Programare II
Programare Orientată Obiect

Curs 11

Curs anterior

- Stanard Template Library
  - Introducere în Stanard Template Library
  - Containere secvenţiale
  - Containere asociative

Curs Curent

- Stanard Template Library
  - Containere asociative
  - Algoritmi
  - Clasa string

Standard Template Library - STL

- STL, biblioteca standard a limbajului C++ oferă cele mai uzuale structuri de date şi algoritmi fundamentali pentru utilizarea lor
- Prima bibliotecă generică a C++ folosită pe scară largă
- Conţinutul biblioteci STL
  - Containere
  - Iteratori
  - Algoritmi
- Performanţă
  - Măsurată prin benckmarks de penalitate

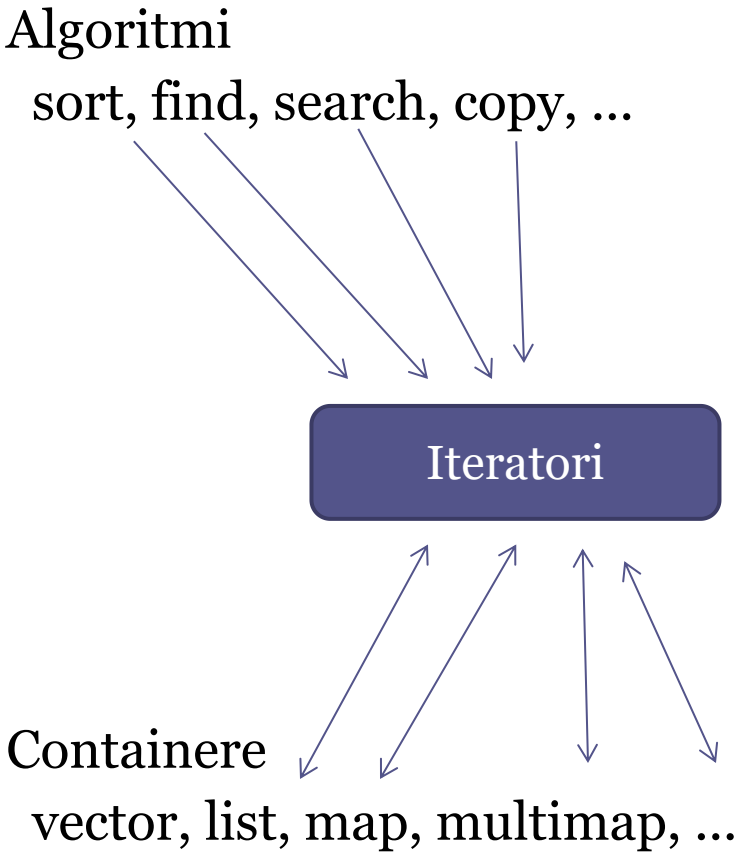
Componente STL

- Containăre
- Iteatori
- Algoritmi

Containere

- Containere secvenţiale
  - vector
  - deque
  - list
- Containere asociative
  - set
  - multiset
  - map
  - multimap
- Adaptori ai containerelor
  - stack
  - queue
  - priority_queue
- Containăre near
  - string
  - bitset
  - valaray

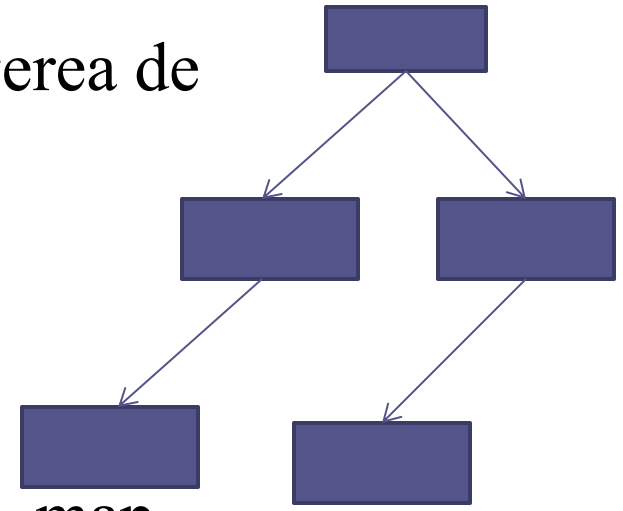
Modelul



- Separarea conceptelor
  - Algoritmi manipulează datele, dar nu ştiu despre containere
  - Containerele depozitează datele, dar nu ştiu despre algoritmi
  - Algoritmi şi containerele interacţionează prin intermediul iteratorilor
    - Fiecare container areun iterator

Containere asociative

- Au acces direct la stocarea/ştergerea de elemente
- Folosesc chei de căutare
- 4 tipuri: multiset, set, multimap, map
  - cheile sunt sortate
  - multiset şi multimap permit chei duplicate
  - multimap şi map conţin valori asociate
  - multiset şi set conţin doar valori



