Índice

Parte I - Programação Orientada a Objetos

1 – Paradigmas de Programação

Programação orientada a objetos é uma técnica aplicável ao desenvolvimento de programas que procura melhorar a qualidade do processo de desenvolvimento de sistemas, enfatizando principalmente o reuso de *software*. Uma linguagem de programação orientada a objetos prove mecanismos de suporte a este estilo de programação.

Cabe destacar que uma linguagem é qualificada como ferramenta de suporte a um determinado estilo de programação se esta oferece facilidades que tornam conveniente o uso do estilo. Uma linguagem não oferece suporte a uma técnica de programação se exigir um esforço excepcional para o desenvolvimento de programas segundo esta técnica. Por exemplo, é possível escrever programas estruturados em Fortran e programas orientados a objetos em C, mas seriam tarefas desnecessariamente difíceis porque essas linguagens não dão suporte a esses estilos.

Programação Procedural

O primeiro paradigma de programação, e provavelmente o mais usado, é:

> Decida que procedimentos são necessários;
> use os melhores algoritmos encontrados para eles.

Neste caso, o foco é no processamento, isto é, no algoritmo necessário para se fazer a computação desejada. Linguagens suportam esse paradigma provendo facilidades para passagem de parâmetros para funções e retorno de valores das funções.

A literatura relacionada a este estilo de programação discute, entre outros assuntos, os modos de passagem de parâmetros, maneiras de distinguir diferentes tipos de parâmetros, diferentes tipos de funções (procedimentos, rotinas, macros,...). Fortran é um exemplo de linguagem de programação que dá suporte a esta técnica.

Programação Modular

Ao longo dos anos, a ênfase no projeto de programas tem se transferido do projeto de procedimentos para a organização dos dados. Entre outras coisas, isto reflete um aumento no tamanho dos programas. Um conjunto de procedimentos relacionados com os dados que eles manipulam é freqüentemente chamado de *módulo*.

> Decida que módulos você quer;
> particione o programa de maneira que os dados fiquem escondidos (protegidos) nos módulos.

Onde não existe um agrupamento de funções relacionadas com um determinado dado, a programação precedural é usada. As técnicas para projetar “bons procedimentos” são agora aplicadas a cada função em um módulo. Modula-2 dá suporte a este estilo de programação, enquanto C meramente permite o seu uso.

> Apesar de C++ não ter sido especificamente projetado para suportar programação modular, seu conceito de classe prove um suporte ao conceito de módulo. Além disso, C++ suporta a noção de módulos de C, que são implementados através de unidades de compilação separadas.

Abstração de Dados

A programação com módulos permite centralizar todo o controle de um determinado tipo de dado em um módulo gerente deste tipo. Isto é certamente um grande avanço sobre a abordagem não estruturada. Porém, “tipos” implementados deste modo são claramente diferentes dos tipos pré-definidos de uma linguagem.

Cada módulo gerente de um tipo deve definir um mecanismo separado para criação de variáveis de seu tipo, não existindo nenhuma norma para atribuição de identificadores de instâncias do tipo. Uma variável de um desses tipos também não obedece as regras usuais de escopo e passagem de parâmetros.

Um tipo criado através de um mecanismo de módulo difere dos tipos primitivos em vários aspectos importantes, e conta com um suporte da linguagem bem inferior ao dado aos tipos primitivos. O problema é que o usuário programa em termos de descritores de *objetos* invés de usar os próprios objetos. Isto implica na perda de controle pelo compilador de determinados erros de manipulação dessas instâncias.

Em outras palavras, o conceito de módulos, que suporta o paradigma de *data hiding*, simplesmente permite o uso da técnica de abstração de dados mas não lhe dá um suporte adequado.

Linguagens como Ada, Clu e C++ resolvem este problema permitindo ao usuário definir tipos que se comportam de uma maneira muito semelhante a dos tipos pré-definidos. Esses tipos são normalmente chamados de *tipos abstratos de dados*.

Onde não existe a necessidade de mais de um objeto de um tipo, a programação com módulos é suficiente.

Decida que tipos são necessários; ofereça um conjunto completo de operações para cada tipo.

Programação Orientada a Objetos

Um tipo abstrato de dados (*TAD*) define uma espécie de caixa preta. Uma vez que ele tenha sido definido, ele não tem como interagir com o resto do programa. Não existe um modo de adaptá-lo para novos usos, exceto pela modificação de sua definição. Isto pode acarretar em várias inflexibilidades e inconveniências, tal como a introdução de erros em um código que já era confiável mas teve que ser alterado para permitir a extensão de um TAD.

A possibilidade de se estender um determinado TAD, ou melhor, especializar um TAD através da criação de um outro, é a característica mais marcante do paradigma de programação orientada a objetos. Linguagens que ofereçam um mecanismo para isto, como o mecanismo de herança introduzido por Simula e adotado por C++, são consideradas linguagens orientadas a objetos. De um modo geral, na literatura de programação e linguagens orientadas a objetos, os TADs recebem o nome de *classes* e suas instâncias são chamadas de objetos.

Decida que classes são necessárias; ofereça um conjunto completo de operações para cada tipo; torne explícita a similaridade usando herança.

2 – Programação Orientada a Objetos

Usando a técnica de orientação a objetos, o desenvolvedor analisa as classes de objetos do sistema. O projeto do sistema será baseado nos sucessivos melhoramentos de sua compreensão dessas classes de objetos.

Esta idéia contraria algumas das regras estabelecidas para o projeto de software, que apresentam o desenvolvimento de sistemas como uma tarefa que visa exclusivamente atender à funcionalidade do sistema expressa por seus requisitos.

Uma das dificuldades iniciais da programação orientada a objetos é a identificação das classes pertinentes a um sistema. Não existe nenhuma regra bem estabelecida para esta identificação, mas algumas considerações podem ser feitas. Programas são usados para se obter respostas para certas questões do ambiente externo ao sistema (como em programas feitos para resolver um problema), para interagir com o mundo (como em um sistema de controle de processos), ou para criar novas entidades (como em um editor de texto ou um compilador). Em cada caso, o sistema deve ser baseado na descrição de alguns aspectos do ambiente externo que são relevantes para a aplicação.

Um sistema bem projetado pode ser visto como um modelo operacional de alguns aspectos de entidades reais. Quando o projeto de software é encarado desta forma, a técnica de orientação a objetos se torna uma abordagem natural: o mundo a ser modelado é composto de objetos, e seria apropriado organizar o modelo

através de representações lógicas desses objetos. Os objetos de um sistema simplesmente refletem objetos externos.

> Definição: a técnica de programação orientada a objetos considera a construção de sistemas de software como uma coleção estruturada de implementações de tipos abstratos de dados.

Em uma arquitetura orientada a objetos, cada classe será a implementação de um tipo abstrato de dados, que consiste de um conjunto de estruturas de dados descrito pelos serviços que fazem parte de sua interface oficial e pelas propriedades destes serviços.

Note como estes conceitos estão de acordo com o princípio de *information hiding*. Se uma classe é a implementação de um TAD, sua interface corresponde exatamente aos serviços da especificação de um TAD. Por exemplo, a interface de uma classe pilha, oferecendo uma visão coerente de pilhas para o ambiente externo, incluiria serviços como *push*, *pop*, *top* e *empty*. A classe pode conter outras operações auxiliares usadas internamente para propósitos de implementação, mas estas não devem fazer parte da interface.

Classes devem ser projetadas para serem reutilizadas em vários sistemas diferentes. Assim, o desenvolvimento de sistemas é visto como uma construção *bottom-up* a partir das classes existentes.

Um aspecto muito importante da programação orientada a objetos é como as classes são agrupadas de uma maneira estruturada, isto é, quais são os relacionamentos existentes entre classes. Especificamente, existem dois tipos de relação: cliente (agregação) e descendente (herança).

- Uma classe é cliente de outra quando ela faz uso dos serviços da outra classe, definidos em sua interface. Por exemplo, uma classe de pilha poderia usar um *array* para sua implementação, e assim ser um cliente de uma classe array. Este é o conceito de agregação.
- Uma classe é descendente de uma ou mais classes quando ela é projetada como uma extensão ou especialização dessas classes. Este é o conceito de herança.

3 – Linguagens Orientadas a Objetos

Tendo em mente as características da programação orientada a objetos, veremos agora as principais características que uma linguagem de programação deve ter para dar suporte a esta técnica:

- *Estrutura modular baseada em objetos:* sistemas são modularizados com base em suas estruturas de dados.
- *Abstração de dados:* objetos devem ser descritos como implementações de tipos abstratos de dados.
- *Gerenciamento automático de memória:* objetos não mais usados devem ser liberados pelo sistema de suporte da linguagem, sem a intervenção do programador.
- *Classes*: todo tipo não-primitivo é um módulo (classe), e todo módulo de alto nível é um tipo.
- *Herança:* uma classe pode ser definida como uma extensão ou restrição de outras.
- *Polimorfismo e Ligação Tardia (Late Binding):* entidades do programa devem poder fazer referência a objetos de mais de uma classe, e operações devem poder ter diferentes implementações em diferentes classes, sendo que em tempo de execução é feita automaticamente a seleção da versão correspondente a instância em questão.

A terceira característica não é propriamente uma característica da linguagem mas sim de seu sistema de suporte. Porém, o projeto da linguagem pode facilitar ou dificultar a implementação de um sistema de coleta de lixo. No caso específico de C++, esta característica não é atendida. Isto é, não existe uma implementação comercial de C++ que apresente um sistema de coleta de lixo. Com o uso da linguagem, fica claro como essa característica faz falta à programação orientada a objetos.

A última característica pode ser implementada em uma linguagem de diferentes maneiras. No projeto de C++, conciliou-se esta característica com a noção de tipos estáticos. Todo objeto em C++ tem um tipo estático (classe), e os tipo dinâmicos que ele pode assumir são restritos (ou pelo menos *quase* restritos) a tipos descendentes de sua classe. A ligação tardia de C++ é implementada através de um mecanismo de redefinição de operações (*overloading*) em classes descendentes (*subclasses*).

Parte II - A Linguagem C++

1 – A Evolução do C++

Em 1980, classes e algumas outras características foram acrescentadas ao C, o que resultou na linguagem chamada *C with Classes*. Esta linguagem foi descrita por Bjarne Stroustrup em alguns artigos.

Em 1983/84, *C with Classes* foi reprojetada, estendida e re-implementada. O resultado disto foi a primeira versão da linguagem C++. As principais extensões foram funções virtuais e sobrecarga de operadores. Depois de mais alguns refinamentos, C++ ficou disponível para o público em geral em 1985 e foi documentado por Bjarne Stroustrup em seu livro *"The C++ Programming Language"*.

Esta apostila apresenta C++ como ele existe atualmente, depois da inclusão de suporte para herança múltipla, classes abstratas, *templates*, tratamento de exceções e outros melhoramentos. Esses dois últimos foram incluídos na linguagem recentemente e podem não estar disponíveis em alguns compiladores.

Como C++ tem sido usado em grandes projetos de software, estabilidade e compatibilidade têm sido importantes considerações no desenvolvimento da linguagem, da mesma forma que eficiência em relação a espaço e tempo de execução.

O principal objetivo em estender C++ tem sido aprimorá-lo como uma linguagem para abstração de dados e programação orientada a objetos em geral e como ferramenta para escrever bibliotecas com uma alta qualidade.

2 – Classes

Classes de C++ são extensões das estruturas de C (`struct`), usadas para definir tipos do usuário. Além dos campos de dados que já podiam ser definidos em estruturas de C, C++ permite que estruturas tenham também funções em sua composição. Essas funções que fazem parte de uma estrutura são chamadas muitas vezes de *métodos* ou *funções membro*.

