UFSC, CTC, Departamento de Engenharia Elétrica

Notas de aula do curso de Linguagem de Programação C++

Eduardo Augusto Bezerra <Eduardo.Bezerra@ufsc.br>

2018/2

Bibliografia utilizada:

[1] "The annotated C++ reference manual"; Ellis, M. A. and Stroustrup, B.; Addison-Wesley; 1990. Versão em português: "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.

Sumário

Classes Derivadas (mecanismo de herança)
(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 são herdados pela classe derivada.
- o operador :: é usado para referenciar explicitamente um membro da classe base.
- isso permite acesso a nome redefinido na classe derivada.

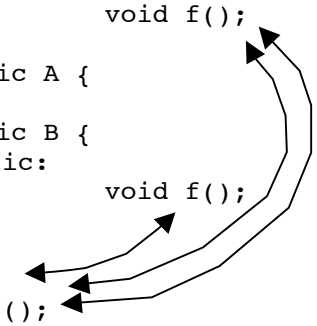
```
class Base {
          public:
                    int a, b;
};
class Derived : public Base {
          int b, c;
};
void f() {
          Derived d;
          d.a = 1;              // acesso ao a da classe Base
          d.Base ::b = 2;       // acesso ao b da classe Base
          d.b = 3;              // 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 é dita **direta** se for mencionada na lista base, e **indireta** se não é uma classe base direta, mas é uma classe base de uma das classes mencionadas na lista base.

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



```
 A
/ \
 |
 |

 B

/ \
 |
 |

 C
```

Classes Bases Múltiplas (herança múltipla)
(pg. 244 [1], pg. 257 [2])

Herança múltipla possibilita a utilização de recursos de diversas classes como ponto de partida para a definição de uma nova classe. O diagrama abaixo representa um exemplo de utilização de herança múltipla. A classe "Clock" armazena a hora, a classe "Calendar" armazena informações sobre a data. A nova classe "ClockCalendar" herda de "Clock" e "Calendar" formando um objeto único com informações 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 funções membro "setClock", "readClock", "setCalendar" e "readCalendar" são todas herdadas por "ClockCalendar" e funcionam em objetos do tipo "ClockCalendar" da mesma forma como funcionam para objetos "Clock" ou "Calendar".

Todas as três classes possuem uma função "advance()". A versão de "advance()" declarada em "ClockCalendar" sobrecarrega (*override*) as versões herdadas de "Clock" e "Calendar", entretanto as funções herdadas podem ser utilizadas na definição de ClockCalendar::advance():

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

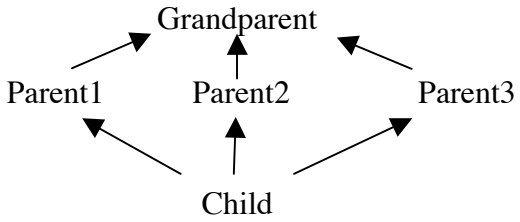
Caso a função “advance()“ não tivesse sido sobrecarregada, então a sequencia a seguir levaria a um erro de ambiguidade. O compilador não saberia qual “advance()” utilizar, o de “Clock” ou o de “Calendar”.

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

Como “advance()” foi sobrecarregada em “ClockCalendar”, a função acima irá chamar ClockCalendar::advance().

Classes Bases Virtuais

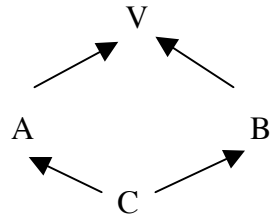
(pg. 248 [1], pg. 260 [2])



No diagrama anterior, as classes “Parent” herdam da classe “Grandparent”. A classe “Child”, por sua vez, recebe herança múltipla das classes “Parent”. O problema aqui é que “Child” possuirá copias duplicadas dos membros herdados de “Grandparent”.

Uma solução para esse tipo de problema é a utilização do conceito de classe virtual. Por exemplo:

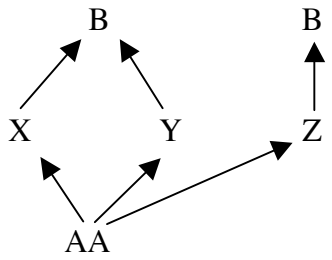
```
class V {
};
class A : virtual public V {
};
class B : virtual public V {
};
class C : public A, public B {
}
```



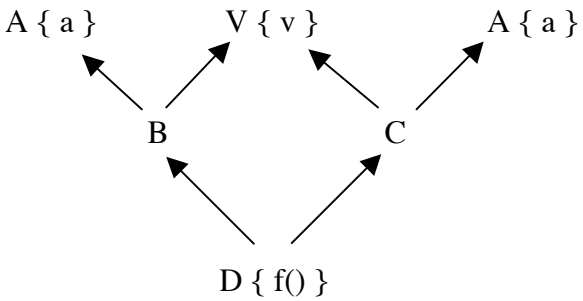
Um único subobjeto de V é compartilhado por todas as classes derivadas que especificam a classe base como virtual. A classe C possui apenas um subobjeto de V.

Uma classe pode possuir classes bases virtuais e não 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 situação a seguir representa um problema de ambiguidade considerando classes virtuais e não-virtuais.

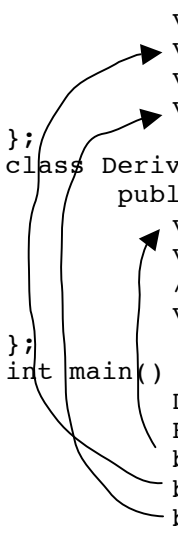