Map/Multimap

Map

- Header <map>
- Nu permite chei duplicate
  - Relaţia one-to-one
    - Ex. O țară poate avea o capitală
- Se poate folosi operatorul[]

Multimap

- Header <map>
- Permite chei duplicate
  - Relaţia one-to-many
    - Ex. Un student poate participa la mai multe cursuri
- Iteratori bidirectionali

EXEMPLU MULTIMAP

```
#include <map>//definitia claselor map si multimap

//definirea unui nume mai scurt pentru multimap
typedef std::multimap< int, double, std::less< int > > mm;

int main()
{
   mm pairs;

   cout << " In acest moment sunt " << pairs.count( 15 ) << " perechi cu cheia 15 in
     multimap\n";

  // inserarea a doua valori folosind value_type
   pairs.insert( mm::value_type( 15, 2.7 ) );
   pairs.insert( mm::value_type( 15, 99.3 ) );

   cout << " Dupa insert sunt " << pairs.count( 15 ) << " perechi cu cheia 15 \n\n";
```

EXEMPLU MULTIMAP

```
  // inseraza cinci valori in multimap
  pairs.insert( mm::value_type( 30, 111.11 ) );
  pairs.insert( mm::value_type( 10, 22.22 ) );
  pairs.insert( mm::value_type( 25, 33.333 ) );
  pairs.insert( mm::value_type( 20, 9.345 ) );
  pairs.insert( mm::value_type( 5, 77.54 ) );

  cout << "Multimap contine :\nCheie\tValoae\n";
  // foloseste const_iterator pentru a itera prin lista de chei
   for ( mm::const_iterator iter = pairs.begin(); iter != pairs.end(); ++iter )
        cout << iter->first << '\t' << iter->second << '\n';

  cout << endl;
  return 0;
} // end main
```

Rezultat rulare

- Rezultat

```
 In acest moment sunt 0 perechi cu cheia 15 in multimap
 Dupa insert sunt 2 perechi cu cheia 15

Multimap contine :
Cheie   Valoare
5       77.54
10      22.22
15      2.7
15      99.3
20      9.345
25      33.333
30      111.11
```

EXEMPLU MAP

```
#include <iostream>
using std::cout;
using std::endl;
#include <map>// definitia clasei map
// definirea unui nume scurt pentru map care va fi folosit in program
typedef std::map< int, double, std::greater< int > > mid;

int main() {
  mid pairs;

   // adaugare valori
  pairs[15] = 2.7 ; // folosirea operatorului de indexare pentru inserarea valorii
  pairs.insert( std::make_pair (30, 111.11 ) ); // folosirea templatului pair inserarea valorii
  pairs.insert( mid::value_type( 5, 1010.1 ) );
  pairs.insert( mid::value_type( 10, 22.22 ) );
  pairs.insert( mid::value_type( 25, 33.333 ) );
  pairs.insert( mid::value_type( 5, 77.54 ) );  // duplicat, va fi ignorat
  pairs.insert( mid::value_type( 20, 9.345 ) );
  pairs.insert( mid::value_type( 15, 99.3 ) ); // duplicat, va fi ignorat
```

EXEMPLU MAP

```
  cout << "perechi:\nCheie\tValoare\n";
   // folosire iterator constant pentru parcurgere
  for ( mid::const_iterator iter = pairs.begin();
          iter != pairs.end(); ++iter )
     cout << iter->first << '\t' << iter->second << '\n';

  // folosirea operatorului de indexare pentru modificarea valorii
   pairs[ 25 ] = 9999.99;
   return 0;
} // end main
```

```
perechi:
Cheie   Valoare
30      111.11
25      33.333
20      9.345
15      2.7
10      22.22
5       1010.1
```

Set/Multiset

Set

- Header <set>
- Reține un șir de valori distince sortat

Multiset

- Header <set>
- Reține un șir de valori sortat

Exemplu Set

```
#include <iostream>
#include <set>
int main () {
   std::set<int> myset;
   std::set<int>::iterator it;
   std::pair<std::set<int>::iterator,bool> ret;
  // setare valori initiale:
  for (int i=1; i<=5; ++i) myset.insert(i*10);
   // set: 10 20 30 40 50
  ret = myset.insert(20); // un nou eleemt e inserat
   if (ret.second==false) it=ret.first; // "it" pointeaza spre elemetul 20

   int myints[]= {5,10,15}; // 10 exista in set nu va fi inserat
   myset.insert (myints,myints+3);
   std::cout << "myset contine:";
   for (it=myset.begin(); it!=myset.end(); ++it)
       std::cout << ' ' << *it; std::cout << '\n';
  return 0;
}
```