Para ilustrar o uso de métodos, vamos considerar a implementação de um tipo data usando o mecanismo de estruturas de C para a representação de uma data e um conjunto de funções para manipular variáveis deste tipo. A implementação disto poderia ser da seguinte forma:

```
struct date { int month, day, year; };
struct date today;
void set_date(struct date* d, int month, int day, int year)
{
   d->month = month;
   d->day   = day;
   d->year  = year;
}
void next_date(struct date* d)
{
   ...
}
void print_date(const struct date* d)
{
   ...
}
```

Desta forma, não existe nenhuma conexão explícita entre as funções de manipulação e o tipo de dado. Esse problema é resolvido por C++ através do uso de funções membro para manipular a estrutura de dados. Essa estrutura em C++ poderia ser definida assim:

```
struct date
{
   int month, day, year;

   void set(int,int,int);
   void get(int*,int*,int*);   //nao seria necessario
   void next();
   void print();
};
```

Métodos são declarados desta forma e só podem ser chamados sobre uma variável específica do tipo apropriado, usando a sintaxe padrão para acesso de membros de uma estrutura:

```
date today;
date my_birthday;

void f()
{
   my_birthday.set(15,8,1970);
   today.set(15,5,1995);

   my_birthday.print();
   today.next();
   today.print();
}
```

Como diferentes estruturas podem ter métodos com o mesmo nome, deve-se especificar a que estrutura um método pertence quando este for definido. Por exemplo:

```
void date::set(int d, int m, int y)
{
   day   = d;
   month = m;
   year  = y;
}
```

O Ponteiro *this*

Em um método, os nomes de membros de sua estrutura podem ser usados sem uma referência explícita para o objeto. Neste caso, os nomes fazem referência para os membros do objeto para o qual a função foi chamada. Porém, as vezes pode ser necessário ter uma maneira de um objeto se referenciar de dentro da chamada de um de seus métodos. Isto pode ser útil para resolver ambigüidades de seus membros com outras variáveis ou para permitir a passagem desta referência como parâmetro para uma outra função.

Para isto, em cada método de uma classe A, o ponteiro `this` é declarado implicitamente como:

```
A* const this;
```

Este ponteiro é inicializado automaticamente com uma referência para o objeto sobre o qual o método foi aplicado. Como o ponteiro `this` é declarado com o modificador `const` (um ponteiro constante), seu valor não pode ser alterado pelo programador. Desta forma, poderíamos ter implementado o método `date::set` da seguinte forma:

```
void date::set(int day, int month, int year)
{
   this->day   = day;
   this->month = month;
```

```
   this->year  = year;
}
```

O uso do `this` para acessar membros de uma classe, de um modo geral, é desnecessário. A sua principal utilidade é para escrever funções que manipulam ponteiros diretamente. Um exemplo típico é uma função que insere um objeto em uma lista duplamente encadeada:

```
struct dlink
{
   dlink* pre;   //antecessor
   dlink* suc;   //sucessor

   void append(dlink*);
   ...
};

void dlink::append(dlink* p)
{
   p->suc = suc;
   p->pre = this;
   suc->pre = p;
   suc = p;
}
```

Funções *Inline*

Na implementação de classes, é muito comum o uso de várias pequenas funções, tipicamente para fazer consultas ao objeto. Usando o mecanismo tradicional de geração de código para chamada de funções, poderíamos ter um problema sério de eficiência com a chamada dessas funções. O tempo gasto com a chamada da função poderia ser maior do que o tempo gasto com o processamento em si feito pela função. Um exemplo disto seria uma classe de números complexos com a seguinte estrutura:

```
struct Complex
{
   float real;
   float img;
   ...
   float get_real();
   float get_img();
};

float Complex::get_real()
{
   return real;
}

float Complex::get_img()
{
   return img;
}
```

O mecanismo de funções *inline* foi projetado para resolver esse problema. Funções *inline* têm um comportamento muito similar às macros do C, e têm limitações quanto a sua complexidade. Funções deste tipo, ao invés de serem chamadas, têm o seu corpo expandido no lugar de sua chamada.

Um método definido (não somente declarado) na declaração de uma classe é feito *inline*, sendo que também é possível definir um método *inline* fora da declaração de sua classe usando o modificador `inline`.

O exemplo a seguir mostra a classe anterior de números complexos refeita usando as duas maneiras de se definir uma função *inline*:

```
struct Complex
{
   float real;
   float img;
   ...
   float get_real() { return real; }  //esta e um funcao inline
   float get_img();
};

inline float Complex::get_img()  //outra funcao inline
{
   return img;
}
```

Membros Estáticos

Em C++, membros de uma classe podem ser **static**. Quando uma variável é declarada **static** dentro de uma classe, todas as instâncias de objetos desta classe compartilham a mesma variável. Uma variável **static** tem seu espaço alocado durante a linkedição do programa, da mesma forma que as variáveis globais.

Como uma variável estática é única para todos os objetos da classe, não é necessário um objeto para referenciar este campo. Como este tipo de variável é semelhante a uma variável global, é necessário defini-la em algum ponto do código, além de sua declaração na classe.

Métodos **static** são semelhantes. Eles são únicos para todos os objetos, e não precisam de um objeto para serem chamados. Por não precisarem de um objeto, somente os dados **static** podem ser acessados por um método **static**.

```
struct Example
{
   static int value; // declara a variável
   static void setvalue(int);
   int id;
};

int Example::value = 0;  // define a variável
void Example::setvalue(int newvalue)
{ value = newvalue; }

void main()
{
   Example ex1, ex2;

   ex1.value = 1;            // Example::value=1
   ex2.value = 2;            // Example::value=2
   Example::setvalue(10);    // Example::value=10

   printf("%d\n", Example::value); // usa value sem nenhum objeto
}
```

Se a definição

```
int Example::value;
```

for omitida, o código compila, mas na hora da linkedição é gerado um erro de símbolo indefinido.

Passagem por Referência

Em C, só existe passagem de parâmetro por valor. Se for necessário passar um parâmetro por referência, é preciso simular isto passando o ponteiro para a variável. Uma característica de C++, que não está diretamente relacionada com orientação a objetos, é que ele permite passagem de parâmetros por referência de uma maneira semelhante a Pascal:

```
int a = 0, b = 0, c = 0;

void f ( int value, int& ref, int* ptr)
{
   value = 1;   //nao altera o valor de a (value e uma copia de a)
   ref   = 2;   //altera o valor de b
   *ptr  = 4;   //altera o valor de c
}

void main()
{
   f(a,b,c);
   //a = 0, b = 2, c = 4
   ...
}
```

Valor *Default* para Parâmetros

Em C++, existe a possibilidade de definir valores *default* para parâmetros de uma função. Por exemplo:

```
void printstr( char* str, int x=-1, int y=-1);

void main()
{
   printstr("na posição 10,10", 10, 10);
   printstr("logo depois");
}

void printstr(char* str, int x, int y)
{
   if (x==-1) x = wherex();
   if (y==-1) y = wherey();
   gotoxy(x,y);
   cputs(str);
}
```

Declarações como esta podem tornar um programa confuso de se ler. Valores *default* só devem ser usados caso um parâmetro vá ter o mesmo valor na maior parte das chamadas.

Exemplo

Para exemplificar o que foi visto até aqui, apresentamos a implementação de uma classe que modela um conjunto de números inteiros. A interface desta classe poderia ser:

```
struct intset
{
   int cursize, maxsize;
   int* set;

   void init(int size);
   void destroy();
```

```
   int  member(int) const;
   void insert(int);
   void start(int& i) const { i = 0; }
   int  ok(int& i ) const { return i < cursize; }
   int  next( int& i ) const { return set[i++]; }
};
```

Esta classe poderia ter a seguinte implementação para seus métodos:

```
void intset::init( int size )
{
   if (size < 1) error("tamanho invalido para o conjunto");
   cursize = 0;
   maxsize = size;
   set = (int*) malloc(maxsize*sizeof(int));
}

void intset::destroy()
{
   free(set);
   set = NULL;
}

void intset::insert(int e)
{
   int i = cursize;

   if (++cursize > maxsize) error("numero de elementos ultrapassou o
limite");

   while ( i>0 && e<set[i-1] ) //mantem o array ordenado
   {
      set[i] = set[i-1];
      i--;
   }
   set[i] = e;
}

int intset::member(int e) const  // busca binaria
{
   int l = 0;
   int u = cursize-1;

   while (l<=u)
   {
      int m = (l+u)/2;
      if (e < set[m])
         u = m-1;
      else if (e > set[m])
         l = m+1;
      else
         return 1;          // achou o elemento
   }
   return 0;                // nao achou o elemento
}
```

Para testar esta classe, faremos um programa que insere números aleatórios em um objeto da classe conjunto de inteiros e depois consulta o conjunto de forma a percorrer todos os seus elementos e os imprime na tela. O programa apresentado a seguir recebe dois parâmetros da linha de comando. O primeiro é o número máximo de elementos que o conjunto criado pelo programa poderá ter, e o segundo é o limite máximo da faixa de valores que poderão ser usados. Eis o programa:

```
void main(int argc, char* argv[])
{
   if (argc != 3) error("dois argumentos sao esperados");

   int count = 0;
   int m = atoi(argv[1]);
   int n = atoi(argv[2]);
   intset set;

   set.init(m);
   while (count < m)
   {
      int t = randint(n);
      if (set.member(t) == 0)
      {
         set.insert(t);
         count++;
      }
   }

   print_in_order(set);
}
```

As seguintes funções auxiliares são usadas neste programa:

```
static void error(const char* s)
{
  cerr << "set: " << s << '\n';
  exit(1);
}

static int randint(int u)
{
   int r = rand();
   if (r<0) r = -r;
   return 1 + r%u;
}

static print_in_order(intset& set)
{
   int var;
   set.start(var);
   while (set.ok(var))
      cout << set.next(var) << '\n';
}
```

3 – Controle de Acesso

Como foi visto anteriormente, o que define um TAD é somente a sua interface, ou seja, como ele é manipulado. Um TAD pode ter diversas implementações possíveis, e, independentemente desta ou daquela implementação, objetos deste tipo serão usados sempre da mesma forma. Os atributos além da interface, ou seja, os atributos dependentes da implementação, não precisam e não devem estar disponíveis para o usuário de um objeto, pois este deve ser acessado exclusivamente através da interface definida. O ato de esconder estas informações é chamado de encapsulamento.

Os mecanismos apresentados até aqui permitem a definição da interface em um tipo, permitindo códigos bem mais modulares e organizados. No entanto, não permitem o encapsulamento de dados e/ou código.

public, private e protected

Parte do objetivo de uma classe é esconder o máximo de informação possível. Então é necessário impor certas restrições na maneira em que uma classe pode ser manipulada, e que dados e código podem ser usados. Existem três tipos de usuário de uma classe:

- A própria classe
- Usuários genéricos
- Classes derivadas (este ponto será visto posteriormente)

Cada um destes três tipos tem privilégios de acesso diferenciados. Cada um destes níveis tem uma palavra reservada associada:

- **private**
- **public**
- **protected**

Os dois exemplos abaixo são equivalentes e ilustram o uso destas novas palavras reservadas.

```
struct Sample
{
 private:
   int a;
   int private_function( char* b );

 protected:
   int b;
   int protected_function( float );

 public:
   int c;
   float d;
   void public_function( controle* );
};
```

```
struct Sample
{
 private:    int a;
 protected:  int b;
 public:
   int c;
   float d;
   void public_function( controle* );

 private:
```

```
    int private_function( char* b );

 protected:
   int protected_function( float );
};
```

A diferença entre declarações com **struct** e **class** está no nível de proteção *default*. Caso não seja especificado nenhum qualificador, os membros de uma **struct** são públicos, enquanto que em uma classe, os membros são privados.

