PUC-RS, Faculdade de Informatica

Laboratorio de Programacao II (C++, Eng. da Computacao)

Prof. Eduardo Augusto Bezerra <eduardob@acm.org>

Notas de aula, 2002/2

Bibliografia utilizada:

[1] "The annotated C++ reference manual"; Ellis, M. A. and Stroustrup, B.; Addison-Wesley; 1990. Versao em portugues: "C++ Manual de Referencia Comentado"; Editora Campus.
[2] "Learning C++"; Graham, N.; McGraw-Hill; 1991.
[3] "Programming with Class – A Practical Introduction to Object-Oriented Programming with C++"; Gray, N. A. B.; John Wiley & Sons; 1994.

Classes Derivadas (mecanismo de heranca) (pg. 242 [1])

- uma classe pode ser derivada de suas classes base (base classes).
- membros de uma classe base podem ser referenciados como se fossem membros da classe derivada.
- membros da classe base sao herdados pela classe derivada.
- o operador :: e' usado para referenciar explicitamente um membro da classe base.
- isso permite acesso a nome redefinido na classe derivada.

Ex:

```
class Base {
          public:
                    int a, b;
};
class Derived : public Base {
          int b, c;
};
void f() {
          Derived d;
          d.a = 1;             // acesso ao a da classe Base
          d.Base ::b = 2;      // acesso ao b da classe Base
          d.b = 3;             // acesso ao b da classe Derived
          d.c = 4;             // acesso ao c da classe Derived
          Base* bp = &d;       // bp e' ponteiro tipo base para objeto
                               // tipo Derived (d e' do tipo Derived)
}
```

classe base == superclasse
classe derivada == subclasse

Uma classe base e' dita **direta** se for mencionada na lista base, e **indireta** se nao e' uma classe base direta, mas e' uma classe base de uma das classes mencionadas na lista base.

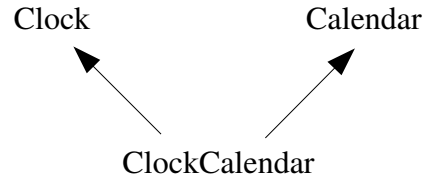
Ex:

```
class A {                                          A
          public:
                      void f();                   / \
};                                                 |
class B : public A {                               |
};
class C : public B {                               B
          public:
                      void f();
};                                                / \
void C::f() {                                      |
          f();                                     |
          A::f();
          B::f();                                  C
}
```

Classes Bases Multiplas (heranca multipla) (pg. 244 [1], pg. 257 [2])

Heranca multipla possibilita a utilizacao de recursos de diversas classes como ponto de partida para a definicao de uma nova classe.

O diagrama abaixo representa um bom exemplo de utilizacao de heranca multipla. A classe “Clock” armazena a hora, a classe “Calendar” armazena informacoes sobre a data. A nova classe “ClockCalendar” herda de “Clock” e “Calendar” formando um objeto unico com informacoes sobre data e hora.




```
class Clock {
 protected:
   int hr, min, sec, is_pm;
 public:
   Clock (int h, int s, int m, int pm);
   void setClock (int h, int s, int m,
                  int pm);
   void readClock (int& h, int& s,
                   int& m, int& pm);
   void advance ();
};
```

```
class Calendar {
 protected:
   int mo, day, yr;
 public:
   Calendar (int m, int d, int y);
   void setCalendar (int m, int d,
                     int y);
   void readCalendar (int& m, int& d,
                      int& y);
   void advance ();
};
```

```
class ClockCalendar : public Clock, public Calendar {
 public:
   ClockCalendar (int mt, int d, int y, int h, int m, int s, int pm);
   void advance ();
};
```

Para inicializar um objeto ClockCalendar, o construtor desse objeto precisa chamar os construtores de Clock e de Calendar:

```
ClockCalendar::ClockCalendar (int mt, int d, int y, int h, int m,
            int s, int pm) : Clock (h, mn, s, pm), Calendar (mt, d, y)
{}
```

As funcoes membro “setClock”, “readClock”, “setCalendar” e “readCalendar” sao todas herdadas por “ClockCalendar” e funcionam em objetos do tipo “ClockCalendar” da mesma forma como funcionam para objetos “Clock” ou “Calendar”.

