INF1010 - Estruturas de dados avançadas
Introdução a C++

Standard Template Library

PUC-Rio

2 de maio de 2018

Standard Template Library (STL)

- Biblioteca de classes e algoritmos parte da biblioteca padrão do C++.
- Fortemente baseada em uma programação genérica, com o uso de templates, fornece ao desenvolvedor:
  - Containers
  - Iteradores
  - Algoritmos

Containers

pair	tuple	array	vector
list	forward_list	deque	queue
priority queue	stack	set	multiset
map	multimap	unordered_set	unordered_multi
unordered_map	unordered_multimap	bitset	valarray

std::pair

- Estrutura que proporciona uma maneira de armazenar dois objetos heterogêneos como uma única unidade. Um caso especial de std::tuple.
- Definição:
  template <class T1, class T2 > struct pair;

```
std::pair<std::string, double> product("shoes",
                                       99.5);
//Acesso publico ao primeiro elemento
product.first  = "black shoes";

//Acesso publico ao segundo elemento
product.second = 149.99;
```

Função útil:

```
product = std::make_pair( "black shoes",
                          149.99 );
```

std::tuple

- Estrutura que proporciona uma maneira de armazenar n objetos heterogêneos como uma única unidade. Uma generalização de std::pair.
- Definição:
  template <class... Types> class tuple;

```
std::tuple<std::string, double, int size>
  product("shoes", 99.5, 42);

//Acesso publico aos elementos atraves
//do metodo get:
std::get<0>(product) = "black shoes";
std::get<1>(product) = 149.99;
std::get<2>(product) = 38;
```

Função útil:

```
product = std::make_tuple( "black shoes",
                           149.99, 38 );
```

std::array

- Encapsula um array de tamanho constante, equivalente ao array de C.
- Definição:
  template <class T, size_t N2 > class array;

```
std::array<int , 10> container;

for (int i = 0; i < 10; i++ )
  container[i]=i;
```

std::vector

- Encapsula arrays de tamanho dinâmico. Gerencia alocação e realocação de memória para os elementos, que são armazenados contiguamente.
- Definição:
  template <class T, class Alloc = allocator<T> > class vector;

```
std::vector<int> container;
for( int i = 0; i < 10; i++ )
  container.push_back(i);

//Saida: Terceiro elemento de 10: 2
std::cout <<
  "Terceiro elemento de " <<
  container.size() << ": " <<
  container[2] << std::endl;
```

std::deque

- Acrônimo irregular para *double-ended queue*, é um container sequencial de tamanho dinâmico, que pode ser expandido ou contraído nas extremidades (início e fim).
- Definição:
  template <class T, class Alloc = allocator<T> > class deque;
- A interface é similar a do std::vector. Porém, internamente, lidam com os elementos de forma diferente: enquanto vectors usam um único array que ocasionalmente necessita ser realocado para crescer, deques podem estar espalhados em diferentes pedaços de memória, e o container necessita manter informação necessária para permitir acesso direto aos seus elementos em tempo constante.
- Conta com funções *push_back* e *push_front* (não presente no std::vector) para inserir elementos.

std::list

- Container de inserção e remoção em tempo constante, usualmente implementado como uma lista duplamente encadeada.
- Definição:
  template <class T, class Alloc = allocator<T> > class list;

```
std::list<int> container;
for( int i = 0; i < 10; i++ )
  container.push_back(i);
```

- Não suporta acesso de tempo constante e direto, com [ ]. Se necessário acessar os elementos, isso deve ser feito com *iteradores*.

std::forward_list

- Versão de std::list implementada como uma simples lista encadeada.
  Provê maior economia de memória, e é uma melhor alternativa quando a iteração bidirecional não é necessária.
- Definição:
  template <class T, class Alloc = allocator<T> > class forward_list;

No fim, é melhor usar vetores ou listas?