ALGORITMI

- Există 60 de algoritmi definiţi în biblioteca standard
- Header <algorithm>
- Pot fi aplicaţi containerelor standard, string şi built-in array
- Definiţi ca funcţii template
- Tipuri
  - Nu modifică secvenţa
  - Modifică secvenţa
  - Secvenţe sortate
  - Altele: mulţimi, heap, minimum, maximum, permutări

Algoritmi

- for_each()
  - Invocă o funcţie pentru fiecare element
- find()
  - Caută prima apariţie a unui argument
- find_if()
  - Prina apariţie care se potriveşte cu predicatul
- count()
  - Numără apariţiile unui element
- count_if()
  - Numără potrivirile predicatului
- replace()
  - Înlocuieşte un element cu noua valoare
- replace_if()
  - Înlocuieşte elementul care se potriveşte cu predicatul cu noua valoare
- copy()
  - Copiază elementele
- unique_copy()
  - Copiază elementele care nu sunt duplicate
- sort()
  - Sortează elementele
- equal_range()
  - Găseşte elemente cu valori echivalente
- merge()
  - Combină secvenţe sortate

EXEMPLU

```
char nextLetter() ;
bool equal_a(char c);
void codificare(char);
int main () {
  std::vector< char > chars( 10 );
  std::ostream_iterator< char > output( cout, " " );

  std::generate_n( chars.begin(), 5, nextLetter );
   cout << "\n\nVectorul de caractere dupa umplerea primelor 5 pozitii:\n";
  std::copy( chars.begin(), chars.end(), output );

  std::fill(chars.begin()+5,chars.end(),'a');
   cout << "\n\nVectorul de caractere dupa umplerea ultimelor 5 pozitii:\n";
   std::copy( chars.begin(), chars.end(), output );

   std::replace_if(chars.begin(),chars.end(), equal_a, '-');
   cout << "\n\nVectorul de caractere dupa inlocuirea lui 'a' cu '-':\n";
  std::copy( chars.begin(), chars.end(), output );
```

Exemplu

```
  cout << " \ n \nVectorul de caractere dupa codificare: \n";
  std::for_each (chars.begin(), chars.end(), codificare);
}

// intoarce urmatoarea litera din alfabel (incepand cu A)
char nextLetter() {
  static char letter = 'A';
  return letter++;
} // end nextLetter

//verifica daca un caracter este egal cu litera a
bool equal_a( char c){
  return c=='a';
} //end equal_a

//adaga la o litera 10
void codificare( char c){
  cout << c+10 << ' \t';
} // end codificare
```

```
Vectorul de caractere dupa umplerea primelor 5 pozitii:
A B C D E a a a a a

Vectorul de caractere dupa umplerea ultimelor 5 pozitii:
A B C D E a a a a a

Vectorul de caractere dupa inlocuirea lui 'a' cu  '-':
A B C D E - - - - -

Vectorul de caractere dupa codificare:
75      76      77      78      79      55      55      55      55      55
```

POINTERI LA FUNCȚII

- Exemplu
  - Sortarea unui liste de persoane
    - Dorim sa sortam după
      - Nume de familie
      - Vârstă
      - Studii
      - ….
  - Cum implementăm aceste cerinţe?

POINTERI LA FUNCTII

- Definim mai multe funcţii de sortare
  - sortNume(sir)
  - sortVarsta(sir)
  - sortStudii(sir)

- Definim o funcţie de sortare care pe lângă şirul de elemente care trebuie sortat primeşte
  - Un criteriu discriminator pentru tipul de sortare
    - sort(sir, ”nume”)
    - sort(sir, ”varsta”)
    - sort(sir, ”studii”)
  - Un pointer la funcţii
    - sort(sir, nume)
    - sort(sir, varsta)
    - sort(sir, studii)

POINTERI LA FUNCȚII

- Utilitate
  - Funcţii ca argumente a altor funcţii
    - Funcţii callback (sau listener)
      - void sort(Peroane**sir, int (*compar) (const Persona&, const Persoana&)){
        ...
        if (compar(sir[i],sir[i+1]){ ...}
        ...
        }
  - int varsta(const Persona &p1, const Persona& p2);

POINTERI LA FUNCȚII

- Exemplu declarare a unui pointer la funcţii şi apelul funcţiei

```
void my_int_func(int x) {
    cout << x;
}
int main() {
   void (*foo)(int); //declare pointer de tip functie
   foo = &my_int_func; //initializare
   foo( 2 ); // apel functie
   (*foo)( 2 ); // apel functie, dar nu este obligatoriu
   return 0;
}
```

FUNCTORI

- Functor (function-like object / function object) este o instanţă a unei clase pentru care sa supraîncărcat operatorul funcţie ()
- Avantaje
  - Suportă operaţii mai complexe decât funcţiile obişnuite
- Mecanism de customizare a comportamentului algoritmilor standard
- Header <functor>

