Programare II
Programare Orientată Obiect

Curs anterior

- Tipuri de date abstracte
- Tipuri de date generice
- Funcţii şablon
- Clase şablon

Curs Curent

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

Caracteristici unei biblioteci generice

- Refolosire
  - Posibilitatea de a opera cu tipuri de date definite de utilizator
- Compoziţia
  - Posibilitatea de a opera cu tipuri de date definite în altă bibliotecă
- Eficienţa
  - Performanțe mai bune decât implementările non-generice (codificării personale)

Ce oferă bibliotecile standard în C++?

- Oferă suport pentru proprietăţile limbajului (gestionarea memoriei, RTTI)
- Oferă informaţii despre implementarea compilatorului (ex. cea mai mare valoare pentru numere de tip float)
- Oferă funcţii care nu pot fi implementate optimal în limbaj pentru orice sistem (sqrt, memmove, etc)
- Oferă suport pentru lucrul cu şiruri de caractere şi streamuri (include suport pentru internaţionalizare şi localizare)
- Oferă un framework pentru containere (vector, map,…) şi algoritmi generici pentru ele (parcurgere, sortare, reuniune, …)
- Suport pentru prelucării numerice (numere complexe, BLAS=Basic Liniar Algebra Subprograms etc.)
- Oferă o bază pentru alte biblioteci

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

De ce să folosim STL?

- Micşorează timpul de implementare
  - Structuri de date deja implementate şi testate
- Cod mai uşor de citit
- Robusteţea
  - Structurile STL sunt actualizate automat
- Cod portabil
- Mentenabilitatea codului
- Uşurinţă

Componente STL

- Containăre
- Iteatori
- Algoritmi

Containere

- Definiţie
  - Un container este un obiect care conţine obiecte

- Exemple:
  - list, vector, stack, etc

- Avantaje containere
  - Simple şi eficiente
  - Fiecare container are ataşat o mulţime de operaţii comune
  - iterarori standard sunt disponibili pentru fiecare container
  - Type-safe şi homogeni
  - Sunt non-intrusive (ex. un obiect nu are nevoie de o clasă de bază pentru a fi membru a unui container)

Exemplu

- Declarare, popularea și afișarea conținutului unei structuri de date de tip map, care stochează informații despre un produs și prețul lui

```
#include <map>
#include <string>
int main() {
//Declarare, popularea și afișarea  conținutului unei structuri de date de tip map
}
```

Includera bibliotecilor care conțin declarerea structuri de date map și a clasei string care ușurează lucrul cu șiruri de caractere

Exemplu

```
int main()
{
   map<string,float> price;
   price["snapple"] = 0.75;
   price["coke"] = 0.50;
   string item;
   double total=0;
   while ( cin >> item )
        total += price[item];
}
```

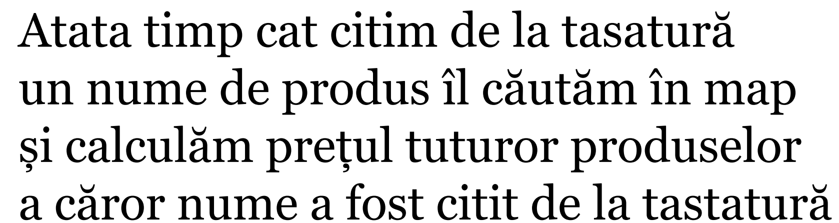
Declararea unei structuri de date de tip map pentru a reține numele produselor și prețurile corespunzătoare

Exemplu

```
int main()
{
   map<string,float> price;
   price[“snapple”] = 0.75;
   price[“coke”] = 0.50;
   string item;
   double total=0;
   while ( cin >> item )
       total += price[item];
}
```

Exemplu

```
int main()
{
   map<string,float> price;
   price[“snapple”] = 0.75;
   price[“coke”] = 0.50;
   string item;
   double total=0;
   while ( cin >> item )
        total += price[item];
}
```



Exemplu

```
int main()
{
   map<string,float> price;
   price[“snapple”] = 0.75;
   price[“coke”] = 0.50;
   string item;
   double total=0;
    while ( cin >> item )
        total += price[item];
}
```

Containere

- Tipuri de containere
  - Containere secvențiale
    - Structuri de date liniare (vector, list, deque)
    - Containere first-class
  - Containere asociative
    - Structuri de date non-liniare, pot identifica rapid elemente (map, multimap, set, multiset)
    - Perechi cheie/valoare
    - Containere first-class
  - Adaptorii ai containerelor
    - Containăre obţinute prin adaptarea unor containăre secvenţiale (stack, queue, priority_queue)

