Conceitos da programação orientada a objetos e tipos abstratos de dados

Siang - 1997

O tratamento aqui será superficial, devendo o aluno consultar livros da área para maiores detalhes. Este material se baseia em vários livros, principalmente os seguintes livros. A ênfase aqui é tão somente mostrar alguns conceitos importantes em projetar estruturas de dados, introduzindo o conceito de tipos abstatos de dados.

1. Takahashi, T. e Liesenberg, H.K.E. *Programação Orientada a Objetos.* VII Escola de Computação, IME/USP, 1990.

2. Stroustrup, B. *The C++ programming language.* Addison Wesley, 1987.

3. Pinson, L.J. and Wiener, R.S. *An Introduction to Object-Oriented Programming and C++.* Addison Wesley, 1988.

Programação orientada a objetos (abreviada o.o.), no sentido mais puro e direto, signifca programação pelo envio de mensagens a objetos. Solução de problemas o.o. consiste em identificar os objetos, mensagens e envio de sequências de mensagens para obter uma solução.

Quatro conceitos principais de orientação a objetos incluem:

- Abstração
- Encapsulamento
- Herança
- Polimorfismo

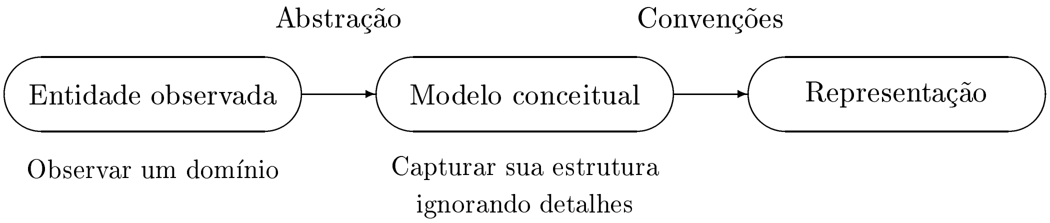
Desafio de novas aplicações

- Tamanho e complexidade: software de uso real frequentemente tem várias dezenas de megabytes de código. Exemplo: X-windows tem 20 Mbytes de código.
- Alta confiabilidade: sistemas automatizados manipulados por usuários não especializados demandam alta confiabilidade e tolerância a falhas.
- Alto desempenho: Novas aplicações como previsão de tempo, controle de processos em tempo real, etc. demandam alto desempenho e curtos tempos de resposta.

Deseja-se construir software:

- rápido
- barato (reutilização de software)
- flexível (fácel alteração)

O paradigma de objetos constitui-se em um possível foco integrador dessas 3 linhas.



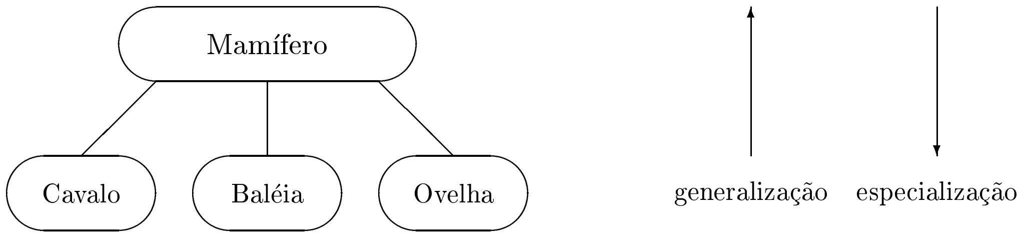
Abstração

Há várias operações de abstração: classificação/instanciação, generalização/especialização, etc.

Categorização ou classificação: agrupamento em classes, com base em algum conjunto de atributos comuns a todos. Exemplo: Classe dos estudantes de graduação.

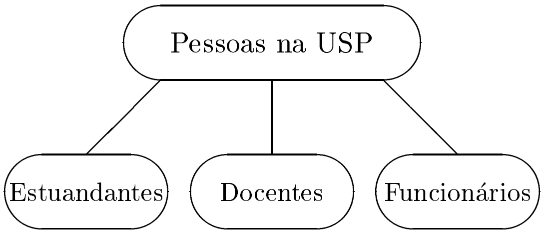
A operação inversa à classificação é a instanciação: obtenção de indivíduos a partir de uma classe.