EXEMPLU

```
template <class T> SumMe {
  public :
     SumMe(T i=0) : sum(i) { }
     void operator() (T x) {
          sum += x;
     }
    T result() const { return sum; }
 private :
    T sum ;
};

void f(vector v) {
   SumMe s;
   for_each (v.begin(), v.end(), s); cout << “Sum is “ << s.result();
}
```

EXEMPLU LISTA DE POINTERI LA OBIECTE

```
class Persoana{
  bool operator < (const Persoana&) const;
};

bool comparePtr(Persoana* a, Persoana* b) {
    return (*a < *b);
}

int main(){
    std::list sir;
    sir.push_back(new persoana(...));
    std::sort(sir.begin(), sir.end());
}
```

EXEMPLU LISTA DE POINTERI LA OBIECTE

```
class Persoana{
  bool operator < (const Persoana&) const;
};

bool comparePtr(Persoana* a, Persoana* b) {
    return (*a < *b);
}

int main(){
    std::list sir;
    sir.push_back(new persoana(...));
    std::sort(sir.begin(), sir.end(), comparePtr);
}
```

Funcții Lambda

- Funcții lambda (C++11)
  - Sunt funcții fără nume
  - Definesc funcționalitatea în locul unde sunt definite
- Funcțiile lambda ar trebui să fie
  - Concise
  - Ușor de înțeles (self explaining)

Funcții lambda

- Sintaxă
  - [] () -> { ...}

- Unde
  - [] – utilizat pentru a captura variabile referință sau copii de variabile
  - () – se specifică parametrii
  - -> - specifică tipul de return pentru expresii lambda mai sofisticate
  - {} – include expresii și bucăți de cod

Funcții lambda

- [] – capturarea variabilelor
  - [a, &b] – variabila a este capturată prin copiere şi variabila b prin referinţă
  - [this] – capturează obiectul curent (*this) prin referinţă
  - [&] – capturează toate variabile automatice folosite în corpul expresiei lambda prin referinţa deasemenea capurează şi obiectul this prin referinţă dacă este cazul
  - [=] – capturează toate variabile automatice folosite în corpul expresiei lambda prin copiere deasemenea capurează şi obiectul this prin referinţă dacă este cazul
  - [] nu capturează nimic

Funcții lambda

- Sortează elementele unui vector
  - vector<int> vec={3,2,1,5,4};

  - C++98

```
class MySort{
public:
bool operator()(int v, int w){ return v > w; }
};
sort(vec.begin(),vec.end(),MySort());
```

  - C++11

```
sort(vec.begin(),vec.end(),[](int v,int w){return v>w;});
```

Funcții lambda

- Numără câte elemente ale unui vector sunt egale cu o valuare
  - C++98

```
class Functor {
public:
int &a;
Functor(int &_a):a(_a){}
bool operator ()(int x) const { return a==x; }
};
int a=45;
count_if(v.begin(), v.end(), Functor(a));
```

  - C++11

```
int a=45;
count_if(v.begin(), v.end(), [&a](int b){return a==b;});
```

  - C++14

```
int a=45;
count_if(v.begin(), v.end(), [&a](auto b){return a==b;});
```

STRING

- String – este o secvenţă de caractere
- Definită în biblioteca <string>prin clasa std::string
- Permite operaţii comune cu şirurile de caractere: concatenare, inserare, atribuire, comparare, adăugare, căutarea unui sub şir, extragerea unui şir
- Suportă orice set de caractere

EXEMPLU

```
string foo() {
  string s1 = “First string” ;
  string s2 = s1, s3(s1, 6, 3);
  wstring ws(s1.begin(), s1.end()); // string of wchar_t
  s3 = s2;
  s3[0] = ‘A’;

  const char* p = s3.data(); // conversion to C
  delete p; // ERROR: the array is owned by string object
  if(s1==s2) cout << “Strings have same content” ;
  s1 += “ and some more.” ;
  s2.insert( 5, “smth ”);

 string::size_type i1 = s1.find(“string”); // i1=6
 string::size_type i2 = s1.find_first_of(“string”); // i2=3
 s1.replace(s1.find (“string”), 3, “STR”);
 cout << s.substr ( 0, 5);
 cin >> s3;
return s1 + ‘ ‘ + s2; // concatenation
}
```

TIPURI NUMERICE - COMPLEX

- Numere complexe
  - Suporta o familie de numere complexe, folosind diferite tipuri scalare pentru a reprezenta părţile reale şi imaginare
  - Suporta operaţii matematice comune

```
void f(complex fl, complex db) {
   complex ld = fl+sqrt(db);
   db += fl*3;
   fl = pow(1/fl, 2);
}
```