A Linguagem C++

Vagner Sacramento e
Humberto Longo
{vagner,longo}@inf.ufg.br

Tópicos Avançados em Programação

1

Biblioteca Padrão

C++ STL - Standard Template Library

2

Biblioteca Padrão

- Parte 1:
  - Visão Geral
  - Contêiners
- Parte 2:
  - Iteradores
  - Algoritmos
  - Objetos Função
- Parte 3:
  - Strings
  - Streams

3

Visão Geral

4

Organização

- Contêiners
  - Estruturas de dados
- Utilidades
  - Operadores, pares, objetos função, alocadores, etc.
- Iteradores
- Algoritmos
  - Busca, ordenação, etc.
- Diagnóstico
  - Exceções, assetivas, etc.
- Strings
- I/O
  - Streams, buffers, arquivos, etc.
- Localização
  - Configurações regionais
- Suporte C++
  - Limites numéricos, RTTI, etc.
- Numérica
  - Complexos, operações sobre vetores e matrizes, etc.

5

STL background

- the Standard Template Library (STL) was developed by Alex Stepanov, originally implemented for Ada (80's - 90's)
- in 1997, STL was accepted by the ANSI/ISO C++ Standards Committee as part of the standard C++ library
- adopting STL also affected strongly various language features of C++, especially the features offered by *templates*
- supports basic data types such as *vectors*, *lists*, associative containers (*maps*, *sets*), and algorithms such as sorting,
  - efficient, and compatible with C/C++ computation model
  - *not object-oriented*: many operations (algorithms) are defined as stand-alone functions
  - uses **template**s for reusability

6

Basic principles of STL

- STL containers are type-parameterized templates, rather than classes with inheritance and dynamic binding
- there is *no common base class* for all of the containers
- no **virtual** functions and late binding
- however, containers implement a (somewhat) uniform container interface with similar operations
- the standard *string* was define independently but later extended to cover STL-like interfaces and services
- STL collections do not directly support I/O operations
  - but s*td*::*istream_iterator* <T> and *std*::*ostream_iterator* <T> can represent IO streams as STL compatible iterators
  - IO can achieved using STL algorithms (*copy*, etc.)

7

Introduction to STL

- STL (*Standard Template Library*) provides three components:
- (1) *containers*, for holding and owning *homogeneous* collections of values; a container itself manages the memory for its elements
- (2) *iterators* are syntactically and semantically similar to C-like pointers; different containers provide different iterators (but with similar interfaces)
- (3) *algorithms* operate on various containers via iterators; algorithms take different kinds of iterators as (generic) parameters; to execute an algorithm on a container, the algorithm and the container must support compatible iterators
  - std::sort, std::reverse

8

Example of STL (Stroustrup)

```
#include <vector>                          // get std::vector
#include <algorithm>      // get std::reverse, std::sort, etc.
...
int main () {
  std::vector <double> v;            // buffer for input data
  double d;
  while (std::cin >> d)                       // read elements
    v.push_back (d);
  if (!std::cin.eof ()) {           // check how input failed
    std::cerr << "format error\n";
    return 1;                                  // error return
  }
  std::cout << "read " << v.size () << " elements\n";
  std::reverse (v.begin (), v.end ());
  std::cout << "elements in reverse order:\n";
  for (int i = 0; i < v.size (); ++i)
    std::cout << v [i] << '\n';
}
```

9

Basic concepts of STL

- *containers* are parameterized class templates; they try to make *minimal* assumptions about the type of elements that they hold -
  but of course need some operations, e.g., for copying elements
- *iterators* are abstractions, compatible to pointers, that provide access to elements within a particular container
- iterators are used for either reading or modifying the elements of the container
- *algorithms* are parameterized function templates; they do not know the actual type of the containers they operate on
- algorithms are purposely decoupled from the containers, and they always use the iterators to access elements in the container

10

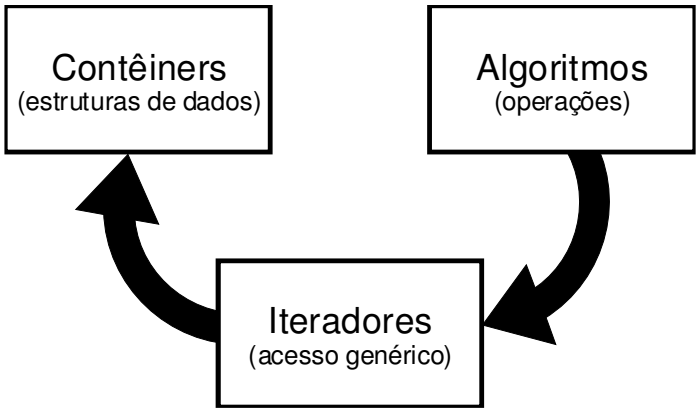
Basic concepts (cont.)

- STL algorithms have an associated *time complexity*, implemented for efficiency (*constant*, *linear*, *logarithmic*)
- they are function templates, parameterized by iterators to access the containers they operate on:

```
std::vector <int> v;
.. // initialize v
std::sort (v.begin (), v.end ());   // instantiate
std::deque <double> d;        // double-ended queue
.. // initialize d
std::sort (d.begin (), d.end ());
// again
```

11

Standart Template Library



12

Contêiners

Containers

- Um objeto que contém outros objetos
  - Listas, vetores e arrays associativos
- Duas categorias
  - Sequenciais
    - vector, list, dqueue, stack,...
  - Associativos
    - map, set,...

Introduction to containers

- a *container* is a class whose objects hold a *homogeneous* collection of values.
- ```
  Container <T> c;                    // initially empty
  ```
- when you *insert* an object into a container, you actually insert a value *copy* of this object
- ```
  c.push_back (value);          // grows dynamically
  ```
- the element type *T* must support a copy constructor (that performs a correct, sufficiently deep *copying* of object data)
- *heterogeneous* collections are represented as containers storing *pointers* to a base class
  => brings out all pointer/memory management problems

15

Intr. to containers (cont.)

- in *sequence containers*, each element is placed in a certain relative position: as first, second, etc.:
- `vector <T>` vectors, sequences of varying length
- `deque <T>` deques (with operations at either end)
- `list <T>` doubly-linked lists
- *associative containers,* used for representing *sorted collections*
- `set <KeyType>` sets with unique keys
- `map <KeyType, ValueType>` maps with unique keys
- `multiset <KeyType>` sets with duplicate keys
- `multimap <KeyType, ValueType>` maps with duplicate keys
- `hash_map <KeyType, ValueType>` provided by many libraries
- but not (yet) by the standard

16

Intr. to containers (cont.)