Generalização/especialização : Observamos várias classes e abstraímos delas uma classe mais genérica.



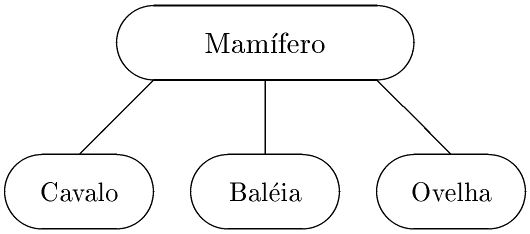
Hierarquia de abstrações

Composta por categorias ou classes, da mais genérica a mais particular. Exemplo:



Herança

A classe mamíferos possui atributos comuns aos elementos classificáveis nessa categoria. As classes "cavalo", "baléia", etc "herdam" os atributos dos mamíferos, além de possuirem atributos particulares da espécie.



Linguagens de programação

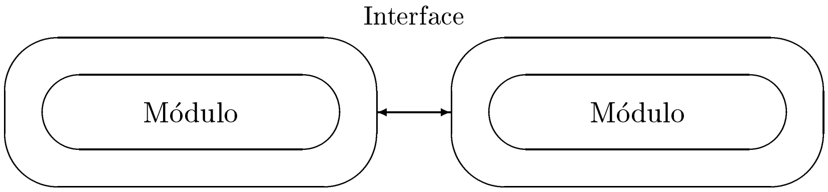
Abstração de dados e abstração procedural (ou procedimental)

As linguagens Algol 60 e Fortran possuem abstração de dados, além de abstração procedural (procedimentos ou sub-rotinas). Além disso, linguagens como Pascal e COBOL possuem também a agregação de dados (**record** de Pascal e estruturas em vários níveis (01 etc) de COBOL). estruturas).

Tipos abstratos de dados

O trabalho pioneiro é de D. Parnas – "On the criteria to be used in decomposing systems into modules", *Communications of the ACM* 15, 2, 1972, pp. 1053–1058. Neste trabalho Parnas apresenta princípios para nortear o projetista na estruturação do software. O principal argumento é o chamadao *ocultamento de informação* ("information hiding").

Módulos devem expor a outros módulos somente a informação necessária para que eles operassem corretamente, escondendo internamente estruturas de dados, lógica procedural, etc.



Especificação formal de tipos abstratos de dados

Na abstração por meio de tipos abstratos de dados, usa-se especificação algébrica para permitir a definição axiomática de um tipo arbitrariamente complexo (composto por dados e operações) sem impor detalhes de implementação. Exemplos de linguagens: Ada, Modula, CLU, Alphard.

Tipo abstrato : conjunto de dados e de operações sobre esses dados.

A visibilidade da estrutura interna do tipo de dado é limitada às operações que forem explicitamente enumerados como exportáveis, incorporando a idéia de "information hiding" de Parnas.

Há a separação de

- definição de um tipo e
- sua representação e implementação

A título de exemplo vamos ver a definição de um tipo abstrato chamado pilha ("stack").

Tipo stack
Operações:

1. Push
2. Pop
3. Top
4. Clear
5. Empty
6. Full

Condições pre e pos: estando **pre** verificada, **pos** é verificada após a operação.
Convenção:

S Valor da pilha antes do procedimento
S' Valor da pilha após o procedimento
$\neg$ não

$\sim$ concatenação de duas cadeias
$<>$ cadeia vazia
length comprimento da cadeia
maxstack tamanho máximo da pilha
RESULT resultado da operação

procedure Push (var S: stack; x: T);
pre $\neg$ Full (S');
pos $S' =< x >\sim S$;

function Pop (var S: stack): T;
pre $\neg$ Empty (S');
pos $S' =\langle\mathrm{RESULT}\rangle \sim S$;

function Top (S: stack): T;
pre $\neg$ Empty (S);
pos RESULT = first (S');

proceudre Clear (var S: stack);
pos $S' =<>$;

function Empty (S: stack):boolean;
pos RESULT $\equiv$ ($S' =<>$);

function Full (S: stack): boolean;
pos RESULT $\equiv$ (length(S) = maxstack);

Implementação

$\vdots$

Os módulos que usam a pilha podem se basear nas condições **pre** e **pos**, mas não podem usar detalhes e informações de implementação.

