguir:

```
void main( ) {
  Identidade nomeCompleto("Marques", "Richard");   // supondo a existência deste construtor
  Identidade xerox1 = nomeCompleto;
```

```
  Identidade xerox2( nomeCompleto);
}
```

geraria as seguintes mensagens:

Inicializando Richard Marques

Copiando Richard Marques

Copiando Richard Marques

O construtor por cópia e tão importante quanto o construtor default. O construtor CI é usado pelo C++ toda vez que for necessário a cópia de um objeto, algo que ocorre com mais frequência que muitos programadores "novatos" possam esperar.

Se o programador não definir um construtor por cópia para a classe, o compilador C++ criará um. O contrutor por cópia default faz uma cópia membro a membro de cada dado do objeto fonte para o objeto destino.

4.5. O ponteiro this (autoreferência)

A organização interna de uma classe mostra como o C++ sabe qual função deve ser invocada quando um método é referenciado. Quando um programador invoca um método tal como **end.buscaCep( )**, o compilador C++ procura **Endereco::** pois este é o tipo de **end** e chama **Endereco::buscaCep( )**.

Considere o código a seguir:

```
Janela umaJanela;
Janela outraJanela;
umaJanela.posicionaCursor(40, 15);
```

Invocando **umaJanela.posicionaCursor( )** alterá a posição do cursor da instância **umaJanela**, mas não têm efeito algum sobre **outraJanela**. A função **Janela::posicionaCursor( )** têm que saber sobre qual objeto ela será invocada.

Todo **método não estático** (métodos estáticos serão abordados em 4.7) de uma classe **recebe** um primeiro argumento escondido que contém o **endereço sobre o qual irá operar**. Este ponteiro escondido não aparece na lista de argumentos e é conhecido dentro da função como **this**.

Vamos assumir que **posicionaCursor( )** é definido como segue:

```
void Janela::posicionaCursor (int xLoc, int yLoc) {
  cursorXLoc = xLoc;
  cursorYLoc = yLoc;
```

```
  escrevaParaJanela(“ ”,xLoc, yLoc);
}
```

Aqui, **this** é do tipo **Janela*** e pode apontar para **umaJanela** ou **outraJanela**, dependendo sobre qual intância ela foi chamada. O uso explicíto do ponteiro **this** é permitido mas usualmente não é necessário, pois na chamada ele já aponta para o objeto em uso. A função anterior poderia ser definida como segue:

```
void Janela::posicionaCursor (int xLoc, int yLoc) {
  this -> cursorXLoc = xLoc;
  this -> cursorYLoc = yLoc;
  this -> escrevaParaJanela(“ ”,xLoc, yLoc);
}
```

O uso explicíto do ponteiro **this** é necessário quando o método deve retornar o endereço do objeto sobre o qual o método foi chamado. Considere a implementação parcial de uma classe de lista encadeada, em particular o método para adicionar um objeto a lista:

```
class ListaEncadeada {
  private:
        ListaEncadeada* proximo;
  public:
        void adicioneObjeto (ListaEncadeada* anterior) {
          proximo = anterior -> proximo;
          anterior - > proximo = this;
        }
};
```

Na segunda linha de **adicioneObjeto( )**, a variável **anterior->proximo** necessita apontar para o objeto sobre o qual o método foi chamado que é possível somente através do ponteiro **this**.

Obs.: Um **método** não só **têm acesso** aos **membros privados** do objeto sobre o qual foi chamado (os quais são apontados por **this**), **mas também aos membros privados de outros objetos da mesma classe**.

4.6. Funções e Classes amigas

A restrição que somente funções membro têm acesso aos membros privados algumas vezes é muito rigída. Existem ocasiões que **funções não-membro** devem ter permissão de acesso a membros privados de um determinada classe. C++ permite que o programador forneça este acesso através da palavra chave **friend**.

A classe **Exemplo,** a seguir, têm o membro privado **dadoPrivado**, o qual somente pode ser acessado pelo seu método **umMetodo( )** e pela função **umAmigo( )** declarada como **friend**.

```
class Exemplo {
        friend void umAmigo (Exemplo&, int);
  private:
        int dadoPrivado;
  public:
        void umMetodo (int);
};
```

As definições para as duas funções diferem ligeiramente:

```
void Exemplo :: umMetodo (int x) {
  dadoPrivado = x;
}

void umAmigo(Exemplo& obj, int x) {
  s.dadoPrivado = x;
}
```

Note que **umMetodo( )** é declarado como um membro de Exemplo, enquanto **umAmigo( )** não. Como uma função membro, **umMetodo( )** têm um ponteiro **this**. Desde que **umAmigo( )** não têm um ponteiro **this**, o programador deve incluir o endereço do objeto para poder acessar seus membros.

A delaração de **friend** deve ocorrer dentro da classe até mesmo na "parte" que contém os membros privados.

Outros tipos de funções podem ser amigas também. Um método de uma classe pode ser declarada como amiga (**friend**) para outra classe, como exemplo que segue:

```
class Casse2;

class Classe1 {
  private:
        int dadoPrivado;
  public:
        void umMetodo(Classe2&);
};

class Classe2 {
        friend void Classe1::umMetodo(Classe2&);
  private:
        int dadoPrivado;
  public:
        void umMetodo(int);
};

void Classe1::umMetodo(Classe2& x) {
  x.dadoPrivado = dadoPrivado;
}
```

Note que **Classe1::umMetodo( )** têm acesso aos dados privados das duas classes. Aos da **Classe1** através do ponteiro **this** e da **Classe2** através do argumento passado **x**. Note também que

a Classe2 é declarada na primeira linha sem ser definida. **Classe2** deve ser declarada antes de poder ser usada na declaração da função membro **Classe1::umMetodo( )**.

Amigos não são exclusividades de funções. Uma **classe** pode ser declarada como **friend** de **outra classe**. Declarar uma outra classe como amiga é equivalente a delarar todas suas funções como amigas.

```
class OutraClasse;
class MinhaClasse {
  private:
        // ...
  public:
        // ...
        int isA(void);
        friend int tudoOk(MinhaClasse&);
        friend class OutraClasse;
};
```

4.7. Membros estáticos

Variáveis estáticas são armazenadas juntas com as variáveis globais (não na pilha) e têm escopo de variável local. Uma variável declarada como **static** dentro de uma função mantém seu valor entre uma chamada e outra da função.

Membros de uma classe também podem ser declarados como estáticos. Um membro estático mantém o mesmo valor para **todos** os objetos. Entretanto, membros estáticos não podem ser armazenados com outros membros da classe.

Quando um **dado membro** é declarado como estático (**static**) existirá uma única cópia desta variável a qual será partilhada por todos os objetos da classe durante a execução de um programa.

Por analogia, quando uma **função membro** é declarada estática esta não se aplicará a nenhum objeto específico da classe, mas poderá agir sobre os membros estáticos. Métodos estáticos **não têm** ponteiro **this**, portanto não têm acesso aos membros normais simplesmente através de seus nomes, como ocorre normalmente.

```
class ContaInstancias {
  private:
        static int numInstancias;                    // não pode atribuir valor aqui
  public:
        ContaInstancias( ) { cout << "Construtor"; numInstancias++; }
        static void quantasInstancias( ) { cout << "\n Temos " << numInstancias << "instancia(s)"; }
};

int ContaInstancias::numInstancias = 0;              // forma de atribuir valor para membro estático

void main( ) {
```

```
  ContaInstancias::quantasInstancias( );
  ContaInstancias a;
  a . quantasInstancias( );
  ContaInstancias b;
  b . quantasInstancias( );
}
```

O programa acima geraria as seguintes mensagens:

Temos 0 instancia(s)

Construtor

Temos 1 instancia(s)

Construtor

Temos 2 instancia(s)

No exemplo os endereços das instâncias a e b não são avaliados para invocar a função **quantasIntancias( )**. Tudo que é necessário para o compilador Turbo C++ para invocar a função membro estática é o tipo da instância.

5. Herança em C++

Para reutilizar classes preexistentes (economizando tempo de programação considerável), o C++ permite que uma classe use as características de outra classe (herde).