- standard containers are somewhat interchangeable - you can choose the one that is the most efficient for your needs
- however, they do provide somewhat different interfaces and services
- changing a container may involve changes to the client code
- different kinds of algorithms require different kinds of iterators
- once you choose a container, you can apply an algorithm that accepts a *compatible* iterator
- *container adapters* are used to adapt containers for the use of specific interfaces
- for example, *stacks* and *queues* are adapters of sequences

17

Containers padrão baseado no container vector

- Tipos de membros
- Iteradores
- Acesso aos elementos
- Construtores
- Operações com pilha, lista
- Tamanho e capacidade
- Funções auxiliares

18

Operações Comuns

Todo container padrão define estes nomes de tipo como membros

```
template <class T, class A = allocator<T> > class std: :vector {
public: // Tipos
    typedef T                                   value_type;
    typedef A                                   allocator_type;
    typedef typename A::size_type               size_type;
    typedef typename A::difference_type         difference_type;
    typedef /* DEP. DE IMPLEMENT. */            iterator; // algo como um T*
    typedef /* DEP. DE IMPLEMENT. */            const_iterator;      // +/- const T*
    typedef std::reverse_iterator<iterator>     reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    typedef typename A::pointer                 pointer; // para elemento
    typedef typename A::const_pointer           const_pointer;
    typedef typename A::reference               reference; // de elemento
    typedef typename A::const_reference         const_reference;
    ...
```

19

Programação Genérica – typenames e templates

```
template<class C> typename C::value_type sum(const C& c)
{
    typename C::value_type s = 0;
    typename C::const_iterator p = c.begin() ; // começa do começo
    while (p!=c.end()) {                       // continue até o fim
        s += *p;                               // pega um valor
        ++p;                    // faça p apontar para o próximo elemento
    }
    return s;
}
```

20

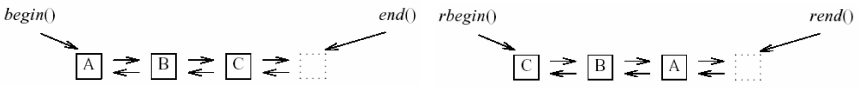
Operações Comuns (cont)

```
template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // iteratores:
    iterator                begin() ; // aponta para o primeiro
    const_iterator          begin() const;
    iterator                end() ;    // aponta para um após o último
    const_iterator          end() const;
    reverse_iterator        rbegin() ;// aponta para o último
    const_reverse_iterator  rbegin() const;
    reverse_iterator        rend() ;   // aponta para um antes do primeiro
    element of reverse      sequence
    const_reverse_iterator  rend() const ;
    // ...
};
```


Iterador

Iterator

Reverse Iterator




```
template<class C>
typename C::iterator find_last(const C& c, typename C::value_type v)
{
    return find_first(c.rbegin(),c.rend(),v).base();
}
```

Iterador de um vetor

Sem iterador reverso teríamos que fazer um loop:

```
template<class C>
typename C::iterator find_last(const C& c, typename C::value_type v)
{
        typename C::iterator p = c.end();   //search backwards from end
        while (p!=c.begin()) {
                --p;
                if (*p==v) return p;
        }
        return p;
}
```

23

Características Comuns

- Tipos parametrizados
  - value_type — Tipo dos elementos
  - allocator_type — Tipo do alocador usado
- Tipos de iteradores
  - iterator — Tipo do iterador
  - const_iterator — Tipo do iterador constante
  - reverse_iterator — Tipo do iterador reverso
  - const_reverse_iterator — Tipo do iterador reverso constante
- Tipos ponteiro e referência
  - pointer — Tipo ponteiro para o elemento
  - const_pointer — Tipo ponteiro constante para o elemento
  - reference — Tipo referência para o elemento
  - const_reference — Tipo referência constante para o elemento

24

Características Comuns

- Criação e destruição
  - Construtor *default* — Cria o contêiner vazio
  - Construtor de cópia — Copia um contêiner de mesmo tipo
  - Construtor(begin, end) — Cria um contêiner e copia em [begin, end[
  - Destrutor — Apaga todos os elementos e libera a memória
- Comparação
  - == — Verifica se o conteúdo é igual
  - != — Verifica se o conteúdo não é igual
  - < — Verifica se o conteúdo é lexicograficamente menor
  - > — Verifica se o conteúdo é lexicograficamente maior
  - <= — Verifica se o conteúdo é lexicograficamente menor ou igual
  - >= — Verifica se o conteúdo é lexicograficamente maior ou igual
  - = — Atribui o conteúdo de outro contêiner

25

Características Comuns

- Acesso
  - size() — Devolve o número de elementos
  - empty() — Verifica se o contêiner está vazio
  - max_size() — Devolve o número máximo de elementos
  - begin() — Iterador para o início
  - end() — Iterador para um após o fim
  - rbegin() — Iterador para o início da iteração reversa
  - rend() — Iterador para um após o fim da iteração reversa
- Alteração
  - swap(c) — Troca o conteúdo com outro contêiner (::swap(c,c))
  - insert(pos,elem) — Insere uma cópia de elem (significado de pos varia)
  - erase(beg,end) — Remove todos os elementos em [beg,end[
  - clear() — Remove todos os elementos

26

Requisitos dos Elementos

- Cópia
  - Os elementos devem poder ser copiados, pois os contêiners guardam cópias de elementos.
- Comparação
  - Containers associativos fazem comparação dos elementos nas buscas (< e =)
    - O que acontece no código abaixo?

```
std::map<char*,int> map;
char name[] = "Renato";
map[name] = 11;
cout << map["Renato"] << '\n';
```

27

Contêiner Vetor

- seqüência de elementos para acesso aleatório
  - ☺ Acesso aleatório
  - ☹ Alterações no meio ou no início
- Cabeçalho
  - #include <vector>
- Parâmetros
  - Tipo dos elementos armazenados
  - Tipo do gerenciador de memória (opcional)
- Exemplos
  - vector<**int**> vi;
  - vector<MyClass> vobj;
  - vector<vector<string> > mtrstr;

28

Acesso aos elementos

```
template <class T, class A = allocator<T> > class vector {
public:
        // ...
        // element access:

        reference operator[] (size_type n);          // unchecked access
        const_reference operator[] (size_type n) const;

        reference at(size_type n);                   // checked access
        const_reference at(size_type n) const;

        reference front();                           // first element
        const_reference front() const;
        reference back();                            // last element
        const_reference back() const;

        // ...
};
```

29

Acesso aos elementos

```
void f(vector<int>& v, int i1, int i2)
try {
        for(int i = 0; i < v.size(); i++) {
                // range already checked: use unchecked v[i] here
        }

        v.at(i1) = v.at(i2); // check range on access

        // ...
}
catch(out_of_range) {
        // oops: out-of-range error
}
```

30

Operações do Contêiner Vetor

- Construtores
  - vector(n) — n posições com valor *default*
  - vector(n,val) — n posições com valor val
- Atribuição
  - assign(beg, end) — Atribui um vetor com os valores de [beg,end[
  - assign(n,val) — Atribui um vetor com n cópias de val
- Acesso
  - [pos] — Indexação sem verificação
  - at(pos) — Indexação com verificação
  - front() — Primeiro elemento
  - back() — Último elemento

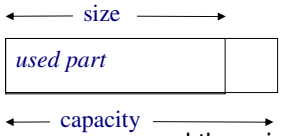
31

Operações do Contêiner Vetor

- Capacidade
  - capacity() — Tamanho de posições alocadas
  - reserve(n) — Reserva espaço para um total de n elemetos
- Tamanho
  - resize(n) — Altera o número de elementos
  - resize(n,val) — Idem, val define o valor de inicialização
- Alteração no final
  - push_back(val) — Adiciona ao final
  - pop_back() — Remove o último elemento
- Alteração na seqüência
  - insert(pos,val) — Insere cópia de val antes de pos
  - insert(pos,n,val) — Insere n cópias de val antes de pos
  - insert(pos,beg, end) — Insere os valores de [beg,end[antes de pos
  - erase(pos) — Apaga elemento em p

32

On STL vectors

- The *vector* template class represents a *resizable* (flexible) array
- *capacity* is the maximum number of elements it may get without a reallocation and copying elements (allocated by *reserve* `()`)
- *size* is the current number of elements actually stored in the vector (always less than or equal to the *capacity*)
- 

- when you *insert* a new element, and there is no more room, i.e., size already equals capacity, then the vector is reallocated
- insertions at the end of a vector are *amortized constant* time (while an individual insertion might be linear in the current size)
- on reallocation, any iterators or references are *invalidated*
- note that *overwriting* operations do not reallocate vectors, so the programmer must prevent any overflow/memory corruption