Uma visão superficial de C++

(Este material não vai ser coberto no curso.)

O que se segue é baseado no manual "Turbo C++ getting started, captítulo 5". Visa apenas dar alguns sabores da linguagem C++.

Veremos como C++ implementa os três conceitos fundamentais de p.o.o.

- Encapsulamento
- Herança
- Polimorfismo

Encapsulamento

Combina uma estrutura de dados com as funções (também chamadas ações ou métodos) para manipular os dados. O acesso à estrutura de dado pode ser protegido.

Class: Numa classe, podemos ter como seus *membros* dados e funções. Os membros são **private** por default, podendo ser tornados públicos com **public**. Pode ainda ser **protected** como veremos daqui a pouco.

```
class Location {
    int X; // private por default
    int Y;
public:
    Location(int InitX, int InitY) // construtor
            {X = InitX; Y = InitY;}
    int GetX() {return X;} // inline function
    int GetY() {return Y;} // inline function
};
```

inline significa que a função é compilada *in situ*, como uma macro, sem overhead de chamadas de funções. Deve ser usada apenas quando a definição envolve apenas uma ou poucas linhas. As funções podem ser declaradas dentro da classe e definidas fora. Neste caso, elas não serão do tipo inline, a menos que se especifique isso explicitamente.

private significa que os dados podem ser acessados apenas através de funções declaradas dentro da mesma classe.

public significa que os membros podem ser acessados de qualquer ponto dentro do mesmo escopo da definição da classe.

construtor serve para especificar como um novo objeto da classe será criado ou inicializado. Note que o construtor leva o mesmo nome da classe (Location). No exemplo acima, o construtor está declarado e também definido dentro da classe (inline). Podemos também declarar o construtor dentro da classe e defini-lo fora:

```
class Location {
...
public:
      Location(int InitX, int InitY); // construtor decl.
...
}
Location::Location(int NewX, int NewY) // construtor def.
{
      X = NewX;
      Y = NewY;
};
```

O construtor pode conter ainda os valores iniciais, como por exemplo:

```
Location::Location(int NewX=0, int NewY=10) // construtor def.
{
...
};
```

Uma vez declarada a classe Location, podemos declarar um objeto novo (uma instância) da classe Location como segue:

```
Location Origin;
```

Ou já chamando o construtor para sua inicialização:

```
Location Origin(1,1);
```

Enquanto o construtor é para a criação, o destrutor, a ser visto mais tarde, serve para eliminação do objeto.

Herança

Queremos agora definir uma nova classe Point, que além de possuir duas coordenadas, possui ainda um valor booloeano que indica a sua visibilidade na tela. Point é portanto parecido com Location, mas com algo mais. C++ trata de situações "semelhantes mas diferentes" com a seguinte solução. Ela permite classes derivadas de classes base, *herdando* suas propriedades. Chamaremos a classe Location como base e a classe derivada Point.

```
// 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);
};
```

Os dados de Location são **protected**, significando que eles são acessíveis apenas por funções membros de Location **e ainda** por funções de classes derivadas de Location.

Uma classe derivada (no caso Point) é definida com

class D: modificador_de_acesso B {
...
}

onde D é o nome da classe derivada, B a classe base. O modificador_de_acesso pode ser **private** ou **public**. Esse modificador (default private) é usado para modificar a acessibilidade de membros da classe base pela classe derivada, conforme a tabela seguinte.

Acesso na classe base	modificador_de_acesso	Acesso herdado da base
public	public	public
private	public	não acessível
protected	public	protected
public	private	private
private	private	não acessível
protected	private	private

Direitos de acesso devem ser passados com cuidado. Uma classe pode ser derivada da classe base de modo **private** ou **public**. A derivação **private** (a default para classes) converte membros **public** e **protected** na classe base para membros **private** na classe derivada. Com a derivação **public**, membros **protected** da classe base continua **protected** na classe derivada, e só podem ser acessados por classes derivadas publicamente ou ainda por **friends** (a ser vistos mais tarde).