Quando construimos uma classe usando outra existente, a nova classe herda as características da classe existente. As características incluem os **dados** e as **funções**, assim como seu acesso (público ou privado). Ao ler sobre programação orientada a objetos, encontraremos os termos **classe base (superclasse)** e **classe derivada (subclasse)**. A classe base é a classe original cujas características estão sendo herdadas. A classe derivada é a que está sendo criada (está herdando). Uma classe base pode ser utilizada por muitas classes diferentes.

Ao derivar uma classe de outra em C++ o seguinte formato será utilizado:

```
class classeDerivada: <modificador_de _acesso> classeBase {
  private:
        // membros privados da classe derivada
  public:
        // membros publicos da classe derivada
};
```

O **modificador de acesso** é usado para modificar o tipo de acesso aos membros herdados como é mostrado na tabela a seguir:

Tipo de acesso na classe base	Modificador de acesso	Tipo de acesso na classe derivada
public	public	public
private	public	sem acesso
protected	public	protected
public	private	private
private	private	sem acesso
protected	private	private

Os acessos do tipo privado (**private**) e público (**public**) já foram abordados, porém existe mais um tipo: **protected**. Os membros seguidos da palavra chave **protected** (protegido) podem ser acessados somente pelas funções membro da mesma classe e pelas funções membro das classes derivadas, ou seja, a classe e suas subclasses têm acesso aos dados protegidos. Portanto o formato geral para declarar uma classe, que terá subclasses, toma a seguinte forma:

```
class NomeDaClasse {
  private:
        unsigned dadoPrivado;
  protected:
        unsigned dadoProtegido;
  public:
        unsigned dadoPublico;
        unsigned lerDadoPrivadoOuProtegido( );
};
```

5.1. Herança Pública

Seja a definição da classe conta corrente em C++:

```
class ContaCorrente {
  private:
        static int          totalDeContas;
        char                senha[10];
  protected:
        int                 numero;
        double              valor;
  public:
        ContaCorrente ( double depositoInicial, char *senhaUsuario);
        ~ ContaCorrente ( );
        void deposito (double quantia);
        void retirada (double quantia);
};
```

Seguindo o exemplo, suponha a necessidade de desenvolver uma classe de conta corrente que seja remunerada diariamente. Para aproveitar as características da classe conta corrente para criar uma classe conta remunerada **com herança pública** procederiamos da seguinte forma:

```
class ContaRemunerada : public ContaCorrente {
  private:
        double indice;
  public:
        void reajusta (void) {
```

```
            valor *= indice;
        }
};
```

Os **métodos** da classe conta remunerada **têm acesso** aos membros **protegidos e públicos** da classe conta corrente, **mas não aos privados**.

A partir de uma instância da classe conta remunerada pode-se utilizar os membros públicos da classe conta corrente (dados e funções membro).

5.2. Herança Privada

Uma outra forma de se aproveitar as características da classe conta corrente para criar uma classe conta remunerada seria utilizar **herança privada** como segue:

```
class ContaRemunerada : private ContaCorrente {
  private:
        double indice;
  public:
        void reajusta (void) ;
};
```

Os **métodos** da classe conta remunerada **têm acesso** aos membros **protegidos e públicos** da classe conta corrente, **mas não aos privados**.

A partir de uma instância da classe conta remunerada **não** se pode utilizar os membros públicos da classe conta corrente (dados e funções membro), ou seja, **não há visibilidade** da classe conta corrente a partir de uma instância da classe conta remunerada.

5.3. Herança Múltipla

Uma classe derivada pode ser construída de várias classes diferentes. Quando uma classe é construída a partir de duas ou mais classes existentes, o termo herança múltipla é utilizado.

Se prestarmos a atenção, a classe conta corrente está nos padrões suiços, ou seja, ela é numerada mas não há qualquer referência ao dono da conta. Para adequá-la aos padrões brasileiros pode-se criar uma classe cliente e uma classe conta brasil que associa um cliente a cada conta utilizando **herança múltipla**.

```
class Cliente {
  private:
        char    nome[100];
        char    CPF[15];
  public:
```

```
        Cliente (char* nomeCliente);
        void mostraCliente (void);
};

class ContaBrasil : public Cliente, public ContaCorrente {
  private:
        double    movimentoMedio;
  public:
        ContaBrasil(char* nomeCliente, double valor);
};
```

Desta forma a classe conta brasil terá todas as características da classe conta corrente, ou seja, poderá fazer movimentações financeiras com as funções membro **deposito( )** e **retirada( )** além de ter acesso aos dados do cliente através da função **mostraCliente( )**.

5.4. Inicializando classes derivadas

Classes derivadas podem ter construtores e destrutores. Os construtores de uma classe derivada podem ser definidos utilizando construtores de classes base. Uma nova sintaxe é necessária para invocar o construtor base sem criar um novo objeto. Por outro lado se o construtor for invocado de dentro do construtor da classe derivada, um novo objeto poderá ser criado em vez de inicializar o objeto existente.

```
class ClasseDerivada : public ClasseBase {
  public:
        ClasseDerivada ( ) : ClasseBase ( ) {
                // definição do construtor da classe derivada
        }
...
```

Invocando o construtor base desta forma, permite **ClasseBase( )** inicializar os membros herdados. O restante do construtor **ClasseDerivada( )**, inicializa somente os novos membros.

A definição para o construtor da classe conta brasil poderia ser definido da seguinte forma:

```
ContaBrasil :: ContaBrasil(char* nomeCliente, double valor) :
                Cliente(nomeCliente) ,          // chama o construtor base Cliente( )
                ContaCorrente(valor,0) {        // chama o construtor base ContaCorrente( )
                  movimentoMedio = 0;
                }
```

6. Polimorfismo em C++

6.1. Sobrecarga de Funções Membro

Uma subclasse pode conter um método com mesmo nome de um método da classe base. Uma função membro na classe derivada **sobrecarrega** uma função membro com **mesmo nome** na classe base, mas a função da classe base ainda pode ser acessada atravé do operador escopo (**::**).

Quando utiliza-se sobrecarga de funções, o compilador C++ escolhe qual função deve ser utilizada através da comparação entre as declarações das funções. Se, entretanto, a sobrecarga for de funções membro o compilador verifica sobre qual instância a função foi chamada. Considere o exemplo a seguir:

```
class Ponto {
  private:
        int xLoc, yLoc;
  public:
        Ponto (int initX, int initY) { xLoc = initX; yLoc =initY; }
        void mostre ( ) { cout << "Eis X: " << xLoc << " e Y: " << yLoc << "\n"; }
};

class Circulo : public Ponto {
  private:
        int raio;
   public:
        Circulo (int initX, int initY, int initR) : Ponto (initX, initY) { raio = initR; }
        void mostre ( ) {
          Ponto::mostre ( );                    // chama mostre ( ) da classe Ponto
          cout << "Eis o Raio: " << raio;
        }

void main ( ) {
  Ponto umPto (20,30);
  Circulo umCirc (10,40,6);

  cout << "Instancia umPto\n";
  umPto.mostre ( );          // chama mostre ( ) da classe Ponto
  cout << "Instancia umCirc\n";
  umCirc.mostre ( );         // chama mostre ( ) da classe Circulo
}
```

O resultado deste programa seria o seguinte:

Instancia umPto

Eis X: 20 e Y: 30

Instancia umCirc

Eis X: 10 e Y: 40

Eis o Raio: 6

No corpo da definição de **Circulo::mostre( )** pode-se notar a chamada da função **Ponto::mostre( )**. Esta sintaxe mostra um outro uso comum do operador escopo, ou seja, o operador escopo (**::**) é usado para especificar uma função em outro escopo em vez da função de mesmo nome no escopo corrente.

Note que a sobrecarga de funções membro, tal como **mostre( )**, permite que diferentes objetos responderem de maneiras diferentes à mesma mensagem, ou seja, permitem o polimorfismo.

6.2. Funções Virtuais (late-binding e early-binding)

Definindo classes para representar figuras tais como pontos, círculos, linhas, quadrados, arcos e outros, pode-se definir uma função membro para cada, a qual mostrará o objeto na tela. No novo método de programação, ou seja, POO, pode-se dizer qua todas as figuras têm a habilidade de se mostrar na tela.