Todas as tres classes possuem uma funcao “advance()”. A versao de “advance()” declarada em “ClockCalendar” sobrecarrega (*override*) as versoes herdadas de “Clock” e “Calendar”, entratanto as funcoes herdadas podem ser utilizadas na definicao de ClockCalendar::advance():

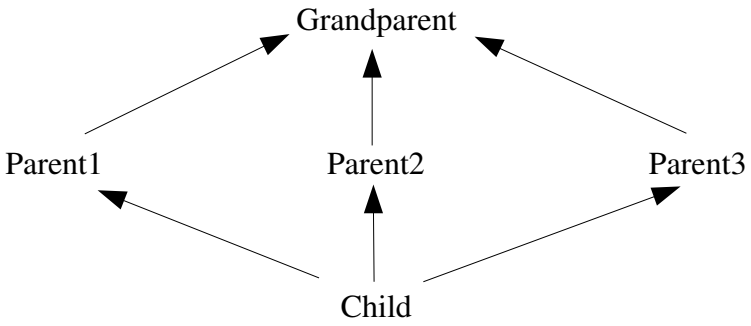
```
ClockCalendar::advance (){     // avancar o calendario, caso o clock
          int wasPm = isPm;    // mude de AM para PM.
          Clock::advance();
          if (wasPm && !isPm)
                    Calendar::advance();
}
```

Caso a funcao “advance()“ nao tivesse sido sobrecarregada, entao a sequencia a seguir levaria a um erro de ambiguidade. O compilador nao saberia qual “advance()” utilizar, o de “Clock” ou o de “Calendar”.

```
ClockCalendar cc;

cc.advance();        // no caso de advance() nao ser sobrecarregado,
                     // em ClockCalendar, sera' preciso especificar
                     // cc.Clock::advance() ou cc.Calendar::advance()
```

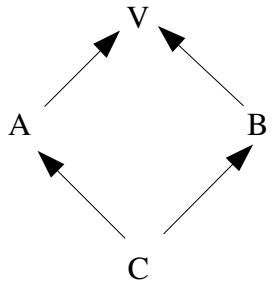
Como “advance()” foi sobrecarregada em “ClockCalendar”, a funcao acima ira' chamar ClockCalendar::advance().

Classes Bases Virtuais (pg. 248 [1], pg. 260 [2])



No diagrama anterior, as classes “Parent” herdam da classe “Grandparent”. A classe “Child”, por sua vez, recebe heranca multipla das classes “Parent”. O problema aqui e' que “Child” possuira' copias duplicadas dos membros herdados de “Grandparent”.

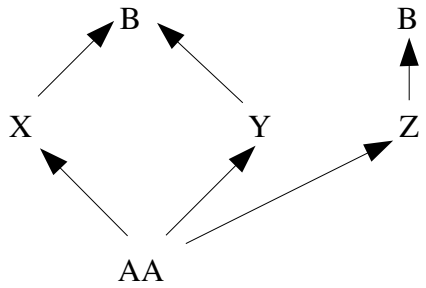
Uma solucao para esse tipo de problema e' a utilizacao do conceito de classe virtual. Por exemplo:




```
class V {
};
class A : virtual public V {
};
class B : virtual public V {
};
class C : public A, public B {
};
```

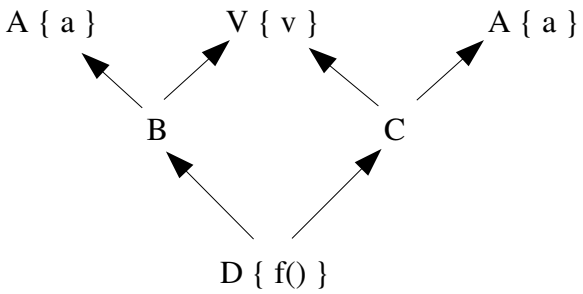
Um unico subobjeto de V e' compartilhado por todas as classes derivadas que especificam a classe base como virtual. A classe C possui apenas um subobjeto de V.

Uma classe pode possuir classes bases virtuais e nao-virtuais. Por exemplo:




```
class B {
};
class X : virtual public B {
};
class Y : virtual public B {
};
class Z : public B {
};
class AA : public X, public Y, public Z {
};
```

A situacao a seguir representa um problema de ambiguidade considerando classes virtuais e nao-virtuais.