Vejamos agora as definições das funções declaradas em point.h acima.

```
// 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
};
```

Estendendo classes

Podemos agora derivar a partir de Location e Point a nova classe Circle, com funções para show, hide, expand, move e contract.

```
// CIRCLE.CPP   A Circle class derived from Point

#include <graphics.h>    // graphics library declarations
#include "point.h"       // Location and Point class declarations
#include <conio.h>       // for getch() function

// link with point2.obj and graphics.lib
```

```
class Circle : Point {   // derived privately from class Point
 // and ultimately from class Location
   int Radius;           // private by default

public:
   Circle(int InitX, int InitY, int InitRadius);
   void Show(void);
   void Hide(void);
   void Expand(int ExpandBy);
   void MoveTo(int NewX, int NewY);
   void Contract(int ContractBy);
};

Circle::Circle(int InitX, int InitY, int InitRadius) : Point(InitX,InitY)
{
   Radius = InitRadius;
 };

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

void Circle::Hide(void)
{
   unsigned int TempColor;    // to save current color
   TempColor = getcolor();    // set to current color
   setcolor(getbkcolor());    // set drawing color to background
   Visible = false;
   circle(X, Y, Radius);      // draw in background color to erase
   setcolor(TempColor);       // set color back to current color
};

void Circle::Expand(int ExpandBy)
{
   Hide();                       // erase old circle
   Radius += ExpandBy;           // expand radius
   if (Radius < 0)               // avoid negative radius
      Radius = 0;
   Show();                       // draw new circle
};

void Circle::Contract(int ContractBy)
{
   Expand(-ContractBy);       // redraws with (Radius - ContractBy)
};

void Circle::MoveTo(int NewX, int NewY)
{
   Hide();                    // erase old circle
   X = NewX;                  // set new location
   Y = NewY;
   Show();                    // draw in new location
};

main()                        // test the functions
{
   // initialize the graphics system
   int graphdriver = DETECT, graphmode;
   initgraph(&graphdriver, &graphmode, "..\\bgi");
```

```
   Circle MyCircle(100, 200, 50);   // declare a circle object
   MyCircle.Show();                 // show it
   getch();                         // wait for keypress
   MyCircle.MoveTo(200, 250);       // move the circle (tests hide
                                    // and show also)
   getch();
   MyCircle.Expand(50);             // make it bigger
   getch();
   MyCircle.Contract(75);           // make it smaller
   getch();
   closegraph();
   return 0;
}
```

Veja acima como foi feita a chamada de uma função membro de uma classe, Circle.Show(), de modo semelhante à referência de um membro de uma estrutura.

Se CirclePointer aponta para um objeto da classe Circle, voce usaria CirclePointer→Show().

Herança múltipla

Através do mecanismo de herança múltipla, uma classe pode herdar as características de várias classes bases.

Suponha uma classe Circle, como já vista. Suponha uma outra classe GMessage que se encarrega de imprimir messagens numa posição da tela dada, com certo font e tamanho. Queremos uma nova classe MCircle que desenha um círculo com uma mensagem dentro do mesmo. Podemos definir a classe MCircle como derivada de Circle e de GMessage.

```
// 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);      // if size > 1, magnifies
   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;
}
```

Polimorfismo

Polimorfismo significa várias formas. Polimorfismo em C++ permite o uso de um mesmo nome para versões diferentes de uma mesma função. Por exemplo usar o nome cubo para calcular a terceira potência de um inteiro ou flutuante:

```
int cubo(int numero-int);

float cubo(float numero-float);
```

C++ se encarrega de chamar a função correta. No caso isso não apresenta dificuldade pois os argumentos são de tipos diferentes.

No caso de argumentos idênticos, o sistema não seria capaz de fazer as distinções em tempo de compilação. Podemos entretanto declarar as funções como sendo virtuais, para as quais a particular versão a ser executada é determinada em tempo de execução (o chamado "late binding"). Por exemplo, use o qualificador virtual na primeira declaração de funções membro:

```
virtual void Show();
virtual void Hide();
```

Se você redeclara Show com os mesmos argumentos formais e mesmo tipo de retorno, então o novo Show automaticamente fica como virtual, nao importando se voce usa virtual ou não. O novo Show virtual **sobrecarrega** ("override") o Show na classe base.