A diferença entre cada tipo de objeto é a forma com que este deve se mostrar na tela. Um ponto é desenhado com uma função que plota um ponto que necessita somente da localização X, Y e talvez o valor da cor. Um circulo necessita de uma rotina gráfica mais complexa para se mostrar, utilizando não somente a localização como também o raio. E um arco necessita, ainda mais, um ângulo inicial e final, e um algoritmo diferente de plotagem. A mesma situação ocorre para esconder, arrastar, mover e outras manipulações básicas com figuras.

Seja o código anterior alterado para operações gráficas:

```
#include <graphics.h>


class Ponto {
  private:
        int xLoc, yLoc;
  public:
        Ponto (int initX, int initY) {
          xLoc = initX; yLoc =initY;
        }
        void mostre ( ) {
          putpixel (xLoc, yLoc, getcolor ( ) ); // usa cor default
        }
        void esconde ( ) {
          putpixel (xLoc, yLoc, getbkcolor ( ) );        // usa a cor de fundo para esconder
        }
        void moverPara (int newX, int newY ) {
          esconde ( );                                   // esconde o objeto ponto
          xLoc = newX;                                   // modifica as coordenadas X e Y
```

```
            yLoc = newY;
            mostre ( );                          // mostra o objeto ponto na nova localização
          }
};

class Circulo : public Ponto {
  private:
          int raio;
   public:
          Circulo (int initX, int initY, int initR): Figura (initX, initY) { raio = initR; }
          void mostre ( ) {
            circle (xLoc, yLoc, raio);           // desenha o circulo com a cor corrente
          }
          void esconde ( ) {
           unsigned int corTemp;                 // para salvar a cor corrente
            corTemp = getcolor ( );
            setcolor ( getbkcolor ( ) );         // coloca a cor de fundo como corrente
            circle (xLoc, yLoc, raio);           // desenha com a cor de fundo para esconder
            setcolor (corTemp);                  // restaura a cor corrente anterior
          }
          void moverPara (int newX, int newY ) {
            esconde ( );                         // esconde o objeto circulo
            xLoc = newX;                         // modifica as coordenadas X e Y
            yLoc = newY;
            mostre ( );                          // mostra o objeto ponto na nova localização
          }
};


void main ( ) {
  Ponto umPto (20,30);
  Circulo umCirc (10,40,6);

  int driver = DETECT, modo;
  initgraph (&driver, &modo, "..\\bgi");


  umPto.mostre ( );                              // mostra umPto na tela
  umPto.moverPara (10,40);                       // move umPto para outra posição
  umCirc.mostre( );                              // mostra umCirc
  umCirc.moverPara (20,30);                      // move umCirc para outra posição
}
```

Em cada classe existe uma função membro **mostre ( )**, porém um ingrediente essencial está faltando. Módulos gráficos baseados em classes já existentes e funções membro necessitariam de alterações de código e recompilações cada vez que uma nova classe de figura fosse incluída com sua própria função membro **mostre ( )**. A razão é que o compilador C++ reconhece qual **mostre ( )** foi referenciado através de uma das três formas abaixo:

1. Diferencia através dos argumentos passados, ou seja, **mostre (int, char)** não é a mesma função que **mostre (char*, float)** por exemplo.
2. Diferencia através do operador de escopo, ou seja, **Circulo::mostre** é diferente de **Ponto::mostre**.

3. Diferencia sobre qual objeto a mensagem foi passada, ou seja, *umCirc.mostre( )* invoca **Circulo::mostre**, enquanto *umPto.mostre( )* invoca **Ponto::mostre**.

Todas essa resoluções de funções são feitas em tempo de compilação. Este mecanismo é conhecido como ***early binding*** (ligação cedo) ou ***static binding*** (ligação estática).

Um pacote gráfico típico deveria fornecer ao programador as definições de classes em arquivos cabeçalhos (**.h**) juntos com os arquivos código (**.obj** ou **.lib**). Com as restrições da ligação estática, o programador não pode adicionar novas classes e desenvolver novas funções para extender o pacote facilmente. O compilador C++ oferece um mecanismo flexível para resolver este problema: ***late binding*** (ligação tarde) ou ***dynamic binding*** (ligação dinâmica) através de funções membro chamadas de **funções virtuais**.

A chamada da função virtual é solucionada em **tempo de execução**, ou seja, a decisão sobre qual função **mostre( )** é chamada pode ser adiada até o tipo de objeto envolvido ser conhecido.

Consirere a função membro **Circulo::moverPara** mostrado anteriormente:

```
void Circulo::moverPara ( int newX, int newY) {
  esconde ( );
  xLoc = newX;
  yLoc = newY;
  mostre ( );
}
```

Note a similaridade desta definição com **Ponto::moverPara** que se encontra definida na classe base de **Circulo** - a classe **Ponto**. De fato, o nome da função, o numero e tipo de argumentos, e o corpo da função são idênticos. Quando o compilador encontra duas chamadas de funções usando o mesmo nome, ele diferencia através de uma das três formas já comentadas.

Como as duas funções **moverPara( )** não apresentam nenhuma diferença, porque não deixar a classe **Circulo** herdá-la? A razão é que as funções **mostre( )** e **esconde( )** chamadas em **Circulo::moverPara** não são as mesmas **mostre( )** e **esconde( )** chamadas em **Ponto::moverPara**. Herdar a função **moverPara( )** da classe **Ponto** chamaria as funções **mostre( )** e **esconde( ) erradas** para mover um círculo, devido a ligação em tempo de compilação (***early binding***). Para podermos **herdar** a função **Ponto::moverPara** devemos declarar as funções **mostre( )** e **esconde( )** como **virtuais**. Portanto na nova declaração das classes ficariam da seguinte forma:

```
class Ponto {
  private:
        int xLoc, yLoc;
```

```
  public:
        Ponto ( );
        Ponto (int initX, int initY) ;
        virtual void mostre ( ) ;
        virtual void esconde ( );
        void moverPara (int newX, int newY ) ;
};

class Circulo : public Ponto {
  private:
        int raio;
   public:
        Circulo (int initX, int initY, int initR);
        void mostre ( );
        void esconde ( );
};
```

As definições das funções membros foram omitidas, pois seriam idênticas as já mostradas. Como as funções **mostre( )** e **esconde( )** foram declaradas virtuais, a função **Ponto::moverPara** pode ser herdada, pois o compilador solucionará qual função **mostre( )** e **esconde( )** será chamada em tempo de execução (***late binding***).

Observações:

- Somente funções membro podem ser declaradas como virtuais.

- Uma vez que a função foi declarada como virtual, nas subclasses ela é automaticamente virtual também, a não ser que tenha alterações nos tipo de retorno ou nos argumentos. No exemplo acima as funções **mostre( )** e **esconde( )** são virtuais na classe **Circulo** também.

Para saber mais:

Manual do Usuário Borland C++ 3.0 - Cap 4 “Object-oriented programming with C++”

7. Generalização da Noção de Herança

7.1. Classe Abstrata

Uma classe abstrata é uma classe com pelo menos uma função puramente virtual**. Uma função virtual é especificada como pura quando é igualada a zero**.Uma classe abstrata pode ser usada somente como classe base para outras classes. Nenhum objeto de uma classe abstrata pode ser criado, ou seja, **a classe abstrata não pode ser instanciada**. Uma classe abstrata **não** pode ser usada como tipo de argumento ou como valor de retorno de função. Por exemplo:

```
class Figura {
  private:
        Ponto centro;
  public:
        Ponto onde( ) { return centro; }
        void mover(Ponto local) { centro = local; desenhe( ); }
        virtual void rodar (int) = 0;          // função puramente virtual
        virtual void desenhar ( ) = 0;         // função puramente virtual
        virtual void esconder ( ) ;
        ...
};


Figura x;                  // Erro: não é possível instanciar classe abstrata
Figura* ptrFig;            // Ok: ponteiro para classe abstrata é possível
Figura func1(void);        // Erro: classe abstrata não pode ser tipo de retorno
int func2(Figura s);       // Erro: classe abstrata não pode ser tipo de argumento
Figura& func3(Figura&);    // Ok: referências para tipo de argumentos ou retorno são possíveis
```