Atributos **private** são os mais restritos. Somente a própria classe[i] pode acessar os atributos privados. Ou seja, somente os métodos da própria classe têm acesso a estes atributos.

```
class Sample
{
   int value;
   void f1(){ value = 0; }
   int  f2() { return value; }
};

int main()
{
   Sample p; // cria um objeto do tipo Sample

   // todos os membros são private, não se pode fazer nada com p

   p.f1();                  // inválido
   printf("%ld", p.f2(); ); // inválido

   return 0;
}
```

No exemplo anterior, todos os membros eram **private**, o que impossibilitava o uso de um objeto da classe `Sample`. Para que as funções `f1` e `f2`, por exemplo, pudessem ser usadas, elas precisariam ser membros **public**. O código a seguir é uma modificação da declaração da classe `Sample` para tornar as funções `f1` e `f2` públicas e portanto passíveis de serem chamadas em `main`.

```
class Sample
{
   int value;
 public:
   void f1(){ value = 0; }
   int  f2() { return value; }
};
```

Membros **protected** serão explicados quando forem apresentados o conceito de herança e classes derivadas.

Para exemplificar, vamos reconsiderar a implementação de um conjunto de inteiros. Nesta nove versão, vamos considerar apenas as operações essenciais de inserir e retirar um elemento, verificar se um elemento pertence ao conjunto e a cardinalidade do conjunto. Então o novo conjunto terá pelos menos a seguinte interface:

```
class intset
{
 public:
```

[i] Ou classes e funções declaradas **friend**, explicado a seguir.

```
   void insert(int n);
   void remove(int n);
   int  member(int n);
   int  size();
};
```

Se formos implementar este conjunto usando listas encadeadas, usaríamos uma estrutura auxiliar elemento que seriam os nós da lista. A nossa classe teria então ainda uma variável **private** que seria o ponteiro para o primeiro elemento da lista. Outra variável **private** seria um contador de elementos. Vamos acrescentar ainda um método para limpar o conjunto, para ser usado antes das funções do conjunto propriamente ditas. Eis a definição da nossa classe:

```
class intset
{
   struct  listElem {
      listElem *next;
      int value;
   };
   listElem* list;
   int nElems;
 public:
   void clean();
   void insert(int n);
   void remove(int n);
   int  member(int n);
   int  size();
};
```

Friend

Algumas vezes duas classes são tão próximas conceitualmente que seria desejável que uma classe tivesse acesso irrestrito aos membros da outra.

No exemplo anterior, não há meio de percorrer o conjunto para, por exemplo, imprimir todos os elementos, como era possível na primeira versão do nosso conjunto de inteiros. Para fazer isto, seria preciso ter acesso ao dado `list` e à estrutura `listElem`, ambos *private*. Uma maneira de resolver este problema seria aumentar a interface do conjunto oferecendo as mesmas funções para percorrer os elementos que faziam parte da primeira versão da classe `intset`. Entretanto, esta solução é artificial, pois estas operações não fazem parte do TAD conjunto.

A solução normalmente usada é a idéia de iteradores. Um iterador atua sobre alguma coleção de elementos e a cada chamada retorna um elemento diferente da coleção, até que já tenha retornado todos. Um iterador para o nosso conjunto seria:

```
class IteratorSet
{
   intset::listElem *current;
 public:
   void init( intset* set) { current = set->list; }
   int theend() { return current==NULL; }
   int nextElem()
   { int n=current->value; current=current->next; return n; }
};

void main()
{
   intset set;  // cria conjunto
   set.clean();  // inicializa para operações
```

```
   set.insert(10); //insere o elemento 10
  // ...
   IteratorSet it; // cria um iterador
   it.init( set ); // inicializa o iterador com set
   while (!it.theend())   // percorre todos os elementos
      printf("%d\n", it.nextElem() );  // imprimindo-os
}
```

O problema aqui é que o iterador usa dados privados de `intset`, o que gera erros durante a compilação. Esse é um caso em que as duas classes estão intimamente ligadas, e então seria conveniente, na declaração de `intset`, dar acesso irrestrito à classe `IteratorSet`.

Para isso existe a palavra reservada **friend**. Ela serve para oferecer acesso especial à algumas classes ou funções. Assim, a classe `intset` poderia ter a seguinte forma final:

```
class intset
{
   friend class IteratorSet;
   struct  listElem {
      listElem *next;
      int value;
   };
   listElem* list;
   int nElems;
 public:
   void clean();
   void insert(int n);
   void remove(int n);
   int  member(int n);
   int  size();
};
```

Funções também podem ser declaradas como **friend** de uma classe. Um exemplo de função **friend** seria:

```
class No {
   friend int readValue( No* ); // dá acesso privilegiado à função
   int value;
 public:
   void setValue( int v ) { value=v; }
};

int readValue( No* n )
{
   return n->value; // acessa dado private de No
}
```

O recurso de classes e funções amigas deve ser usado com cuidado, pois isto é um furo no encapsulamento dos dados de uma classe. Projetos bem elaborados raramente precisam lançar mão de classes ou funções **friend**.

4 – Sobrecarga

Apesar de o nome sobrecarga estar associado com uma coisa a ser evitada (sobrecarga de circuitos elétricos etc.), em C++ sobrecarga é mais um novo e poderoso recurso. Sobrecarga se refere a carregar uma função com mais de um significado. A possibilidade de existir funções com o mesmo nome significa que uma função não é mais identificada somente pelo seu nome, mas sim pela sua *assinatura* completa.

Para um programador C isto pode parecer uma aberração, mas no dia a dia nos deparamos com situações idênticas no uso da nossa linguagem. Consideremos o verbo tocar. Podemos usá-lo em diversas situações. Podemos tocar violão, tocar um *compact disc* etc. Em C++, diríamos que o verbo tocar está sobrecarregado. Cada contexto está associado a um significado diferente, mas todos estão conceitualmente relacionados.

Sobrecarga de funções

Voltando ao C++, consideremos agora uma função `display`, que pega um argumento e o imprime na tela. Seria interessante se pudéssemos usar esta função com qualquer tipo de argumento, deixando o compilador escolher qual código usar em cada situação. Por exemplo:

```
display("funcao display com string");
display( 1230 );
display( 3.1416 );
```

Na realidade, muitas linguagens de programação usam sobrecarga internamente, mas não deixam este recurso disponível para o programador. C é uma dessas linguagens. Consideremos o código:

```
a = a + 125;
b = 3.1416 + 1.48;
```

O operador + está sobrecarregado; no primeiro caso estamos nos referindo à uma função que recebe `(int, int)`. No segundo, a soma recebe `(float, float)`.

Seria desagradável se os operadores usados para soma de inteiros e reais fossem diferentes, já que conceitualmente eles fazem a mesma coisa. O motivo pelo qual se usou sobrecarga aqui foi a simplicidade. Em C++ é possível definir funções sobrecarregadas.

O compilador se encarrega de escolher qual função usar de acordo com os parâmetros passados.

Obs.: Isto significa que o compilador, quando se depara com uma chamada
```
display("string");
```
não vai procurar o nome _display, como em C. O nome da função internamente será _display mais uma codificação com os tipos dos parâmetros, por exemplo, _display@pc. Por isso, o compilador precisa diferenciar uma função C pura de uma C++ na hora da chamada, para saber que nome procurar. Para declarar uma função C no código C++, usa-se
```
extern "C" void f(int);
```
ou
```
extern "C"{ ... }
```

É importante saber que o tipo de retorno não é usado para distinguir funções. As funções

```
void display(int);
int  display(int);  // errado!!!!!
```

não são consideradas diferentes, o que gera um erro de compilação.

Tanto funções globais como funções membro podem ser sobrecarregadas. Exemplo de uso de sobrecarga:

```
#include <stdio.h>

void display( char *v ) { printf("%s", v); }
void display( int   v ) { printf("%d", v); }
void display( char  v ) { printf("%c", v); }
void display( float v ) { printf("%f", v); }

void main()
{
   display("funcao display com string");
```

```
   display( 100 );
   display( 'A' );
   display( 3.1416 );
}
```

As regras para decidir que função chamar em caso de sobrecarga são extremamente complexas. Eventualmente, o compilador pode dar um erro se não for possível uma escolha.

Sobrecarga de operadores

O uso de funções sobrecarregadas não só uniformiza chamadas de funções para diferentes tipos de objetos como também permite que os nomes sejam mais intuitivos. Se um dos objetivos da sobrecarga é permitir que as funções sejam chamadas pelo nome mais natural possível, não importa se o nome já foi usado, porque não deixar o programador sobrecarregar também os operadores?

Na realidade, um operador executa algum código com alguns parâmetros, assim como qualquer função. A aplicação de um operador é equivalente à chamada de uma função. Em C++ é permitido sobrecarregar um operador, com o objetivo de simplificar a notação e uniformizar a expressão.

Existem duas maneiras de implementar operadores para classes de objetos: como funções membro e como funções globais. Por exemplo, dado um objeto w e um operador unário !, a expressão

```
!w
```

é equivalente às chamadas de funções

```
w.operator!();   // usando uma função membro
operator!(w);    // usando uma função global
```

Vejamos agora como seria com um operador binário, por exemplo, &. A expressão

```
x & y
```

é equivalente às chamadas

```
x.operator&(y); // usando uma função membro
operator&(x,y); // usando uma função global
```

Um detalhe importante é que uma função `y::operator&(x)` nunca será considerada pelo compilador para resolver a expressão `x&y`, já que isto implicaria que o operador é comutativo.

Antes do primeiro exemplo, precisamos ter em mente que C++ não permite a criação de novos operadores; só podem ser redefinidos os operadores que já existem na linguagem. Isto implica que, por exemplo, o operador / será sempre binário. Outra característica é que a prioridade também não pode ser alterada, é preservada a original do C.

Para exemplificar, vamos definir uma classe que modela números complexos:

```
struct  Complex {
   float i, j;
}
```

Para trabalhar numericamente com objetos desta classe, seria ótimo se pudéssemos manipulá-los assim como qualquer inteiro ou real, ou seja, usando operadores. Afinal, as operações +, -, *, / etc. são bem definidas para números complexos. Podemos então fazer:

```
struct  Complex {
   float i, j;
   int operator==(Complex& c)
   {
      return i==c.i && j==c.j;
   }
   Complex operator+(Complex& c)
   {
      Complex n;
```

```
      n.i = i + c.i;
      n.j = j + c.j;
      return n;
   }
   Complex operator-(Complex& c)
   {
      Complex n;
      n.i = i - c.i;
      n.j = j - c.j;
      return n;
   }
   Complex operator*(Complex& c)
   {
      Complex n;
      n.i = i * c.i - j * c.j;
      n.j = i * c.j + j * c.i;
      return n;
   }
   Complex operator/(Complex& c)
   {
       Complex n;
       float den = c.i * c.i + c.j * c.j;
       n.i = ( i * c.i + j * c.j ) / den;
       n.j = ( j * c.i - i * c.j ) / den;
       return n;
   }
};
```

A maior parte dos operadores podem ser sobrecarregados. São eles:

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

Tanto as formas unárias como as binárias de

```
+ - * &
```

podem ser sobrecarregadas, assim como as formas pré-fixadas ou pós-fixadas de

```
++ --
```

Os seguintes operadores não podem ser sobrecarregados:

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

já que estes operadores já tem um significado predefinido (exceto ?:) para objetos de qualquer classe.

