Generic Programming

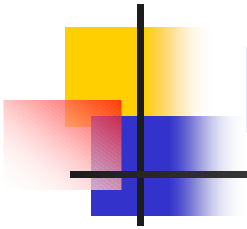
Daniel Terzella

Leandro Amancio

Eduardo Bruno Hadlich

Autores

- David R. Musser
- Alexander A. Stepanov
  - Artigo apresentado em 1988, em Roma – Itália.



Programação Genérica

- Definição dos Autores
  - "Generic Programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software."

Programação Genérica

- Outra definição
  - Programação Genérica pode ser definida como “programação com conceitos”, onde um conceito é definido como uma família de abstrações que são relacionadas por um conjunto comum de requisitos.

Analogia

vários bolos
fôrmas
sabores
Classes genéricas
Instâncias
Abstrações de dados

Classificação das Abstrações

- Abstração de dados
  - Tipos de dados e operações definidas sobre eles
  - Podem ser entendidas independentemente de sua implementação
    - Pilhas, Listas, Filas

Classificação das Abstrações

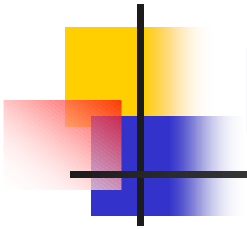
- Abstração de algoritmos
  - Famílias de abstrações de dados com operações comuns entre eles
    - Algoritmos de ordenação para listas, pilhas e filas

Idéias-chave

- Elevar o nível de abstração de algoritmos concretos sem perder eficiência.
- Utilizar algoritmos sem se preocupar com abstração de dados.
- Representações largamente adaptáveis para suportar seu uso direto na construção de software.

Utilização

- Definição de uma classe genérica é feita uma única vez e ela é reutilizada através do envio de parâmetros. (Parametrização)
- Especificação dos algoritmos em tempo de compilação.
- STL em C++



Exemplo abstrato

- Classes concretas
  - Real_stack, Integer_stack, Book_stack
- Classe genérica
  - Stack[G]
    - onde G é um parâmetro genérico
    - Classe reutilizável em diversas aplicações

Templates

- **Definição**: são uma abordagem para programação genérica, uma estratégia de programação na qual conjuntos de algoritmos são parametrizados por uma variedade de tipos de dados/estruturas de dados adequadas.
- O mecanismo de Templates em C++ permite que um tipo seja uma parâmetro na definição de uma classe ou função.
- Seu uso abre novas possibilidades na linguagem, tais como a descrição de algoritmos que se executam em tempo de compilação, a geração otimizada de códigos e o reuso de código.

Templates

- O uso de Templates se mostrou essencial no projeto da biblioteca padrão de C++ (STL), uma vez que as estruturas e algoritmos lá definidos devem ser de uso geral sem comprometer sua eficiência. Assim, vários algoritmos podem ser usados com várias estruturas de dados

Templates

- **Um *template* não causa perda de desempenho**: a transformação de um *template* "abstrato" em uma classe "concreta" é feita em tempo de compilação. E também não implica na redução da quantidade de código gerado: se tenho em meu projeto duas classes derivadas de um *template*, código será gerado de forma equivalente à definição direta de duas classes não relacionadas.

Templates

- **Declaração de template:** começa com a palavra chave template seguida por uma lista de parâmetros de tipos formais colocada entre os símbolos < e >. Todo parâmetro formal é precedido pela palavra-chave **typename** ou pela palavra-chave **class**.

- **Parâmetros de tipos formais** são tipos primitivos ou tipos definidos pelo programador, usados para especificar os tipos de parâmetros da classe ou função, o tipo de retorno da função e para declarar variáveis dentro do corpo da função.