std::queue

- Container que provê a funcionalidade de uma estrutura de dados do tipo fila (FIFO - first-in, first-out).
- Definição:
  template <class T, class Container = deque<T> > class queue;

```
std::queue<int> container;
for( int i = 0; i < 10; i++ )
  container.push(i);

//Saida: Primeiro da fila: 0
std::cout << "Primeiro da fila: "
          << container.front();

//Remove o primeiro da fila
container.pop();
```

std::stack

- Container que provê a funcionalidade de uma estrutura de dados do tipo pilha (FILO - first-in, last-out).
- Definição:
  template <class T, class Container = deque<T> > class stack;

```
std::stack<int> container;
for( int i = 0; i < 10; i++ )
  container.push(i);

//Saida: Topo da pilha: 9
std::cout << "Topo da pilha: "
          << container.top();

//Remove o elemento do topo
container.pop();
```

std::priority_queue

- Container que provê a funcionalidade de uma estrutura de dados do tipo heap.
- Definição:
  template <class T, class Container = vector<T>, class Compare = less<typename Container::value_type> > class priority_queue;

std::priority_queue

- Exemplo:

```
std::priority_queue<int> container;
container.push(30);
container.push(100);
container.push(25);
container.push(40);

//Saida: Em ordem: 25 30 40 100
std::cout << "Em ordem:";
while (!container.empty())
{
  std::cout << " " << container.top();
  container.pop();
}
```

std::set

- Container associativo que armazena um conjunto de objetos únicos em uma ordem específica. Implementado utilizando uma árvore binária de busca, executa busca, remoção e inserção em complexidade logarítmica.
- Definição:

```
template < class T,
           class Compare = <less<T>,
           class Alloc = allocator<T>
         > class set;
```

std::set

- Exemplo:

```
std::set<int> container;
for (int i=1; i<=5; ++i)
  container.insert(i*10);
//O set contem: 10 20 30 40 50

//Tentando inserir o elemento 20
std::pair<std::set<int>::iterator, bool> ret;
ret = container.insert(20);

if( container.second == false )
  std::cout << "Elemento 20 nao inserido!";
```

std::map

- Container associativo que armazena pares de chave-valor, onde as chaves são únicas. Também são implementados utilizando árvores binárias de busca, e portanto, busca, remoção e inserção são realizadas em complexidade logarítmica.
- Definição:

```
template < class Key,
           class T,
           class Compare = less<Key>,
           class Alloc =
             allocator< pair<const Key,T> >
         > class map;
```

std::map

- Exemplo:

```
std::map<int ,std::string> container;
container[12]="joao";
container[60]="maria";
container[5]="jose";

//Substituindo o elemento 5:
container[5] = "paula";
```

Iteradores

- São objetos usados para referenciar elementos armazenados em containers.
- Facilitam acesso e operações sobre containeres inteiros ou sobre partes delimitadas deles.
- Iteradores **apontam** para itens que são parte de um container.
- Funções begin() e end() retornam iteradores respectivamente para o início e para o fim dos containers.
- Operações realizadas com iteradores:
  i1++, ++i1, i1–, –i1,
  i1 = i2,
  i1 == i2, i1 != i2,
  *i1 (Derreferenciação: Acesso ao elemento apontado)

Iteradores

- Operações realizadas com iteradores de alguns containers sequenciais (vector e deque, p. e.):
  $i1 < i2, i1 <= i2, i1 > i2, i1 >= i2,$
  $i1 + n, i1 - n,$
  $i1 += n, i1 -= n,$
  $i1[n]$
- Nem todos os containers suportam iteradores, e nem todo iterador suporta todas as operações acima.

Iteradores

- Loop sem iterador:

```
std::vector<int> myIntVector;

myIntVector.push_back(1);
myIntVector.push_back(4);
myIntVector.push_back(8);

for( unsigned int i = 0;
     i < myIntVector.size();
     ++i )
{
    std::cout << myIntVector[i] << " ";
    //Saida deve ser: 1 4 8
}
```

Iteradores

- Loop com iterador:

```
std::vector<int> myIntVector;

myIntVector.push_back(1);
myIntVector.push_back(4);
myIntVector.push_back(8);

std::vector<int>::iterator it;
for( it = myIntVector.begin();
     it != myIntVector.end();
     ++it )
{
    std::cout << *it << " ";
    //Saida deve ser: 1 4 8
}
```

Iteradores

- Por que usar iteradores?