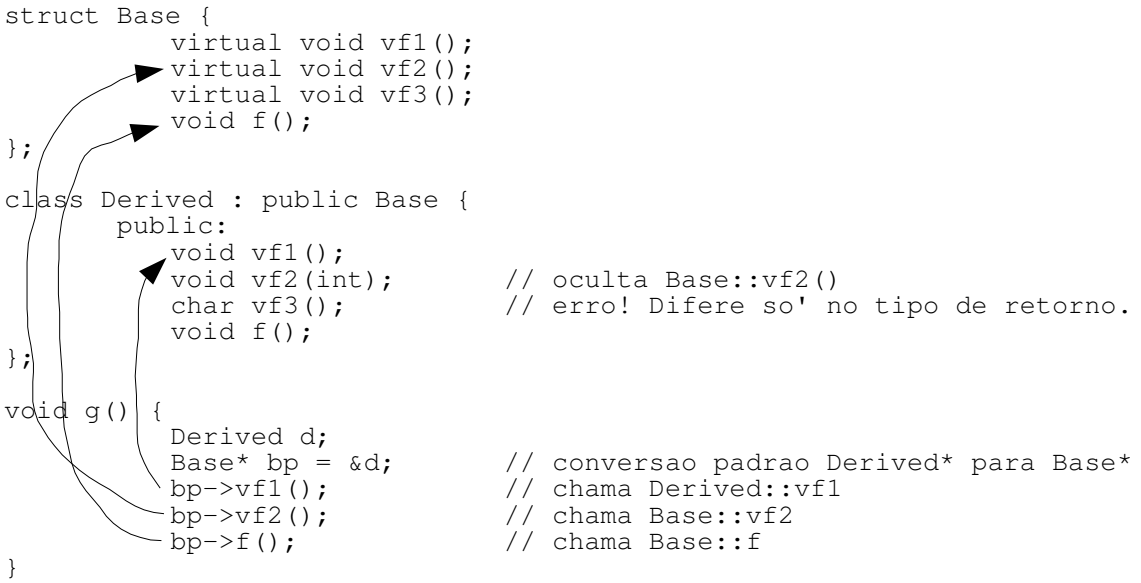

```
class V {   public:
                 int v;
};
class A {   public:
                 int a;
};
class B : public A, virtual public V {
};
class C : public A, virtual public V {
};
class D : public B, public C {
             public:
                 void f();
};
void D::f() {
     v++;      // ok, somente um 'v' em 'D'
     a++;      // erro, ambiguidade: 2 'a's em 'D'
}
```

Funcoes Virtuais (pg. 257 [1])

Classe derivadas e funcoes virtuais sao a chave para o projeto de muitos programas em C++ (orientacao a objeto):

- Classe base define uma interface para a qual uma variedade de implementacoes sao fornecidas por classes derivadas.
- O mecanismo de funcao virtual assegura que o objeto seja manipulado pelas funcoes definidas para ele.
- A habilidade para chamar uma serie de funcoes usando exatamente a mesma interface e' algumas vezes chamado de polimorfismo (funcoes virtuais proporcionam isso).

Ex:



```
struct Base {
           virtual void vf1();
           virtual void vf2();
           virtual void vf3();
           void f();
};

class Derived : public Base {
        public:
           void vf1();
           void vf2(int);       // oculta Base::vf2()
           char vf3();          // erro! Difere so' no tipo de retorno.
           void f();
};

void g() {
           Derived d;
           Base* bp = &d;       // conversao padrao Derived* para Base*
           bp->vf1();           // chama Derived::vf1
           bp->vf2();           // chama Base::vf2
           bp->f();             // chama Base::f
}
```

Observacoes:

- Se uma classe base (“Base”) contem uma funcao virtual “f” e uma classe derivada dela (“Derived”) tambem contem uma funcao “f” do mesmo tipo, entao uma chamada de “f” por um objeto da classe derivada chama Derived::f, (mesmo se o acesso for atraves de um ponteiro ou referencia a base).
- A funcao da classe derivada e' dita “ignorar” a funcao da classe base.
- Se os tipo de funcoes forem diferentes entao as funcoes sao consideradas diferentes e o mecanismo virtual nao e' invocado.
- “bp->vf1()” chama “Derived::vf1()” pois “bp” aponta para um objeto de classe “Derived” no qual “Derived::vf1()” tinha ignorado a funcao virtual “Base::vf1()”.
- “bp->vf2()” e' virtual. “d” e' do tipo “Derived”, mas nao existe “vf2()” em “Derived”, apenas “vf2(int)”.
- “bp->f()” nao e' virtual, pois “bp” e' do tipo “Base”.
- Chamada de funcao virtual depende do tipo do objeto. Ex: “d” e' do tipo “Derived”.
- Chamada de funcao membro nao-virtual depende apenas do tipo do ponteiro ou referencia. Ex: “bp” e' do tipo “Base”.

Friends (pg. 303 [1], pg. 168 [2])

Funcoes Friend compartilham a maioria dos privilegios das funcoes membro, mas nao estao fortemente ligadas a nenhuma classe. Uma funcao pode ser friend de diferentes classes, mas nao pode ser membro de apenas uma classe. Construtores, destrutores e funcoes virtuais (virtual) precisam ser funcoes membro e, consequentemente, nao podem ser Friend. Uma friend de uma classe e' uma funcao que nao e' um membro da classe, mas tem permissao para usar os membros privados e protegidos da classe. A declaracao da friend nao esta' no escopo da classe, e a friend nao e' chamada com os operadores de acesso a membro, a menos que ela seja um membro de outra classe.

Ex:

```
class X {
           int a;
           friend void friendSet(X*, int);
       public:
           void memberSet(int);
};
void friendSet(X* p, int i) {
           p->a = i;
}
void X::memberSet(int i) {
           a = i;
}
void f() {
           X obj;
           friendSet(&obj, 10);
           obj.memberSet(10);
}
```

Observacoes:

- Membros privados (private) de uma classe base sao acessiveis apenas por funcoes membro e funcoes friend dessa classe base.
- Funcoes membro e funcoes friend de uma classe derivada nao podem ver membros privados de uma classe base.
- Membros protegidos (protected) de uma classe base sao acessiveis a funcoes membro e funcoes friend da classe base e de qualquer classe derivada.

Assim como o uso de comandos “go to” nao e' recomendado em linguagens que seguem o paradigma da programacao procedural (devido a quebra na estrutura do programa), o uso de funcoes friend tambem deve ser evitado ou utilizado com bastante cuidado ao se utilizar o paradigma da orientacao a objetos. O uso de friend pode possibilitar o acesso a informacoes que deveriam estar “ocultas” e encapsuladas em uma determinada classe. Um exemplo de uso importante de funcoes friend e' a sobrecarga de operadores, discutida mais a frente.

Classes Abstratas (pg. 264 [1], pg. 278 [2])

- Suportam a nocao de um conceito geral (ex. Shape) onde somente variantes mais concretas podem ser utilizadas (ex. Circle, Square).
- Uma classe abstrata pode ser utilizada para definir uma interface para as classes derivadas que fornecerao uma variedade de implementacoes.
- Pode ser usada unicamente como uma classe basica para outra classe.
- Objetos de uma classe abstrata nao podem ser criados.
- Para ser abstrata precisa ter pelo menos uma funcao virtual pura.
- Uma funcao virtual e' pura pelo uso de um especificador-puro (= 0) na declaracao da funcao dentro da classe.

Ex:

```
class Point {
};
class Shape {                              // classe ABSTRATA
        Point centre;
      public:
        Point where() {
                  return centre;
        }
        void move (Point p){
                  centre = p;
                  draw();
        }
        virtual void rotate (int) = 0;           // virtual pura
        virtual void draw (int) = 0;             // virtual pura
};
```

Uma classe abstrata nao pode ser utilizada como:

- tipo de argumento;
- tipo de retorno de funcao;
- tipo para conversao explicita.

Classes abstratas podem ser ponteiros e referencias.

Ex:

```
Shape x;             // Erro! Objeto de classe abstrata
Shape* p;            // ok
Shape f();           // Erro
void g(Shape);       // Erro
Shape& h(Shape&);    // ok
```

Funcoes virtuais puras sao herdadas como funcoes virtuais puras.