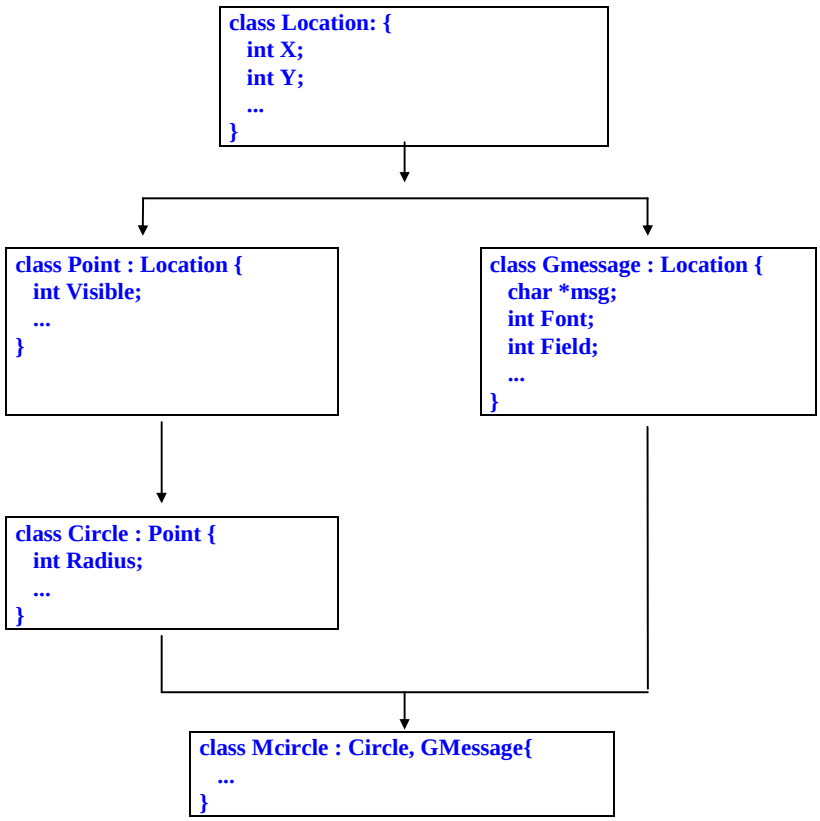
Suponha que uma classe **D** seja derivada imediata de uma classe abstrata **B** (classe base). Então para cada função puramente virtual em **B**, se **D** não providenciar uma definição para a mesma, esta função membro transforma-se em puramente virtual em **D**, e **D** também será uma classe virtual. Seja o exemplo:

```
class Circulo : public Figura {
  private:
        int raio;
  public:
        void rodar (int) { }        // função virtual definida: nenhuma ação para rodar um circulo
        void desenhar ( );          // Circulo::mostrar deve ser definida em algum lugar
};
```

Como a função **rodar( )** foi definida (mesmo não tendo código algum) na classe **Circulo** e **desenhar( )** foi redeclarada, a classe **Circulo** pode ser instanciada, ou seja, não é uma classe virtual.

7.2. Um exemplo gráfico

A **Borland** fornece vários exemplos para explicar vários conceitos envolvidos na programação orientada a objetos em C++. Um dos exemplos mostra como começar a criar uma pacote gráfico. A estrutura de classes do projeto "**mcircle.prj"** (fornecido pela Borland, que é apresentado no Adendo A) é apresentado abaixo:



A unidade fundamental num gráfico é um simples ponto na tela (**um pixel**). Uma classe **Location** que define um localização através de sua localização X e Y, um construtor para criar e iniciar a localização de um ponto. Esta classe servirá como base para todas as outras classes. Antes de desenhar qualquer coisa, entretanto, deve-se distinguir se o "pixel" está ligado ou desligado. A classe **Point** define um pixel herdando os dados membros de **Location** e adicionando o dado membro **Visible**, que indica se o pixel está ligado ou desligado. A classe **GMessage**, subclasse de **Location**, mostra uma sequência de caracteres (*string*) na tela. A classe

Circle define um circulo herdando os dados de **Location** e **Point,** e adicionado o dado membro **Radius**. Por fim, a classe **MCircle** através de herança múltipla herda as características de **GMessage** e **MCircle**, ou seja, mostra um círculo em conjunto com uma mensagem. Os programas deste projeto encontram-se no adendo.

7.3. Classes Base Virtuais

Com herança múltipla, a classe base não pode ser especificada mais de uma vez na classe derivada:

```
class B { ... };
class D : B, B { ... };          // ERRO!
```

Entretanto, uma classe base pode ser pode derivada indiretamente mais de uma vez:

```
class X : public B { ... };
class Y : public B { ... };

class Z : public X, public Y { ... };      //Ok!
```

Neste caso, cada objeto da classe Z herdará duas vezes as características da classe B. Se isso causar problemas,a palavra chave **virtual** pode ser adicionada ao modificador de acesso a classe base:

```
class X : virtual public B { ... };
class Y : virtual public B { ... };

class Z : public X, public Y { ... };
```

Como classe **B** agora é uma **classe base virtual** então a classe **Z** herdará somente uma vez as características da classe **B**.

7.4. Compatibilidade entre objetos da classe de base e da classe derivada

Na programação orientada a objeto, uma instância (objeto) da classe derivada é também instância da classe de base.

Em C++, pode-se atribuir:

- uma instância da classe derivada a uma instância da classe base.

- um apontador (ou referência) de uma instância da classe derivada a um apontador (ou referência) de uma instância da classe de base.

```
class Ponto {
  private:
        int xLoc, yLoc;
  public:
        Ponto (int initX, int initY) ;
  ...
};

class Circulo : public Ponto {
  private:
        int raio;
   public:
        Circulo (int initX, int initY, int initR);
   ...
};

void main( ) {
  Ponto         pto(10,10);
  Circulo       circ(20,20,5);

  pto = circ;                           // ok!
  circ = pto;                           // ERRO!

  Ponto         *ptrP;
  Circulo       *ptrC;

  ptrP = ptrC;                          // Ok!
  ptrC = ptrP;                          // ERRO!
  ptrC = (Circulo *) ptrP;              // Ok! Mas... PERIGOSO!

  Ponto         &refP = pto;
  Circulo       &refC = circ;


  refP = refC;                          // Ok!
  refC = refP;                          // ERRO!
}
```

8. Objetos Especiais

8.1. Array de Objetos e Objetos Dinâmicos

Como ocorre com outros tipo de dados, pode-se declarar ponteiros para classes e vetores de objetos de classe:

```
Ponto Origem;                   // declara o objeto Origem do tipo Ponto
Ponto Linha[80];                // declara um vetor de 80 objetos do tipo Ponto
Ponto *ptrPonto;                // declara um ponteiro para um objeto do tipo ponto
ptrPonto = &Origem;             // ptrPonto aponta para o objeto Origem
ptrPonto = Linha;               // ptrPonto aponta para o objeto Linha[0]
```

C++ permite usar as funções de alocação dinâmica do C tal como **malloc**. Entretanto, como já foi comentado, C++ inclue extensões poderosas que tornam a alocação e desalocação de memória mais simples e consistente. Mais importante ainda, é que permitem que construtores e destrutores sejam chamados:

```
Ponto *pto1     = new Ponto(15,30);     // aloca espaço para um objeto ponto
Ponto &pto2     = *new Ponto(10,25);    // idem
Ponto *ptos     = new Ponto[10];        // aloca espaço para 10 objetos ponto

delete pto1;                            // desaloca um objeto ponto
delete[10] ptos;                        // desaloca 10 objetos ponto
```

8.2. Objetos que são objetos de outras classes

Seja a classe **Ponto** abaixo:

```
class Ponto {
  private:
        int x, y;
  public:
        Ponto ( ) { x = y =0; }
        Ponto (int a, int b) { x =a; y =b; }
        int  igual (Ponto &p) { return (x == p.x) && (y == p.y); }
};
```

A classe **Reta** definida abaixo, contêm dois membros que são instâncias da classe **Ponto**. O contrutor da classe **Reta** neste caso, deve chamar explicitamente o construtor para os pontos.

```
class Reta {
  private:
        Ponto p1, p2;
  public:
        Reta ( ) : p1 (0, 0) , p2 (0, 5){
          cout << "Construtor default: p1(0, 0) e p2(0, 5)"
         }
```

```
        Reta (int x1, int y1, int x2, int y2) : p1(x1, y1) , p2(x2, y2) {
          if (p1.igual (p2) ) cout << "Erro: Pontos coincidentes!"
        }
};
```