A função de atribuição `operator=()` é definida por default como a atribuição ao nível de membro, e não pode ser redefinido como uma função global.

5 – Construtores e Destrutores

Como o nome já diz, um construtor é uma função usada para construir um objeto de uma dada classe. Ele é chamado assim que um objeto é criado. Analogamente, os destrutores são chamados assim que os objetos são destruídos.

No exemplo onde foi implementada a primeira versão da classe `intset`, foi necessária a introdução da função `init`, usada para inicializar o estado do objeto. Se esta função não for chamada antes de tudo, os resultados são imprevisíveis. Deixar esta chamada a cargo do programador é aumentar o potencial de erro do programa. Na realidade, a função `init` deveria ser um construtor de `intset`. O compilador garante que o construtor é a primeira função a ser executada sobre um objeto.

A mesma coisa acontecia em `IteratorSet`. O método `init` deveria ser um construtor, pois antes desta chamada o estado do objeto é inconsistente.

Os construtores e destrutores são métodos especiais. Nenhum dos dois tem um tipo de retorno, e o destrutor não pode receber parâmetros, ao contrário do construtor.

Para definir um construtor de uma classe *Classe*, definimos um método nomeado *Classe*. O destrutor deve ter nome ~*Classe*. Usando sobrecarga, pode-se definir vários construtores para uma mesma classe. O exemplo do conjunto de inteiros passaria a

```
class intset
{
   struct  listElem {
      listElem *next;
      int value;
   };
   listElem* list;
   int nElems;
 public:
   intset() { nElems=0; list=NULL; }
   ~intset(); //retira todos os elementos
   void insert(int n);
   void remove(int n);
   int  member(int n);
   int  size();
};


class IteratorSet
{
   intset::listElem *current;
 public:
   IteratorSet(intset* set) { current = set->list; }
   int theend() { return current==NULL; }
   int nextElem()
   { int n=current->value; current=current->next; return n; }
};

void main()
{
   intset set;  // cria conjunto ja inicializado
   set.insert(10); //insere o elemento 10
   // ...
   IteratorSet it( set ); // cria um iterador
   while (!it.theend())   // percorre todos os elementos
      printf("%d\n", it.nextElem() );  // imprimindo-os

   // quando set sai do escopo, o destrutor é chamado
}
```

Outro exemplo para construtores e destrutores é a modelagem do tipo arquivo:

```
class File
{
   FILE *file;
 public:
   File( char* name ) { file=fopen(name, “r”); }
   ~File()  { fclose(file); file = NULL; }
   char read() { return file?fgetc(file):EOF; }
   int isOpen() { return file!=NULL; }
}

void main(int argc, char *argv[])
{
   if (argc != 2) return 1;

   char c;
   File file(argv[1]);
   if (!file.isOpen()) return 1;
   while ( (c = file.read()) != EOF )
      printf(“%c”, c);
   // destrutor chamado quando file sai do escopo,
   // garante o fechamento do arquivo
}
```

Construtores gerados automaticamente

Neste ponto existe uma pergunta a ser feita: mesmo quando a classe `intset` não tinha construtor, foi possível criá-la e usá-la como se ela tivesse um construtor que não recebia nenhum parâmetro e não fazia nada. Será que na classe `File` ou `IteratorSet`, onde o construtor recebe um parâmetro, também poderíamos criar um objeto sem nenhum parâmetro e usar um construtor vazio? A resposta, felizmente, é não.

Existem dois construtores gerados automaticamente. Um deles é exatamente o construtor vazio que não recebe parâmetros. Este construtor só é gerado caso a classe não defina nenhum construtor.

O outro construtor gerado automaticamente é o construtor de cópia. Este construtor recebe um objeto por referência, e cria um objeto que é uma cópia deste. Este construtor pode ser chamado de duas formas, sempre na criação do objeto: passando-se o objeto a ser copiado como parâmetro do construtor ou inicializando o novo objeto na criação com =.

O exemplo abaixo mostra várias maneiras de se criar um objeto:

```
class Sample
{
   int a;
 public:
   Sample(int v) { a=v;}
};

void main()
{
   Sample obj1;        // ERRO!! não tem construtor vazio
   Sample obj2(6);     // chama construtor declarado
   Sample other(obj2); // cria uma cópia de obj2
                       // (construtor de cópia)
   Sample another = other;    // cria uma cópia de other
                              // (construtor de cópia)
                              // NÃO chama operador =
}
```

Objetos temporários

Quando queremos somar um valor, por exemplo 10, a um inteiro, não é necessário criar uma variável int com o valor 10 para depois somar. Simplesmente fazemos o seguinte:

```
x = x + 10;
```

O que estamos fazendo? É como se tivéssemos criado uma variável int com o valor 10 somente para usar nesta expressão, ou seja, criamos uma variável int temporária.

Em C++ é possível criar objetos temporários de qualquer tipo. Para ficar clara a utilidade deste recurso, vamos supor que a classe *Complexo* definida anteriormente tivesse um construtor

```
Complexo( float re, float im) { i=re; j=im; }
```

Agora temos um objeto *x* do tipo *Complexo* e queremos somar a ele (3 + 6.2j). Isto pode ser feito da seguinte maneira:

```
x = x + Complexo(3, 6.2);
```

Primeiro, será construído um objeto temporário do tipo Complexo usando o construtor (float, float). Depois, será usado o operador + da classe *Complexo* para somar os dois números, e o resultado será atribuído a *x*.

6 - Operadores **new** e **delete**

Até agora, todos os exemplos só usavam objetos criados estaticamente, na pilha. Na realidade, na maioria das vezes os objetos não são criados na pilha, mas são criados dinamicamente. Em C, esta gerência era feita por uma biblioteca (`malloc`, `free` etc.). Em C++ existem operadores dedicados à criação e destruição de objetos:

```
new
delete
```

O operador **new** é similar à função malloc, mas é um operador (assim como **delete**), é uma parte da linguagem; não é necessário incluir nenhum arquivo .h para usá-los. Uma das vantagens de **new** em relação a malloc é o fato de não ser necessária nenhuma conversão de tipo. Também não é necessário calcular o tamanho que a classe ocupa na memória. Tudo isso é feito automaticamente. Se quiséssemos criar um `intset` e um `IteratorSet` dinamicamente, seria simples:

```
intset* set;
set = new intset;
...
IteratorSet *it = new IteratorSet(*set);
...
delete it;
delete set;
```

Vetores de objetos

Em C, quando era necessário alocar dinamicamente vários elementos, era preciso multiplicar o tamanho do tipo pretendido pelo número de cópias:

```
int *i = (int*) malloc( sizeof(int) * 10 );
```

Com o operador **new**, esta criação ficou simplificada:

```
int *i = new int[10];
```

Se for usado um construtor que recebe parâmetros, seria:

```
IteradorConj *it = new IteradorConj [n] (*conj);
```

Para desalocar um vetor de objetos, usamos o operador **delete** da seguinte forma:

```
delete [] i;
```

7 – Herança Simples

Provavelmente herança é o recurso que torna o conceito de classe mais poderoso. Em C++, o termo herança se aplica apenas às classes. Variáveis não podem herdar de outras variáveis e funções não podem herdar de outras funções.

Herança permite que se construa e estenda continuamente classes desenvolvidas por você mesmo ou por outras pessoas, sem nenhum limite. Começando da classe mais primitiva, pode-se derivar classes cada vez mais complexas que não são apenas mais fáceis de depurar, como também são mais simples.

O objetivo de um projeto em C++ é desenvolver classes que resolvam um determinado problema. Estas classes são geralmente construídas incrementalmente através de herança, começando de uma *classe básica* simples. Cada vez que se deriva uma nova classe começando de uma já existente, pode-se herdar algumas ou todas as características da classe pai, adicionando novas quando for necessário. Um projeto completo pode ter centenas de classes, mas normalmente estas classes são derivadas de algumas poucas classes básicas. C++ permite não apenas herança simples, mas também múltipla, permitindo que uma classe incorpore comportamentos de mais de uma classe base.

Reutilização em C++ se dá através do uso de uma classe já existente ou da construção de uma nova classe a partir de uma já existente.

Classes derivadas

A descrição anterior pode ser interessante, mas um exemplo é a melhor forma de mostrar o que é herança e como ela funciona. Aqui está um exemplo de duas classes, a segunda herdando as propriedades da primeira:

```
class Box
{
 public:
   int height, width;
   void Height (int h) { height=h; }
   void Width (int w) { width=w; }
};

class ColorBox : public Box
{
 public:
   int color;
   void Color(int c) { color=c; }
};
```

Usando a terminologia de C++, a classe `Box` é chamada classe base para a classe `ColorBox`, que é chamada classe derivada. Classes base são também chamadas de classes pai. A classe `ColorBox` foi declarada com apenas uma função, mas ela herda duas funções e duas variáveis da classe base. Sendo assim, o seguinte código é possível:

```
void main()
{
   ColorBox cb;
   cb.Color(5);
   cb.Height(3);  // herdada
   cb.Width(50);  // herdada
}
```

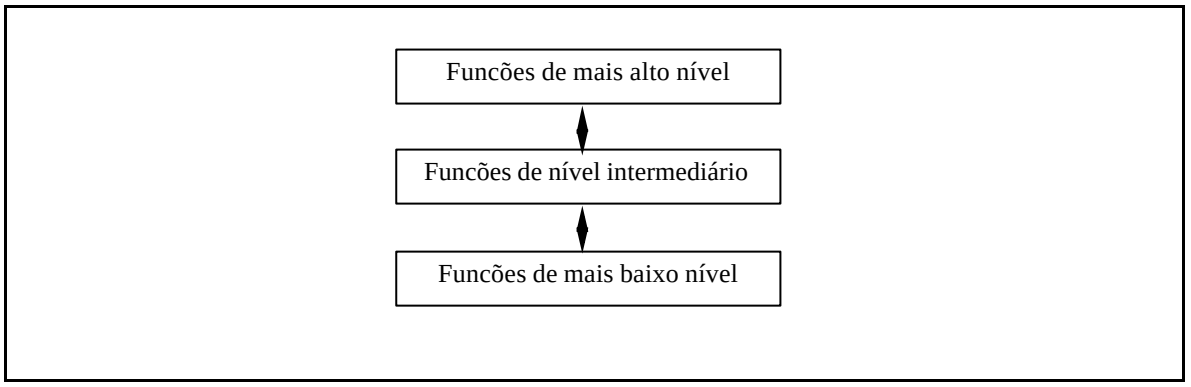
Note que as funções herdadas são usadas exatamente como as não herdadas. A classe Colorida não precisou sequer mencionar o fato de que as funções `Box::Height()` e `Box::Width()` foram herdadas. Esta uniformidade de expressão é um grande recurso de C++. Usar um recurso de uma classe não requer que se saiba se este recurso foi herdado ou não, já que a notação é invariante. Em muitas classes pode existir uma cadeia de classes base derivadas de outras classes base. Uma classe derivada de uma árvore de herança

como esta herdaria características de muitas classes pai diferentes. Entretanto, em C++, não é preciso se preocupar onde ou quando um recurso foi introduzido na árvore.

Derivar uma classe de outra aumenta a flexibilidade a um custo baixo. Uma vez que já existe uma classe base sólida, apenas as mudanças feitas nas classes derivadas precisam ser depuradas. Mas quando exatamente se usa uma classe base, e que tipos de modificações precisam ser feitas? Quando se herda características de uma classe base, a classe derivada pode estender, restringir, modificar, eliminar ou usar qualquer dos recursos sem qualquer modificação.