Ex:

```
class AbCircle : public Shape {
          int radius;
      public:
          void rotate (int) {}
          // AbCircle::draw() e' uma virtual pura
};
```

Sobrecarga de funcoes (pg. 375 [1], pg. 168 [2])

Quando diversas declaracoes de funcoes diferentes sao especificadas por um unico nome no mesmo escopo, diz-se que esse nome esta' sobrecarregado. Quando esse nome e' utilizado, a funcao correta e' selecionada pela comparacao dos tipos dos argumentos reais com os tipos dos argumentos formais.

Ex:

```
double abs (double);
int abs (int);

abs(1);              // chamada a abs (int)
abs(1.0);            // chamada a abs (double)
```

Funcoes que diferem apenas no tipo de retorno nao podem ter o mesmo nome.

Uma funcao membro de uma classe derivada NAO esta' no mesmo escopo de uma funcao membro do mesmo nome em uma classe base. No exemplo a seguir, “D::f(char*)” oculta “B::f(int)”, em vez de sobrecarrega-la.

Ex:

```
class B {
      public:
          int f (int);
};

class D : public B {
      public:
          int f (char*);
};

void h(D* pd) {
          pd->f(1);            // erro! D::f(char*) oculta B::f(int)
          pd->B::f(1);         // ok
          pd->f(“Ben”);        // ok, chama D::f
}
```

Uma funcao declarada localmente nao esta' no mesmo escopo que uma funcao em escopo de arquivo.

Ex:

```
int f(char*);
void g() {
          extern f(int);
          f("asdf");           // erro! f(int) oculta f(char*) logo nao
                               // existe nenhuma f(char*) neste escopo.
}
```

Uma chamada de um determinado nome de funcao escolhe, entre todas as funcoes desse nome que estao no escopo e para as quais existe um conjunto de conversoes tal que uma funcao poderia ser possivelmente chamada, a funcao que melhor corresponde aos argumentos reais. A funcao de melhor correspondencia e' a intersecao entre os conjuntos de funcoes que melhor correspondem a cada argumento. A menos que essa intersecao tenha exatamente um membro, a chamada e' ilegal.

Sobrecarga de operadores (pg. 401 [1], pg. 168 [2])

A maior parte dos operadores podem ser sobrecarregados:

```
new   delete
+   -   *   /   %   ^   &   |   ~
!   =   <   >   +=   -=   *=   /=   %=
^=   &=   |=   <<   >>   >>=   <<=   ==   !=
<=   >=   &&   ||   ++   --   ,   ->*   ->
()   []
```

O s seguintes operadores nao podem ser sobrecarregados:

```
.    .*    ::    ?:    sizeof
```

Novos operadores nao podem ser criados, apenas os existentes podem ser sobrecarregados de forma a implementar uma nova funcionalidade. Uma outra restricao e' que em uma expressao utilizando um operador sobrecarregado, pelo menos um dos operandos precisa ser um objeto instanciado de uma classe.

Para redefinicao de um operador e' utilizada a palavra reservada "operator" seguida do operador a ser sobrecarregado:

Ex:

```
double operator+= (double newValue);
```

Nesse exemplo o operador += foi sobrecarregado, e para operandos do tipo double, o novo operador += retornara' um valor do tipo double.

Para um outro exemplo de sobrecarga de operadores, considerar a seguinte estrutura:

```
struct ClockTime {
          int hr;
          int min;
          int sec;
}
```

Considerando a seguinte definicao da variavel t:

```
ClockTime t = {11, 45, 20};
```

E supondo a necessidade de imprimir o conteudo da variavel t utilizando o seguinte operador de saida:

```
cout << t;                    // Deve imprimir 11:45:20
```

O operador << pode ser sobrecarregado da seguinte forma:

```
ostream& operator<< (ostream& c, ClockTime t){
          c << t.hr << ":";
          c << t.min << ":";
          c << t.sec;
          return c;
}
```

Uma observacao importante e' que como ClockTime e' uma struct logo seus membros sao public por default, e podem ser referenciados livremente na redefinicao do operador. Porem, caso as variaveis da instancia fossem privadas, o operador teria que ser declarado como friend da classe ClockTime para possuir acesso as variaveis membro. A sobrecarga de operadores e' uma situacao onde funcoes friend sao necessarias.

