Minicurso de C++

Aula 05
Objetos const, Templates, Containers

Responsáveis:
Gabriel Daltro Duarte
Rafael Guerra de Pontes
Walisson da Silva
Weslley da Cunha Santos
Ianes Grécia

Objetos const e Funções-membro const

Objetos const

- Alguns objetos precisam ser modificáveis e alguns não. O programador pode utilizar a palavra chave **const** para especificar que um objeto não pode ser modificado e que qualquer tentativa de modificação gerará um erro de compilação.

  Exemplo: const Ponto Origem;

Objetos const

- A “constância” de um objeto só é imposta após a inicialização do objeto pelo construtor e é encerrada quando o destrutor é chamado;

Funções-membro const

- Os compiladores C++ não permitem a chamada de funções membro para objetos const, a menos que a própria função membro também seja declarada como const. Isso é válido até para funções como get que são apenas de leitura.

- O compilador não permite que funções membro declaradas como const modifiquem o objeto.

Funções-membro constantes

- Declarar funções membro como const reforça o controle efetuado pelo compilador e permite uma programação mais segura, sem nenhum custo adicional de execução.
- Para definir uma função membro de dados com const deve-se colocar a palavra const após a lista de parâmetro da função. Veja o exemplo abaixo:
  Exemplo: `void imprimePonto () const;`

Exemplo

```
class Ponto {
    public:
        Ponto(double = 0, double = 0);
        void setX (double);
        void setY (double);
        void setXY (double, double);
        double getX (void) const;
        double getY (void) const;
        double distanciaDaOrigem (void) const;
    private:
        double x,y;

};
```

Exercício

- Declare uma classe Data com os membro de dados dia, mes e ano private;
- Declare funções set e get públicas para que o usuário da classe possa acessar dia, mes, ano;
- Declare um construtor que obrigatoriamente recebe argumentos para inicializar dia, mes e ano.
- Declare uma função public responsável por imprimir a data.
- Declare as funções que não modificam os membros de dados como const;
- Declare um objeto data const e chame a função para imprimir a data.

Templates

Templates

- É uma poderosa ferramenta do C++.
- Único segmento de código cria uma família de funções sobrecarregadas ou classes correlacionadas.
- Técnica chamada “programação genérica”.
- Exemplo:
  - Podemos criar uma função template para fazer uma ordenação genérica que pode ser usada para múltiplos tipos de dado (int, float, string, etc).

Templates de Funções

Templates de funções

- Funções sobrecarregadas geralmente realizam operações similares para tipos de dados diferentes.
- Se a operação e lógica do programa são idênticas para qualquer tipo, pode-se realizar um *overloading* mais compacto com templates de funções.
- Um template de função é basicamente uma família de funções sobrecarregadas.

Templates de funções

- Inicialmente, escreve-se a definição.
- Baseando-se nos tipos de argumento passados explicitamente ou inferidos pela chamada da função, o compilador gera códigos-fontes separados (especializações) para lidar com cada chamada apropriadamente.
- É melhor que macros da linguagem C, pois há um controle maior de checagem de tipo de dado.

Templates de funções

- Todo template de função deve começar com a palavra reservada `template` seguida da lista de parâmetros entre “< >”.
- Se o parâmetro definir um tipo, é precedido por `typename`.
- Exemplos:
  - template < typename A>
  - template < typename A, typename B >
- Cada parâmetro na lista de parâmentros do template é denominado “parâmetro de tipo formal”.
- Templates são geradores de código.

Exemplo 0

```
template < typename T > // or template< typename T >
T maximum( T value1, T value2, T value3 ){
    T maximumValue = value1; // assume value1 is maximum
    // determine whether value2 is greater than maximumValue
    if ( value2 > maximumValue )
    maximumValue = value2;
    // determine whether value3 is greater than maximumValue
    if ( value3 > maximumValue )
    maximumValue = value3;
    return maximumValue;
} // end function template maximum
```

Exemplo 1

```
// Function template maximum test
program.
#include <iostream>
#include "maximum.h" // include
definition of function template maximum
using namespace std;
int main()
{
// demonstrate maximum with int values
int int1, int2, int3;
cout << "Input three integer values: ";
cin >> int1 >> int2 >> int3;
// invoke int version of maximum
cout << "The maximum integer value is:
"
<< maximum( int1, int2, int3 ) ;
// demonstrate maximum with double
values
double double1, double2, double3;
cout << "\n\nInput three double values:
";
cin >> double1 >> double2 >> double3;
// invoke double version of maximum
cout << "The maximum double value is: "
<< maximum( double1, double2, double3 )
;
// demonstrate maximum with char values
char char1, char2, char3;
cout << "\n\nInput three characters: ";
cin >> char1 >> char2 >> char3;
// invoke char version of maximum
cout << "The maximum character value
is: "
<< maximum( char1, char2, char3 ) <<
endl;
} // end main
```

Execução do exemplo anterior

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

Templates de classes

Templates de Classes

- É possível implementar classes que tratem de maneira genérica um tipo de dado ainda desconhecido no momento da implementação.
- São chamados de tipos parametrizados, por precisar de um ou mais parâmetros de tipo para customizar uma classe genérica.
- Oportunidade de reutilização de código.
- Quando se instancia um objeto da classe, especifica-se os tipos de dados específicos.