Objetos dinâmicos

Os objetos em exemplos anteriores são declarados e têm sua memória alocada em tempo de compilação. Veremos agora objetos que são criados em tempo de execução, com sua memória alocada de modo dinâmico do espaço livre do sistema. Criação de objetos dinamicamente é uma técnica importante em aplicações em que a quantidade de dados não é conhecida antes da execução. Uma árvore binária, por exemplo, pode crescer durante a execução de um programa.

Para a criação de objetos dinâmicos, usamos **new**, como por exemplo:

```
Circle *ACircle = new Circle (151,82,50);
```

Ao ponteiro ACircle é dado o endereço de um bloco de memória, de tamanho suficiente para conter um objeto do tipo Circle. ACircle aponta portanto para um objeto Circle alocado do espaço livre do sistema.

Quando se cria um objeto via new, você é responsável pela sua dealocação, usando o operador delete:

```
delete pointer;
```

onde pointer é o apontador usado para alocar a memória por meio de new.

Exemplo: lista linear

Cria uma lista ligada de objetos gráficos e posteriormente removidos quando não mais necessitados.

Vamos criar uma lista (tipo List) de objetos do tipo Point.

```
// LISTDEMO.CPP           Demonstrates dynamic objects

// Link with FIGURES.OBJ and GRAPHICS.LIB

#include <conio.h>          // for getch()
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <graphics.h>
#include "figures.h"

class Arc : public Circle {
   int StartAngle, EndAngle;
public:
   // constructor
   Arc(int InitX, int InitY, int InitRadius, int InitStartAngle,
       int InitEndAngle);
   // virtual functions
   void Show();
   void Hide();
};

struct Node {     // the list item
   Point *Item;   // can be Point or any class derived from Point
   Node  *Next;   // point to next Node object
};

class List {      // the list of objects pointed to by nodes
   Node *Nodes;   // points to a node
public:
   // constructor
   List();
   // destructor
   ~List();
   // add an item to list
   void Add(Point *NewItem);
   // list the items
   void Report();
};
```

```
// definitions for standalone functions

void OutTextLn(char *TheText)
{
   outtext(TheText);
   moveto(0, gety() + 12);   // move to equivalent of next line
}

void MemStatus(char *StatusMessage)
{
   unsigned long MemLeft;  // to match type returned by
   // coreleft()
   char CharString[12];    // temp string to send to outtext()
   outtext(StatusMessage);
   MemLeft = long (coreleft());

   // convert result to string with ltoa then copy into
   // temporary string
   ltoa(MemLeft, CharString, 10);
   OutTextLn(CharString);
}

// member functions for Arc class

Arc::Arc(int InitX, int InitY, int InitRadius, int InitStartAngle,
         int InitEndAngle) : Circle (InitX, InitY,InitRadius)
                              // calls Circle
                              // constructor
{
   StartAngle = InitStartAngle;
   EndAngle = InitEndAngle;
}

void Arc::Show()
{
   Visible = true;
   arc(X, Y, StartAngle, EndAngle, Radius);
}

void Arc::Hide()
{
   unsigned TempColor;
   TempColor = getcolor();
   setcolor(getbkcolor());
   Visible = false;
   arc(X, Y, StartAngle, EndAngle, Radius);
   setcolor(TempColor);
}

// member functions for List class

List::List () {
   Node *N;
   N = new Node;
   N->Item = NULL;
   N->Next = NULL;
   Nodes = NULL;             // sets node pointer to "empty"
                             // because nothing in list yet
}

List::~List()                // destructor
```

```
{
   while (Nodes != NULL) {   // until end of list
      Node *N = Nodes;       // get node pointed to
      delete(N->Item);       // delete item's memory
      Nodes = N->Next;       // point to next node
        delete N;            // delete pointer's memory
   };
}

void List::Add(Point *NewItem)
{
   Node *N;              // N is pointer to a node
   N = new Node;         // create a new node
   N->Item = NewItem;    // store pointer to object in node
   N->Next = Nodes;      // next item points to curent list pos
   Nodes = N;            // last item in list now points
                         // to this node
}

void List::Report()
{
   char TempString[12];
   Node *Current = Nodes;
   while (Current != NULL)
   {
      // get X value of item in current node and convert to string
      itoa(Current->Item->GetX(), TempString, 10);
      outtext("X = ");
      OutTextLn(TempString);
      // do the same thing for the Y value
      itoa(Current->Item->GetY(), TempString, 10);
      outtext("Y = ");
      OutTextLn(TempString);
      // point to the next node
      Current = Current->Next;
   };
}

void setlist(void);

// Main program
main()
{
   int graphdriver = DETECT, graphmode;
   initgraph(&graphdriver, &graphmode, "..\\bgi");

   MemStatus("Free memory before list is allocated: ");
   setlist();
   MemStatus("Free memory after List destructor: ");
   getch();
   closegraph();
}

void setlist() {

   // declare a list (calls List constructor)
   List AList;

   // create and add several figures to the list
   Arc *Arc1 = new Arc(151, 82, 25, 200, 330);
   AList.Add(Arc1);
   MemStatus("Free memory after adding arc1: ");
```

```
    Circle *Circle1 = new Circle(200, 80, 40);
    AList.Add(Circle1);
    MemStatus("Free memory after adding circle1: ");
    Circle *Circle2 = new Circle(305, 136, 35);
    AList.Add(Circle2);
    MemStatus("Free memory after adding circle2: ");
    // traverse list and display X, Y of the list's figures
    AList.Report();
    // The 3 Alist nodes and the Arc and Circle objects will be
    // deallocated automatically by their destructors when they
    // go out of scope in main(). Arc and Circle use implicit
    // destructors in contrast to the explicit ~List destructor.
    // However, you could delete explicitly here if you wish:
    // delete Arc1; delete Circle1; delete Circle2;
    getch();   // wait for a keypress
    return;
}
```