No exemplo a seguir uma classe Vector e' criada para operacoes com vetores, e os operadores + e * sao sobrecarregados para executar, respectivamente, a soma de dois vetores, e o produto escalar de dois vetores.

Ex:

```
const SIZE = 10;
class Vector {
          double c [SIZE];     // armazena o vetor
      public:
          Vector();            // cria vetor vazio
          friend Vector operator+ (const Vector& v, const Vector& w);
          friend double operator* (const Vector& v, const Vector& w);
};

Vector::Vector() {
          for (int i = 0; i < SIZE; i++)
                    c[i] = 0.0;
}
```

```
Vector operator+ (const Vector& v, const Vector& w) {
          Vector u;
          for (int i = 0; i < SIZE; i++)
                    u.c[i] = v.c[i] + w.c[i];
          return u;
}

double operator* (const Vector& v, const Vector& w) {
          double sum = 0.0;
          for (int i = 0; i < SIZE; i++)
                    sum += v.c[i] * w.c[i];
          return sum;
}
```

Polimorfismo (pg. 266 [2])

Polimorfismo e' a situacao na qual objetos pertencentes a diferentes classes podem responder a uma mesma mensagem, normalmente de maneiras diferentes. Polimorfismo pode ser utilizado para simplificar, por exemplo, o trabalho para impressao dos valores de nodos em uma lista heterogenea. Em uma lista heterogenea cada nodo pode possuir objetos de classes diferentes, e o polimorfismo simplifica a manipulacao desse tipo de lista.

No exemplo a seguir o vetor “list” (estrutura de dados homogenea) e' utilizado para armazenar dois objetos diferentes, “parent” e “child”. Isso e' possivel devido a utilizacao de ponteiros combinado com o conceito de heranca.

```
class parent {
   protected:
      int j, k;
   public:
      void setJ(int newj);
      void setK(int newk);
      int getJ();
      int getK();
};
class child : public parent {
   protected:
      int m, n;
   public:
      void setM(int newm);
      void setN(int newn);
      int getM();
      int getN();
};

void parent::setJ(int newj) {
      j = newj;
}

void parent::setK(int newk) {
      k = newk;
}
```

```
int parent::getJ() {
      return j;
}

int parent::getK() {
      return k;
}

void child::setM(int newm) {
      m = newm;
}

void child::setN(int newn) {
      n = newn;
}

int child::getM() {
      return m;
}

int child::getN() {
      return n;
}
```

```
int main() {
      parent prnt, prntl;
      child chld, chldl;

      prnt = chld;            // can access chld.j and chld.k only
      // chld = prnt;         // error

      parent* pp = &chld;     // can access all 4 variables
      parent& rr = chld;

      // child* cc = &prnt;   // error

      pp = &prntl;
      pp = &chldl;

      // Heterogeneuos list
      parent* list[4];
      list[0] = &prnt;        // prnt has 2 variables: j & k
      list[1] = &chld;        // chld has 4 variables: m, n, j & k
      list[2] = &prntl;       // prntl has 2 variables: j & k
      list[3] = &chldl;       // chldl has 4 variables: m, n, j & k
}
```

Uma maneira mais interessante para implementar polimorfismo em C++ e' por intermedio de funcoes virtuais. Ainda com relacao ao exemplo anterior de implementacao de listas heterogeneas em uma estrutura homogenea, uma forma mais eficiente para essa implementacao deve considerar funcoes que possam ser utilizadas para qualquer objeto independentemente de sua classe. O exemplo anterior e' re-implementado na listagem a seguir utilizando-se funcoes virtuais.

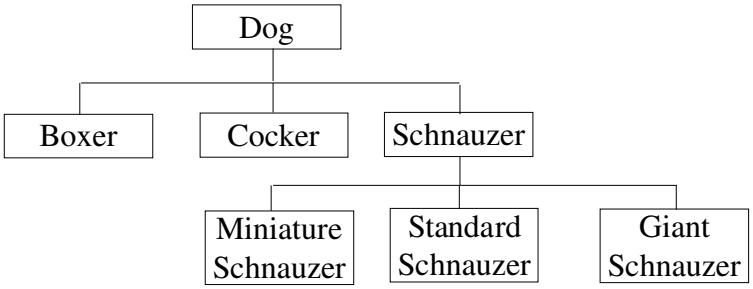
Notar que a palavra “virtual” e' utilizada nas funcoes membro na classe base. As funcoes na classe derivada que sobrecarregam as funcoes da classe base sao automaticamente assumidas como virtuais e nao ha' necessidade de escrever a palavra “virtual” na frente da declaracao dessas funcoes.