Exemplo Função Genérica

```
template < class T>
T maximum (T value1, T value2, T value3)
{
        T max = value1;
        if (value2 > max)
            max = value2;

        if (value3 > max)
            max = value3;

        retunr  max;
}
```

Uso da Função Genérica

```
//versão para inteiros
int main ()
{
    int    int1, int2, int3
    cout << “Forneça 3 valores inteiros: ”;
    cin  >> int1  >> int2  >> int3;
    cout << “O maior valor inteiro é: ”
          << maximum (int1, int2, int3);
…
```

Uso da Função Genérica

```
...
//versão para double
double   double1, double2, double3

cout << “Forneça 3 valores double: ”;
cin  >> double1  >> double2  >> double3;
cout << “O maior valor double é: ”
        << maximum (double1, double2, double3);
...
```

Uso da Função Genérica

```
...
//versão para caracteres
char   char1, char2, char3

cout << “forneça 3  caracteres: ”;
cin  >> char1  >> char2  >> char3;
cout << “O maior valor de caractere é: ”
        << maximum (char1, char2, char3);

return 0;
} //fim do main
```

Resultado do Uso da Função

Forneça 3 valores inteiros: 1 2 3
O maior valor inteiro é: 3

Forneça 3 valores double: 3.3 2.2 1.1
O maior valor double é: 3.3

Forneça 3 caracteres: A C B
O maior valor de caractere é: C

Resultado do uso da Função

- Resutado da compilação é o mesmo que se fossem escritas 3 funções diferentes para cada tipo, ou seja, é compilada uma função para operar com int, uma para double e outra para char.

- Quando é feita a compilação, todas as referências feitas a class T na função genérica são substituídas pelo tipo enviado por parâmetro.

Exemplo da Função Compilada

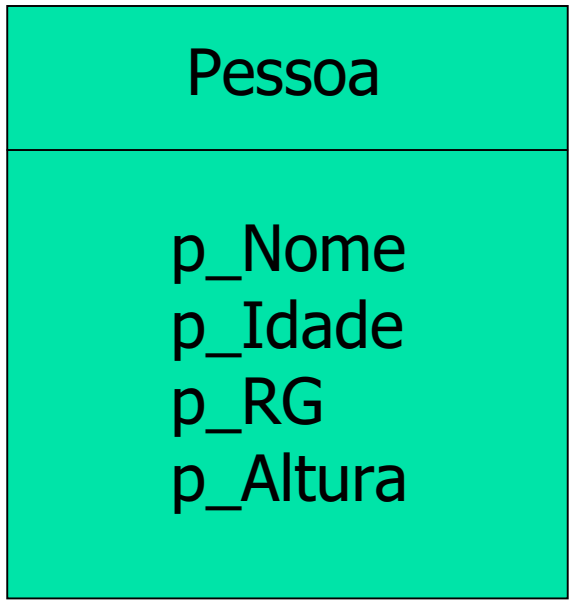
```
int maximum (int value1, int value2, int value3)
{
        int max = value1;
        if (value2 > max)
            max = value2;

        if (value3 > max)
            max = value3;

        retunr  max;
}
```

Uso da Função por uma Classe

Se fossem passados para a função 3 Objetos da classe Pessoa?



Função Compilada com Classe Pessoa

```
Pessoa maximum (Pessoa value1, value2, value3){
        Pessoa max = value1;
        if (value2 > max)
            max = value2;

        if (value3 > max)
            max = value3;

        retunr  max;}
```

Como fazer “Pessoa1 > Pessoa2” ?

Função Genérica com Classe Pessoa

- Quando se escreveu a função genérica não se sabia a priori qual é o método que compara os dados, justamente porque a classe que chama esse método é genérica (template).

- Apenas sabe-se que para que uma classe de entrada possa usar essa função, é preciso que o método **operator>** esteja definido.

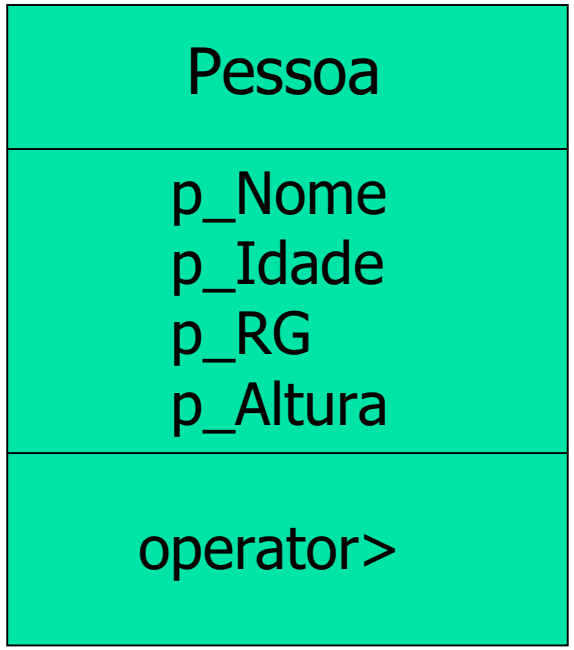
Sobrecarga

- Sobrecarga objetiva fornecer para tipos definidos pelo usuário as mesmas expressões que C++ fornece para seus tipos primitivos.