33

```
vector <int> v;
v.reserve (100);  // allocate space for 100 integers
                        // capacity = 100 , size = 0
int i;
while (cin >> i)      // read from the standard input
  v.push_back (i);        // will expand v if needed
for (i = 0; i < v.size (); ++i)
  cout << v[i] << " ";
try {                             // use checked access
  cout << v.at (100);                     // may throw
} catch (std::out_of_range&) {
  cout << "doesn't have 101 elements" << endl;
}
for (int i = 0; i < v.size () / 2; ++i)
   v.pop_back ();               // remove second half
vector <int> v1 (v);                     // copy to v1
v1.insert (v1.begin ()+1, 117);// insert after first
```

34

Contêiner Lista

- seqüência de elementos para acesso seqüêncial
  - ☺ Alterações na ordem dos elementos
  - ☺ Inserções em quaisquer pontos da seqüência
  - ☹ Acesso aleatório
- Cabeçalho
  - #include <list>
- Parâmetros
  - Tipo dos elementos armazenados
  - Tipo do gerenciador de memória (opcional)
- Exemplos
  - list<**int**> li;
  - list<MyClass> lobj;
  - list<vector<string> > lvet;

35

Operações do Contêiner Lista

- Construtores
  - list(n) — n posições com valor *default*
  - list(n,val) — n posições com valor val
- Atribuição
  - assign(beg, end) — Atribui um vetor com os valores de [beg,end[
  - assign(n,val) — Atribui um vetor com n cópias de val
- Acesso
  - front() — Primeiro elemento
  - back() — Último elemento

36

Operações do Contêiner Lista

- Alteração nas extremidades
  - push_back(val) Adiciona ao final
  - pop_back() Remove o último elemento
  - push_front(val) Adiciona um novo primeiro elemento
  - pop_front() Remove o primeiro elemento
- Alteração na seqüência
  - insert(pos,val) Insere cópia de val antes de pos
  - insert(pos,n,val) Insere n cópias de val antes de pos
  - insert(pos,beg, end) Insere os valores de [beg,end[antes de pos
  - erase(pos) Apaga elemento em p

37

Operações do Contêiner Lista

- Remoção
  - remove(val) Remove todos os elementos com valor val
  - remove_if(op) Remove os elementos que op(elem)==**true**
  - unique() Remove duplicatas
  - unique(op) Remove duplicatas que op(elem)==**true**
- Alterações sem cópia
  - splice(pos,list) incorpora todos os elementos de list para antes de pos
  - splice(pos,list,p) move *p de list para antes de pos
  - splice(pos,list,beg,end) move os elementos de list em [beg,end[... idem
  - sort() Ordena usando operador <
  - sort(op) Ordena usando comparação cmp
  - merge(list) Intercala com list mantendo a ordenação (<)
  - merge(list,op) Intercala com list mantendo a ordenação (op)
  - reverse() Inverte a ordem dos elementos

38

Operações do Contêiner Lista

fruit:
  apple pear
citrus:
  orange grapefruit lemon
---
list<string>: :*iterator p = find_ if*(*fruit.begin*()
  ,*fruit.end*() ,*initial*(´*p*´)) ;

fruit.splice(*p*,*citrus*,*citrus.begin*()) ;

39

Operações do Contêiner Lista

fruit:
  apple orange pear
citrus:
  grapefruit lemon
fruit.splice(*fruit.begin*() ,*citrus*) ;

  fruit:
    grapefruit lemon apple orange pear
  citrus:
    <empty>

40

Operações do Contêiner Lista

f1:
    apple quince pear
f2:
    lemon grapefruit orange lime

```
f1.sort() ;
f2.sort() ;
f1.merge(f2) ;
```

f1:
    apple grapefruit lemon lime orange pear quince
f2:
    <*empty*>

Operações do Contêiner Lista

f1:
    apple orange lemon orange lime pear quince pear
    fruit.remove("*orange*") ;
fruit:
    apple lemon lime pear quince pear
    fruit.remove_ if(*initial*(´*l*´)) ;
fruit:
    apple pear quince pear
    fruit.sort() ;
    fruit.unique() ; // Só remove elementos iguais consecutivos
fruit:
    apple pear quince

Operações do Contêiner Lista

fruit:

pear pear apple apple (Remover somente alguns itens duplicados)

fruit.unique(*initial2*(´*p*´)) ;

fruit:

pear apple apple

43

Operações do Contêiner Lista

fruit:

banana cherry lime strawberry

fruit.reverse();

fruit:

strawberry lime cherry banana

44

Contêiner Fila Dupla

- Seqüência de elementos para acesso nas extremidades
  - ☺ Acesso nas extremidades
  - ☹ Alterações no meio da seqüência
- Cabeçalho
  - #include <deque>
- Parâmetros
  - Tipo dos elementos armazenados
  - Tipo do gerenciador de memória (opcional)
- Exemplos
  - deque<**int**> dqi;
  - deque<MyClass> dqobj;
  - deque<vector<string> > dqvet;

45

Operações do Contêiner Fila Dupla

- *deques* are similar to *vectors*
- *deque* iterators are random access
- additionally two operations to insert/remove elements in front:
- `push_front ()` add new first element
- `pop_front ()` remove the first element
-
- *deques* do not have operations *capacity* ( ) and *reserve* ( )

46

Operações do Contêiner Fila Dupla

- Construtores
  - deque(n) — n posições com valor *default*
  - deque(n,val) — n posições com valor val
- Atribuição
  - assign(beg, end) — Atribui um vetor com os valores de [beg,end[
  - assign(n,val) — Atribui um vetor com n cópias de val
- Acesso
  - [pos] — Indexação sem verificação
  - at(pos) — Indexação com verificação
  - front() — Primeiro elemento
  - back() — Último elemento

47

Operações do Contêiner Fila Dupla

- Tamanho
  - resize(n) — Altera o número de elementos
  - resize(n,val) — Idem, val define o valor de inicialização
- Alteração nas extremindades
  - push_back(val) — Adiciona ao final
  - pop_back() — Remove o último elemento
  - push_front(val) — Adiciona um novo primeiro elemento
  - pop_front() — Remove o primeiro elemento
- Alteração na seqüência
  - insert(pos,val) — Insere cópia de val antes de pos
  - insert(pos,n,val) — Insere n cópias de val antes de pos
  - insert(pos,beg, end) — Insere os valores de [beg,end[antes de pos
  - erase(pos) — Apaga elemento em p

48

Contêiner Pilha

- Adaptador de contêiner seqüencial para acesso LIFO
  - Interface mais restritiva que oferece métodos para manipular uma pilha
- Cabeçalho
  - #include <stack>
- Parâmetros
  - Tipo dos elementos armazenados
  - Tipo dos contêiner seqüêncial utilizado (*default* deque<T>)
- Exemplos
  - stack<**int**> si;
  - stack<MyClass, vector<MyClass>> stkobj;
  - stack<int, vector<int>> stkobj;

49

Operações do Contêiner Pilha

```
template<class T, class C = deque<T> > class stack {
protected:
    C c;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;
    explicit stack(const C& a =C()) : c(a)  { }
    bool empty() const                      { return c.empty() ; }
    size_type size() const                  { return c.size() ; }
    value_type& top()                       { return c.back() ; }
    const value_type& top() const           { return c.back() ; }
    void push(const value_type& x)          { c.push_back(x) ; }
    void pop()                              { c.pop_back() ; }
};
```

➔ Não possui parâmetro para alocador de memória

50

Operações do Contêiner Pilha

```
void f()
{
        stack<int> s;
        s.push(2) ;
        if (s.empty()) { / / underflow is preventable
        / / don't pop
}
else{ / / but not impossible
        s.pop() ; / / fine: s.size() becomes 0
        s.pop() ; / / undefined effect, probably bad
        }
}

void f(stack<char>& s)
{
        if (s.top()==´c´) s.pop() ; / / remove optional initial 'c'
        / / ...
}
```