- Clase non-STL cu caracteristici și comportament (containere near)
  - Asemănătoare cu containerele, dar cu funcţionalitate redusă (string, bitset, valarray)
  - Pot fi folosite cu iteratorii cea ce le face accesibile manipulări algoritmilor definiţi în STL

Containere

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

Funcții membre comune pentru container

- Funcţii membre pentru toate containerele
  - Constructor implicit (default), constructor de copiere, destructor
  - empty
  - max_size, size
  - = < <= > >= == !=

- Funcţii pentru containere
  - begin, end
  - rbegin, rend
  - erase, clear

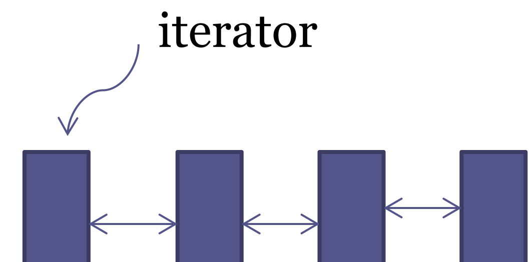
Componente STL

- Containăre
- Iteatori
- Algoritmi

Iteratori

- Iteratori asemănători cu ponteri
  - Pointează spre primul element al unui container
  - Operatori comuni pentru toate containerele
    - * deferenţiere
    - ++ pointeză spre următorul element
    - begin() întoarce un iterator sper primul element
    - end() întoarce un iterator sper ultimul element

- Utilizare iteratori cu secvenţe
  - Containere
  - Secvenţe de intrare; istream_iterator
  - Secvenţe de ieşire: ostream_iterator



Iteratori

- Utilizare
  - std::istream_iterator< int > inputInt(cin)
  - Poate citi intrări de la cin
  - *inputInt
    - Deferenţiat pentru a citi primul int de la cin
  - ++inputInt
    - Trece la următorul int din flux (stream)

- std::ostream_iterator< int > outputInt(cout)
  - Poate scrie int la cout
  - *outputInt = 7
    - Afişează 7 la cout
  - ++outputInt
    - Avansează iteratorul astfel încât să se poată afişa următorul int

Iteratori. Exemplu

```
#include <iostream>
#include <iterator> // ostream_iterator şi isteam_iterartor
using namespace std;
int main()
{
 cout << "Introdu 2 intregi: ";
 istream_iterator <int> inputInt (cin);
 int nr1 = *inputInt;
 ++inputInt;
 int nr2 = *inputInt;

 ostream_iterator <int> outputInt (cout);
 cout<< "rezultatul este: ";
 *outputInt = nr1 + nr2;
 cout<<endl;
}
```

Output
Introdu 2 intregi: 5 7 rezultattul este: 12

Tipuri de iteratori

- Intrare (input)
  - Citeşte elemente dintr-un container, se poate deplasa doar înainte
- Ieşire (output)
  - Scrie elemente într-un container, se poate deplasa doar înainte
- Înaintare (forward)
  - Combină iteratori de intrare şi ieşire
  - Pot parcurge o secvenţă de mai multe ori
- Bidirecţionali
  - La fel ca cei de înaintare, dar se pot deplasa şi în sens invers
- Acces aleatoriu (random)
  - La fel ca cei bidirecţionali dar pot sări şi la orice element

Tipuri de iteratori suportaţi de cotainere

- Containere secvenţiale
  - vector: acces aleator
  - deque: acces aleator
  - list: bidirecţional

- Containere asociative (pt. toate acces bidirecţional)
  - set
  - multiset
  - map
  - multimap

- Adaptori ai containerelor (nu au iteratori ataşaţi)
  - stack
  - queue
  - priority_queue

Iteratori operaţii

- Toţi
  - ++p, p++
- Iteratori de intrare
  - *p
  - p = p1
  - p == p1, p != p1
- Iteratori de ieşire
  - *p
  - p = p1
- Iteratori de înaintare
  - Conţin operaţiile iteratorilor de intrare şi ieşire
- Bidirecţionali
  - --p, p—
- Acces aleatoriu
  - p + i, p += i
  - p - i, p -= i
  - p[i]
  - p < p1, p <= p1
  - p > p1, p >= p1