- Como podemos saber quem é a maior pessoa?
  R: Pode ser visto pela Altura. A maior pessoa é a mais alta!

Sobrecarga

- Deve-se sobrecarregar o operador de **>** para que a classe pessoa possa usar a função.



Sobrecarga

Exemplo de função sobrecarregada:

```
//recebe objeto pessoa como parâmetro
bool  operator> (Pessoa &A)
{  return  (p_Altura  >  A.p_Altura)  }
```

Na função genérica quando for feito:

```
if (value1 > max)
```

seria como se estivesse chamando uma função da classe:

```
value1.operator>(max);
```

OBS: Outro operador que deveria ser sobrecarregado seria o operador de atribuição (=).

Sobrecarga

- Sobrecarga é um procedimento bem comum na Linguagem C++, por exemplo o operador soma (+) executa de maneiras diferentes para ints, floats, doubles pois já foram sobrecarregados na linguagem C++.

- Existem várias maneiras de se sobrecarregar porém, basta-se saber que sobrecarga é um recurso facilitador para a programação genérica usando templates em C++.

- Mas para que isso ocorra deve-se ter classes e funções bem documentadas para que seu usuário tenha noção de que métodos criar na sua classe para usar determinada classe ou função genérica.

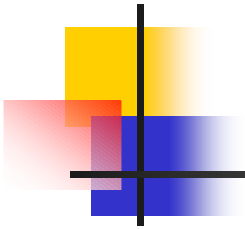
Mais de um Parâmetro

- um *template* pode ter mais de um parâmetro. Por exemplo:

```
template<class T, int i> class Buffer {
T v[i];
int sz;
public:
Buffer() : sz(i) {};
//...};
Buffer<float, 127> cbuf;
```

- <u>observação importante</u>: como precisamos definir a **instanciação** em tempo de compilação, certos parâmetros devem ser instanciados por literais:

```
void f(int i) { Buffer<int, i> bx; //erro }
```



Especialização

- *O template* genérico deve ser declarado antes de qualquer especialização:

- **template<class T> class List {/*...*/};** //template genérico

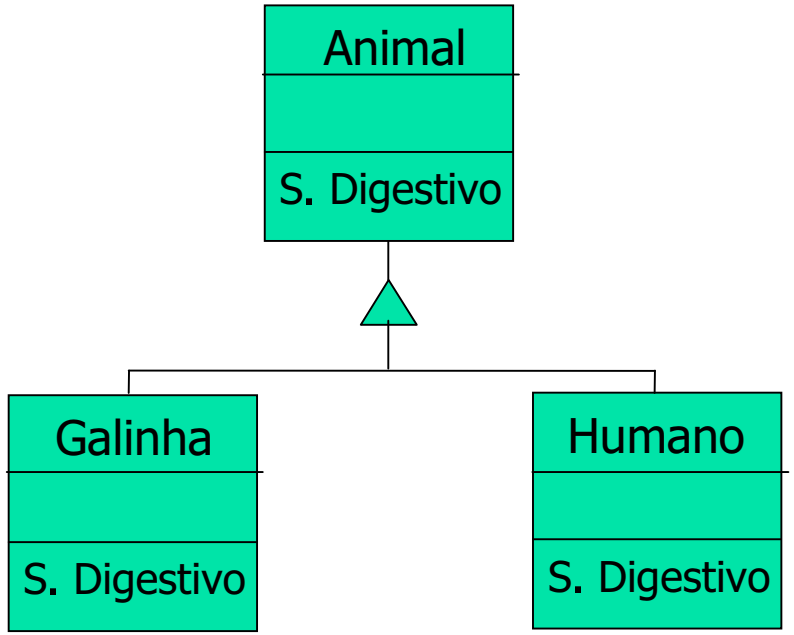
- **template<class T> class List<T*> {/*...*/};** //template especializado em ponteiros

- Nesta situação, o compilador primeiramente irá procurar o *template especializado (* List<ponteiro>) antes de qualquer *template* List parametrizado por um ponteiro.
- Desta forma, cria-se com menos trabalho uma interface para uma classe mais genérica e ao mesmo tempo uma única classe para todos os tipos de Lista de ponteiro;

Polimorfismo X Templates

- **templates e funções virtuais tem objetivos bastante semelhantes:** permitem a criação de conjuntos de tipos de dados compartilhando uma interface em comum.
- Funções virtuais fornecem um tipo de comportamento polimórfico em tempo de execução, enquanto templates agem em tempo de compilação.
- Quando utilizar um ou outro?