Uma classe não pode ter como membro uma instância dela mesma, mas pode ter como membro um apontador do mesmo tipo.

```
class Matriz {
  Matriz A;        // Erro!
  ...
};
```

```
class Matriz {
  Matriz *A;       // Ok!
  ...
};
```

9. Sobrecarga de Operadores

C++ permite ao programador redefinir a ação de vários operadores, de modo que realizem funções específicas quando usados com objetos de uma classe particular. Considere a classe Complex:

```
class Complex {
        double real, imag;                  // privado por default
  public:
        ...
        Complex ( ) { real = imag = 0; }          // construtor inline
        Complex (double r, double i = 0) {        // idem
          real = r;
          imag = i;
        }
        ...
};
```

Para fazer a soma de complexos uma função tal como:

```
Complex somaComplex (Complex c1, Complex c2);
```

poderia ser desenvolvida, mas seria mais natural se fosse possível escrever:

```
Complex p1(0,1), p2(1,0), p3;
p3 = p1 + p2;
```

ao invés de:

```
p3 = somaComplex(p1,p2);
```

Utilizando a palavra-chave **operator,** o operador **+** pode facilmente ser sobrecarregado incluindo a declaração a seguir dentro da classe Complex:

```
Complex operator+ (Complex& c) {
  return Complex (real + c.real, imag + c.imag);
}
```

A sobrecarga de operador não pode alterar o número de argumentos, a precedência e as regras de associação aplicados ao uso normal dos operadores.

A tabela abaixo apresenta todos operadores que podem ser sobrecarrecados em C++:

+	-	*	/	%	^	&	|
~	!	,	=	<	>	<=	>=
++	--	<<	>>	==	!=	&&	||
+=	-=	*=	/=	%=	^=	&=	|=

<<=	>>=	[ ]	( )	->	->*	new	delete

Os operadores que não podem ser sobrecarregados são:

.	.*	::	?:

Os símbolos utilizados pelo preprocessador # e ## também não podem ser sobrecarregados.

9.1. Formas de redefinir operadores

A redefinição de um operador pode ser feita de duas formas: **função membro** ou **função amiga**. Nos dois casos, para se definir um operador **@**, deve-se implementar uma função de nome **operator @**.

Para operadores **unários**:

método: tipoRetorno **operator @** ( );

função amiga: tipoRetorno **operator @** (tipoArgumento);

Para operadores **binários**:

método: tipoRetorno **operator @** (TipoArgumento);

função amiga: **friend** tipoRetorno **operator @** (tipoArg1, tipoArg2);

9.2. Exemplos de sobrecarga

9.2.1. Operadores “ ++ ” e “ -- ”

A partir da versão C++ 2.1, a sobrecarga dos operadores incremento (++) ou decremento (--) são feitos de forma separada para os casos: **posposto e anteposto**.

Para ANTEPOSTO:

tipoRetorno **operator** ++( );

tipoRetorno **operator** --();

ou

friend tipoRetorno **operator** ++(tipoDaClasse);

friend tipoRetorno **operator** --(tipoDaClasse);

Para POSPOSTO:

tipoRetorno **operator** ++(**int**);

tipoRetorno **operator** --(**int**);

ou

friend tipoRetorno **operator** ++(tipoDaClasse, **int**);

friend tipoRetorno **operator** --(tipoDaClasse, **int**);

Por exemplo:

```
class Foo {
  void operator ++( );                    // método anteposto
  friend void operator --(Foo &);         // função amiga anteposto
  friend void operator ++(Foo&, int);     // função amiga posposto
  void operator --(int);                  // método posposto
};

// Definições das funções amigas e funções membro

void main ( ) {
  Foo x;
  x++;          // chama operator++ (x, 0)
  ++x;          // chama x.operator++ ( )

  x--;          // chama x.operator-- ( )
  --x;          // chama operator-- (x)
}
```

9.2.2. Operador " = "

A sobrecarga do operador atribuição (=) pode ser feita através da declaração de uma função membro não estática. Por exemplo:

```
class String {
  ...
  public:
  String& operator = (String& str);
  ...
  String (String&);
  ~Sting( );
}
```

O código acima, com a definição apropriada de **String::operator = ( )** , permite a atribuição de string str1 = str2, tal como ocorre em outras linguagens. Diferente das outras funções de operadores, a função do **operador atribuição não pode ser herdado** por classes derivadas. Se, para qualquer classe **X**, não existe o operador atribuição definido pelo programador, o compilador gera automaticamente um default que faz atribuição membro a membro da **classe X**:

```
X& X::operator = (const X& fonte) {
```

```
        // atribuição membro a membro
}
```

9.2.3. Operador " cast "

O nome do operador **cast** é sempre o nome do tipo de destino. Este operador deve sempre ser **redefinido como uma função membro** e o tipo de retorno **nunca** é indicado, pois é sempre igual ao nome do operador. Seja a definição e utilização do operador *cast* para a classe Racional:

```
class Racional {
  int num, den;
  public:
        Racional (int n, int d) { num = d; den =d; }
        ...
        operator float ( ) {
          return num / (float) den;
        }
        ...
};

void main ( ) {
  Racional q(1,3), r(1,7);
  float x, y;
  x = (float) q;
  y = float(r);                  // notação funcional
}
```

9.2.4. Operador " [ ] "

A sobrecarga do operador indexação (**[ ]**) pode ser definida pelo programador através da declaração de uma função membro não estática. Por exemplo:

```
class VetorInt {
  int *p;
  int tamanho;
  public:
  int& operator [ ] (int i) { return p[i]; }
  ...
  VetorInt (const VetorInt&);
  ~VetorInt( );
}
```

A expressão **x[a]**, onde **x** é uma instância da classe **VetorInt**, é interpretado como **x.operator[](a)**.

9.2.5. Operadores " new " e " delete "

Os operadores **new** e **delete** podem ser sobrecarregados para fornecer funções alternativas de alocação de memória. O operador **new** sobrecarregado, definido pelo programador, deve retornar **void*** e receber como primeiro argumento **size_t**. O operador **delete** sobrecarregado, também definido pelo usuário, deve retornar **void** e receber como primeiro argumeto **void***; um segundo argumento **size_t** é opcional. Por exemplo:

```
#include <stdlib.h>
class X {
  ...
  public:
        void* operator new(size_t tamanho) { return newalloc (tamanho); }
        void operator delete(void* ptr) { newfree (ptr); }
        X( ) { /* definição do construtor default */ }
        X(char ch) { /* definição do construtor */ }
        ~X( ) { /* definição do destrutor */ }
  ...
};
```

O argumento **tamanho** fornece o tamanho do objeto a ser criado, e **newalloc( )** e **newfree( )** são as funções usadas para alocação e desalocação de memória. Construtores e destrutores chamados para objetos da **classe X** (ou objetos de classes derivadas de **X** que não tiverem seus próprios operadores **new** e **delete** sobrecarregados) invocarão as funções **X::operator new( )** e **X::operator delete( )**, respectivamente.

As funções **X::operator new( )** e **X::operator delete( )** são membros estáticos de **X** se forem explicitamente declaradas como **static** ou **não**. Portanto elas não podem ser funções virtuais.

Os operadore **new** e **delete** predefinidos ainda podem ser utilizados dentro do escopo da classe **X**, explicitamente com o operador escopo global (**::operator new** e **::operator delete**), ou implicitamente quando na criação e destruição de objetos que não sejam da classe **X** ou **derivada**. Por exemplo, seja uma outra forma de sobrecarregar operadores **new** e **delete** para **X**:

```
void* X::operator new (size_t tamanho) {
  void* ptr =new char[tamanho]              // chamada do new padrão
  ...
  return ptr;
}

void* X::operator delete (void* ptr) {
  ...
  delete (void*) ptr;                       //chamada do delete padrão
}
```

9.3. Sobrecarga dos operadores " << " (inserção) e " >> " (extração)

9.3.1. Operador inserção (<<)

O C++ fornece quatro objetos **stream** predefinidos como segue:

- **cin:** Entrada padrão, usualmente o teclado, correspondente a **stdin** no C;
- **cout**: Saída padrão, usualmente o vídeo, correspondente a **stdout** no C;
- **cerr:** Saída de erro padrão, usualmente o vídeo, correspondente a **stderr** no C;
- **clog:** Versão totalmente "bufferizada" de **cerr**, sem correspondente no C.