Componente STL

- Containăre
- Iteatori
- Algoritmi

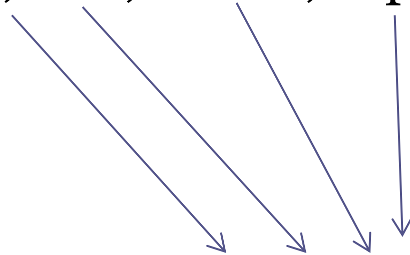
Algoritmi

- Algoritmi STL utilizaţi de obicei împreună cu containerele
  - Operează cu elementele containărelor indirect prin intermediul iteratorilor
  - De multe ori operează pe secvenţe de operatori
    - Perechi de operatori
    - Primul şi ultimul element
  - De multe ori întorc iteratori
    - Ex: find()
    - Într-un iterator la un element al secvenţei sau end() dacă nu este găsit
  - Economisesc timpul şi efortul necesar implementări lor

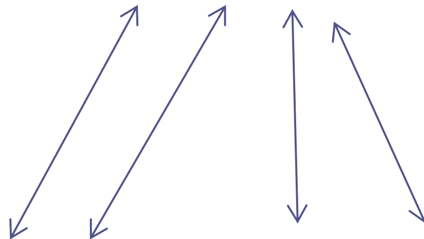
Modelul

Algoritmi

sort, find, search, copy, …



Iteratori



Containere

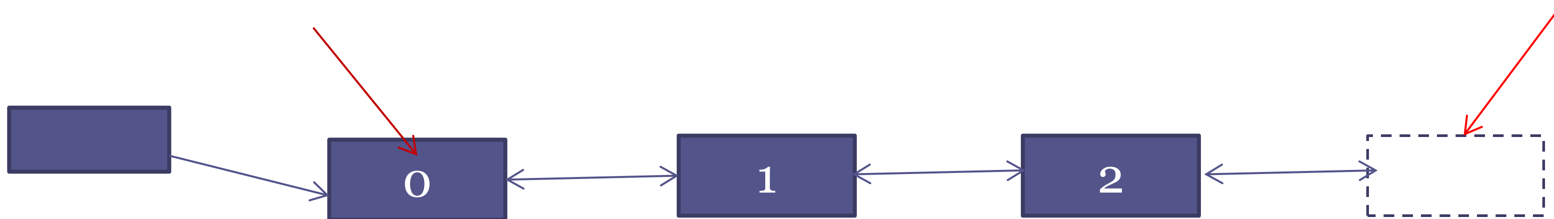
vector, list, map, multimap, …

- 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 secvenţiale

- vector

- list (lista dublu înlănţuită)



Containărul vector

- Header <vector>
- Inserare / ştergere rapidă de la sfârşitul tabloului
- reverse(), capacity()
- Blocuri continue de memorie
  - Accesarea elementelor folosind []
- Tablou dinamic
  - Creşte cu un anumit factor atunci când mărimea (size()) depăşeşte capacitatea (capacity())
    - Alocă o zonă de memorie continuă mai mare
    - Face o copie a lui însuşi
    - Dealocă vechea memorie
- Iteratori de acces aleatori

Vector

- Declarare
  - std::vector < tip > variabila;
- Iteratori
  - std::vector ::const_iterator iterVar;
    - Nu poate modifica valorile elementelor
  - std::vector ::reverse_iterator iterVar;
    - Parcurge elementele in ordine inversă
    - Pentru a obţine punctul de început se foloseşte rbegin
    - Pentru a obţine punctul de sfârşit se foloseşte rend
  - Std::ostream_iterator nume (outputStream, separator)
    - Separator – caracter care separă ieşirile

Vector

- Funcţii
  - v.push_back(valoare)
    - Adaugă un element la sfârşitul vectorului
  - v.size()
    - Dimensiune actuală a vectorului
  - v.capacity()
    - Cât de multe elemente poate să conţină vectorul înainte de a fi redimensionat
    - La realocare spaţiul se dublează
  - vector v(array, array+ SIZE)
    - Crează un vector v cu elementele tabloului a pană la array+SIZE

Vector

- Funcţii
  - v.insert (iterator, valoare)
    - Inserează valoarea înaintea locaţiei iteratorului
  - v.insert (iterator, array, array+SIZE)
    - Inserează un tablou de elemente înaintea poziţiei iteratorului
  - v.erase(iterator)
    - Şterge elementul din container
  - v.erase(iterator1, iterato2)
    - Şterge elementele începând cu iterator1 până la iterator2
  - v.clear()
    - Şterge întreg containerul