Em uma funcao **nao-virtual**, a **declaracao da variavel ponteiro** e' quem define qual definicao de funcao utilizar (no exemplo: base ou derivada). Como a variavel “p” no main (exemplo a seguir) e' declarada do tipo “parent*”, as funcoes definidas no parent serao sempre utilizadas.

No caso de funcoes **virtuais**, o que define a funcao a ser utilizada e' a **classe do objeto apontado**. Por exemplo, se a variavel “p” declarada no main como “parent*” apontar para um objeto “parent”, a definicao das funcoes do parent serao utilizadas. Por outro lado, caso a variavel “p” (ainda declarada como parent*) apontar para um objeto “child”, entao as funcoes do child serao utilizadas. Notar no main do exemplo a seguir que a variavel “p” e' declarada como um ponteiro para “parent” porem a classe do objeto apontado por “p” e' “child”. Nesse caso o comando “p->setJ(4);” ira' acionar a funcao “setJ” da classe “child” (a classe do objeto apontado e' “child”). Por outro lado, caso “p” apontasse para um objeto “parent”, entao a funcao “setJ” executada seria a da classe “parent”.

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

class parent {
   protected:
      int j, k;
   public:
      virtual void setJ(int newj);
      virtual void setK(int newk);
      virtual int getJ();
      virtual int getK();
};

class child : public parent {
   protected:
      int m, n;
   public:
      void setJ(int newj);
      void setM(int newm);
      void setN(int newn);
      int getM();
      int getN();
      int getJ();
};

void parent::setJ(int newj) {
   j = newj;
   cout << "parent::setJ" << endl;
}

void child::setJ(int newj) {
   j = newj;
   cout << "child::setJ" << endl;
}

void parent::setK(int newk) {
   k = newk;
}

int parent::getJ() {
   return j;
}

int child::getJ() {
   return j;
}

int parent::getK() {
   return k;
}

void child::setM(int newm) {
   m = newm;
}

void child::setN(int newn) {
   n = newn;
}

int child::getM() {
   return m;
}

int child::getN() {
   return n;
}

int main() {
   child c;

   parent* p = &c;

   p->setJ(4);
   c.setJ(5);
}
```

Um exemplo mais tradicional de uso de polimorfismo e' representado na figura a seguir. Diversas racas de cachorros sao herdadas da classe “dog” que possui uma funcao para impressao da raca do objeto “dog” em uso. A funcao “print_breed” a ser executada sera' aquela pertencente ao objeto em uso.



```
#include <iostream>
using namespace std;
class Dog {
   public:
    virtual void printBreed(){
     cout << "Breed not defined\n";
    }
};
class Boxer : public Dog {
   public:
      void printBreed() {
         cout << "Boxer\n";
      }
};
class Cocker : public Dog {
   public:
      void printBreed() {
        cout << "Cocker Spaniel\n";
      }
};
class Schnauzer : public Dog {
   public: void printBreed() {
              cout << "Schnauzer\n";
           }
};
class MiniatureSchnauzer :
                  public Schnauzer {
   public:
      void printBreed() {
         cout << "Miniature"
              << "Schnauzer\n";
      }
};
class StandardSchnauzer :
                  public Schnauzer {
   public:
      void printBreed() {
         cout << "Standard"
              << "Schnauzer\n";
      }
};
class GiantSchnauzer :
                  public Schnauzer {
   public:
      void printBreed() {
       cout << "Giant Schnauzer\n";
      }
};

int main() {
   Dog d;
   Boxer b;
   Cocker c;
   Schnauzer s;
   MiniatureSchnauzer ms;
   StandardSchnauzer ss;
   GiantSchnauzer gs;

   Dog* dp;

   dp = &d; dp->printBreed();
   dp = &b; dp->printBreed();
   dp = &c; dp->printBreed();
   dp = &s; dp->printBreed();
   dp = &ms; dp->printBreed();
   dp = &ss; dp->printBreed();
   dp = &gs; dp->printBreed();

   cout << "\n";