List consiste em um apontador a uma lista de Node, que por sua vez consiste em um apontador Item a um Point, mais um apontador Next ao próximo Node.

O programa declara uma variável AList, do tipo List, e constrói uma lista ligada de três nós. Graças ao destrutor, espaço é removido automaticamente no fim.

Funções friend

Normalmente acesso a membros **private** de uma classe é restrito a funções membro da classe. As vezes se torna necessário dar acesso a dados **private** a funções externas. A declaração **friend** dentro de uma classe permite a especificação de funções que podem acessar os membros **private** da classe.

Suponha que exista uma função chamada Fancy_Print para a qual você deseje dar acesso ao conteúdo de seus objetos da classe String.

```
classe String {
    ...
friend void Fancy_Print(String &AString);
    ...
```

Desse modo, a função Fancy_Print pode acessar aos membros da classe String.

Exemplos de programas em C++

O que se segue é um resumo do Capítulo 6 do Manual Turbo ++ Getting Started. Os programas exemplificam vários conceitos de C++. Tais programas estão presentes no diretório de Turbo++ de nome:

`\tc\examples`

Programa 1: uso de cout

```
// ex1.cpp:   A First Glance
// from Chapter 6 of Getting Started
#include <iostream.h>

main()
{
   cout << "Frankly, my dear...\n";
   cout << "C++ is a better C.\n";
}
```

O arquivo **iostream.h** possui a maioria da funcionalidade de **stdio.h**.

cout é uma saída padrão para a tela. Substitui **stdout** em C. O operador << envia os dados da direita a cout. (Note que << também denota o operador de shift-left; este é mais um exemplo de "overloading".)

Programa 2: uso de cin

```
// ex2.cpp:   An interactive example
// from Chapter 6 of Getting Started
#include <iostream.h>

main()
{
   char name[16];
   int age;

   cout << "Enter your name: ";
   cin >> name;
   cout << "Enter your age: ";
   cin >> age;

   if (age < 21)
      cout << "You young whippersnapper, " << name << "!\n";
   else if (age < 40)
      cout << name << ", you're still in your prime!\n";
   else if (age < 60)
      cout << "You're over the hill, " << name << "!\n";
   else if (age < 80)
      cout << "I bow to your wisdom, " << name << "!\n";
   else
      cout << "Are you really " << age << ", " << name << "?\n";
}
```

cin é a entrada padrão associada ao teclado.