Pode-se redirecionar estas **stream** padrões de/para outros dispositivos e arquivos. (Em C, somente **stdin** e **stdout** podiam ser redirecionados).

A classe **istream** inclue a definição para o operador >> para os tipo padrões (**int, long, double, float, char** e **char***). Assim a declaração **cin >> x**, chama o função operador >> apropriada para a instância **istream cin** definida em iostream.h e a usa para direcionar esta **stream** de entrada dentro da localização de memória representada pela variável **x**. Da mesma forma, a classe **ostream** sobrecarregou o operador <<, o qual permite que a declaração **cout >> x** direcione o valor de x para **ostream cout** que manda para saída.

Um benefício real com as **stream's** do C++ é a facilidade com que se pode sobrecarregar os operadores >> e << para permitir I/O com tipos definidos pelo programador. Considere uma estrutura de dados simples definida pelo programador:

```
struct Empregado {
        char* nome;
        int depto;
        long vendas;
};
```

Para sobrecarregar o operador << para a saída do objeto do tipo **Empregado**, deve-se fazer a seguinte definição:

```
ostream& operator << (ostream& str, Empregado& e)  {
  str << "Nome: " << e.nome;
  str << "\n Departamento: << e.depto;
  str << "\n Quantidade de vendas: r$ "<< e.vendas;
  return str;
}
```

Note que a função operador << deve retornar **ostream&**, uma referência para **ostream**, assim pode-se usar o **novo operador** << da mesma forma que o operador predefinido. Desta forma, pode-se mostrar um objeto do tipo **Empregado** como segue:

```
#include <iostream.h>
...
void main( ) {
  Empregado Rick ={ “Richard”, 30, 10000 };
  ...
  cout << Rick;
  ...
}
```

Fornecendo a seguinte saída:

Empregado: Richard

Departamento: 30

Quantidade de vendas: r$ 10000

Da mesma forma, para sobrecarregar o operador >> para a entrada dos dados de um objeto do tipo **Empregado**, deve-se fazer a seguinte definição:

```
istream& operator >> (ostream& str, Empregado& e)  {
  cout << “ Entre com o nome do empregado”
  str >> e.nome;
  cout << “\n Qual departamento?”
  str >> e.depto;
  cout << “\n Quanto vendeu?”
  str >> e.vendas;
  return str;
}
```

A função operador >> deve retornar **istream&**, uma referência para **istream**, assim pode-se usar o **novo operador** >> da mesma forma que o operador predefinido. Pode-se entrar com os dados de um objeto do tipo **Empregado** como segue:

```
#include <iostream.h>
...
void main( ) {
  Empregado Rick;
  ...
  cin << Rick;                  // Entrada de dados
  ...
  cout << Rick;
  ...
}
```

Para o exemplo anterior os operadores >> e << foram redefinidos para trabalhar com um tipo novo que foi definido com a palavra-chave **struct**, ou seja, todos os seus dados são públicos por default. Se a palavra-chave **class** for utilizada, as funções devem ser declaradas como

amigas, para poderem ter acesso a **dados protegidos** e **privados**, dentro da declaração da classe **Empregado**:

```
class Empregado {
        char* nome;
        int depto;
        long vendas;

        friend istream& operator >> (ostream& , Empregado& );
        friendostream& operator << (ostream& , Empregado& );
};
```

Para saber mais:

Manual do Usuário Borland C++ 3.0 - Cap 16 “Using C++ streams”

10. Acesso a disco em C++

10.1. Classes para arquivos em C++

No C++ existem as classes para trabalhar com arquivos: **fstream**, **ifstream** e **ofstream**, as quais são derivadas de **iostream**, **istream** e **ostream**, respectivamente. Estas classes contêm objetos da classe **filebuf**, que é derivada de **streambuf**, para permitir interface para I/O de arquivo. As descrições das classes estão contidas no arquivo cabeçalho “**fstream.h**”, o qual já inclue “**iostream.h**”.

Estas três classes fornecem construtores para abrir objetos **stream** e associá-los com arquivos. Os construtores mais utilizados são declarados como segue:

```
ifstream (const char*, int = ios::in, int = filebuf  ::openprot);
ofstream (const char*, int = ios::out, int = filebuf  ::openprot);
fstream (const char*, int , int = filebuf  ::openprot);
```

As três classes também **herdam** os operadores **inserção** e **extração** de suas classes bases. Considere um exemplo que copia o arquivo “file.in” para o arquivo “file.out”:

```
#include <fstream.h>                    // não é necessário incluir <iostream.h>

int main( ) {
  char ch;
  ifstream f1(“file.in”);
  ofstream f2(“file.out”);

  if (!f1) cerr << “O arquivo file.in nao pode ser aberto para entrada”;
  if (!f2) cerr << “O arquivo file.out nao pode ser aberto para saída”;

  while ( f2 && f1.get(ch) )
        f2.put(ch);
```

```
  return 0;
}
```

Para as classes **stream**, o operador **!** é sobrecarregado para voltar 0 se nenhum erro ocorreu e diferente de zero se um erro ocorreu.

O segundo argumento para os construtores é um inteiro para indicar o modo de abertura para o arquivo. A lista dos modos possíveis é mostrada a seguir:

```
enum ios::open_mode {
        in          = 0x01,     // abre para leitura
        out         = 0x02,     // abre para escrita
        ate         = 0x04,     // procura eof quando aberto
        app         = 0x08,     // todas inclusões no final do arquivo
        trunc       = 0x10,     // trunca se já existe
        nocreate    = 0x20,     // não abre se o arquivo não existe
        noreplace   = 0x40,     // não abre se arquivo já existe
        binary      = 0x80      // arquivo binário (não texto)
};
```

Por exemplo, para prevenir que o arquivo destino não seja criado se o mesmo já existe no programa anterior, o programador poderia alterar a chamada do construtor para:

```
ofstream f2("file.out", ios::out | ios::nocreate);
```

Quando um objeto **stream** é aberto com o construtor default, um objeto é criado, mas nenhum arquivo é aberto. Antes de sua utilização, o objeto deve ser associado a um arquivo usando a função membro **open( )**. Os argumentos para **open( )** são iguais ao do construtor normal. Portanto, o arquivo fonte no exemplo poderia ser alterado como segue:

```
ifstream f1;
f1.open("file.in");
```

A função membro **close( )** fecha um objeto **stream**. Um objeto pode ser **aberto** e **fechado repetidamente**.

Para saber mais:

Manual do Usuário Borland C++ 3.0 - Cap 16 "Using C++ streams"

Tópico: "Stream class reference"

Manual do Usuário Borland C++ 3.0 - Cap 4 "Object-oriented programming with C++"

Tópico: "The C++ streams libraries"

10.2. Buscando dentro de uma "stream"

O tipo **streampos** definido dentro de "**iostream.h**" armazena a localização do ponteiro de leitura ou escrita dentro de uma **stream** de arquivo. Dois métodos podem ser usados para indicar a localização do ponteiro. O método **tellg( )** indica a localização do ponteiro de extração (ou seja, a próxima posição de leitura no arquivo), enquanto o método **tellp( )** indica a localização do ponteiro de inserção (próxima posição de escrita). A expressão a seguir armazena a posição corrente para escrita:

```
// declaração
ofstream out("saida.txt");

// dentro de algum lugar no programa
streampos outPos = out.tellp( );
```

Os métodos **seekg( )** e **seekp( )** reposicionam o ponteiro **stream** para uma **streampos**. Existem duas variantes destes métodos: uma para localização absoluta e outra para relativa.

```
out.seekp(outPos);          // retorna para localização armazenada

out.seekp(-10, ios::curr);          // move para 10 bytes antes da posição corrente

out.seekp(10, ios::beg);            // move para 10 bytes depois do começo do arquivo

out.seekp(-10, ios::end);           // move para 10 bytes antes do fim do arquivo
```

10.3. Ponteiro para "stream"

Muitos programas aceitam entrada de um arquivo especificado na linha de comando ou da entrada padrão se nenhum arquivo for especificado. Os objetos **stream** padrões predefinidos são declarados para permitir o redirecionamento para uma **stream** diferente. Isto é possível através da utilização de ponteiros **stream**.