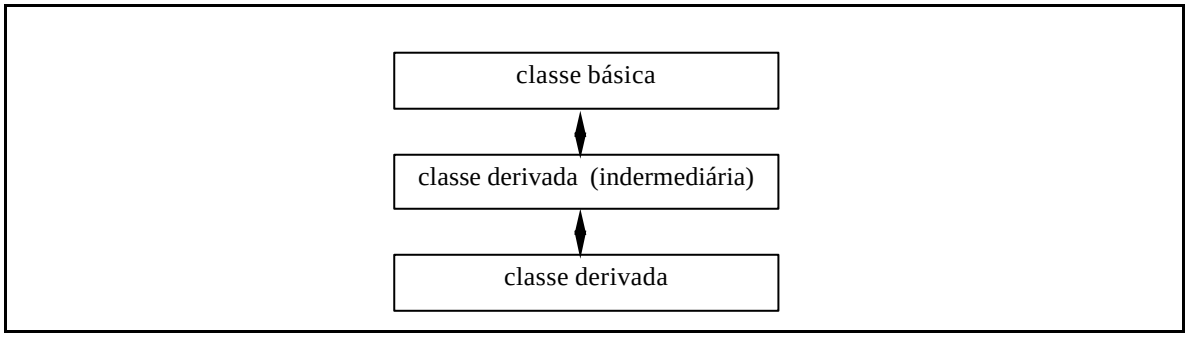
Outra perspectiva para herança

O método *top-down* tem sido usado há anos para desenvolver projetos de software. Usando esta metodologia, um problema é subdividido continuamente em subproblemas menores, até que as subtarefas sejam facilmente gerenciáveis. Esta subdivisão leva essencialmente a uma função de alto nível que usa várias funções de um nível menor. De certa maneira, as funções de mais alto nível incorporam os recursos implementados nas outras funções. Esta noção não é muito diferente de herança. A figura seguinte mostra um programa *top-down*:



Arquitetura básica de um programa desenvolvido usando técnicas estruturadas

A árvore para um programa *top-down* tem grande similaridade com a árvore para a herança de classes, mostrada na figura abaixo:



Arquitetura básica de uma hierarquia de classes

Considerando apenas os grafos, a principal diferença entre os dois é a perspectiva. No mundo da programação estruturada, a árvore é olhada por cima. Funções de um dado nível na árvore invocam funções abaixo dela. As funções de nível mais baixo tem um escopo mais estreito e comportamento mais especializado. Em uma herança de classes, normalmente se tem uma classe herdando características dos pais. Então se olha a árvore de baixo para saber que recursos ela tem.

É claro que existem muitas outras diferenças entre os dois métodos, mas é importante mencionar que os conceitos por trás dos dois não são totalmente distintos.

O que não é herdado

Nem tudo é herdado quando se declara uma classe derivada. Alguns casos são inconsistentes com herança por definição:

- Construtores

- Destrutores
- Operadores **new**
- Operadores de atribuição (=)
- Relacionamentos **friend**
- Atributos **private**

Classes derivadas invocam o construtor da classe base automaticamente, assim que são instanciadas.

Membros protected

Na seção de controle de acesso, vimos como deixar disponíveis ou ocultar atributos das classes, usando os especificadores **public** e **private**. Além desses dois, existe também o especificador **protected**. Do ponto de vista de fora da classe, um atributo **protected** funciona como **private**: não é acessível fora da classe; a diferença está na herança. Enquanto um atributo **private** de uma classe base não é visível na classe derivada, um **protected** é, e continua sendo **protected** na classe derivada. Por exemplo:

```
class A {
 private:
   int a;
 protected:
   int b;
 public:
   int c;
};

class B : public A {
 public:
   int geta() { return a; }  // ERRO!! a não é visível
   int getb() { return b; }  // válido (b protected)
   int getc() { return c; }  // válido (c public)
};

void main()
{
   A ca;
   B cb;

   ca.a = 1; // ERRO! a não é visível (private)
   ca.b = 2; // ERRO! b não é visível de fora (protected)
   ca.c = 3; // válido (c é public)

   cb.a = 4; // ERRO! a não é visível nem internamente em B
   cb.b = 5; // ERRO! b continua protected em B
   cb.c = 6; // válido (c continua public em B)
}
```

Especificadores de acesso para classes base

Nos exemplos acima, em toda declaração de uma classe derivada, usou-se o especificador **public**:

```
class B : public A { ... };
```

Na realidade, os especificadores de acesso **private** e **public** podem ser usados na declaração de uma herança. Por *default*, as heranças são **private**, por isso usou-se **public** nos exemplos acima.

Estes especificadores afetam o nível de acesso que os atributos terão na classe derivada:

- **private**: todos os atributos herdados (**public**, **protected**) tornam-se **private** na classe derivada;
- **public**: todos os atributos **public** são **public** na classe derivada, e todos os **protected** também continuam **protected**.

Na realidade, isto é uma conseqüência da finalidade real de heranças **public** e **private**, que voltará a ser discutida quando estivermos tratando de compatibilidade de tipos.

Construtores e destrutores

Quando uma classe é instanciada, seu construtor é chamado. Se a classe foi derivada de alguma outra, o construtor da classe base também precisa ser chamado. A ordem de chamada dos construtores é fixa em C++. Primeiro a classe base é construída, para depois a derivada ser construída. Se a classe base também deriva de alguma outra, o processo se repete recursivamente até que uma classe não derivada é alcançada.

Desta forma, quando um construtor para uma classe derivada é chamado, todos os procedimentos efetuados pelo construtor da classe base já foram realizados. Considere a seguinte árvore de herança:

```
class Primeira {};
class Segunda: public Primeira {};
class Terceira: public Segunda {};
```

Quando a classe `Terceira` é instanciada, os construtores são chamados da seguinte maneira:

```
Primeira::Primeira();
Segunda::Segunda();
Terceira::Terceira();
```

Esta ordem faz sentido, já que uma classe derivada é uma especialização de uma classe mais genérica. Isto significa que o construtor de uma classe derivada pode usar atributos herdados.

Os destrutores são chamados na ordem inversa dos construtores. Primeiro, os atributos mais especializados são destruídos, depois os mais gerais. Então a ordem de chamada dos destrutores quando `Terceira` sai do escopo é:

```
Terceira::~Terceira();
Segunda::~Segunda();
Primeira::~Primeira();
```

Passando argumentos para uma classe base

Como os construtores das classes base são chamados automaticamente, deve existir alguma maneira de passar os argumentos corretos para estes construtores, no caso de eles necessitarem de parâmetros. Existe uma notação especial para este caso, ilustrada abaixo, para funções **inline** e não **inline**:

```
class Primeira {
   int a, b, c;
 public:
   Primeira(int x, int y, int z) { a=x; b=y; c=z; }
};

class Segunda : public Primeira {
   int valor;
 public:
   Segunda(int d) : Primeira(d, d+1, d+2) { valor = d; }
   Segunda(int d, int e);
};

Segunda::Segunda(int d, int e) : Primeira(d, e, 13)
{
   valor = d + e;
}
```

A partir do exemplo acima, não é difícil perceber que, se uma classe base não possui um construtor sem parâmetros, a classe derivada tem que, obrigatoriamente, declarar um construtor, mesmo que este construtor seja vazio:

```
class Base {
 protected:
   int valor;
 public:
   Base(int a) { valor = a; } // esta classe não possui um construtor
                              // sem parâmetros
};

class DerivadaErrada : public Base{
 public:
   int pegaValor() { return valor; }
// ERRO! classe não declarou construtor, compilador não
// sabe que parâmetro passar para Base
};

class DerivadaCerta: public Base {
 public:
   int pegaValor() { return valor; }
   DerivadaCerta() : Base(0) {}
// CERTO: mesmo que não haja nada a fazer para inicializar a classe,
// é necessário declarar um construtor para dizer com que parâmetro
// construir a classe Base
};
```

8 – Herança e Polimorfismo

A origem da palavra polimorfismo vem do grego: poli (muitos) e morphos (forma) - múltiplas formas. Polimorfismo descreve a capacidade de um código C++ se comportar de diferentes formas dependendo do contexto em tempo de execução. Mais precisamente, é o processo onde diferentes implementações de uma função podem ser acessadas usando o mesmo nome.

Conversão de tipos

Em C++, normalmente não se usam objetos de classes isoladas, mas sim objetos em uma hierarquia de classes. Para isto, é necessário que certas conversões sejam feitas entre objetos de classes em níveis diferentes da hierarquia. Considere as seguintes classes:

```
class A {};
class B: public A {};
```

Como B é derivado de A, todos os membros disponíveis em A também estarão disponíveis em B. Então B é um superconjunto de A. Nesse caso, é possível converter um objeto do tipo B em um do tipo A, mas não no sentido inverso. O código abaixo é um exemplo:

```
void main()
{
   A a;
   B b;

   A* ap = new A;
   B* bp = new B;

   a = b;   // permitido, a é superclasse de b
   ap = bp; // mesmo motivo

   b = a;   // ERRO! a não é subclasse de b
```

```
   bp = ap; // ERRO! mesma coisa
}
```

As conversões podem ocorrer implicitamente durante chamadas de funções, como mostrado abaixo:

```
class Base {
   int i;
 public:
   int getInt() { return i; }
   void setInt(int v) { i = v; }
};

class Primeira: public Base {
 public:
   float f;
};

class Segunda: public Base {
 public:
   char c;
};

void showInt( Base& b )
{
   printf("%d\n", b.getInt());
}

void main()
{
   Base  base;
   Primeira *p = new Primeira;
   Segunda   s;

   base.setInt(0);
   p->setInt(1);
   s.setInt(2);
   showInt( base ); // valido, base é do tipo Base
   showInt( *p );   // tambem valido, p é subclasse de Base
   showInt( s );    // idem
}
```

Redefinindo métodos em uma hierarquia

Um método pode ser redefinido em uma classe derivada. Por exemplo:

```
class A {
 public:
   int f(int i) { return i+1; }
};

class B: public A {
 public:
   int f(int i) { return i+10; }
};

void main()
{
```

```
   A a;
   B b;

   printf("%d\n", a.f(10) );
   printf("%d\n", b.f(10) );
}
```

No caso acima, a função `f` foi redefinida na classe `B`. Para redefinir um método, é preciso que os parâmetros sejam iguais.

Não existe sobrecarga na hierarquia de classes, apenas em uma mesma classe. Ou seja, se a função `B::f` no exemplo acima recebesse como parâmetro um `float`, esta nova função esconderia a função `A::f`, mesmo que `A::f` receba um `int`.

Early x late binding

Em C++, uma chamada de método pode estar apenas indicada no código, sem especificar exatamente que função será chamada. Isto é conhecido como *late binding*. Em linguagens de programação mais tradicionais, como C e Pascal, o compilador chama identificadores de funções fixos baseados no código fonte. O linkeditor substitui estes identificadores com o endereço físico da função. Este processo é chamado de *early binding*, porque os identificadores de funções são associados com endereços físicos antes da execução do programa.

O problema de *early binding* é que o programador precisa saber que objetos serão usados em todas as chamadas de função em todas as situações. Isto é uma limitação. A vantagem de *early binding* é a eficiência; os únicos passos para a chamada de uma função são a passagem de parâmetros, passar o controle para a função e limpar a pilha.

O problema de *late binding* é exatamente a eficiência. O código deve descobrir que função chamar durante a execução do programa. Algumas linguagens, como Smalltalk, usam exclusivamente *late binding*. O resultado é uma linguagem extremamente poderosa, mas com algumas penalidades em relação ao tempo de execução. Por outro lado, ANSI C usa somente *early binding*, resultando em alta velocidade mas falta de flexibilidade.