Programa 3: uso de inline

```
// ex3.cpp:   Inline Functions
// from Chapter 6 of Getting Started
#include <iostream.h>

const float Pi = 3.1415926;

inline float area(const float r) {return Pi * r * r;}

main()
{
   float radius;

   cout << "Enter the radius of a circle: ";
   cin >> radius;
   cout << "The area is " << area(radius) << "\n";
}
```

A palavra-chave **inline** instrui o compilador a inserir o código diretamente no local da chamada da função, evitando "overhead" da chamada de funções. Seu uso é recomendado no lugar de #define.

Programa 4: chamada por referência e uso de argumento default

```
// ex4.cpp:   Default arguments and Pass-by-reference
// from Chapter 6 of Getting Started
#include <iostream.h>
#include <ctype.h>

int get_word(char *, int &, int start = 0);

main()
{
   int word_len;
   char *s = "  These words will be printed one-per-line  ";

   int word_idx = get_word(s,word_len);           // line 13
   while (word_len > 0)
   {
      cout.write(s+word_idx, word_len);
               cout << "\n";
      //cout << form("%.*s\n",word_len,s+word_idx);
      word_idx = get_word(s,word_len,word_idx+word_len);
   }
}

int get_word(char *s, int& size, int start)
{
   // Skip initial whitespace
   for (int i = start; isspace(s[i]); ++i);
   int start_of_word = i;

   // Traverse word
   while (s[i] != '\0' && !isspace(s[i]))
      ++i;
   size = i - start_of_word;
   return start_of_word;
}
```

O segundo argumento da função **get_word** é passado por referência, permitindo a alteração de seu valor pela rotina. (Equivale ao parâmetro **var** de Pascal.)

O terceiro argumento de **get_word** é default (com valor default especificado como 0). Isso significa que ele pode ser omitido (como na linha 13), e nesse caso o valor 0 é passado automaticamente.

Programas 5 e 6

Não examinaremos os programas 5 e 6, passando diretamente para o programa 7.

Programa 7: classe List

```
// list.h:   A Integer List Class
// from Chapter 6 of Getting Started
const int Max_elem = 10;

class List
{
   int *list;        // An array of integers
   int nmax;         // The dimension of the array
   int nelem;        // The number of elements

public:
   List(int n = Max_elem) {list = new int[n]; nmax = n; nelem = 0;};
   ~List() {delete list;};
   int put_elem(int, int);
   int get_elem(int&, int);
   void setn(int n) {nelem = n;};
   int getn() {return nelem;};
   void incn() {if (nelem < nmax) ++nelem;};
   int getmax() {return nmax;};
   void print();
};

// list.cpp:   Implementation of the List Class
// from Chapter 6 of Getting Started
#include <iostream.h>
#include "list.h"

int List::put_elem(int elem, int pos)
{
   if (0 <= pos && pos < nmax)
   {
      list[pos] = elem;    // Put an element into the list
      return 0;
   }
   else
      return -1;           // Non-zero means error
}

int List::get_elem(int& elem, int pos)
{
   if (0 <= pos && pos < nmax)
   {
      elem = list[pos];    // Retrieve a list element
      return 0;
   }
   else
      return -1;           // non-zero means error
```

```
}

void List::print()
{
   for (int i = 0; i < nelem; ++i)
      cout << list[i] << "\n";
}

// ex7.cpp:   Using the List class
// from Chapter 6 of Getting Started
#include "list.h"

main()
{
   List l(5);
   int i = 0;

   // Insert the numbers 1 through 5
   while (l.put_elem(i+1,i) == 0)
      ++i;
   l.setn(i);

   l.print();
}
```

O exemplo acima define listas em alocação sequencial por meio de vetores.