Templates de classes

Os templates de classes requerem parâmetros de tipo (int, float, char, etc) para especificar como personalizar um template de “classe genérica” para formar uma especialização de template de classe.

Exemplo de Template de Classe: Pilha

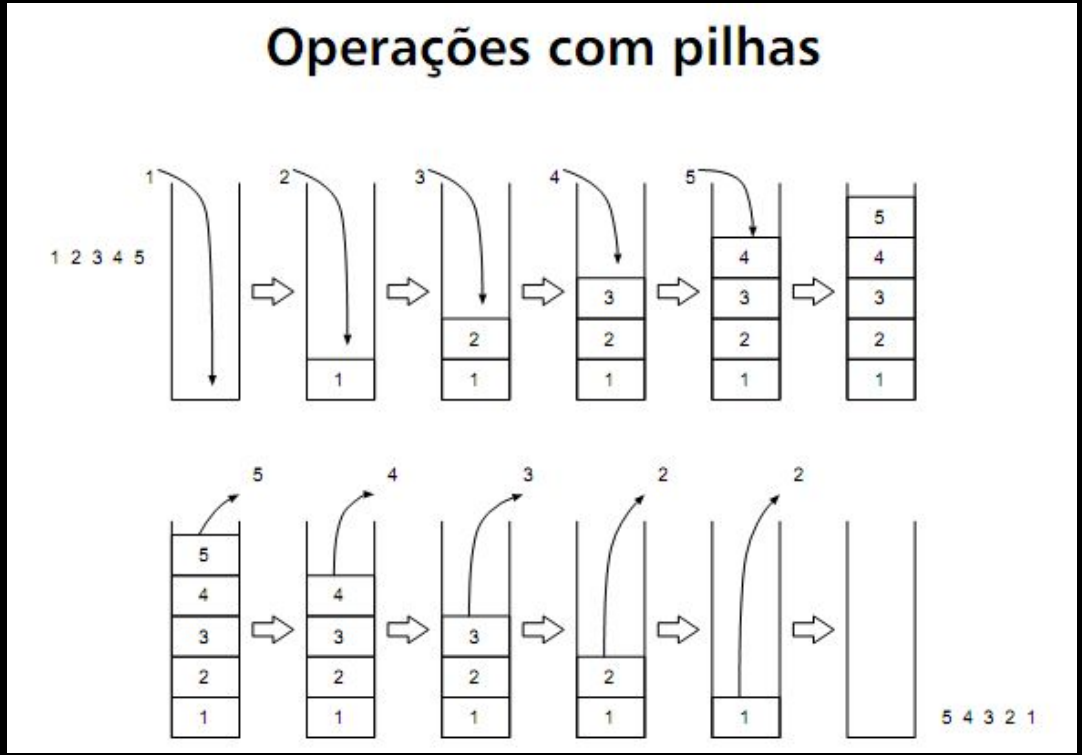
Definição de pilha:

Pilha é uma estrutura de dados na qual inserimos itens na parte superior e os recuperamos itens a partir do último que entrou

⇒ LIFO (*Last In, First Out*).

Templates de classes

Representação gráfica de uma pilha:



Exemplo de Template de Classe: Pilha

É possível entender o conceito de pilha independentemente do tipo dos itens que serão colocados na pilha;

Entretanto, para instanciarmos uma pilha, é necessário definir um tipo de dado (float, int, chat, etc.);

Exemplo de Template de Classe: Pilha

Seria útil a capacidade de escrevermos uma pilha genérica para tipos de dados genéricos e instanciar versões específicas de tipo dessa classe pilha genérica;

O C++ fornece essa capacidade por meio dos templates de classe;