```
class V {   public:
                 int v;
};
class A {   public:
                 int a;
};
class B : public A, virtual public V {
};
class C : public A, virtual public V {
};
class D : public B, public C {
             public:
                 void f();
};
void D::f() {
     v++;      // ok, somente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

Funções Virtuais (pg. 257 [1])

Classes derivadas e funções virtuais são a chave para o projeto de muitos programas em C++ (orientação a objeto). Classe base define uma interface para a qual uma variedade de implementações são fornecidas por classes derivadas. O mecanismo de função virtual assegura que o objeto seja manipulado pelas funções definidas para ele.

A habilidade para chamar uma serie de funções usando exatamente a mesma interface é algumas vezes chamado de polimorfismo (funções virtuais proporcionam isso).



```
struct Base {
          virtual void vf1(){}
          virtual void vf2(){}
          virtual void vf3(){}
          void f(){}
};
class Derived : public Base {
       public:
          void vf1(){}          // Derived::vf1() ignora Base::vf1()
          void vf2(int){}       // nao existe vf2() em Derived
          //char vf3(){}        // erro! Difere so' no tipo de retorno.
          void f(){}            // f() nao e' virtual
};
int main() {
          Derived d;
          Base* bp = &d;        // conversao padrao Derived* para Base*
          bp->vf1();            // chama Derived::vf1()
          bp->vf2();            // chama Base::vf2()
          bp->f();              // nao eh virtual, logo chama Base::f()
}
```

- Se uma classe base (“Base”) contém uma função virtual “f” e uma classe derivada (“Derived”) também contém uma função “f” do mesmo tipo, então uma chamada de “f” por um objeto da classe derivada chama Derived::f, mesmo se acesso for por ponteiro ou referência a base.
- A função da classe derivada é dita “ignorar” a função da classe base.
- Se os tipos de funções forem diferentes então as funções são consideradas diferentes e o mecanismo virtual não é invocado.
- “bp->vf1()” chama “Derived::vf1()” pois “bp” aponta para um objeto de classe “Derived” no qual “Derived::vf1()” tinha ignorado a função virtual “Base::vf1()”.
- “bp->vf2()” é virtual. “d” é do tipo “Derived”, mas não existe “vf2()” em “Derived”, apenas “vf2(int)”.
- “bp->f()” não é virtual. Como “bp” é do tipo “Base”, logo chama base.
- Chamada de função virtual depende do tipo do objeto. Como “d” é do tipo “Derived”, “bp” chama os métodos virtualizados de “Derived”. Se “d” fosse do tipo “Base”, então chamaria apenas os métodos de “Base”.
- Chamada de função membro não virtual depende apenas do tipo do ponteiro ou referencia. Ex: “bp” é do tipo “Base”.