Programa 8: classe derivada Stack

```
// stack2.h:   A Stack class derived from the List class
// from Chapter 6 of Getting Started
#include "list2.h"

class Stack : public List                 // line 5
{
   int top;

public:
   Stack() {top = 0;};
   Stack(int n) : List(n) {top = 0;};      // line 11
   int push(int elem);
   int pop(int& elem);
   void print();
};
```

A linha 5 informa que a classe Stack é derivada da classe base List. Como List tem um construtor que possui um argumento, o construtor Stack chama o construtor List diretamente.

```
// stack.cpp:   Implementation of the Stack class
// from Chapter 6 of Getting Started
#include <iostream.h>
#include "stack.h"

int Stack::push(int elem)
{
   int m = getmax();
   if (top < m)
   {
      put_elem(elem,top++);
      return 0;
```

```
   }
   else
      return -1;
}

int Stack::pop(int& elem)
{
   if (top > 0)
   {
      get_elem(elem,--top);
      return 0;
   }
   else
      return -1;
}

void Stack::print()
{
   int elem;

   for (int i = top-1; i >= 0; --i)
   {  // Print in LIFO order
      get_elem(elem,i);
      cout << elem << "\n";
   }
}
```

Note que as funções públicas da classe List podem ser usadas diretamente, pois Stack é uma List. Entretanto, os membros **private** da porção List de um objeto Stack não podem ser referenciados diretamente.

```
// ex8.cpp:   Using the Stack Class
// from Chapter 6 of Getting Started
#include "stack.h"

main()
{
   Stack s(5);
   int i = 0;

   // Insert the numbers 1 through 5
   while (s.push(i+1) == 0)
      ++i;

   s.print();
}
```

Programa 9: uso de protected e polimorfismo

As vezes é conveniente permitir uma classe derivada ter acesso direto a alguns membros da uma classe base. Tais membros são ditos **protected**.

```
// list2.h:   A Integer List Class
// from Chapter 6 of Getting Started
const int Max_elem = 10;

class List
```

```
{
protected:     // The protected keyword gives subclasses
               // direct access to inherited members
   int *list;        // An array of integers
   int nmax;         // The dimension of the array
   int nelem;        // The number of elements

public:
   List(int n = Max_elem) {list = new int[n]; nmax = n; nelem = 0;};
   ~List() {delete list;};
   int put_elem(int, int);
   int get_elem(int&, int);
   void setn(int n) {nelem = n;};
   int getn() {return nelem;};
   void incn() {if (nelem < nmax) ++nelem;};
   int getmax() {return nmax;};
   virtual void print();                   // line 22
};
```

Podemos agora substituir chamadas a funções de List por referências a dados de List na implementação de Stack.

```
// stack.cpp:   Implementation of the Stack class
// from Chapter 6 of Getting Started
#include <iostream.h>
#include "stack.h"

int Stack::push(int elem)
{
   int m = getmax();
   if (top < m)
   {
      put_elem(elem,top++);
      return 0;
   }
   else
      return -1;
}

int Stack::pop(int& elem)
{
   if (top > 0)
   {
      get_elem(elem,--top);
      return 0;
   }
   else
      return -1;
}

void Stack::print()
{
   int elem;

   for (int i = top-1; i >= 0; --i)
   {  // Print in LIFO order
      get_elem(elem,i);
      cout << elem << "\n";
   }
}
```

Completamos agora com:

```
// ex9.cpp:   Using the print() virtual function
// from Chapter 6 of Getting Started
#include <iostream.h>
#include "stack2.h"

main()
{
   Stack s(5);
   List l, *lp;
   int i = 0;

   // Insert the numbers 1 through 5 into the stack
   while (s.push(i+1) == 0)
      ++i;

   // Put a couple of numbers into the list
   l.put_elem(1,0);
   l.put_elem(2,1);
   l.setn(2);

   cout << "Stack:\n";
   lp = &s;           // line 22
   lp->print();       // Invoke the Stack print() method; line 23

   cout << "\nList:\n";
   lp = &l;
   lp->print();       // Invoke the List print() method; line 27
}
```

O exemplo ilustra polimorfismo. O tipo de um objeto não é identificado até o tempo de execução. Definindo a função **print** como virtual (linha 22 de "list2.h"), podemos chamar funções diferentes **print** através de um apontador à classe base. Na linha 22, *lp* aponta a um objeto Stack, assim o método de impressão de Stack é chamada na linha 23. Analogamente, a função **print** de Stack é executada na linha 27.