51

Contêiner Fila

- Adaptador de contêiner seqüencial para acesso FIFO
  - Interface para um container que permite a inserção de elementos no back() e a extração no front()
- Cabeçalho
  - #include <queue>
- Parâmetros
  - Tipo dos elementos armazenados
  - Tipo dos contêiner seqüêncial utilizado (*default* deque<T>)
- Exemplos
  - queue<**int**> qi;
  - queue<MyClass, dequeue<MyClass>> queobj;

52

Operações do Contêiner Fila

```
template<class T, class C = deque<T> > class queue {
protected:
    C c;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;
    explicit queue(const C& a =C()) : c(a)  { }
    bool empty() const                      { return c.empty() ; }
    size_type size() const                  { return c.size() ; }
    value_type& front()                     { return c.front() ; }
    const value_type& front() const         { return c.front() ; }
    value_type& back()                      { return c.back() ; }
    const value_type& back() const          { return c.back() ; }
    void push(const value_type& x)          { c.push_back(x) ; }
    void pop()                              { c.pop_front() ; }
};
```

53

Operações do Contêiner Fila

```
struct Message {
    / / ...
};
void server(queue<Message>& q)
{
        while(!q.empty()) {
                Message& m = q.front() ; / / get hold of message
                m.service() ; / / call function to serve request
                q.pop() ; / / destroy message
    }
}

void server2(queue<Message>& q, Lock& lck)
{
        while(!q.empty()) {
        Message m;
        { LockPtr h(lck) ; / / hold lock only while extracting message (see §14.4.7)
        if (q.empty()) return; / / somebody else got the message
        m = q.front() ;
        q.pop() ;
}
    m.service() ; / / call function to serve request
```

54

Contêiner Fila de Prioridades

- Adaptador de contêiner seqüêncial para acesso de elementos de maior prioridade
  - É uma queue na qual cada elemento recebe uma prioridade que controla a ordem nas quais os elementos atingem o top()
- Cabeçalho
  - #include <queue>
- Parâmetros
  - Tipo dos elementos armazenados
  - Tipo dos contêiner seqüêncial utilizado (*default* deque<T>)
  - Comparador de elementos que define relação de prioridades
- Exemplos
  - priority_queue<**int**> qi;
  - qi.push(2);
  - qi.push(4);
  - cout << qi.pop();

55

Operações da Fila de Prioridades

```
template <class T, class C = vector<T>, class Cmp = less<typename C::value_type> >
class priority_queue {
protected:
    C c;
    Cmp cmp;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;
    explicit priority_queue(const Cmp& a1 =Cmp() , const C& a2 =C())
            : c(a2) , cmp(a1) { }
    template <class In>
    priority_queue(In first, In last, const Cmp& =Cmp() , const C& =C()) ;
    bool empty() const { return c.empty() ; }
    size_type size() const { return c.size() ; }
    const value_type& top() const { return c.front() ; }
    void push(const value_type&) ;
    void pop() ;
};
```

56

Operações da Fila de Prioridades

```
struct Message {
int priority;
bool operator<(const Message& x) const { return priority < x.priority; }
// ...
};
priority_ queue<Message>& q;
```

Outras formas:

```
priority_ queue<string,Nocase> pq; // use Nocase::operator()() for
    comparisons

int myints[]= {10,60,50,20};
priority_queue< int, vector<int>, greater<int> > third (myints);
```

57

Container MAP

```
template <class Key, class T, class Cmp = less<Key>,
class A = allocator< pair<const Key,T> > >
class std::map {
public:
    // types:
    typedef Key key_type;
    typedef T mapped_type;
    typedef pair<const Key, T> value_type;
    typedef Cmp key_compare;
    typedef A allocator_type;
    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    typedef implementation_defined1 iterator;
    typedef implementation_defined2 const_iterator;
    typedef typename A::size_type size_type;
    typedef typename A::difference_type difference_type;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    // ...
};
```

58

Iteradores e Pares

- A iteração sobre MAP é uma iteração sobre uma sequência de elementos pair<const key,mapped_type>

```
void f(map<string,number>& phone_ book)
{
    typedef map<string,number>: :const_ iterator CI;
    for (CI p = phone_ book.begin() ; p!=phone_ book.end() ; ++p)
    cout << p->first << ´\t´ << p->second << ´\n´;
}
```

- Apresenta os elementos em ordem crescente de suas chaves

59

Iteradores e Pares

```
template <class T1, class T2> struct std::pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
    pair() : first(T1()) , second(T2()) { }
    pair(const T1& x, const T2& y) : first(x) , second(y) { }
    template <class U, class V>
        pair(const pair<U,V>& p) : first(p.first) , second(p.second) { }
};
```

- Conversão na inicialização:

```
pair<int,double> f(char c, int i)
{
return pair<int,double>(c,i) ; / / conversions required
}
```

60

Contêiners Mapa e Multimapa

- Conjunto de pares (chave,valor) para acesso baseado na chave
- Cabeçalho
  - #include <map>
- Parâmetros
  - Tipo dos elementos da chave
  - Tipo dos elementos armazenados
  - Tipo do comparador dos elementos (opcional)
  - Tipo do gerenciador de memória (opcional)

61

Contêiners Mapa e Multimapa

```
template <class Key, class T, class Cmp = less<Key>,
class A = allocator< pair<const Key,T> > >
class map {
public:
// ...
mapped_ type& operator[](const key_ type& k) ; // access element with key k
// ...
};
```

- Faz busca baseado na chave. Se a chave não for encontrada, um elemento com a chave e o valor default de mapped_type é inserido no map.
- Exemplo:

```
void f()
{
    map<string,int> m; // map starting out empty
    int x = m["Henry"] ; // create new entry for "Henry", initialize to 0, return 0
    m["Harry"] = 7; // create new entry for "Harry", initialize to 0, and assign 7
    int y = m["Henry"] ; // return the value from "Henry"'s entry
    m["Harry"] = 9; // change the value from "Harry"'s entry to 9
}
```

62

Contêiners Mapa e Multimapa

- Exemplos
  - map<string,**int**> m;
  - ++m["int"];
  - cout << m["int"];
- Valor da saida
  - ?

63

Contêiners Mapa e Multimapa

nail 100 hammer 2 saw 3 saw 4 hammer 7 nail 1000 nail 250

```
void readitems(map<string,int>& m)
{
    string word;
    int val = 0;
    while (cin >> word >> val) m[word] += val;
}
```

64

Contêiners Mapa e Multimapa

```
int main()
{
        map<string,int> tbl;
        readitems(tbl) ;
        int total = 0;
        typedef map<string,int>: :const_  iterator CI;
        for (CI p = tbl.begin() ; p!=tbl.end() ; ++p) {
                total += p>
                second;
                cout << p->first << ´\t´ << p->second << ´\n´;
        }
        cout << "\ntotal\t" << total << ´\n´;
        return !cin;
}
```

```
hammer 9
nail 1350
saw 7
total 1366
```


Contêiners Mapa e Multimapa

```
map<Key,T>::iterator it;
(*it).first;            // the key value (of type Key)
(*it).second;           // the mapped value (of type T)
(*it);                  // the "element value" (of type pair<const Key,T>)
```

OU

```
it->first;              // same as (*it).first   (the key value)
it->second;             // same as (*it).second  (the mapped value)
```

Contêiners Mapa e Multimapa

```
#include <iostream>
#include <map>
using namespace std;

bool fncomp (char lhs, char rhs) {return lhs<rhs;}

struct classcomp {
  bool operator() (const char& lhs, const char& rhs) const
  {return lhs<rhs;}
};

int main ()
{
  map<char,int> first;
  first['a']=10;   first['b']=30;   first['c']=50;   first['d']=70;
  map<char,int> second (first.begin(),first.end());
  map<char,int> third (second);
  map<char,int,classcomp> fourth;                 // class as Compare

  bool(*fn_pt)(char,char) = fncomp;
  map<char,int,bool(*)(char,char)> fifth (fn_pt); // function pointer as Compare
  return 0;
}
```

67

Contêiners Mapa e Multimapa

void f(map<string,int>& m)

{

map<string,int> mm; / / compare using < by default

map<string,int> mmm(m.key_ comp()) ; / /
compare the way m does

/ / ...