Polimorfismo X Templates

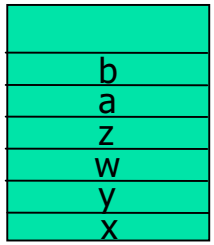


funções virtuais são adequadas quando existe uma hierarquia entre as classes que compartilham aquela interface, possibilitando uma forma elegante de permitir diferentes implementações de um mesmo método

Polimorfismo X Templates

- templates tratam suas classes instanciadas em igual nível hierárquico. A implementação de um método é única: podem-se utilizar especializações para determinar as ligeiras diferenças entre as classes (se necessário), enquanto o algoritmo principal é o mesmo para este conjunto de classes.
- Além disso, por questões de eficiência, pode ser mais interessante reorganizar determinados métodos em templates ao invés de distribuí-los em funções virtuais presentes em diferentes classes;

Polimorfismo X Templates



Pilha Genérica (LIFO)

- exemplo genérico: conhecemos o conceito e o comportamento de uma pilha ("*último a entrar é o primeiro a sair*") sem mesmo saber dos tipos de itens que estão inseridos nela. Mas quando chega no momento de efetivamente instanciar uma pilha, um tipo de dados deve ser especificado. Necessitamos de meios de descrever a noção de pilha genericamente e instanciar classes que são versões desta classe genérica de pilha para tipos específicos de pilha. Este recurso é fornecido por Templates.

Exemplo de Classe Genérica

```
//declaração da classe Pilha
template< class T> class Stack{
public:
        Stack (int = 10);                    //construtor default (tamanho da pilha 10)
        ~Stack() { delete[] stackPtr;}       //destruidor
        boll  push ( const T&);              //insere um elemento na pilha
private:
        int   size ;                         //quantidade de elementos na Pilha
        int   top;                           //posição do elemento no topo da pilha
        T     *stackPtr;                     //ponteiro para pilha

        boll   isEmpty()    const    {retunr top == -1;}
        boll   isFull()    const    {retunr top ==  size - 1;}
};
```

Exemplo de Classe Genérica

```
//Construtor com tamanho default  10
template<class T>
Stack<T>::Stack ( int  s ) {
        size  =  (s  > 0  ?  s :  10);
        top  =  -1;                          //inicialmente stack está vazia
        stackPtr  =  new  T[ size ]          //aloca espaço para elementos
}
// insere um elemento na pilha (retorna 1 se bem sucedida e 0 caso contrário)
template < class T >
boll  stack <T>:: push ( const  T &pushValue){

   if  (  ! IsFull()  ) {
        stackPtr[ ++top ] = pushValue;       //coloca item em Satck
        retunr  true;                        //push bem sucedida
   }
  else { retunr   false; }                   //push mal sucedida
}
```

Exemplo de Classe Genérica

```
//retira um elemento da pilha
template< class T >
boll  stack <T>:: pop (  T &popValue){

if  (  ! isEmpty()  ){
            popValue  =  stackPtr[ top- ];        //remove item de stack
            return   true;                        //operação pop bem sucedida
}
return  false;                                    //pop mal sucedido
}
```

Uso da Classe Pilha Genérica

```
int main(){
//uso da pilha com double
stack < double > doubleStack(5);
double f = 1.1;
cout << "Inserindo elementos em doubleStack\n"

  while ( doubleStack.push( f )  )              //sucesso true retornado
  {        cout  <<  f  << ' ';
           f  +=  1.1;   }

cout << "\nStack está cheia. Não pode inserir  " <<  f
           << "\nRetirando elementos de  doubleStack\n ";

  while  (doubleStack.pop ( f ) )               //sucesso: true retornado
           cout << f  << '  ';

cout << "\nStack está vazia.  Não pode retirar\n";
```