Considere o programa:

```
#include <fstream.h>
#include <iomanip.h>                // include necessária para os manipuladores de stream

void main(int argc, char* argv[ ]) {
  istream *entrada = &cin;
  char buffer[80];
  if (argc ==2)
        entrada = new ifstream(argv[1]);

  while (! entrada->eof( ) ) {
        (*entrada) >> setw(80) >> buffer;
        cout << buffer;
  }
```

```
}
```

O ponteiro **entrada** primeiramente recebe o endereço da **stream padrão** (teclado). O programa usa o ponteiro **entrada** para ler uma **string** e mandá-la para a entrada padrão. Entretanto, se um argumento é fornecido, o programa assume que é um nome de um arquivo e uma nova instância **ifstream** é ligada a este arquivo. O endereço do novo objeto é armazenado em **entrada** assumindo com que os dados vêm do arquivo ao invés do teclado.

Para saber mais:

Hands On Turbo C++ - Cap 9 “Stream Input / Output”

11. Tipos Parametrizados ou Gabaritos ("template")

Template, também chamado de tipo parametrizado ou gabarito, permite construir uma família de funções relacionadas.

11.1. Funções Template

Considere uma função **max(**x,y**)** que retorna o maior entre dois argumentos. Os dois argumentos (x e y) pode ser de qualquer tipo que seja possível ordenar. Porém é importante lembrar, que o compilador C++ espera que os tipos de parâmetros x e y sejam declarados em tempo de compilação. Sem usar **templates**, muitas versões sobrecarregadas de **max( )** seriam necessárias, uma para cada tipo utilizado, mesmo sendo idêntico o código para todas as funções. Cada versão compararia os dois argumentos e retornaria o maior. Por exemplo:

```
int max(int x, int y) {
  return (x > y) ? x: y;
}

long max(long x, long y) {
  return (x > y) ? x: y;
}

// definição de max( ) para outros tipos
```

Com a utilização de **template**, pode-se definir um gabarito para uma família de funções sobrecarregadas deixando que o tipo de dado seja também um parâmetro:

```
template <class Tipo>
Tipo max (Tipo x, Tipo y) {
  return (x > y) ? x: y;
}
```

O tipo de dado é representado por argumento template: **<class Tipo>**. Quando usado em uma aplicação, o compilador gera a função apropriada de acordo com o tipo de dado que a chama:

```
int i;
Complex a, b;

int j =max(i, 0);                  // argumentos são inteiros
Complex m = max(a, b);             // argumentos são do tipo Complex
```

11.2. Classes Template

Classe Template (também conhecida como classe genérica ou classe geradora) permite definir um gabarito para definição de classe. Considere o seguinte exemplo de uma classe vetor. Se uma classe de vetores de inteiros ou de qualquer outro tipo, as operações básicas executadas sobre o tipo serão iguais (criação, apagar, indexar, etc). Com o tipo de elemento tratado como um parâmetro para a classe, o sistema gerará um gabarito para classe:

```
#include <iostream.h>

template <class Tipo>
class Vetor {
  private:
        Tipo *dado;
        int tamanho;
  public:
        Vetor (int);
        ~Vetor ( ) { delete[ ] dado; }
        Tipo& operator[ ] (int i) { return dado[i]; }
};

// Definição de função não inline
template <class Tipo>
Vetor<Tipo>::Vetor (int n) {
  dado = new Tipo[n];
  tamanho = n;
}

main( ) {
  Vetor<int> x(5);          // gera um vetor de inteiros

  for (int i = 0; i<5 ; ++i)
        x[i] = i;

  for (i = 0; i<5; ++i)
        cout << x[i] << ' ';

  cout << ' \n';
  return 0;
}
```

A saída na tela para programa será:

0 1 2 3 4

Para saber mais:

Manual do Usuário Borland C++ 3.0 - Cap 13 "C++ specifics"

• Bibliografia

- Davis, Stephen R., “Hands-On Turbo C++”, Addison-Wesley Publishing Company, Inc, Reading, Massachusetts, 1991.

- Turbo C++ Version 3.0, “User’s Guide”, Borland International, Inc, Scotts Valley, 1992.

A. Adendo

```
// Borland C++ - (C) Copyright 1991 by Borland International

/* point.h--Example from Getting Started */

// point.h contains two classes:
// class Location describes screen locations in X and Y coordinates
// class Point describes whether a point is hidden or visible

enum Boolean {false, true};

class Location {
protected:          // allows derived class to access private data
  int X;
  int Y;

public:             // these functions can be accessed from outside
  Location(int InitX, int InitY);
  int GetX();
  int GetY();
};
class Point : public Location {      // derived from class Location
// public derivation means that X and Y are protected within Point

protected:
  Boolean Visible;  // classes derived from Point will need access

public:
  Point(int InitX, int InitY);      // constructor
  void Show();
  void Hide();
  Boolean IsVisible();
  void MoveTo(int NewX, int NewY);
};
// Borland C++ - (C) Copyright 1991 by Borland International
```

```
/* POINT2.CPP--Example from Getting Started */

// POINT2.CPP contains the definitions for the Point and Location
// classes that are declared in the file point.h

#include "point.h"
#include <graphics.h>

// member functions for the Location class
Location::Location(int InitX, int InitY) {
  X = InitX;
  Y = InitY;
};

int Location::GetX(void) {
  return X;
};

int Location::GetY(void) {
  return Y;
};

// member functions for the Point class: These assume
// the main program has initialized the graphics system

Point::Point(int InitX, int InitY) : Location(InitX,InitY) {
  Visible = false;              // make invisible by default
};

void Point::Show(void) {
  Visible = true;
  putpixel(X, Y, getcolor());      // uses default color
};

void Point::Hide(void) {
  Visible = false;
  putpixel(X, Y, getbkcolor()); // uses background color to erase
};

Boolean Point::IsVisible(void) {
  return Visible;
};

void Point::MoveTo(int NewX, int NewY) {
  Hide();        // make current point invisible
  X = NewX;      // change X and Y coordinates to new location
  Y = NewY;
  Show();        // show point at new location
};
```

```
// Borland C++ - (C) Copyright 1991 by Borland International

/* MCIRCLE.CPP--Example from Getting Started */

// MCIRCLE.CPP       Illustrates multiple inheritance

#include <graphics.h> // Graphics library declarations
#include "point.h"    // Location and Point class declarations
#include <string.h>   // for string functions
#include <conio.h>    // for console I/O

// link with point2.obj and graphics.lib

// The class hierarchy:
// Location->Point->Circle
// (Circle and CMessage)->MCircle

class Circle : public Point {  // Derived from class Point and
                       // ultimately from class Location
protected:
  int Radius;
public:
  Circle(int InitX, int InitY, int InitRadius);
  void Show(void);
};


class GMessage : public Location {
// display a message on graphics screen
  char *msg;              // message to be displayed
  int Font;              // BGI font to use
  int Field;             // size of field for text scaling

public:
  // Initialize message
  GMessage(int msgX, int msgY, int MsgFont, int FieldSize,
        char *text);
  void Show(void);        // show message
};


class MCircle : Circle, GMessage {  // inherits from both classes
public:
  MCircle(int mcircX, int mcircY, int mcircRadius, int Font,
       char *msg);
  void Show(void);               // show circle with message
};


// Member functions for Circle class

//Circle constructor
Circle::Circle(int InitX, int InitY, int InitRadius) :
            Point (InitX, InitY)       // initialize inherited members
//also invokes Location constructor
{
  Radius = InitRadius;
};
```

```
void Circle::Show(void)
{
  Visible = true;
  circle(X, Y, Radius); // draw the circle
}

// Member functions for GMessage class

//GMessage constructor
GMessage::GMessage(int msgX, int msgY, int MsgFont,
                   int FieldSize, char *text) :
                   Location(msgX, msgY)
//X and Y coordinates for centering message
{
  Font = MsgFont;    // standard fonts defined in graph.h
  Field = FieldSize; // width of area in which to fit text
  msg = text;        // point at message
};

void GMessage::Show(void)
{
  int size = Field / (8 * strlen(msg));     // 8 pixels per char.
  settextjustify(CENTER_TEXT, CENTER_TEXT); // centers in circle
  settextstyle(Font, HORIZ_DIR, size);      // magnify if size > 1
  outtextxy(X, Y, msg);                     // display the text
}

//Member functions for MCircle class

//MCircle constructor
MCircle::MCircle(int mcircX, int mcircY, int mcircRadius, int Font,
            char *msg) : Circle (mcircX, mcircY, mcircRadius),
            GMessage(mcircX,mcircY,Font,2*mcircRadius,msg)
{
}

void MCircle::Show(void)
{
  Circle::Show();
  GMessage::Show();
}

main()     //draws some circles with text
{
  int graphdriver = DETECT, graphmode;
  initgraph(&graphdriver, &graphmode, "..\\bgi");
  MCircle Small(250, 100, 25, SANS_SERIF_FONT, "You");
  Small.Show();
  MCircle Medium(250, 150, 100, TRIPLEX_FONT, "World");
  Medium.Show();
  MCircle Large(250, 250, 225, GOTHIC_FONT, "Universe");
  Large.Show();
  getch();
  closegraph();
  return 0;
}
```

B. Adendo

- O que é uma **stream**?