}

68

Operações do Cont. (Multi)Mapa

- Construtores
  - *contêiner*(op) — vazio, usando op como comparação op
  - *contêiner*(beg,end,op) — Os pares em [beg,end], usando op como comparação op
- Acesso
  - [k] — Apenas para o contêiner mapa. Devolve uma referência para o valor mapeado por k. Se não houver o elemento este é inserido com o valor *default*
- Mapeamento
  - count(k) — Número de elementos com a chave k
  - find(k) — Devolve a primeira posição com a chave k ou end()
  - lower_bound(k) — Encontra o primeiro elemento com chave K
  - upper_bound(k) — Encontra o primeiro elemento com chave maior do que K
  - equal_range(k) — Devolve a primera e a última posição onde um elemento com a chave k seria inserido

69

Operações do Cont. (Multi)Mapa

```
void f(map<string,int>& m)
{
    map<string,int>: :iterator p = m.find("Gold") ;
    if (p!=m.end()) { / / if "Gold" was found
    / / ...
    }
    else if (m.find("Silver")!=m.end()) { / / look for "Silver"
    / / ...
    }
    / / ...
}
```

70

Operações do Cont. (Multi)Mapa

```
void f(multimap<string,int>& m)
{
    multimap<string,int>: :iterator lb = m.lower_
       bound("Gold") ;
    multimap<string,int>: :iterator ub = m.upper_
       bound("Gold") ;
    for (multimap<string,int>: :iterator p = lb; p!=ub; ++p) {
    / / ...
    }
}
```

71

Operações do Cont. (Multi)Mapa

```
// map::equal_elements
#include <iostream>
#include <map>
using namespace std;

int main ()
{
  map<char,int> mymap;
  pair<map<char,int>::iterator,map<char,int>::iterator> ret;

  mymap['a']=10;
  mymap['b']=20;
  mymap['c']=30;

  ret = mymap.equal_range('b');

  cout << "lower bound points to: ";
  cout << ret.first->first << " => " << ret.first->second << endl;

  cout << "upper bound points to: ";
  cout << ret.second->first << " => " << ret.second->second << endl;

  return 0;
}

lower bound points to: 'b' => 20
upper bound points to: 'c' => 30
```

72

Operações do Cont. (Multi)Mapa

- Atribuições

```
phone_ book["Order department"] = 8226339;

template <class Key, class T, class Cmp = less<Key>,
class A = allocator< pair<const Key,T> > >
class map {
public:
// ...
// list operations:
pair<iterator, bool> insert(const value_ type& val) ; // insert (key,value) pair
iterator insert(iterator pos, const value_ type& val) ; // pos is just a hint
template <class In> void insert(In first, In last) ; // insert elements from sequence
void erase(iterator pos) ; // erase the element pointed to
size_ type erase(const key_ type& k) ; // erase element with key k (if present)
void erase(iterator first, iterator last) ; // erase range
void clear() ;
// ...
};
```

73

Operações do Cont. (Multi)Mapa

- Atribuições

```
void f(map<string,int>& m)
{
    pair<string,int> p99("Paul",99) ;
    pair<map<string,int>: :iterator,bool> p = m.insert(p99) ;
    if (p.second) {
        // "Paul" was inserted
    }
    else {
        // "Paul" was there already
    }

    map<string,int>: :iterator i = p.first; // points to m["Paul"]
    // ...
}
```

74

Operações do Cont. (Multi)Mapa

- Remoção

```
void g(map<string,int>& m)
{
    m.erase(m.find("Catbert")) ;
    m.erase(m.find("Alice") ,m.find("Wally")) ;
}
```

75

Multimap

- *É como um map, exceto pelo fato de que ele permite chaves duplicadas*
- *Acesso a múltiplos valores com a mesma chave*
  - *lower_bound, upper_bound, equal_range*
  - *Encontrar o limite superior e inferior usando duas operações não é elegante*

```
void f(multimap<string,int>& m)
{
     typedef multimap<string,int>: :iterator MI;
     pair<MI,MI> g = m.equal_  range("Gold") ;
     for (MI p = g.first; p!=g.second; ++p) {
     / / ...
     }
}
```

76

Multimap

```
void print_ numbers(const
  multimap<string,int>& phone_ book)
{
   typedef multimap<string,int>: :const_ iterator I;
   pair<I,I> b = phone_
     book.equal_range("Stroustrup") ;
   for (I i = b.first; i != b.first; ++i) cout <<
   i->second << ´\n´;
}
```

77

Contêiners Conjunto e Multiconjunto

- Conjunto de valores para verificação de pertinência
- **map** no qual os valores são irrelevantes
- Cabeçalho
  - #include <set>
- Parâmetros
  - Tipo dos elementos armazenados
  - Tipo do comparador dos elementos (opcional)
  - Tipo do gerenciador de memória (opcional)
- Exemplos
  - set<**int**> conj;
  - conj.insert(4);
  - conj.insert(1);
  - if (conj.find(2) == conj.end()) /* ... */ ;

78

Operações do Cont.(Multi)Conjunto

- Construtores
  - *contêiner*(op) vazio, usando op como comparação op
  - *contêiner*(beg,end,op) Os pares em [beg,end[, usando op como comparação op
- Mapeamento
  - count(e) Número de elementos com o valor e
  - find(e) Devolve a primeira posição com o valor e ou end()
  - lower_bound(e) Devolve a primeira onde um elemento com o valor e k seria inserido
  - upper_bound(e) Devolve a última posição onde um elemento com o valor e seria inserido
  - equal_range(e) Devolve a primera e a última posição onde um elemento com o valor e seria inserido

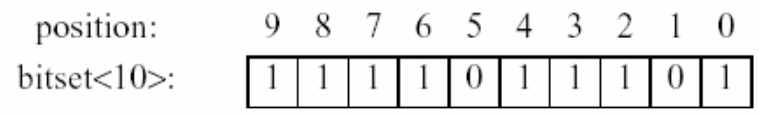
79

Contêiner Conjunto de Bits

- Conjunto de valores para verificação de pertinência
- Não é possível endereçar um bit diretamente usando ponteiro primitivo
- Cabeçalho
  - #include <bitset>
- Parâmetros
  - Número de bits
- Exemplos
  - **enum** RGB { red, green, blue, count }
  - bitset<count> bs;
  - bs.set(red);
  - bs.set(green);

80

Contêiner Conjunto de Bits



- Valor padrão é 0

81

Operações do Conjunto de Bits

- Construtores
  - bitset(long) Ativa os bits do inteiro longo long
  - bitset(str) Ativa os bits correspondentes aos caracteres '1' e da string str
  - bitset(str,i) Ativa os bits correspondentes aos caracteres nas posições após i da string str que tenham valor '1'
  - bitset(str,i,j) Ativa os bits correspondentes aos caracteres nas posições em [i,j] da string str que tenham valor '1'
- Acesso
  - size() Devolve o número de bits
  - count() Devolve o número de bits ativos
  - any() Informa se algum bit está ativo
  - none() Informa se nenhum bit está ativo
  - test(idx) Informa se o bit de indice idx está ativo