Exemplo de Template de Classe: Pilha

Por exemplo, um template de classe “PILHA” pode tornar-se a base para criar muitas classes PILHAS (“PILHAS DE DOUBLE”, “PILHAS DE INT”, etc);

```
// Stack class template.
#ifndef STACK_H
#define STACK_H
template< typename T >
class Stack {
public:
explicit Stack( int = 10 ); // default constructor (Stack size 10)
// destructor
~Stack(){
delete [] stackPtr; // deallocate internal space for Stack
} // end ~Stack destructor
bool push( const T & ); // push an element onto the Stack
bool pop( T & ); // pop an element off the Stack
// determine whether Stack is empty
bool isEmpty() const {
        return top == -1;
} // end function isEmpty
// determine whether Stack is full
bool isFull() const {
        return top == size - 1;
} // end function isFull
private:
int size; // # of elements in the Stack
int top; // location of the top element (-1 means empty)
T *stackPtr; // pointer to internal representation of the Stack
}; // end class template Stack
// constructor template
template< typename T >
Stack< T >::Stack( int s )
: size( s > 0 ? s : 10 ), // validate size
top( -1 ), // Stack initially empty
stackPtr( new T[ size ] ) { // allocate memory for elements
// empty body
} // end Stack constructor template
// push element onto Stack;
// if successful, return true; otherwise, return false
template< typename T >
bool Stack< T >::push( const T &pushValue ){
if ( !isFull() ){
stackPtr[ ++top ] = pushValue; // place item on Stack
return true; // push successful
} // end if
return false; // push unsuccessful
} // end function template push
// pop element off Stack;
// if successful, return true; otherwise, return false
template< typename T >
bool Stack< T >::pop( T &popValue ) {
        if ( !isEmpty() ) {
                popValue = stackPtr[ top-- ]; // remove item from Stack
                return true; // pop successful
        } // end if
        return false; // pop unsuccessful
} // end function template pop
#endif
```

```
// Stack class template test program.
Function main uses a
// function template to manipulate objects
of type Stack< T >.
#include <iostream>
#include <string>
#include "Stack.h" // Stack class template
definition
using namespace std;
// function template to manipulate Stack<
T >
template< typename T >
void testStack(
Stack< T > &theStack, // reference to
Stack< T >
T value, // initial value to push
T increment, // increment for subsequent
values
const string stackName ) // name of the
Stack< T > object
{
cout << "\nPushing elements onto " <<
stackName << '\n';
// push element onto Stack
while ( theStack.push( value ) )
{
cout << value << ' ';
value += increment;
} // end while
cout << "\nStack is full. Cannot push " << value
<< "\n\nPopping elements from " << stackName <<
'\n';
// pop elements from Stack
while ( theStack.pop( value ) )
cout << value << ' ';
cout << "\nStack is empty. Cannot pop" << endl;
} // end function template testStack
int main()
{
Stack< double > doubleStack( 5 ); // size 5
Stack< int > intStack; // default size 10
testStack( doubleStack, 1.1, 1.1, "doubleStack"
);
testStack( intStack, 1, 1, "intStack" );
} // end main
```

Exemplo de saída

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Template de classe Vector

- O template de classe vector da C++ Standard Libary instancia objetos que representam um tipo de array mais robusto que possui muitas capacidades adicionais.
- Os arrays nativos do C e do C++ tem grande potencial para erros;

Template de classe Vector

- É possível facilmente ultrapassar qualquer uma das extremidades do array;
- Dois arrays não podem ser significadamente comparados com os operadores de igualdade, maior que ou menor que.
- Um array não pode ser atribuído a outro com o operador de atribuição;

Template de classe Vector

- Todas essas operações foram implementadas no tamplate de classe vector e podem ser feitas em seus objetos;
- O template de classe vector está disponível para qualquer pessoa que constrói aplicativos em C++ ;

Template de classe Vector

```cpp
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include <vector>
using std::vector;

void printVector ( vector <int> &ref);
void lerVector ( vector <int> &ref);

int main ()
{
   int i;
   vector <int> integers1 (5);
   vector <int> integers2 (10);

   cout << "Tamanho de integers1: " << integers1.size () << endl;
   cout << "Tamanho de integers2: " << integers2.size () << endl;

   lerVector (integers1);
   printVector (integers1);
   printVector (integers2);

   integers2 = integers1;

   printVector (integers2);

   integers2[6] = 1;
   integers2.at (6) = 1;

   return 0;
}
```

Template de classe Vector

```
Tamanho de integers1: 5
Tamanho de integers2: 10
1
2
3
4
5

1 2 3 4 5

0 0 0 0 0 0 0 0 0 0

1 2 3 4 5
terminate called after throwing an instance of 'std::out_of_range'
  what():  vector::_M_range_check

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.

Process returned 3 (0x3)   execution time : 279.763 s
Press any key to continue.
```

Standard Template Library

Standard Template Library (STL)

- Define poderosos e reutilizáveis componentes que implementam diversas estruturas de dados e algoritmos que as processam, tudo baseado em templates genéricos.
- É dividida em:
  - Containers
  - Iterators
  - Algoritmos

Containers

- Estruturas de dados capazes de armazenar objetos de praticamente qualquer tipo de dado.
- Cada um possui funções-membro próprias.
- São dinâmicos.
- Podem ser sequenciais, associativos ou adaptativos.

Standard Library container classes.

Standard Library container class	Description
Sequence containers	
`vector`	Rapid insertions and deletions at back. Direct access to any element.
`deque`	Rapid insertions and deletions at front or back. Direct access to any element.
`list`	Doubly linked list, rapid insertion and deletion anywhere.
Associative containers	
`set`	Rapid lookup, no duplicates allowed.
`multiset`	Rapid lookup, duplicates allowed.
`map`	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
`multimap`	One-to-many mapping, duplicates allowed, rapid key-based lookup.
Container adapters	
`stack`	Last-in, first-out (LIFO).
`queue`	First-in, first-out (FIFO).
`priority_queue`	Highest-priority element is always the first element out.