Vector

- Funcţii
  - v.front(), v.back()
    - Returnează primul, ultimul element al vectorului
  - v[elementnumber] = valoare
    - Atribuie valoarea elementului
  - v.at(elementnumber) = valoare
    - Atribuie valoarea elementului, verifică corectitudinea indexului
    - Excepţie out_of_bounds

Parcurgere vector

- for(int i = 0; i<v.size(); ++i)   // de ce int? ...
  ... // utilizare v[i]
- for(vector<int>::size_type i = 0; i<v.size(); ++i)
  // mai lungă dar întotdeauna corectă
  ... // utilizare v[i]
- for(vector<int >::iterator p = v.begin(); p!=v.end(); ++p)
  … // utilizare *p

- Toate variantele sunt corecte
  - Nu există avantaje fundamentale ale unei abordări
  - Cea care foloseşte iterator este comună pentru toate containerele
  - Preferă size_type in loc de int pentru a împiedica erori rare

Vector exemplu

```
#include <iostream>
#include <vector>

int main () {
  std::vector<int> a; //definirea unui vector fara elemente

  std::vector<int> b(4,100);  //definirea unui vector care contine 4 elemente egale cu 100

  std::vector<int> c(b.begin(), c(b.begin()+2); //creaza un vector cu primele 2 elemete ale vectorului
    b

  int myints[] = {16,2,77,29};
  std::vector<int> d(myints, myints + sizeof(myints) / sizeof(int) ); // initializare vector cu un
    tablou de intregi

  std::cout << “Afisare continut vector d:";
  for (std::vector<int>::iterator it = d.begin(); it != d.end(); ++it)
        std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Vector exemplu

```
#include <vector>
using namespace std;
int main(){
      vector<double> values(10);
      // creaza un vector cu 10 elemete initializate cu 0.0

      values.insert (values.begin() + 5, 1.4142);
      // inseraza 1.4142 pe a cincea pozitie in vector

       values.remove (values.begin() + 3);
      // Sterge  elementul de pe pozitia a treia din vector
}
```

Containerul list

- Containerul list
  - Header <list>
  - Adăugare/ştergere eficientă oriunde în container
  - Liste dublu înlănţuite
  - Iteratori bidirecţionali
  - Utilizare
    - std::list < tip > nume;

Containerul list

- Funcţii ataşate unui obiect, T, de tip list
  - T.sort ()
    - Sortare în ordine crescătoare
  - T.splice (iterator, altObiect)
    - Inserează valorile din variabila altObiect înaintea iterarorului
  - T.merge (altObiect)
    - Şterge altObiect şi îl inserează în T sortat
  - T.unique()
    - Şterge elementele duplicate

Containerul list

- Funcţii ataşate unui obiect, T, de tip list
  - T.swap (altObiect)
    - Interschimbă conţinutul
  - T.assign (iterator1, iterator2)
    - Înlocuieşte conţinutul cu elementele din domeniul iteratorului
  - T.remove (valoare)
    - Şterge toate instanţele valorii 'valoare'

Containerul list.Exemplu

```
#include <list>
#include <string>
int main () {
  std::list<std::string> mylist;  //declararea unei liste de tip string
  std::list<std::string>::iterator it;  //declararea unui iterator de tip list<string>

  mylist.push_back ("one"); //adaugare de elemente in lista
  mylist.push_back ("two");
  mylist.push_back ("Three");

  mylist.sort(); // sortare lista

  std::cout << “lista contine:";
 for (it=mylist.begin(); it!=mylist.end(); ++it)
         std::cout << ' ' << *it;
   std::cout << '\n';
}
```

Deque

- Coadă cu intrări în ambele părţi
- Inserare rapidă la începutul şi sfârşitul cozi
- Header <deque>
- Functii
  - push_back(), push_front()
  - pop_back(), pop_front()
  - empty()
  - insert()
  - erase()

Deque. Example

```
#include <iostream>
#include <deque>
int main () {
   std::deque<int> mydeque;  //declarea unei cozi goale

   mydeque.push_back (100);  //adaugare elemente in coada
   mydeque.push_back (200);
   mydeque.push_back (300);

   std::cout << “Scoatere elemente din coada:";
   while (!mydeque.empty()) {
        std::cout << ' ' << mydeque.front();
        mydeque.pop_front();
    }
    std::cout << "\n Dimeniunea finala a coze este:" << int(mydeque.size()) << '\n';
    return 0;
}
```