Uso da Classe Pilha Genérica

```
... //uso da pilha com inteiro
Stack< int >  inStack;                    //usa construtor default
Int  i = 1;
cout << "Inserindo elementos em intStack\n"

while ( intStack.push( i )  )             //sucesso true retornado
{           cout << i  << '  ' ;
            ++i;  }

cout << "\nStack está cheia. Não pode inserir  " <<  i
            << "\nRetirando elementos de  intStack\n ";

while  (intStack.pop ( i ) )              //sucesso: true retornado
            cout << i  << '  ';
cout << "\nStack está vazia.  Não pode retirar\n";

return  0;
}  //fim do main
```

Resultado do Uso da Pilha

Inserindo elementos em doubleStacK
1.1 2.2 3.3 4.4 5.5
Stack está cheia não pode inserir 6.6

Retirando elementos de doubleStack
5.5 4.4 3.3 2.2 1.1
Stack está vazia. Não pode retirar

Inserindo elementos em intStack
1 2 3 4 5 6 7 8 9 10
Stack está cheia não pode inserir 11

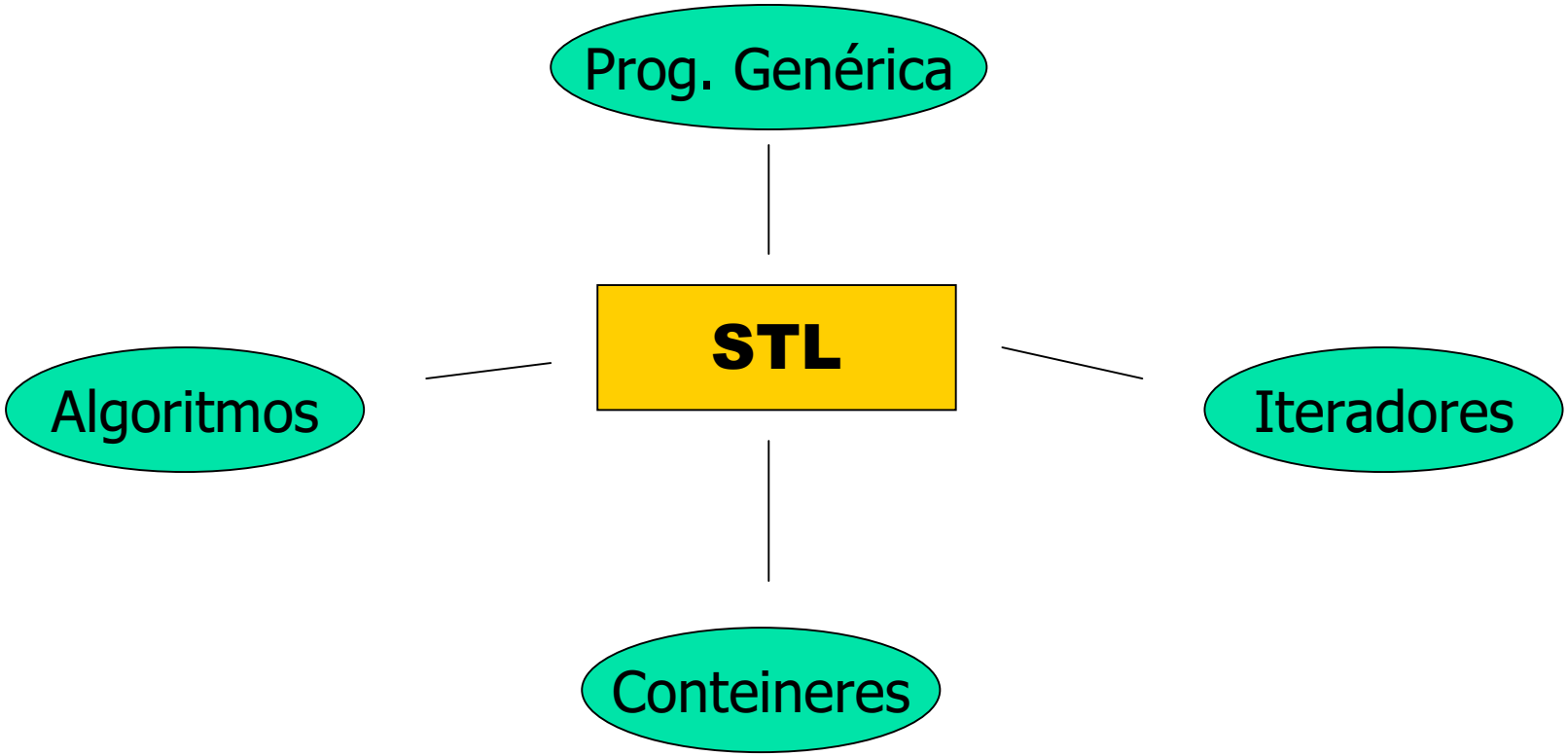
Retirando elementos de intStack
10 9 8 7 6 5 4 3 2 1
Stack está vazia. Não pode retirar