82

Operações do Conjunto de Bits

```
void f()
{
    bitset<10> b1; / / all 0
    bitset<16> b2 = 0xaaaa; / / 1010101010101010
    bitset<32>b3 =0xaaaa; / / 00000000000000001010101010101010
    bitset<10>b4("1010101010") ; / / 1010101010
    bitset<10>b5("10110111011110",4) ; / / 0111011110
    bitset<10> b6("10110111011110",2,8) ; / / 0011011101
    bitset<10> b7("n0g00d") ; / / invalid_argument thrown
    bitset<10> b8 = "n0g00d"; / / error: no char* to bitset conversion
}
```


Operações do Conjunto de Bits

- Operadores (mesmo significado quando aplicado a inteiros)
  - Relacionais
    - == !=
  - Boleanos
    - & | ^ << >> ~
  - Atribuição
    - &= |= ^= <<= >>=
- [idx] Acessa o bit de índice idx

Operações do Conjunto de Bits

- Manipulação
  - set() Ativa todos os bits
  - set(idx) Ativa o bit de indice idx
  - set(idx, val) Ativa o bit de indice idx de acondo com val
  - reset() Desativa todos os bits
  - reset(idx) Desativa o bit de indice idx
  - reset(idx, val) Desativa o bit de indice idx de acondo com val
  - flip() Alterna todos os bits
  - flip(idx) Alterna o bit de indice idx
- Conversão
  - to_ulong() Retorna um ulong com o mesmo padrão de bits
  - to_string() Devolve uma string com o binário correspondente

85

Operações do Conjunto de Bits

```
// bitset::set
#include <iostream>
#include <bitset>
using namespace std;

int main ()
{
        bitset<4> mybits;

        cout << mybits.set() << endl;       // 1111
        cout << mybits.set(2,0) << endl;    // 1011
        cout << mybits.set(2) << endl;      // 1111

        return 0;
}
```

86

Operações do Conjunto de Bits

```
// bitset::flip
#include <iostream>
#include <string>
#include <bitset>
using namespace std;

int main ()
{
        bitset<4> mybits (string("0001"));

        cout << mybits.flip(2) << endl;     // 0101
        cout << mybits.flip() << endl;      // 1010

        return 0;
}
```

87

Operações do Conjunto de Bits

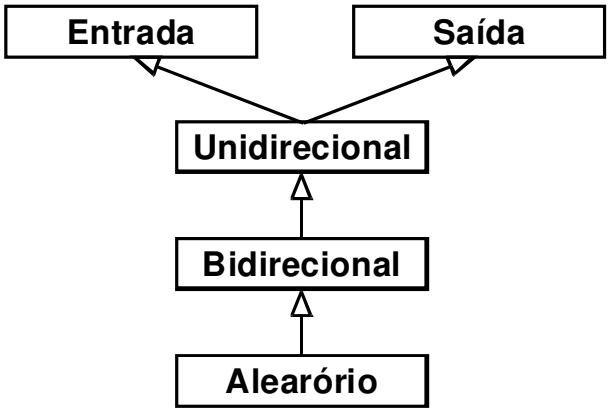
```
// bitset::to_string
#include <iostream>
#include <string>
#include <bitset>
using namespace std;

int main ()
{

  string mystring;
  bitset<4> mybits;     // mybits: 0000

  mybits.set();         // mybits: 1111

  mystring=mybits.to_string<char,char_traits<char>,allocator<char> >();

  cout << "mystring: " << mystring << endl;

  return 0;
}
```

88

Iteradores

89

Categorias



90

Operações

Categ.	**Saíd.**	**Entr.**	**Uni.**	**Bi.**	**Aleatório**
Leitura		=*p	=*p	=*p	=*p
Acess.		->	->	->	-> [ ]
Escrita	*p=		*p=	*p=	*p=
Iteraç.	++	++	++ --	++ --	++ -- + - += -=
Comp.		== !=	== !=	== !=	== != < > >= <=

91

Iteradores de Entrada

- Uso
  - Acesso somente de leitura e unidirecional
- Operações
  - construtor de cópia
  - *i Leitura do elemento apontado
  - i->*membro* Leitura de um membro do elemento apontado
  - ++i Passo a frente (retorna a nova posição)
  - i++ Passo a frente (retorna a posição antiga)
  - i1 == i2 Testa se os iteradores apontam pro mesmo lugar
  - i1 != i2 Testa se os iteradores não apontam pro mesmo lugar

92

Iteradores de Saída

- Uso
  - Acesso somente de escrita e unidirecional
- Operações
  - construtor de cópia
  - *i Escrita no elemento apontado
  - ++i Passo a frente (retorna a nova posição)
  - i++ Passo a frente (retorna a posição antiga)

93

Iteradores Unidirecionais

- Uso
  - Iteração unidirecional
- Operações
  - construtor de cópia e *default*
  - *i Acesso do elemento apontado
  - i->*membro* Acesso a um membro do elemento apontado
  - ++i Passo a frente (retorna a nova posição)
  - i++ Passo a frente (retorna a posição antiga)
  - i1 == i2 Testa se os iteradores apontam pro mesmo lugar
  - i1 != i2 Testa se os iteradores não apontam pro mesmo lugar
  - i1 = i2 Atribuição

94

Iteradores Bidirecionais

- Uso
  - Iteração bidirecional
- Operações
  - Mesmas operações de iterad. unidirecionais
  - --i Passo a trás (retorna a nova posição)
  - i-- Passo a trás (retorna a posição antiga)

95

Iteradores de Acesso Aleatório

- Uso
  - Iteração de acesso aleatório
- Operações
  - Mesmas operações de iterad. bidirecionais
  - i[off] Acesso ao n-ésimo elemento a partir da posição apontada
  - i+=n Salta n elementos para frente
  - i-=n Salta n elementos para trás
  - i+n Iterador apontando para n posições para frente
  - i-n Iterador apontando para n posições atrás
  - i1-i2 Número de elementos entre iteradores
  - i1<i2 Se i1 está antes de i2
  - i1>i2 Se i1 está depois de i2
  - i1<=i2 Se i1 está antes ou na mesma posição de i2
  - i1>=i2 Se i1 está depois na mesma posição de i2

96

Funções Auxiliares

- advance(pos, n) — Avança o iterador n posições
- distance(i1, i2) — Número de elementos entre os iteradores i1 e i2
- iter_swap(i1, i2) — Troca os valores dos elementos apontados por i1 e i2

97

Adaptadores de Iteradores

- Iteradores reversos
  - Permitem iterar na seqüência na ordem inversa
- Iteradores de inserção (insersores)
  - Permitem que a atribuição do valor apontado resulte na inserção de um novo elemento naquela posição
  - Operações para obtenção de insersores

```
inserter = back_inserter(contêiner);
inserter = front_inserter(contêiner);
inserter = inserter(contêiner, out_iter);
```

98

Algoritmos

99

Visão Geral

- Cabeçalho
  - #include <algorithm>
- Categorias
  - Sem modificação
  - Com modificação
  - De remoção
  - De mutação
  - De ordenação
  - De Intervalo Ordenado

100

Busca

- Contar o número de elementos com um valor
  - count, count_if
- Encontrar o menor ou o maior valor
  - min_element, max_element
- Buscar um elemento com um valor
  - find, find_if
- Buscar n ocorrências seqüênciais de um valor
  - search_n
- Buscar uma subseqüência
  - search
- Buscar última ocorrência de uma subseqüência
  - find_end
- Buscar um os elementos de uma outra seqüência
  - find_first_of
- Buscar dois elementos adjacentes seguindo um critério
  - adjacent_find

101

Comparações

- Testar se duas seqüências são iguais
  - equal
- Buscar a primeira diferença em duas seqüências
  - mismatch
- Comparação lexicográfica
  - lexicographical_compare