C++ não é uma linguagem procedural tradicional como Pascal, mas também não é uma linguagem orientada a objetos pura. C++ é uma linguagem híbrida. C++ usa *early* e *late binding* procurando oferecer o melhor de cada um dos métodos. O programador controla quando usar um ou outro. Para um código em que a execução é determinística, pode-se usar *early binding*. Em situações mais complexas, usa-se *late binding*. Desta forma, pode-se conciliar alta velocidade com flexibilidade.

Métodos virtuais

Em C++, *late binding* é especificado declarando-se um método como *virtual*. *Late binding* só faz sentido para objetos que fazem parte de uma hierarquia de classes. Declarar uma função como virtual em uma classe que não é usada como classe base é sintaticamente correto, mas desnecessário.

A melhor maneira de mostrar o que acontece quando uma função é declarada virtual é através de um exemplo. Suponha que, no nosso projeto, todas as classes tem um método que retorna o nome dela:

```
class A {
 public:
   char* getClassName() { return "A"; }
};

class B {
 public:
   char* getClassName() { return "B"; }
};
void showClassName(A& a) {  printf("%s\n", a.getClassName());  }

void main()
{
   A a;
```

```
   B b;
   showClassName(a);
   showClassName(b);
}
```

Na primeira chamada de `showClassName`, que recebe um `A&`, o objeto passado é do tipo `A`, então não há dúvida: o método chamado é `A::getClassName`. Já na segunda chamada, o objeto passado é do tipo `B`, mas é convertido para o tipo `A` na chamada da função, e dentro dela, o objeto é visto como se fosse do tipo `A`. O que acontece é que o método chamado será novamente `A::getClassName`. Não era esse o comportamento desejado. Parece que a função `showClassName` não pode ser usada nos dois casos. Será que é necessário escrever uma função para cada classe? A resposta é não.

Na realidade, o problema não está na função `showClassName`, e sim na declaração das classes. O que aconteceu foi que o compilador amarrou a chamada `getClassName` com o método de `A`. Ou seja, *early binding*. O que faltou foi dizer ao compilador para não amarrar a chamada em tempo de compilação, mas descobrir, em tempo de execução, qual o tipo real do objeto e chamar o método certo. Ou seja, *late binding*.

Se o método `A::getClassName` for declarado virtual, sempre será usado *late binding* na chamada desta função. Modificando a declaração da classe `A`:

```
class A {
 public:
   virtual char* getClassName() { return "A"; }
};
```

Nesse caso, as chamadas de `showClassName` mostrarão os nomes corretos das classes.

Destrutores virtuais

Considere o seguinte código:

```
class VetorPonteiros {
 protected:
   int  numElems;
   int **elems;
 public:
   VetorPonteiros(int e) { numElems = e; elems = new int*[e]; }
   ~VetorPonteiros()     { delete [] elems; }
};

class Matriz: public VetorPonteiros {
   int cols;
 public:
   Matriz(int m, int n) : VetorPonteiros(m)
   {
      int i;
      cols = n;
      for (i=0; i<numElems; i++) elems[i] = new int[cols];
   }
   ~Matriz()
   {
      int i;
      for (i=0; i<numElems; i++) delete [] elems[i];
   }
};

void apaga( VetorPonteiros* vp )  {  delete vp;  }
```

```
void main()
{
   Matriz* m = new Matriz(10,20);
   apaga(m);
}
```

A função `apaga` destrói o objeto, o que causa a chamada do destrutor. Mas para o compilador, o destrutor é uma função como outra qualquer, e nesse caso, na hora em que o objeto é destruído, ele é visto como um objeto do tipo `VetorPonteiros`. Como o destrutor de `VetorPonteiros` não é virtual, o destrutor `Matriz::~Matriz` não é chamado, deixando vários ponteiros perdidos no programa.

Este caso é o mesmo da seção anterior, e pode ser resolvido declarando o método `VetorPonteiros:~VetorPonteiros` como virtual:

```
class VetorPonteiros {
 protected:
   int  numElems;
   int **elems;
 public:
   VetorPonteiros(int e) { numElems = e; elems = new int*[e]; }
   virtual ~VetorPonteiros() { delete [] elems; }
};
```

9 – Classes abstratas

Nós pderíamos pensar em varias implementações para uma classe *Pilha*. Por exemplo: uma que fosse implementada com listas encadeadas, e outra com um vetor. Portanto, a implementação não faz parte do tipo, e, ao declarar o tipo no arquivo cabeçalho (.h), não faz diferença nenhuma que campos representarão a pilha, apenas sua interface.

Se declarássemos um *Pilha* sem os campos de representação da lista encadeada, os métodos também não poderiam ser definidos, já que eles se baseiam na representação interna.

Métodos virtuais nulos

Para resolver o problema apresentado acima foi introduzido em C++ o mecanismo de métodos virtuais nulos. Esses métodos são definidos sem um corpo, e indicam que a classe tem aquele método mas a implementação fica para as classes derivadas:

```
class Pilha {
 public:
   virtual void push(int) = 0;
   virtual int  pop() = 0;
   virtual int  isEmpty() = 0;
};
```

Dada esta definição, pode-se escrever uma aplicação que utiliza uma pilha. A única restrição é a criação de objetos deste tipo. Como `Pilha` tem métodos virtuais nulos, ela não pode ser instanciada. Ou seja, não é possível criar objetos desta classe.

Para que pudéssemos usá-la, seria preciso então definir uma nova classe (por exemplo `Pilha_LE`) que derivasse de `Pilha`, implementando os métodos com listas encadeadas. Ou então `Pilha_V`, que usasse um vetor para representar a pilha.

Uma alternativa para que pudéssemos também criar objetos `Pilha`, seria colocar um protótipo de uma função:

```
Pilha *newPilha();
```

que retornasse um objeto derivado de `Pilha`. Um arquivo `.h` que contenha estas duas declarações permite o uso completo de pilhas, sem saber absolutamente nada da implementação. Na hora de linkar o programa, podemos escolher entre vários módulos com várias implementações diferentes.

A classe que faria parte de um módulo `pilha_le.c` poderia ser:

```
#include "pilha.h"

class Pilha_LE : public Pilha {
   struct  elemPilha {
      elemPilha* prox;
      int val;
      elemPilha(elemPilha*p, int v) { prox=p; val=v; }
   };
   elemPilha* topo;
 public:
   int  vazia()     { return topo == NULL }
   void push(int v) { topo = new elemPilha(topo, v); }
   int  pop();
};

int Pilha_LE::pop()
{
   if (topo)
   {
      elemPilha *ep = topo;
      int v = ep->val;
      topo  = ep->prox;
      delete  ep;
      return v;
   }
   return -1;
}

Pilha *newPilha() { return new Pilha_LE; }
```

Compatibilidade de tipos

Vamos supor que existe uma classe `ListaEncadeada` e desejamos implementar uma classe `Fila`, reutilizando a implementação de `ListaEncadeada`. `Fila` deve herdar **public** ou **private** de `ListaEncadeada`?

O que deve ser levado em consideração é se `Fila` é um tipo compatível com `ListaEncadeada` ou não.

`Fila` será um tipo compatível com `ListaEncadeada` se pudermos utilizar um objeto do tipo `Fila` como uma `ListaEncadeada` em qualquer situação. Ou seja, se todas as operações[ii] de `ListaEncadeada` podem ser aplicadas sobre um objeto `Fila`.

Neste exemplo, o caso é somente de reutilização, já que não podemos inserir um objeto no meio de uma `Fila`, o que é permitido em `ListaEncadeada`.

Em C++, heranças podem ser **public** ou **private**. No caso da herança **private**, os métodos da classe base não ficam disponíveis, e, mais do que isso, não são permitidas conversões de um objeto da classe derivada

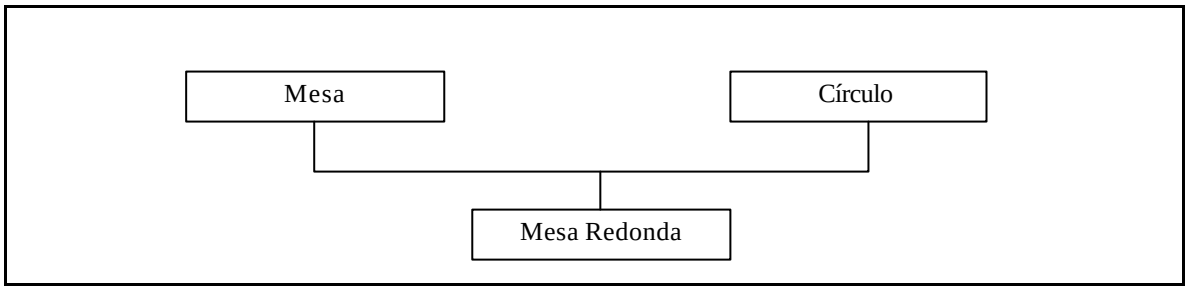
[ii]Importante: operação não está amarrada à implementação. Compatibilidade de tipos não quer dizer que todas as implementações da classe base valem para a classe derivada; a classe derivada pode e deve fornecer novas implementações quando for o caso.

para um da classe base. Isto impede que um objeto `Fila` seja convertido para uma `ListaEncadeada`, o que garante que as operações da classe base não serão usadas sobre `Fila`.

Herança **private** existe exatamente para estes casos exclusivos de reutilização. Quem usa a classe não "sabe" que o objeto herdou de alguma classe base se a herança for **private**.

10 – Herança Múltipla

Em C++, a herança não se limita a uma única classe base. Uma classe pode ter vários pais, herdando características de todos eles. Este tipo de herança introduz grande dose de complexidade na linguagem e no compilador, mas os benefícios são substanciais. Considere a criação de uma classe `MesaRedonda`, tendo não só propriedades de mesas, mas também as características geométricas de ser redonda. A figura abaixo mostra os relacionamentos:



Criando uma mesa redonda através de herança múltipla

O código abaixo é uma possível implementação:

```
class Circulo {
   float raio;
 public:
   Circulo(float r) { raio = r; }
   float area()     { return raio*raio*3.14159; }
};

class Mesa {
   float ipeso;
   float ialtura;
 public:
   Mesa(float p, float a)  { ipeso = p; ialtura=a; }
   float peso()   { return ipeso; }
   float altura() { return ialtura; }
};

class MesaRedonda: public Circulo, public Mesa {
   int icor;
 public:
   MesaRedonda(int c, float a, float p, float r);
   int cor() { return icor; }
};

MesaRedonda::MesaRedonda(int c, float a, float p, float r)
 : Mesa(p, a), Circulo(r)
{
   icor = c;
}
```

```
void main()
{
   MesaRedonda mesa(5, 1, 20, 3.5 );

   printf("Peso:   %f\n", mesa.peso());
   printf("Altura: %f\n", mesa.altura());
   printf("Area:   %f\n", mesa.area());
   printf("Cor:    %d\n", mesa.cor());
};
```

Ordem de chamada dos construtores e destrutores

Assim como em herança simples, os construtores das classes base são chamados antes do construtor da classe derivada. A ordem de declaração na classe define a ordem de chamada dos construtores. No exemplo acima, declaramos a classe assim:

```
class MesaRedonda: public Circulo, public Mesa {
```

e definimos o construtor da seguinte maneira:

```
MesaRedonda::MesaRedonda(int c, float a, float p, float r)
 : Mesa(p, a), Circulo(r)
```

Como a ordem de declaração na classe é a que define a ordem dos construtores, então teremos as seguintes chamadas:

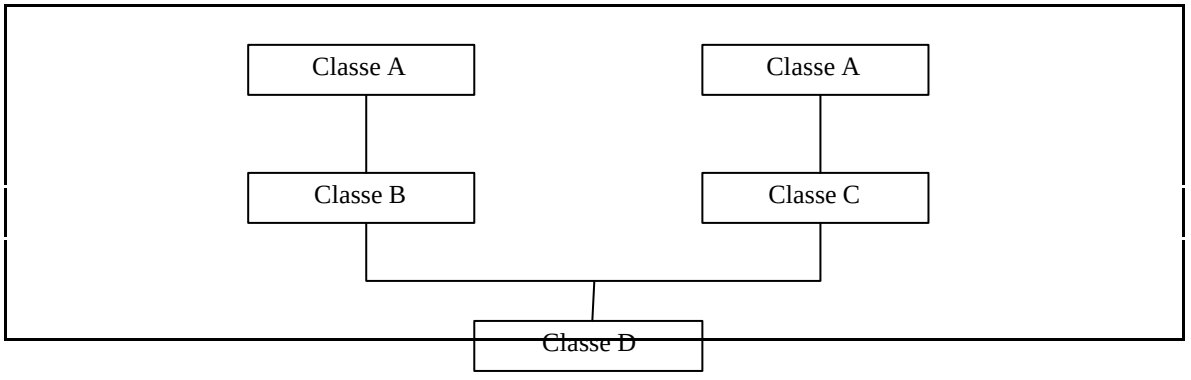
```
Circulo::Circulo
Mesa::Mesa
MesaRedonda::MesaRedonda
```

Classes base virtuais

Classes base virtuais só são usadas com herança múltipla. É uma maneira de o programador controlar como as classes devem der herdadas. Por exemplo:

```
class A {
 public:
   int a;
};
class B: public A {};
class C: public A{};
class D: public B, public C {
 public:
   int valor() { return a; }
};
```

Gera a seguinte hierarquia:



Herança com uma mesma classe aparecendo duas vezes

Portanto, o código acima gera um erro de compilação:

```
"Field 'a' is ambiguous in 'D' in function D::valor()"
```

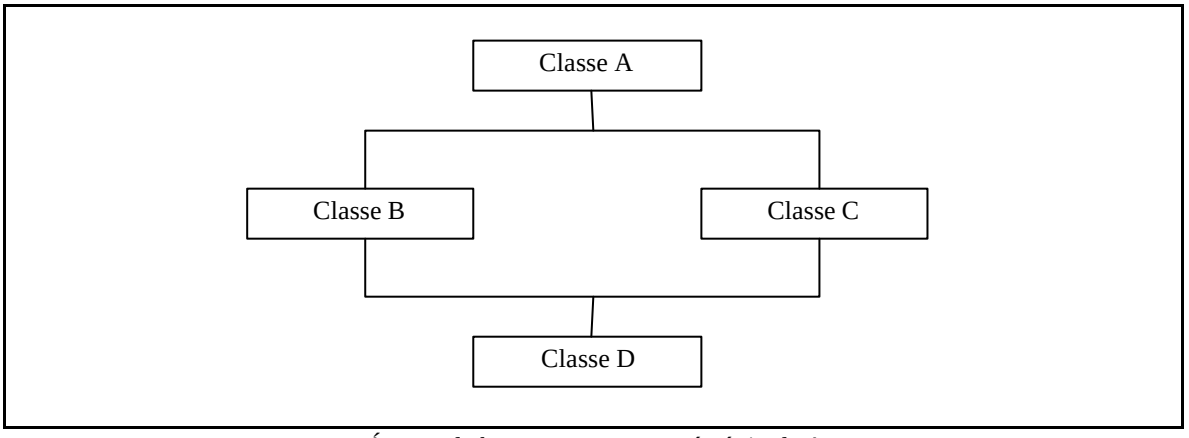
O compilador não sabe que cópia de *a* esta sendo referenciada. O operador de escopo poderia ser utilizado para retirar o erro:

```
int valor() { return C::a; }
```

Às vezes o programador quer montar uma árvore onde só exista uma cópia de A. É o caso de usar uma classe base virtual. Declarando uma classe base como virtual força o compilador a só permitir uma cópia de uma dada classe base na declaração da classe derivada. A classe *D* poderia ser redefinida da seguinte maneira:

```
class D: public virtual B, public virtual C {
 public:
   int valor() { return a; }
};
```

Agora a função valor não precisa mais do operador de escopo, e a árvore gerada será:



Árvore de herança com uma só cópia de A

11 – Templates

Templates de classes

Uma classe C++ normalmente é projetada para armazenar algum tipo de dado. Muitas vezes a funcionalidade de uma classe também faz sentido com outro tipo de dado. É o caso de muitos exemplos apresentados, por exemplo, *Pilha*.

Se uma classe é vista simplesmente como um manipulador de dados, pode fazer sentido separar a definição desta classe da definição dos tipos manipulados. Nesse caso a classe trabalharia com um tipo genérico T. Com isto não criaríamos uma classe real, mas uma descrição de classe, chamada *class template*. Uma *class template* é utilizada pelo compilador para criar uma classe real em tempo de compilação, usando um tipo de dado específico.

Consideremos novamente a classe `Pilha`, que só armazenava inteiros apesar de sua funcionalidade nada ter a ver com o tipo de dado envolvido. Seria o caso de definir uma *template* para a classe `Pilha`, onde o tipo armazenado é um tipo `T` qualquer:

```
template<class T> class Pilha {
   struct  elemPilha {
      elemPilha* prox;
      T val;
      elemPilha(elemPilha*p, T v) { prox=p; val=v; }
   };
   elemPilha* topo;
 public:
   int  vazia()   { return topo == NULL }
   void push(T v) { topo = new elemPilha(topo, v); }
   T pop();
};
template<class T> T Pilha<T>::pop()
{
   if (topo)
   {
      elemPilha *ep = topo;
      T v = ep->val;
      topo  = ep->prox;
      delete  ep;
      return v;
   }
   return -1;
}
```

Para criar um objeto desta classe, é preciso dizer o tipo na declaração do objeto:

```
void main()
{
   Pilha<int> intPilha;
   Pilha<char*> stringPilha;
   Pilha<Pilha<int>> intPilhaPilha;

   intPilha.push(10);
   charPilha.push( “teste” );

   intPilhaPilha.push( intPilha );
}
```

Tendo em vista que uma *template* é simplesmente uma descrição de uma classe, é necessário que toda esta descrição tenha sido lida antes de alguma declaração que envolva esta *template*. Isto significa, em se tratando de templates, que é necessário colocar no arquivo .h não apenas a declaração de classe, mas também a implementação dos métodos desta classe.

Outra conseqüência de templates serem apenas descrições é que erros semânticos só aparecem na hora de usar a *template*. Durante a declaração da *template* apenas erros de sintaxe são checados. Mesmo que a *template* em si tenha sido compilada sem erros, na hora de usar podem aparecer erros.

Este teste semântico é realizado todas as vezes que a *template* é instanciada para algum tipo novo. Isto porque na definição do código da *template* não há nenhuma restrição quanto às operações que podem ser aplicadas ao tipo T. O teste então tem que ser feito para cada tipo.

Templates com vários argumentos genéricos

Templates não estão limitadas a terem apenas um argumento. A *Pilha* poderia armazenar dois tipos de dados:

```
template<class T, class R> class Pilha {
   struct  elemPilha {
```

```
      elemPilha* prox;
      T t_val;
      R r_val;
      elemPilha(elemPilha*p, T t, R r)
         { prox=p; t_val=t; r_val=r; }
   };
   elemPilha* topo;
 public:
   int  vazia()     { return topo == NULL }
   void push(T t, R r) { topo = new elemPilha(topo, t, r); }
   void pop(T& t, R& r);
};
```

Templates com argumentos não genéricos

Templates não estão limitadas a argumentos genéricos. Por exemplo, uma pilha que armazene um número fixo de elementos de um mesmo tipo:

```
template<class T, int S> class Pilha {
   struct  elemPilha {
      elemPilha* prox;
      T t_val[S];
      elemPilha(elemPilha*p, T* t);  };
   elemPilha* topo;
 public:
   int  vazia();
   void push(T* t);
   void pop(T* t);
};
```

Templates de funções

Templates também podem ser usadas para definir funções. A mesma motivação para classes vale neste caso. Algumas vezes funções realizam operações sobre dados sem utilizar o conteúdo deles, ou seja, independentemente de que tipo de dado seja.

Suponha que precisamos testar a magnitude de dois elementos quaisquer. A seguinte macro resolve este problema:

```
#define max(a,b) ((x>y) ? x : y)
```

Por muito tempo macros como esta foram usadas em programas C, mas isto tem os seus problemas. A macro funciona, mas impede que o compilador teste os tipos dos elementos envolvidos. A macro poderia ser utilizada para comparar um inteiro com um ponteiro, sem que sejam gerados erros.

Poderíamos usar uma função como esta:

```
int max(int a, int b)
{
   return a > b ? a : b;
}
```

Mas esta função só funciona para inteiros. Se o nosso programa só trabalha com escalares, poderíamos escrever uma função que trabalhe com **double**:

```
double max(double a, double b)
{
   return a > b ? a : b;
}
```

Nesse caso, o compilador se encarrega de converter os tipos **char, int** etc. para **double**, e a função funcionaria para todos estes casos.

Existem pelo menos duas limitações nesta versão:

- apenas tipos que podem ser convertidos para **double** podem usar esta função. Isto significa que objetos e ponteiros não podem ser utilizados.
- o tipo de retorno é sempre **double**, independente do tipo passado. Suponha que a função `display` seja sobrecarregada para imprimir vários tipos de dados. Agora considere o código:

```
display(max('1', '9'));
```

Apesar de estarmos trabalhando com caracteres, a função `display` a ser chamada será a versão que trabalha com **double**, e o resultado será 57.00, que é o código ASCII do caractere '9'.

Em C++, pode-se definir uma *template* para esta função, da seguinte maneira:

```
template<class T> T max( T a, T b )
{
   return a > b ? a : b;
}
```

Esta função pode ser chamada normalmente:

```
void main()
{
   printf("%c", max('1', '9'));
}
```

12 – Tratamento de Exceções

Funcionamento básico

No desenvolvimento de bibliotecas, é possível escrever código capaz de detectar erros de execução mas, em geral, não é possível fazer seu tratamento. Por outro lado, o usuário de uma biblioteca é capaz de fazer o correto tratamento de uma exceção mas não é capaz de detectá-la.

O conceito de exceção é introduzido para ajudar neste tipo de problema. A idéia fundamental é que uma função que detecte um problema e não seja capaz de resolvê-lo "acuse a exceção" esperando que quem a chamou seja capaz de realizar o tratamento adequado.

Por exemplo, considere como representar e tratar o erro de indexação fora dos limites de um array dada pela classe *Vector*:

```
class Vector {
  int* p;
  int sz;
public:
  class Range{ }; //classe de tratamento de exceção
  int& operator[]( int i );
  // outras funções .....
}
```

A acusação de uma exceção é feita pelo comando `throw`. No nosso exemplo, os objetos da classe `Range` são utilizados como exceções e são acusados da seguinte forma:

```
int& Vector::operator[]( int i )
{
  if ( i>=0 && i<sz ) return p[i];
```

```
  throw Range();
  // Quando indexamos o vetor com limites inválidos é
  // acusada a exceção correspondente
  // se a função que chamou operator[] souber tratá-la,
  // teremos o tratamento adequado.
}
```

A função que precisa detectar a utilização de índices fora do limite indica seu interesse pelo tratamento colocando código correspondente na seguinte forma:

```
void f( Vector& v )
{
  //.. código qualquer sem tratamento
  try{
    //.. código qualquer com tratamento
    operação_qualquer( v );
  }
  catch( Vector::Range ) {
    // Aqui se encerra o código de tratamento da exceção
    // Range.
    // a função operação_qualquer apresentou a exceção
    // que está sendo tratada.
    // Esse código somente será executado se e somente se
    //operação_qualquer fizer uso de indexação inválida.
  }
  //.. código qualquer sem tratamento
}
```

A construção

```
catch( /* nome da exceção */ ) { /* código */ }
```

é denominada manipulador de exceção (*exception handler*). Esta construção somente pode ser utilizada depois de um bloco prefixado com a palavra reservada `try` ou após outra construção `catch`. Os parênteses encerram a declaração dos tipos de objetos aonde o manipulador pode executar. Se a função `operação_qualquer()` ou qualquer outra função chamada por `operação_qualquer()` causar uma indexação inválida no array, será gerada uma exceção que será pega pelo manipulador e seu código executado.