Standard Template Library

- O que é STL ?
  - Aprovada em 1994 pelo comitê de padrões C++
  - Criadores: Alexander Stepanov, Meng Lee e David Musser
  - Biblioteca padrão de componentes reutilizáveis
  - A STL é formada por três elementos chaves – contêineres, iteradores e algoritmos

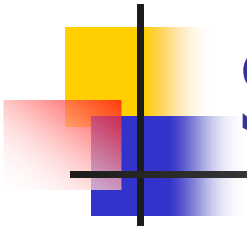
Standard Template Library

- STL:



Standard Template Library

- Contêineres
  - Estruturas de dados populares padronizadas, criados apartir de templates, utilizando-se o paradigma da programação genérica
    - Ex.: Pilhas, Listas, Filas...



Standard Template Library

- Algoritmos
  - Utilizados para manipular os conteineres
    - Ex: Inserir, Deletar, Procurar, Classificar...

Standard Template Library

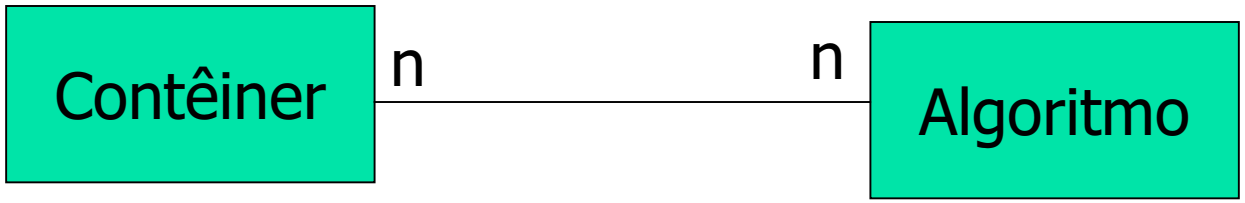
- Iteradores
  - São utilizados para acessar elementos de contêineres

Standard Template Library

- Conteineres
  - A STL foi cuidadosamente planejada de maneira que os contêineres forneçam funcionalidades similares, de forma que existam operações genéricas e outras operações que se aplicam a subconjuntos de conteineres similares.
    - Ex: função size (aplicada a todos conteineres)
    - Ex: função eraser (apenas contêineres da subclasse Adaptadores)

Standard Template Library

- Relação n x n



- Um algoritmo pode ser utilizado por vários conteineres
- Um contêiner pode utilizar vários algoritmos

Standard Template Library

- Outras bibliotecas
  - Algoritmos construidos nas classes de conteineres

Contêiner
Algoritmo A
Algoritmo B
Algoritmo C

Standard Template Library

- Exemplo:

```
// container
list<string> Friends;

// iterator to the begin
list<string>::iterator begin = Friends.begin()of the container

// iterator to the end of the container
list<string>::iterator end = Friends.end() of the container

// algorithm find() search for "Philippe" in the container

if (find(Friends.begin(),Friends.end(),"Philippe") != Friends.end())

cout << "Philippe is a friend of mine" << end;
```

Standard Template Library

- Vantagem
  - Usando a STL, podemos economizar tempo e esforço consideráveis de programação e obter como resultado softwares de maior qualidade.
  - O desenvolvedor não perde tempo na construção de componentes que já existem na biblioteca
  - Softwares de maior qualidade

Programação Genérica

- Links:
  - Generic Programming
    - http://www.cs.rpi.edu/~musser/gp
  - STL
    - http://www.sgi.com/Technology/STL
    - http://www.del.ufrj.br/~villas
    - http://www.cs.rpi.edu/projects/STL/stl/stl.html
    - http://www.isel.ipl.pt/Agenda/seminario2.html