Uma **stream** é uma referência abstrata (“alça”) para qualquer fluxo de dados de uma **fonte** (ou produtor) para um **destino** (ou consumidor). Os termos extrair, retirar e buscar são usados quando se fala de entrada de caracteres; e os termos inserir, colocar e armazenar no caso de saída de dados.

- A biblioteca **iostream**

Esta biblioteca têm duas classes paralelas: **streambuf** e **ios**. As duas classes são de baixo nível, cada uma fazendo um conjunto diferente de operações.

streambuf

Esta classe fornece métodos para armazenamento (buffer) e suporte para **stream** quando um pouco ou nenhum formato é necessário.

ios

Esta classe contém um ponteiro para **streambuf**. Ela possui duas subclasses: **istream** (para entrada) e **ostream** (para saída). Existe uma outra classe, **iostream**, que é derivada das duas anteriores através de herança múltipla:

```
class ios;
class istream : virtual public ios;
class ostream : virtual public ios;
class iostream : public istream, public ostream;
```

Por fim, existe três classes **withassign** derivadas de **istream, ostream** e **iostream**:

```
class istream_withassign : public istream;
class ostream_withassign : public ostream;
class iostream_withassign : public iostream;
```

Está última fornece quatro **streams** “padrões”: **cin, cout, cerr** e **clog**. A declaração encontra-se no arquivo “**istream.h**”:

```
extern istream_withassign cin;
extern ostream_withassign cout;
extern ostream_withassign cerr;
extern ostream_withassign clog;
```

Funções membro **put( )** e **write( )**

Para saída de dados binários ou um caracter binário, pode-se usa a função **put( )** declarada em **ostream** como segue:

```
ostream& ostream :: put (char) ;
```

Com a declaração *int ch ='x';* as duas linhas a seguir são equivalentes

```
cout.put(ch);
cout << (char) ch;
```

A função membro **write( )** permite "jogar" para a saída objetos maiores:

```
ostream& ostream :: write(const signed char* ptr, int n);
ostream& ostream :: write(const unsigned char* ptr, int n);
```

Esta função "coloca" na saída ***n*** caracteres (inclusive NULL) em fomato binário. Por exemplo:

```
cout.write((char *)&x, sizeof(x));
```

mandará a representação binária de x para a saída.

Formatação de saída

Formato de entrada e saída são determinados por vários *flags de estado de formato* declarado como **enum** na classe **ios**. Os estados são determinados por bits como **long int** como segue:

```
public:
enum {
        skipws          = 0x0001,       // despreza espaço em branco na entrada
        left            = 0x0002,       // alinhamento a esquerda
        right           = 0x0004,       // alinhamento a direita
        internal        = 0x0008,       // posiciona depois do sinal ou indicador de base
        dec             = 0x0010,       // conversão decimal
        oct             = 0x0020,       // conversão octal
        hex             = 0x0040,       // conversão hexadecimal
        showbase        = 0x0080,       // mostra a base decimal na saída
        showpoint       = 0x0100,       // mostra com ponto decimal
        uppercase       = 0x0200,       // saída hexadecimal maiúscula
        showpos         = 0x0400,       // mostra '+' com inteiros positivos
        scientific      = 0x0800,       // mostra em notação cientifíca
        fixed           = 0x1000,       // mostra em notação de ponto flutuante
        unitbuf         = 0x2000,       // descarrega todas as strams depois de inserção
        stdio           = 0x4000,       // descarrega stdout e stderr depois de inserção
```

```
};
```

Função membro **width( )**

O default para inserção é colocar na saída o mínimo de caracteres necessários para representar o operando com alinhamento à direita. Para alterar isto, pode-se utilizar a função **width( )**:

```
int ios :: width (int w);      // configura o tamanho do campo para w e retorna o tamanho anterior
int ios :: width ( );          // retorna a tamanho corrente, sem alterações
```

Por exemplo:

```
int i = 123;
int old_w = cout.width(6);
cout << i;                     // mostra bbb123 onde b = espaço em branco
cout.width(old_w)              // restaura o tamanho antigo
```

Manipuladores

Manipuladores podem ser colocados em cadeia nas operações de inserção e extração para **alterar** o estado da **stream** como um efeito.

Manipulador	Sintaxe	Ação
dec	cout << **dec** cin >> **dec**	Configura conversão decimal
hex	cout << **hex** cin >> **hex**	Configura conversão hexadecimal
oct	cout << **oct** cin >> **oct**	Configura conversão octal
ws	cin >> **ws**	Extrai os caracteres em branco
endl	cout << **endl**	Insere uma linha e descarrega a stream
ends	cout << **ends**	Insere caracter null na string
flush	cout << **flush**	Descarrega uma stream
setbase(int)	cout << **setbase(n)**	Configura base numérica (0, 8, 10 ou 16). 0 significa default: decimal
resetiosflag(long)	cout << **resetiosflag(l)** cin >> **resetiosflag(l)**	Limpa os flags de estado para **cin** ou **cout** de formato especificados em l
setiosflag(long)	cout << **setiosflag(l)** cin >> **setiosflag(l)**	Configura os flags de estado para **cin** ou **cout** de formato especificados em l
setfill(int)	cout << **setfill(n)** cin >> **setfill(n)**	Configura o caracter de preenchimento para n
setprecision(int)	cout << **setprecision(n)** cin >> **setprecision(n)**	Configura o número de dígitos de precisão para ponto flutuante
setw(int)	cout << **set(w)** cin >> **set(w)**	Configura o tamanho do campo para n

Por exemplo:

```
cout << setw(4) << i << setw(6) << j;
```

é equivalente a

```
cout.width(4);
cout << i;
cout.width(6);
cout << j;
```

Função membro **fill( )**

O caracter a direção de preenchimento depende da configuração dos flags de estado correspondentes. O caracter default de preenchimento é o espaço em branco. Pode-se alterar o caracter usando a função **fill( )**:

```
int  i = 123;
cout.fill('*');
cout.width(6);
cout << i;                    // mostra ***123
```

A direção default de preenchimento é alinhamento a direita. Pode-se alterar o default com as funções **setf( )** e **unsetf( )**:

```
int  i = 56;
...
cout.width(6);
cout.fill('#');
cout .setf (ios::left,);
cout << i;                    // mostra 56###
```

O segundo argumento, **ios::adjustified**, indica quais bits configurar. O primeiro argumento, **ios::left**, indica o que configurar nestes bits.

Função membro **get( )**

Para o tipo **char** (signed ou unsigned), o operador >> despreza o operador espaço e armazena o próximo caracter. Se for necessário ler um caracter, espaço ou não, pode-se usar a função membro **get( )**:

```
char ch;
cin.get (ch);
```

Uma variante desta função possibilita o controle sobre o número de caracteres extraídos, onde serão colocados e com qual caracter termina:

```
istream& istream :: get (char *buf, int max, int tem = '\n');
```

Função membro **putback( )**

A função membro

```
istream& istream :: putback(char c);
```

coloca o caracter c de volta em **istream**; se não for possível colocar, o estado da **stream** é configurado para “fail”.