   Dog& rd = d;
   Dog& rs = s;
   Dog& rm = ms;

   rd.printBreed();
   rs.printBreed();
   rm.printBreed();

}
```

No programa acima um objeto e' declarado para cada classe (d, b, c, s, ms, ss, gs). O ponteiro “dp” e' criado para apontar para a classe base (Dog). Esse ponteiro apesar de ser do tipo Dog, pode ser utilizado para apontar para as classes derivadas, e no exemplo acima as chamadas “dp->printBreed()” acionam as funcoes virtuais das classes derivadas produzindo a seguinte saida:

```
Breed not defined
Boxer
Cocker Spaniel
Schnauzer
Miniature Schnauzer
Standard Schnauzer
Giant Schnauzer
```

O mesmo principio e' utilizado a seguir onde “rd”, “rs” e “rm” sao referencias a objetos “Dog” porem ao atribuir os objetos de classes derivadas a essas referencias, o conceito de polimorfismo se torna bastante evidente, pois “Dog” assumira' diversas formas, e as tres ultimas chamadas ao metodo “printBreed()” resultarao na seguinte saida:

```
Breed not defined
Schnauzer
Miniature Schnauzer
```

Template (pg. 415 [2])

Template e' um modelo a ser seguido por uma classe ou funcao. Um modelo de classe define o layout e as operacoes para um conjunto ilimitado de tipos relacionados. Por exemplo, um unico modelo de classe List poderia fornecer uma definicao comum para listas de “int”, “float” e ponteiros para um objeto qualquer. Um modelo de funcao define um conjunto ilimitado de funcoes relacionadas. Por exemplo, um unico modelo de funcao “sort()” poderia fornecer uma definicao comum para a classificacao de todos os tipos definidos pelo modelo de classe List.

Para uma explicacao mais detalhada de Templates incluindo exemplos de programas em C++ ver o arquivo “ApostilaCPP.pdf”.

Graphical User Interface (GUI)

Uma interfaces graficas para os programas desenvolvidos na disciplina podem ser facilmente desenvolvidas utilizando a ferramenta QT designer. Seguir os passos abaixo para desenvolver uma GUI para um programa generico, e utilizar os mesmos passos caso tenha interesse em desenvolver GUIs para outros programas:

1 – Executar o QT designer. No prompt do linux digitar:

```
% designer &
```

2 – Construir a GUI para uma aplicacao qualquer utilizando as facilidades existentes no QT designer (selecionar “File New” para chamar o wizard). Como exemplo sugere-se escrever uma aplicacao simples com apenas caixas de dialogo (Text Box), labels e alguns botoes (Next, Finish, Cancel, Help, ...). A aplicacao selecionada no wizard poderia ser um Dialog.

3 – Salvar a GUI utilizando “File/Save” (ou “Save as”). Isso vai criar um arquivo ASCII representando a GUI em XML (no exemplo foi utilizado o nome gui.ui para salvar a GUI criada).

4 – Utilizando novamente a linha de comando no Linux, trocar para o diretorio onde o arquivo XML foi criado (extencao ".ui") e digitar:

% uic -o gui.h gui.ui

Isso ira' criar um arquivo .h contendo o header para a classe dialog (escolhida como exemplo no wizard). A seguir digitar:

% uic -i gui.h -o gui.cpp gui.ui

Isso ira' criar o arquivo com a implementacao para as funcoes membro da classe dialog do arquivo gui.h

5 – Criar o seguinte programa em c++ para testar a GUI:

```
#include <qapplication.h>
#include "gui.h"

int main (int argc, char *argv[])
{
   QApplication app (argc, argv);

   Form1 test_gui;   // Form1 e' o campo "name" definido no
                     // QT designer para o form.
   app.setMainWidget(&test_gui);
   test_gui.show();
   int ret = app.exec();
   return ret;
}
```

6 – Gerar o executavel para o programa utilizando o compilador g++:

% moc -o moc_gui.cpp gui.h
% g++ -I$QTDIR/include gui.cpp testgui.cpp moc_gui.cpp -L$QTDIR/lib -lqt

7 – Executar o programa:

% ./a.out &