Se um manipulador pegou uma determinada exceção, esta foi devidamente tratada e qualquer outro manipulador ainda existente se torna irrelevante. Em outras palavras, apenas o manipulador mais recentemente encontrado pelo controle da linguagem será executado. Por exemplo, dado que a função `f()` pega uma exceção `Vector::Range`, uma função que chame `f()` jamais pegará a exceção `Vector::Range`.

```
int ff( Vector& v )
{
  try{
    f(v);
  }
  catch( Vector::Range )
  { // este código jamais será executado ...
  }
}
```

Naturalmente, um programa é capaz de tratar diversas exceções. Esses erros são mapeados com nomes distintos. Continuando o exemplo de *Vector*, trataremos mais um caso: criação de array com tamanho fora dos limites. Temos:

```
class Vector {
  int* p;
  int sz;
  int max 512; // número máximo de elementos
 public:
  class Range{ }; //classe de tratamento de exceção
  class Size{ }; //classe de tratamento de exceção

  int& operator[]( int i );
  Vector( int sz )
  {
     if ( sz < 0 || max < sz ) throw Size();
     // continuação do construtor...
  }
}
```

O usuário da classe *Vector* pode discriminar a exceção pondo diversos manipuladores dentro do bloco precedido por `try`:

```
void f( Vector& v )
{
  try {
    qualquer_operação(v);
  }
  catch( Vector::Range ) {
  // código de tratamento de indexação inválida
  }
  catch( Vector::Size ) {
  // código de tratamento para criação de vetor muito grande
  }
// esse código é executado se não tiver ocorrido nenhuma
// exceção.
}
```

Nomeação de exceções

Uma exceção é tomada pelo manipulador não pelo seu tipo mas sim por um objeto. Havendo necessidade de transmitir alguma informação do levantamento da exceção para o manipulador, é necessário haver algum mecanismo de colocar tal informação neste objeto. Isto é feito colocando-se campos dentro da classe que representa a exceção e tomando seus valores nos manipuladores. Para tomarmos estes valores nos *handles*, é necessária a definição de um nome para o objeto criado na acusação.

No exemplo criado, é importante saber qual o valor que foi usado como índice na exceção `Vector::Range` (indexação fora dos limites):

```
class Vector{
// ...
public:
  class Range {
   public:
     int index;
     // criação do campo que diz o valor inválido
     Range( int i ) { index = i; }
     // a criação do objeto indica o valor inválido
  };
int& operator[]( int i )
// ...
```

```
};

int& Vector::operator[]( int i )
{
  if ( 0<=i && i<sz ) return p[i];
  throw Range(i);
  // acusa-se a exceção indicando
  // o valor índice invalido ao construir Range
}
```

Para examinar o índice incorreto, o manipulador deve dar um nome ao objeto da exceção:

```
void f( Vector& v )
{ /...
  try {
    qualquer_operação(v);
  }
  catch( Vector::Range r ) {
  // ‘r’ é o nome do objeto Range acusado no operador []
    printf( “índice errado: %d \n”, r.index );
    exit(0);
  }
```

É interessante notar que no caso de templates, tem-se a opção de nomear a exceção de modo que cada classe instanciada pela `template` tenha sua própria classe de exceção:

```
template<class T> class Allocator {
  // ...
  class Exhausted {};  // classe do tratamento de exceção
  // ...
};

void f (Allocator<int>& ai, Allocator<double>& ad )
{
  try {
    // ...
  }
  catch (Allocator<int>::Exhausted) {
  // ...
  // tratamento de inteiros
  }
  catch (Allocator<double>::Exhausted) {
  // ...
  // tratamento de doubles
  }
}
```

Alternativamente, uma exceção pode ser comum a todas as classes instanciadas pela `template`:

```
class Allocator_Exhausted {};
template<class T> class Allocator {
  // ...
};

void f( Allocator<int>& ai, Allocator<double>& ad )
{
  try {
```

```
    // ...
  }
  catch( Allocator_Exausted ) {
  // ...
  // tratamento para ambos os tipos
  }
}
```

Agrupamento de exceções

Normalmente as exceções podem ser categorizadas em famílias. Por exemplo, pode-se imaginar um erro matemático que inclua as exceções de *overflow*, *underflow*, divisão por zero, etc. A exceção de erro matemático (MATHERR) pode ser determinada pelo conjunto de erros que podem ser produzidos em uma biblioteca de funções numéricas.

Uma maneira de fazer MATHERR é determiná-la como um tipo de todos os possíveis erros numéricos:

```
enum MATHERR { Overflow, Underflow, ZeroDivide };
```

e na função que trata os erros:

```
void f( .... )
{
  try {
    // ...
  }
  catch( MATHERR m ) {
    switch( m ) {
    case Overflow:
      // ...
    case Underflow:
      // ...
    case ZeroDivide:
      // ...
    }
  }
}
```

De outra maneira, C++ usa a capacidade de herança e de funções virtuais para evitar este tipo de switch(). É possível a utilização de herança para descrever coleções de exceções. Por exemplo:

```
class MATHERR {};
class Overflow: public MATHERR {} ;
class Underflow: public MATHERR {} ;
class ZeroDivide: public MATHERR {} ;
// ....
```

Para este caso, existem muitas ocasiões em que deseja-se fazer o tratamento de MATHERR sem saber precisamente de que tipo é o erro. Com a utilização de herança, é possível dizer:

```
try {
  // ...
}
catch( Overflow ) {
//  tratamento de overflow ou tudo derivado de overflow
}
catch ( MATHERR ) {
//  tratamento de qualquer outro erro numérico
}
```

A organização de exceções em hierarquias pode ser importante para a robustez do código de um programa. Consideremos o tratamento de todas as exceções de nossa biblioteca numérica sem o agrupamento destas. Neste caso, as funções que utilizam esta biblioteca teriam que exaustivamente determinar e tratar toda a lista de erros.

```
try {
// ...
}
catch (Overflow) { /* ..... */ }
catch (Underflow) { /* ..... */ }
catch (ZeroDivide) { /* ..... */ }
// e todas as outras exceções!!!
```

Isto não somente é tedioso mas da margem ao esquecimento de alguma exceção. Além, uma determinada função que desejar fazer o tratamento de qualquer erro numérico (sem saber que tipo de erro) precisa ser constantemente atualizada quando do aparecimento de novas exceções. Por exemplo: o logaritmo de número menor ou igual a zero; o que implica também em recompilação das funções clientes.

Neste sentido é muito mais prático fazer:

```
try {
// ...
}
catch (MATHERR) { /* ..... */ }
// trata qualquer erro matemático.
```

que garante sempre o tratamento sem a necessidade de manutenção do código quando da introdução de novas exceções.

Exceções derivadas

A utilização de hierarquias de exceções naturalmente direciona os manipuladores que estão interessados somente em um subconjunto da informação carregada pelas exceções. Em outras palavras, uma exceção é normalmente tomada por um manipulador da classe básica ao invés de um da classe exata. A semântica para a tomada de um manipulador e nomeação de uma exceção é idêntica para a passagem de argumentos de funções vista anteriormente. Por exemplo:

```
class MATHERR {
  // ...
  virtual void debug_print() {};
};

class int_overflow : public MATHERR {
public:
  char op;
  int opr1, opr2;
  int_overflow( const char p, int a, int b )
  { op=p; opr1=a; opr2=b; }
  virtual void debug_print()
  { printf(“ operador:%c:( %d, %d )”, op, opr1, opr2 ); }
};

void f()
{
   try{
    g();
  }
```

```
  catch( MATHERR m ) { /* ... */ }
}
```

Quando um manipulador `MATHERR` é encontrado, *m* é um objeto `MATHERR` mesmo que a chamada de `g()` tenha acusado um `int_overflow`. Isto implica que a informação extra encontrada em `int_overflow` está inacessível.

No entanto, ponteiros e referências podem ser utilizados para evitar esta perda de informação. Para tal, pode-se escrever:

```
int add( int x, int y )
{
  if ( x>0 && y>0 && x>MAXINT - y
    || x<0 && y<0 && x<MININT + y )
    throw int_overflow( '+', x, y );
  return x + y;
}

void f()
{
  try {
    add( 1, 2 ); // ok
    add( MAXINT, 3 ); // causa exceção
  }
  catch( MATHERR& m ) {
    // ...
    m.debug_print();
    // chama o método de int_overflow!!!
  }
}
```

Re-throw

Dada uma função que capture uma exceção, é comum para um manipulador chegar a conclusão que nada pode ser feito a respeito do erro. Neste caso, a coisa típica a ser feita é o acusação da exceção novamente (*re-throw*), esperando que outro manipulador possa fazê-lo melhor. Por exemplo:

```
void h()
{
  try {
    // ...
  }
  catch( MATHERR ) {
    if ( posso_tratar ) tratamento();
    else throw; // re-throw
  }
}
```

Um *re-throw* é indicado pelo comando `throw` sem argumentos. A exceção de relevantamento é a exceção original tomada e não somente a parte que era acessível como `MATHERR`. Em outras palavras, se um `int_overflow` foi acusado, a função que chamou `h()` pode ainda tomar um `int_overflow` que `h()` tomou como `MATHERR` e decidiu reacusar.

```
void k()
{
  try {
    h();
```

```
    // ...
  }
  catch( int_overflow ) {
    // ...
  }
}
```

A versão abaixo deste tipo de comportamento pode ser útil. Assim como em funções, pode-se utilizar '...' (indicando qualquer argumento) de modo que `catch(...)` signifique qualquer exceção. Por exemplo:

```
void m()
{
  try {
    // ...
  }
  catch(...) {
    limpeza();
    throw;
  }
}
```

Isto é, se qualquer exceção ocorrer, resultado da execução de parte de `m()`, a função `limpeza()` será chamada no manipulador e a exceção que causou a chamada da função `limpeza()` será reacusada.

Devido ao fato de que exceções derivadas podem ser tratadas por manipuladores para mais de um tipo de exceção, a ordem em que os estes aparecem após o bloco de `try` é relevante. Os tratadores são escolhidos em ordem. Por exemplo:

```
try {
  // ...
}
catch( ibuf ) {
  // tratador de input overflow
}
catch( io ) {
  // tratador de qualquer erro de I/O
}
catch( stdlib ) {
  // tratador de qualquer erro em bibliotecas
}
catch( ... ) {
  // tratador de qualquer outra exceção
}
```

Bibliografia

- Ellis, Margareth, and Bjarne Stroustrup. *The Annotated C++ Reference Manual*.Addison Wesley, 1990.
- Meyer, Bertrand. *Object-oriented Software Construction*. Prentice-Hall, 1988.
- Stroustrup, Bjarne. *The C++ Programming Language*.2nd edition. Addison Wesley, 1991.