Iteradores

Range-based loops (C++11)

```
for( int number : myIntVector )
{
    std::cout << number << " ";
}
```

```
for( int& number : myIntVector )
{
    number++;
}
```

Iteradores

Escrevendo mais rápido e legível com a palavra-chave *auto* (C++11)

- Quando o compilador é capaz de determinar o tipo de uma variável no momento da sua inicialização, você não precisa necessariamente especificar o seu tipo.
  Por exemplo, no trecho de código abaixo, o compilador deduzirá que y é um inteiro.

```
int x = 3;
auto y = x;
```

Iteradores

- Isso é bastante útil para lidar com códigos com template, STL e iteradores.

Por exemplo, a inicialização de it, que normalmente seria:

```
std::vector< std::tuple< std::string,
                         double,
                         int > > products;

std::vector< std::tuple< std::string,
                         double,
                         int >
           >::iterator it = products.begin();
```

Pode ficar:

```
auto it = products.begin();
```

Algoritmos

- A STL provê funções genéricas (algoritmos) para uma grande variedade de propósitos: busca, ordenação, contagem, manipulação, etc.
- Muitos desses algoritmos são aplicados a ranges de elementos definidos como [first, last), onde last se refere ao elemento exatamente depois do range desejado.
- Definidos no cabeçalho <algorithm>.

Algoritmos

Algumas das funções mais utilizadas:

- Troca o valor de duas variáveis ou arrays:

```
template<class T>
void swap( T& a, T& b );

template<class ForwardIt1, class ForwardIt2>
ForwardIt2 swap_ranges( ForwardIt1 first1,
                        ForwardIt1 last1,
                        ForwardIt2 first2 );
```

- Inverte a ordem dos elementos no range passado:

```
template< class BidirIt >
void reverse( BidirIt first, BidirIt last );
```

Algoritmos

Algumas das funções mais utilizadas:

- Embaralha um conjunto de elementos:

```
template<class RandomIt>
void random_shuffle( RandomIt first,
                     RandomIt last );
```

- Elimina elementos repetidos, mantendo somente o primeiro encontrado e mantendo a ordem relativa:

```
template<class ForwardIt>
ForwardIt unique( ForwardIt first,
                  ForwardIt last );
```

Algoritmos

Algumas das funções mais utilizadas:

- Verifica se um determinado elemento está presente num container ordenado (ao menos parcialmente), fazendo uma busca binária.

```
template<class ForwardIt, class T>
bool binary_search( ForwardIt first,
                    ForwardIt last,
                    const T& value );
```

- Busca o maior/menor elemento no range passado:

```
template< class T >
const T& max( const T& a, const T& b );

template< class T >
const T& min( const T& a, const T& b );
```

Algoritmos

- Constrói um Max-Heap a partir de um conjunto de elementos:

```
template<class RandomIt>
void make_heap( RandomIt first,
                RandomIt last );
```

- Constrói um heap a partir de um conjunto de elementos e uma função de comparação:

```
template<class RandomIt, class Compare>
void make_heap( RandomIt first,
                RandomIt last,
                Compare comp );
```

Mais em: http://en.cppreference.com/w/cpp/algorithm

Referências

- Tutoriais, referências de funções, fórum, etc: http://www.cplusplus.com
- Livros:
  - Drozdek, Adam. Data Structures and Algorithms in C++. Pacific Grove, CA: Brooks/Cole, 2001.
  - Paul J. Deitel. 2010. C++ how to Program. P.J. Deitel, H.M. Deitel (7th ed.). Pearson Education.
  - Scott Meyers. 1998. Effective C++ (2nd Ed.): 50 Specific Ways to Improve Your Programs and Designs. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
  - Scott Meyers. 2014. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14 (1st ed.). O'Reilly Media, Inc.
- E muito material na Internet...