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

Funções Friend compartilham a maioria dos privilégios das funções membro, mas não são fortemente ligadas a nenhuma classe. Uma função pode ser friend de diferentes classes, mas não pode ser membro de apenas uma classe. Construtores, destrutores e funções virtuais (virtual) precisam ser funções membro e, consequentemente, não podem ser Friend. Uma friend de uma classe é uma função que não é um membro da classe, mas tem permissão para usar os membros privados e protegidos da classe. A declaração da friend não está no escopo da classe, e a friend não é chamada com os operadores de acesso a membro, a menos que ela seja um membro de outra classe.

```
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);
}
```

- Membros privados (private) de uma classe base são acessíveis apenas por funções membro e funções friend dessa classe base.
- Funções membro e funções friend de uma classe derivada não podem ver membros privados de uma classe base.
- Membros protegidos (protected) de uma classe base são acessíveis a funções membro e funções friend da classe base e de qualquer classe derivada.

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

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

A maior parte dos operadores podem ser sobrecarregados:

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

Os seguintes operadores não podem ser sobrecarregados:

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

Novos operadores não podem ser criados, apenas os existentes podem ser sobrecarregados de forma a implementar uma nova funcionalidade. Uma outra restrição é que em uma expressão utilizando um operador sobrecarregado, pelo menos um dos operandos precisa ser um objeto instanciado de uma classe.

Para redefinição de um operador é utilizada a palavra reservada "operator" seguida do operador a ser sobrecarregado: `double operator+= (double newValue);`

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

Outro exemplo de sobrecarga de operadores:

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

Considerando a seguinte definição da variável t: `ClockTime t = {11, 45, 20};`

E supondo a necessidade de imprimir o conteúdo da variável t utilizando o seguinte operador de saída:

```
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 observação importante é que como ClockTime é uma struct logo seus membros são public por default, e podem ser referenciados livremente na redefinição do operador. Porem, caso as variáveis da instancia fossem privadas, o operador teria que ser declarado como friend da classe ClockTime para possuir acesso as variáveis membro. A sobrecarga de operadores é uma situação onde funções friend são necessárias.

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

```
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;
}
```

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

Quando diversas declarações de funções diferentes são especificadas por um único nome no mesmo escopo, diz-se que esse nome está sobrecarregado. Quando esse nome é utilizado, a função correta é selecionada pela comparação dos tipos dos argumentos reais com os tipos dos argumentos formais.

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

Funções que diferem apenas no tipo de retorno não podem ter o mesmo nome.

Uma função membro de uma classe derivada NÃO está no mesmo escopo de uma função membro do mesmo nome em uma classe base. No exemplo a seguir, “D::f(char*)” não sobrecarrega “B::f(int)”.

```
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), que deixa
                       // de existir no escopo de D
   pd->B::f(1);        // ok
   pd->f("Ben");       // ok, chama D::f
}
```

Uma função declarada localmente não está no mesmo escopo que uma função em escopo de arquivo.

```
int f(char*);
void g() {
   extern f(int);
   f(“asdf”);   // erro! f(int) oculta f(char*), e nao ira' existir
                // nenhuma funcao f(char*) no escopo da funcao g()
}
```

Uma chamada de um determinado nome de função escolhe, entre todas as funções desse nome que estão no escopo e para as quais existe um conjunto de conversões tal que uma função poderia ser possivelmente chamada, a função que melhor corresponde aos argumentos reais. A função de melhor correspondência é a interseção entre os conjuntos de funções que melhor correspondem a cada argumento. A menos que essa interseção tenha exatamente um membro, a chamada é ilegal.

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

- Suportam a noção 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 fornecerão uma variedade de implementações.
- Pode ser usada unicamente como uma classe básica para outra classe.
- Objetos de uma classe abstrata não podem ser criados.
- Para ser abstrata precisa ter pelo menos uma função virtual pura.
- Uma função virtual é pura pelo uso de um especificador-puro (= 0) na declaração da função 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 não pode ser utilizada como:

- tipo de argumento;
- tipo de retorno de função;
- tipo para conversão explicita.

Classes abstratas podem ser ponteiros e referencias:

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

Funções virtuais puras são herdadas como funções virtuais puras:

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

Polimorfismo (pg. 266 [2])

Polimorfismo é a situação na qual objetos pertencentes a diferentes classes podem responder a uma mesma mensagem, normalmente de maneiras diferentes.

Considerar o programa a seguir:

```
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
}
```

No exemplo anterior, o vetor "list" (estrutura de dados homogênea) é utilizado para armazenar dois objetos diferentes, "parent" e "child". Isso é possível devido a utilização de ponteiros combinado com o conceito de herança.

Polimorfismo pode ser utilizado para simplificar, por exemplo, o trabalho para impressão dos valores de nodos em uma lista heterogênea. Em uma lista heterogênea cada nodo pode possuir objetos de classes diferentes, e o polimorfismo simplifica a manipulação desse tipo de lista.

Uma maneira mais interessante para implementar polimorfismo em C++ é por intermédio de funções virtuais. Ainda com relação ao exemplo anterior de implementação de listas heterogêneas em uma estrutura homogênea, uma forma mais eficiente para essa implementação deve considerar funções que possam ser utilizadas para qualquer objeto independentemente de sua classe. O exemplo anterior é reimplementado na listagem a seguir utilizando-se funções virtuais.

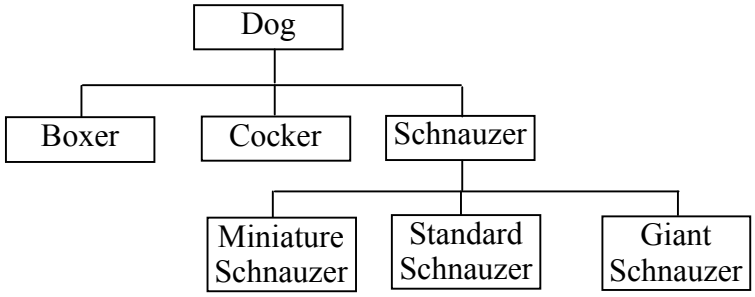
Notar que a palavra "virtual" é utilizada nas funções membro na classe base. As funções na classe derivada que sobrecarregam as funções da classe base são automaticamente assumidas como virtuais e não há necessidade de escrever a palavra "virtual" na frente da declaração dessas funções.

Em uma função **não-virtual**, a **declaração da variável ponteiro** é quem define qual definição de função utilizar (no exemplo: base ou derivada). Como a variável "p" no main (exemplo a seguir) é declarada do tipo "parent*", as funções definidas no parent serão sempre utilizadas.

No caso de funções **virtuais**, o que define a função a ser utilizada é a **classe do objeto apontado**. Por exemplo, se a variável "p" declarada no main como "parent*" apontar para um objeto "parent", a definição das funções do parent serão utilizadas. Por outro lado, caso a variável "p" (ainda declarada como parent*) apontar para um objeto "child", então as funções do child serão utilizadas. Notar no main do exemplo a seguir que a variável "p" é declarada como um ponteiro para "parent" porem a classe do objeto apontado por "p" é "child". Nesse caso o comando "p->setJ(4);" irá acionar a função "setJ" da classe "child" (a classe do objeto apontado é "child"). Por outro lado, caso "p" apontasse para um objeto "parent", então a função "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 é representado na figura a seguir. Diversas raças de cachorros são herdadas da classe “dog” que possui uma função para impressão da raça do objeto “dog” em uso. A função “print_breed” a ser executada será 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 anterior um objeto é declarado para cada classe (d, b, c, s, ms, ss, gs). O ponteiro "dp" é 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 funções virtuais das classes derivadas produzindo a seguinte saída:

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

O mesmo principio é utilizado a seguir onde "rd", "rs" e "rm" são 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" assumirá diversas formas, e as três ultimas chamadas ao método "printBreed()" resultarão na seguinte saída:

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

Template
(pg. 415 [2])

Template é um modelo a ser seguido por uma classe ou função. Um modelo de classe define o layout e as operações para um conjunto ilimitado de tipos relacionados. Por exemplo, um único modelo de classe List poderia fornecer uma definição comum para listas de "int", "float" e ponteiros para um objeto qualquer. Um modelo de função define um conjunto ilimitado de funções relacionadas. Por exemplo, um único modelo de função "sort()" poderia fornecer uma definição comum para a classificação de todos os tipos definidos pelo modelo de classe List.

Para uma explicação mais detalhada de Templates incluindo exemplos de programas em C++ ver o arquivo "ApostilaCPP.pdf".

Graphical User Interface (GUI)

Interfaces gráficas 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 genérico, 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 aplicação qualquer utilizando as facilidades existentes no QT designer (selecionar "File New" para chamar o wizard). Como exemplo sugere-se escrever uma aplicação simples com apenas caixas de dialogo (Text Box), labels e alguns botões (Next, Finish, Cancel, Help, ...). A aplicação 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 diretório onde o arquivo XML foi criado (extinção ".ui") e digitar: **% uic -o gui.h gui.ui**

Isso irá 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 irá criar o arquivo com a implementação para as funções 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 executável 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 &**