102

Alterações

- Executar operação sobre os elementos
  - for_each
- Copiar uma subseqüência
  - copy, copy_backward
- Transforma os valores e armazena em outra
  - transform
- Intercala ou troca valores entre duas seqüências
  - merge, swap_ranges
- Atribui valores os elementos
  - fill, fill_n, generate, gerenate_n
- Substitui as ocorrências de um valor
  - replace, replace_if, replace_copy, replace_copy_if

Remoções

- Remover elementos de uma sequência
  - remove
  - remove_if
  - remove_copy
  - remove_copy_if
- Remover repetições
  - unique
  - unique_copy

Reordenação

- Inverter a ordem dos elementos
  - reverse, reverse_copy
- Rotacionar os elementos na seqüência
  - rotate, rotate_copy
- Permutar os elementos
  - next_permutation, prev_permutation
- Aleatorizar a ordem dos elementos
  - ramdom_shuffle
- Colocar no início os elementos que seguem um critério
  - partition, stable_partition

105

Ordenação

- Ordenar os elementos
  - sort
  - stable_sort
  - partial_sort
  - partial_sort_copy
  - nth_element
- Manipulação de heaps
  - make_heap
  - push_heap
  - pop_heap
  - sort_heap

106

Sobre Seqüências Ordenadas

- Buscas
  - binary_search, includes, lower_bound, upper_bound, equal_range
- Combinação
  - merge, set_union, set_intersection, set_difference, set_symetric_difference, inplace_merge

107

Sem Modificação

- count(i,j,val) Conta elementos com o valor val
- count_if(op) Conta elementos em que op(elem) == true
- min_element(i,j) Elemento de menor valor (operador <)
- min_element(i,j,cmp) Elemento de menor valor com (comparação cmp)
- max_element(i,j) Elemento de maior valor
- max_element(i,j,cmp) Elemento de maior valor com (comparação cmp)
- find(i,j,val) Primeiro elemento com valor val
- find_if(i,j,op) Primeiro elemento que op(elem)==true
- search_n(i,j,n,val) Primeiro de n elementos consecutivos de valor val
- search_n(i,j,n,val,op) Primeiro de n elementos consecutivos tal que op(elem)==true
- search(i,j,beg,end) Primeira subseqüência [beg,end[em [i,j[
- search(i,j,beg,end,op) Primeira subseqüência [beg,end[em [i,j[tal que op(elm, proc)
- find_end(i,j,beg,end) Última subseqüência [beg,end[em [i,j[
- find_end(i,j,beg,end,op) Última subseqüência [beg,end[em [i,j[tal que op(elm, proc)

108

Objetos Função

Predicados

- equal_to<*type*>() ==
- not_equal_to<*type*>() !=
- less<*type*>() <
- greater<*type*>() >
- less_equal<*type*>() <=
- greater_equal<*type*>() >=
- logical_not<*type*>() !
- logical_and<*type*>() &&
- logical_or<*type*>() ||

Objetos Função Aritméticos

- negate<*type*>() - (unário)
- plus<*type*>() +
- minus<*type*>() -
- multiplies<*type*>() *
- divides<*type*>() /
- modulus<*type*>() %

111

Adaptadores de Função

- bind1st(f,val) Associa val ao primeiro parâmetro da função
  - a = bind1st(f,x);
  - a(y); // f(x,y);
- bind2nd(f,val) Associa val ao segundo parâmetro da função
  - a = bind2nd(f,x);
  - a(y); // f(y,x);
- not1(f) Nega uma operação unária
  - a = not1(f)
  - a(f) // !f(a)
- not2(f) Nega uma operação binária
  - a = not2(f)
  - a(x,y) // !f(x,y)

112

Adaptadores de Função

- ptr_fun(op) Cria objeto função que executa a função
  - a = ptr_fun(pfunc);
  - a(obj); // pfunc(obj);
- mem_func_ref(op) Cria objeto função que executa uma função membro do seu parâmetro
  - a = mem_func_ref(pmemb);
  - a(obj); // obj.*pmemb();
- mem_func(op) Cria objeto função que executa uma função membro do seu parâmetro
  - a = mem_func(pmemb);
  - a(obj); // obj->*pmemb();

113

Strings

114

Template basic_string

- Similar ao contêiner vector
- Otimizado para cadeias de caracteres
- Apresenta apenas operações típicas de strings
- Declaração:
  - **template**<**class** charT, /* ... */> **class** basic_string;
- Usos do template:
  - **typedef** basic_string<**char**> string;
  - **typedef** basic_string<wchar_t> wstring;

115

Tipos

- Tipos parametrizados
  - value_type — Tipo do caracter
  - allocator_type — Tipo do alocador usado
- Tipos de iteradores
  - iterator — Tipo do iterador
  - const_iterator — Tipo do iterador constante
  - reverse_iterator — Tipo do iterador reverso
  - const_reverse_iterator — Tipo do iterador reverso constante
- Tipos ponteiro e referência
  - pointer — Tipo ponteiro para o caracter
  - const_pointer — Tipo ponteiro constante para o caracter
  - reference — Tipo referência para o caracter
  - const_reference — Tipo referência constante para o caracter

116

Construtores e Destrutores

- string() String vazia
- string(str) Cópia da string str
- string(str,idx) Cópia da substring de str que inicia no índice idx
- string(str,idx,len) Cópia da substring de str que inicia no índice idx e comprimento len
- string(cstr) String a partir de uma string C
- string(chars,len) String com os len primeiros caracteres do vetor chars
- string(num,c) String com num repetições do caracter c
- string(beg,end) String os os caracteres em [beg,end[
- ~string() Destrói todos os caracteres e libera a memória

117

Conversão para Strings C

- data() Retorna um vetor com os caracteres da string (sem o '\0')
- c_str() Retorna uma string C com os mesmos caracteres (i.e. com o '\0')
- copy(buff) Copia os caracteres da string para um vetor fornecido

118

Comprimento e capacidade

- size(), length() Número de caracteres
- empty() Verifica se é a string vazia
- max_size() Número máximo de caracteres
- capacity() Número de caracteres que pode comportar sem realocar memória

```
string s("Maia");
cout << s.length();
```

119

Acesso aos Caracteres

- [idx] idx-ésimo caracter da string
- at(idx) idem, mas com verificação

```
string s("Marcelo");
s.at(6) = s[1];    // s == " Marcela"

// Cuidado
char& r = s[5];
s = "Renato";      // realocação libera vetor original
r = 'a';           // cuidado! r foi desalocado
```

120

Comparações

- s1 == s2 Conteúdo de s1 é o mesmo de s2
- s1 < s2 s1 é lexicograf. menor que s2
- s1 > s2 s1 é lexicograf. maior que s2
- s1 <= s2 s1 é lexicograf. menor ou igual que s2
- s1 >= s2 s1 é lexicograf. maior ou igual que s2
- s1.compare(s2) como o strcmp(s1,s2) para strings C

  - string s1("abcd");
  - if (s1 == "abcd") cout << s1.compare("dcba"); // algum valor > 0

121

Atribuições

```
const string aString("Othelo");
string s;

s = aString;        // atribui "Othelo"
s = "two\nlines"; // atribui conteúdo de string C
s = ' ';            // atribui um único caracter

s.assign(aString);
s.assign(aString,1,3);
s.assign("two\nlines");
```

122

Apagando

```
string s = "Meu texto";

s = "";
s.clear();
s.erase();
```

123

Apendando

```
const string aString("othello");
string s;

a += aString;              // apenda "othello"
a += "world";              // apenda string C
a += '\n';                 // apenda único caracter

a.append(aString,1,3);     // apenda "the"
a.append("nico", 3);       // apenda "nic"
a.append('x',5);           // apenda "xxxxx"
```

