

Capítulo 9

Templates

Uma classe C++ normalmente é projetada para armazenar algum tipo de dado. Muitas vezes a funcionalidade de uma classe também faz sentido com outros tipos de dados. Este 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; isto é, pode-se fazer uma descrição de uma classe sem definir o tipo dos dados que ela manipula. Nesse caso, definição da classe é parametrizada por um tipo genérico T, sendo chamada C<T>. Esta construção não é uma classe realmente, mas uma descrição do conjunto de classes com o mesmo comportamento e que operam sobre um tipo T qualquer. Esta construção é denominada **class template**.

Com class templates, é permitido ao programador criar um série de classes distintas com a mesma descrição, mas sobre tipos distintos. Por exemplo, pode-se construir uma pilha de inteiros (Pilha<int>) ou pilha de strings (Pilha<char*>) apartir da mesma descrição. Esta descrição abstrata da template é utilizada pelo compilador para criar uma classe real em tempo de compilação, usando o tipo dos dados especificados quando de seu uso.

Em alguns textos acadêmicos, esta funcionalidade é chamada de polimorfismo paramétrico.

Declaração de templates

Consideremos novamente a classe *Pilha*, que já apareceu com várias implementações. Em todos os exemplos, a classe só armazenava inteiros, apesar de a funcionalidade nada ter a ver com o tipo de dado envolvido. Neste caso, ao invés de reescrevermos uma nova classe pilha para cada novo tipo demandado, pode-se definir uma template para a classe *Pilha*, onde o tipo armazenado é um T qualquer. A forma deste declaração é mostrada abaixo:

```
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()
    { if (topo)
      {
          elemPilha *ep = topo;
          T v = ep->val;
          topo = ep->prox;
          delete ep;
          return v;
      }
      return -1;
    }
};
```

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 de seus métodos.

Outra consequência 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, podem aparecer erros quando de sua utilização.

Esta checagem semântica é realizada 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. A checagem então tem que ser feita para cada tipo.

Usando templates

Definida a implementação de nossa template de pilhas, pode-se utilizá-la para qualquer tipo T criando pilhas de inteiros, strings, etc. Na criação de objetos destas classes pilhas, é necessário especificar o tipo de pilha na declaração do objeto:

```
void main()
{
    Pilha<int> intPilha;           // pilha de inteiros
    Pilha<char*> stringPilha;      // pilha de strings
    Pilha<Pilha<int>> intPilhaPilha; // pilha de pilha de inteiros

    intPilha.push(10);
    stringPilha.push( "teste" );

    intPilhaPilha.push( intPilha );
}
```

Declaração de métodos não inline

Quando da declaração de uma template, não é obrigatória a implementação de seus métodos na forma inline; isto é, sua implementação pode não estar no corpo da declaração da classe. Para tal, basta especificar, de maneira análoga a descrição de classes, a template a que o método pertence.

Segue abaixo a implementação do método pop fora do corpo da template:

```
template<class T> class Pilha {
// estrutura interna da template...
public:
    int vazia() { return topo == NULL }
    void push(T v) { topo = new elemPilha(topo, v); }
    T pop();      // protótipo do método pop
                  // implementado abaixo (fora da classe)
};
// Fim da declaração da classe

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;
}
```

Templates com vários argumentos genéricos

Templates não estão limitadas a terem apenas um único argumento. Pode-se utilizar diversos argumentos para parametrizar a descrição da classe. No exemplo abaixo, A *Pilha* armazena dois tipos de dados e ambos são parâmetros da template:

```

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; ou seja, tipos definidos pelo programador. Pode-se utilizar como argumentos da template tipos primitivos da linguagem como inteiros ou caracteres. No exemplo abaixo, tem-se uma pilha que armazena 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. Uma delas diz respeito ao tipo dos parâmetros: apenas tipos que podem ser convertidos para double podem usar esta função. Isto significa que objetos e ponteiros não podem ser utilizados. A outra limitação se refere ao tipo de retorno: este 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'.

A opção de sobrecarregar max para vários tipos também não é boa, pois teríamos que reescrever o código, que seria idêntico, para todos os tipos que quiséssemos usar. Ainda assim o problema não estaria resolvido, pois novos tipos não poderiam ser usados sem que se criasse outra versão sobrecarregada de max.

Na verdade, qualquer tipo de dado que possua operações de comparação pode ser usado com max, e sempre da mesma maneira. Não é possível escrever uma única função que trate todos os tipos, mas o mecanismo de templates possibilita descrever, para o compilador, como estas funções podem ser implementadas:

```

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

```

Esta função faz sentido para qualquer tipo T. Na realidade existirá uma implementação para cada tipo que for usado com esta função dentro do programa, mas estas funções serão geradas transparentemente pelo compilador. O uso de templates de funções não exige que se especifique explicitamente os tipos genéricos na chamada, como em templates de classes. Basta usar como se existisse uma função específica para o tipo envolvido:

```

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

```

Repare que, apesar de a função ser usada para comparar caracteres, em nenhum momento aparece a palavra char. O compilador sabe qual max precisa ser chamada pelo tipo dos parâmetros usados. Por causa disto, os tipos genéricos de templates de funções devem sempre aparecer nos parâmetros da função. Caso isto não aconteça, será sinalizado um erro de sintaxe na linha da declaração da template.

Assim como em templates de classes, templates de funções podem ter vários parâmetros genéricos.

Exercício 14 - Classe Vector para qualquer tipo.

Implementar com templates uma classe *Vector* com a mesma funcionalidade da desenvolvida na seção de sobrecarga de operadores, mas que possa ser usada com qualquer tipo.

Tratamento de Exceções

Quando do 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.

Funcionamento básico

O funcionamento dos tratadores de exceção é composto de diversas etapas:

Definição das exceções que uma classe pode levantar;

Definição de quando uma classe acusa uma exceção;

Definição do(s) tratador(es) das exceções.

A definição das exceções que podem ser levantadas é feita na construção da classe. Neste momento define-se as condições de erro e os representa sob a forma de uma classe. Desta forma, cada condição de exceção é descrita por uma classe de exceção. 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 que
                    // representa acesso com índice inválido
    int& operator[]( int i );
};
```

A classe *Range*, a princípio sem dados internos, representa a condição de erro para acesso com índice não válido (menor que zero, maior que o espaço alocado, etc.)

A definição dos momentos da exceção são também definidos na construção da classe. A acusação das exceções é feita normalmente nos métodos da classe que encontrem alguma situação de erro. O levantamento de uma exceção é feita pelo comando `throw` que, conforme no nosso exemplo, é acionado quando o índice do array é inválido. Nestas situações, os objetos da classe *Range* são utilizados como exceções e são acusados da seguinte forma:

```
int& Vector::operator[]( int i )
{
    if ( i<0 || i>size-1 ) throw Range();
                        // Range() cria um objeto da classe 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 definição dos tratadores aparecem, normalmente, nas funções que utilizam serviços das classes que levantam exceções. No caso de C++, estas funções podem seleccionar quais as exceções que serão tratadas e trechos onde estas podem surgir.

No nosso exemplo, 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
        // a 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 size;
    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 ( i<0 || i>size-1 ) throw Range(i);
                        // acusa-se a exceção indicando
                        // o valor índice inválido ao construir Range
    return p[i];
}

```

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_Exhausted ) {
        // ...
        // 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 dá 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 destas funções.

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. 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() // redefinição de 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; isto é, se dentro do tratador chamarmos a função *debug_print*, não conseguiremos ver nada sobre o erro de overflow de inteiros. Isto é devido ao fato do compilador não fazer late-binding com o objeto.

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 ) { // recebe no manipulador uma referência
        // ...
        m.debug_print();
        // chama o método de int_overflow!!!
    }
}
```

Re-throw

Dada uma função que capture uma exceção, não é incomum 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 reavertimento 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 recusar.

```
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 recusada.

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
}

```

Especificação de exceções

A acusação e o tratamento de exceções afetam o relacionamento entre as funções. Neste sentido, é interessante haver um mecanismo de especificar quais exceções que podem ser levantadas como parte da declaração de uma função. Por exemplo:

```
void f( int a ) throw (x2, x3, x4);
```

especifica que a função *f* só pode acusar as exceções *x2*, *x3*, *x4* e suas derivadas, nada mais. Deste modo, está garantindo para quem a chama que durante sua execução nenhuma outra exceção será levantada. Se, por acaso, algo acontecer que invalide esta garantia, a tentativa de acusação de uma exceção indevida será transformada em uma chamada para a função *unexpected*. O significado default para *unexpected* é a chamada a *terminate*, que normalmente representa um *abort*.

Desta forma, escrever:

```
void f( int a ) throw (x2, x3, x4)
{ /* implementacao qualquer */ }
```

significa a mesma coisa que:

```
void f( int a )
{
    try{
        /* implementação qualquer */
    }
    catch(x2) { throw; } // re-throw
    catch(x3) { throw; } // re-throw
    catch(x4) { throw; } // re-throw
    catch(...){ unexpected(); }
}
```

Mais que economia de digitação, o uso de especificação de exceções explicita as exceções que podem surgir na definição da função (.h) o que nem sempre aconteceria se esta definição ficasse em sua implementação.

Uma função sem especificação pode levantar qualquer exceção.

```
int f ();
```

enquanto que uma função sem a possibilidade de acusar qualquer exceção é declarada com uma lista explicitamente vazia:

```
int g() throw();
```

Exceções indesejadas

O mal uso de especificações de exceções pode levar a chamadas a função *unexpected*, que é indesejável a não ser no caso de testes. Pode-se evitar isto por uma boa estruturação e organização das exceções ou pela interceptação das chamadas a *unexpected*.

A função *set_unexpected* serve para interceptarmos estes casos. Esta função redefine o comportamento do sistema quando de uma exceção indesejada retornando o tratador antigo.

Abaixo temos um exemplo deste mecanismo. Neste caso, cria-se uma classe que representa um trecho onde exceções não previstas devem ser tratadas.

```
typedef void(*functype)();
functype set_unexpected( functype );

class MyPart {
    functype old;
public:
    MyPart( functype f ) { old = set_unexpected( f ); }
    ~MyPart() { set_unexpected( old ); }
}

void new_trat() { printf("novo tratamento.\n"); }

void f()
{
    MyPart( &new_trat ); // construtor implica em redefinição
    g();
} // destrutor reseta unexpected() anterior
```

Neste caso, a execução de *f* é protegida contra erros de exceções não desejadas.

Exceções não tratadas

Uma exceção acusada e não tratada implica na chamada da função *terminate*. Esta também é chamada se o mecanismo de exceções de C++ encontrar a pilha corrompida. *terminate* executa a última função recebida como argumento da função *set_terminate*.

Este mecanismo serve como mais um nível para erros de exceção. É normalmente utilizado para medidas mais drásticas no sistema como: aborto da execução do processo, reinicialização do sistema, etc.

A redefinição deste comportamento é feito de modo análogo ao *unexpected*.

```
typedef void (*PFV) ();
PFV set_terminate (PFV);
```

Exemplo: calculadora RPN com tratamento de exceções

Como a calculadora usa a classe *Stack*, esta será atualizada para gerar exceções em caso de erro. Os erros possíveis são *push* com pilha cheia ou *pop* com pilha vazia:

```
// Excecoes da pilha
class StackError{};
class StackFull : public StackError {
public:
    int size;
    StackFull( int s ) { size = s; }
};
class StackEmpty : public StackError {};

class Stack {
    friend class StackIterator;
    int size;

    int top;

    int *elems;
```

```

public:
    Stack(int s = 50) { size = s; top = 0;
                        elems = new int[size]; }
    ~Stack()           { delete [] elems; }
    void push(int i)
    { if (top>=size) throw StackFull(size);
      elems[top++]=i; }
    int pop()
    { if (top==0) throw StackEmpty();
      return elems[--top]; }
    int empty()        { return top == 0; }
};

```

A própria calculadora tem seus erros:

```

class RpnError{};
class RpnNoOperands : public RpnError, public StackEmpty {};

class RPN : public Stack {
    void getop(int* n1, int* n2) throw(RpnNoOperands);
public:
    void sum() { int n1, n2; getop(&n1, &n2); push(n1+n2); }
    void sub() { int n1, n2; getop(&n1, &n2); push(n1-n2); }
    void mul() { int n1, n2; getop(&n1, &n2); push(n1*n2); }
    void div() { int n1, n2; getop(&n1, &n2); push(n1/n2); }
};

void RPN::getop(int* n1, int* n2) throw(RpnNoOperands)
{
    try {
        *n2 = pop();
        *n1 = pop();
    }
    catch ( StackEmpty )
    { throw RpnNoOperands(); }
}

```

Exercício 15 - Template Array com tratamento de exceções

Introduzir tratamento de exceções na template Array.