124

Alterando

```
const string aString("ente");
string s("g");
s.insert(1,aString);              // "gente"
s.insert(1,"er");                 // "gerente"
s.replace("ger", "graficam"); // graficamente
s.erase(7,12);                    // grafica
```

125

Subcadeias e Concatenação

```
string s("interoperabilidade");

s.substring(5,5); // retorna string("opera")

cout << "in" + s.substring(5,5) + "ção";
```

126

Busca

- Funções sobrecarregadas para receber strings, strings C, strings e um intervalo defindindo uma subcadeia, etc.

find(val)	Primeira ocorrência de val
rfind(val)	Última ocorrência de val
find_first_of(val)	Primeiro caracter de val
find_last_of(val)	Último caracter de val
find_first_not_of(val)	Primeiro caracter que não está em val
find_last_not_of(val)	Último caracter que não está em val

127

Streams

128

Motivação

- Canal para transferência de dados como bytes (caracteres)
- Permite a conversão dos tipos de dados para bytes. Uma espécie de serialização

129

Streams de Saída

- Template Básico
  - **typedef** basic_ostream<char> ostream;
  - **typedef** basic_ostream<wchar_t> wostream;
- Funções
  - put(c) // escreve caracter c
  - write(cstr, sz) // escreve sz caracteres da string C cstr;
- Operadores
  - << // serializa valor e escreve no stream
- Exemplos
  - cout << "local " << &p << ", free store " << p << '\n';
  - // local 0x7fffead0, free store 0x500c

130

Streams de Entrada

- Template Básico
  - **typedef** basic_istream<char> istream;
  - **typedef** basic_istream<wchar_t> wistream;
- Funções
  - get(c) // lê um caracter e armazena em c
  - read(cstr, sz) // lê sz caracteres para o buffer cstr de sz chars;
- Operadores
  - >> // lê o valor do stream e escreve na variável
- Exemplos
  - int prioridade; string comando; string expressao;
  - cout >> prioridade >> comando >> expressao;

131

Leitura não Formatada

```
streamsize gcount() const;              // number of char read by last get()
int_type get();                         // read one Ch (or Tr::eof())
basic_istream& get(Ch& c);              // read one Ch into c
basic_istream& get(Ch* p, streamsize n);          // newline is terminator
basic_istream& get(Ch* p, streamsize n,Ch term);
basic_istream& getline(Ch* p, streamsize n);      // newline is terminator
basic_istream& getline(Ch* p, streamsize n,Ch term);
basic_istream& ignore(streamsize n = 1, int_type t = Tr::eof()) ;
basic_istream& read(Ch* p, streamsize n);         // read at most n char
```

132

Estado do Stream

```
bool good() const;                              // next operation will succeed
bool eof() const;                               // end of input seen
bool fail() const;                              // next operation will fail
bool bad() const;                               // stream is corrupted
iostate rdstate() const;                        // get io state flags
void clear(iostate f = goodbit) ;               // set io state flags
void setstate(iostate f) {clear(rdstate()|f);} // add f to io state flags
operator void*() const;                         // nonzero if !fail()
bool operator!() const { return fail() ; }
```

133

Exceções

- Por *default* os streams não lançam exceções
- Para lançar exceções em situações de erro usa-se a operação exception

```
cout.exceptions(ios_base::badbit|ios_base::failbit|ios_base::eofbit) ;
cin.exceptions(ios_base::badbit|ios_base::failbit)
```

134

Formatação

```
cout << "default:\t" << 1234.56789 << ´\n´;
// formato científico
cout.setf(ios_base::scientific,ios_base::floatfield);
cout << "scientific:\t" << 1234.56789 << ´\n´;
// formato de ponto fixo
cout.setf(ios_base::fixed,ios_base::floatfield);
cout << "fixed:\t" << 1234.56789 << ´\n´;
// volta ao default (ou seja, format genérico)
cout.setf(0,ios_base::floatfield);
cout << "default:\t" << 1234.56789 << ´\n´;
```

135

Manipuladores

- flush
  - cout << x << flush << y << flush;
- endl
  - cout << "Hello, World!" << endl;
- noskipwsnão ignora os espaços na entrada
  - cin >> noskipws >> x;

136

Formação Básica

- boolalpha // symbolic representation of true and // false (input and output)
- noboolalpha // s.unsetf(ios_base::boolalpha)
- showbase // on output prefix oct by 0 and hex by 0x
- noshowbase // s.unsetf(ios_base::showbase)
- showpoint // print trailing zeros
- noshowpoint // s.unsetf(ios_base::showpoint)
- showpos // explicit '+' for positive numbers
- noshowpos // s.unsetf(ios_base::showpos)

137

Formatação Textual

- skipws // skip whitespace
- noskipws // s.unsetf(ios_base::skipws)
- uppercase // X and E rather than x and e
- nouppercase // x and e rather than X and E

138

Formação Numérica

- dec // integer base is 10
- hex // integer base is 16
- oct // integer base is 8
- fixed // floatingpoint format dddd.dd
- scientific // scientific format d.ddddEdd

139

Constantes

- endl // put '\n' and flush
- ends // put '\0' and flush
- flush // flush stream
- ws // eat whitespace (input)

140

Com Parâmetro

- clear flags
  - resetiosflags(ios_base::fmtflags f);
- set flags
  - setiosflags(ios_base::fmtflags f);
- output integers in base b
  - setbase(int b);
- make c the fill character
  - setfill(int c);
- n digits after decimal point
  - setprecision(int n);
- next field is n char
  - setw(int n);

Streams de Arquivos

- Criados a partir de templates
  - typedef basic_ifstream<char> ifstream;
  - typedef basic_ofstream<char> ofstream;
  - typedef basic_fstream<char> fstream;
  - typedef basic_ifstream<wchar_t> wifstream;
  - typedef basic_ofstream<wchar_t> wofstream;
  - typedef basic_fstream<wchar_t> wfstream;

Modo de Abertura de Arquivos

```
namespace io_base {
    app,        // append
    ate,        // open and seek to end of file
    binary,     // I/O to be done in binary mode in, open for reading
    out,        // open for writing
    trunc;      // truncate file to 0 length
}

ofstream mystream(name.c_str() ,ios_base::app);
fstream dictionary("concordance",ios_base::in|ios_base::out);

mystream.close(); dictionary.close();
```

143

Streams de Strings

- Principais usos

```
typedef basic_istringstream<char> istringstream;
typedef basic_ostringstream<char> ostringstream;
typedef basic_stringstream<char> stringstream;

typedef basic_istringstream<wchar_t> wistringstream;
typedef basic_ostringstream<wchar_t> wostringstream;
typedef basic_stringstream<wchar_t> wstringstream;
```

144

Escrevendo numa String

```
extern const char* std_message[] ;
string compose(int n, const string& cs)
{
   ostringstream ost;
   ost << "error(" << n << ") " << std_message[n]
        << " (user comment: " << cs << ´)´;
   return ost.str() ;
}
```

145

Revisão

146

Visão Geral do Curso

- Extensão através de novos tipos
  - Classes, construtores, cópia, operadores
- Classes abstratas como interfaces
  - Funções membro virtuais, polimorfismo de inclusão
- Hierarquia de classes para POO
  - Herança, controle de acesso
- Software genérico fortemente tipado
  - Templates de classe, templates de funções
- Tratamento de erros regulares
  - Lançamento e captura de exceções
- Modularização de software em larga escala
  - Espaços de nomes, cabeçalhos, compilação por partes
- Contêiners padrão e algoritmos
  - Standard Template Library
- Strings padrão e streams de E/S
  - Operações com strings, streams

147

Fim

148