Parametrizare (Programare generică) - plan

- Clase parametrizate
  - Declarație template
  - Definirea membrilor
  - Instanțiere (Specializare), Specializare utilizator
  - Friend în template
  - Membri statici în template
- Introducere in STL
  - Conținutul STL
  - Containere – clasificare
  - Iteratori – clasificare
  - Operații de bază
  - Vector: declarare, acces, iteratori, inserări, ștergeri, comparări
- Ierarhia de clase STL:: IOS

Clase parametrizate

- Programare generică : programare ce utilizează tipurile ca parametri
- În C++ - mecanismul template: clase template, funcții template
- Programatorul scrie o singură definiție template iar C++, pe baza parametrilor, generează specializări care sunt compilate cu restul programului sursă
- O funcție template poate fi supraîncărcată cu:
  - Funcții template cu acelaşi nume dar parametri template diferiți
  - Funcții non – template cu acelaşi nume dar cu alți parametri
- Clasele template se mai numesc tipuri parametrizate

Template-uri

```
//Supraincarcare functii
void swap(int& x, int& y){
  int aux = x;
  x = y;
  y = aux;
}
void swap(char& x, char& y){
  char aux = x;
  x = y;
  y = aux;
}
void swap(float& x, float& y){
  float aux = x;
  x = y;
  y = aux;
}
```

Template-uri

```
int x = 23, y = 14;
char c1 = 'o', c2 = '9';
double u = .5, v = 5.5;
swap(x, y);
swap(u, v);
swap(c1, c2);
```

- Polimorfism parametric:
  - utilizarea aceluiaşi nume pentru mai multe înțelesuri(polimorfism).
  - înțelesul este dedus din tipul parametrilor (parametric).

Template-uri

- Funcție generică de interschimbare:

```
template <class T>
void swap(T& x, T& y) {
  tip aux = x;
  x = y;
  y = aux;
}

int m = 8, n = 15;
swap(m, n); // swap<int>(m, n);
cout << endl << m << ", " << n << endl;
double a = 10.0, b = 20.0;
swap(a,b); // swap<double>(m, n);
cout << endl << a << ", " << b << endl;
complex z1(1.2, 2.3), z2(5.5, 2.2);
swap(z1, z2); // swap<complex>(m, n);
```

Template-uri

- Funcție generică de sortare:

```
template <class T>
void naiveSort(T a[], int n)
{
    for (int i=0; i<n-1; i++)
          for(int j=i+1; j<n; j++)
                if (a[i] > a[j])
                      swap<T>(a[i], a[j]);
};
```

- Parametrii generici pot fi şi tipuri de bază

```
template <class T, int n>
void naivSort2(T a[])
{
    for (int i=0; i<n-1; i++)
          for(int j=i+1; j<n; j++)
                if (a[i] > a[j])
                      swap<T>(a[i], a[j]);
};
```

Template-uri

- La apel, pentru n trebuie transmisă o constantă:

```
int i;
double x[5] = {2,1,3,5,4};
naivSort<double>(x, 5);
for (i=0; i<5; i++)
  cout << x[i] << ", ";
cout << endl;
double y[5] = {2,1,3,5,4};
naivSort2<double, 5>(y);
for (i=0; i<5; i++)
  cout << x[i] << ", ";
cout << endl;
int n = 5;
double z[5] = {2,1,3,5,4};
naivSort2<double, n>(z);    // eroare
```

Clase template (parametrizate)

- Declarație **`template`**:

`template <` *lista_argumente_template* **`>`** *declaratie*

lista_argumente_template ::= *argument_template* | *lista_argumente_template*, *argument_template*

argument_template ::= *tip_argument* | *declaratie_argument*

tip_argument ::= **`class`** *identificator* | **`typename`** *identificator*

declaratie_argument::=<*tip*> *identificator*

declaratie::= *declarația unei clase sau funcții*

Clase parametrizate

```
template< class T, int i > class MyStack{};

template< class T1, class T2 > class X{};
template< typename T1, typename T2 > class X{};

template<class T> class allocator {};
template<class T1, typename T2 = allocator<T1> >
class stack { };
stack<int> MyStack; // al doilea argument este implicit

class Y {...};
template<class T, T* pT> class X1 {...};
template<class T1, class T2 = T1> class X2 {...};
Y aY;
X1<Y, &aY> x1;
X2<int> x2;
```

Clase parametrizate

- Funcțiile membru ale unei clase **`template`** sunt funcții template parametrizate cu parametrii clasei template respective

- Definirea membrilor unei clase **`template:`**
  - Definiție **`inline`**, la specificarea clasei – nu este specificat explicit **`template`**
  - Definiție în afara specificării clasei – trebuie declarată explicit **`template:`**

`template` <*lista_argumente_template* >
nume_clasa_template<*argum*>::*nume_functie_membru* **(***parametri***)**
{
    //Corp functie
}

- Compilatorul nu alocă memorie la declararea unui **`template`**

Clase parametrizate

- *Instanțiere* **`template`**: procesul de generare a unei declarații de clasă (funcție) de la o clasă(funcție) **`template`** cu argumente corespunzătoare

```
template class MyStack<class T,int n>{…};
template class MyStack<int, 6>;
template MyStack<int, 6>::MyStack();

template<class T> void f(T) {...}
template void f<int> (int);
template void f(char);
```

Clase parametrizate - Exemplu

```
template <class T>
class Vector
{
public:
        Vector(int=0);
        T& operator [](int);
        const T& operator [](int) const;
        // …
private:
        T* tab;
        int n;
};
template <class T>
class Matrice
{
public:
        Matrice(int=0, int=0);
        Vector<T>& operator [](int);
        const Vector<T>& operator [](int) const;
private:
        Vector<T>* tabv;
        int m, n;
};
```

Exemplu(cont)

```
template <class T>
T& Vector<T>::operator [](int i)
{
        cout << "Vector<T>::operator [](int i)" << endl;
        return tab[i];
}
template <class T>
const Vector<T>& Matrice<T>::operator [](int i) const
{
        cout << "Matrice<T>::operator [](int i) const" << endl;
        if (i < 0 || i >= m) throw "index out of range";
        return tabv[i];
}
        Vector<int> v(5);
        v[3] = 3; // apel varianta nonconst
        const Vector<int> vv(5);
        vv[4] = 4;// apel varianta const

        Matrice<double> m(3,5);
        m[1][2] = 5.0;
        const Matrice<double> mm(3,5);
        mm[2][3] = 7.0;
```

Specializări

- O versiune a unui template pentru un argument template particular este numită o *specializare*

- O definiție alternativă pentru o clasă(funcție) template (de ex. pentru a funcționa când argumentul template este pointer) este numită *specializare definită de utilizator*

Clase parametrizate - Specializări

```
template <class T>
class Vector
{
public:
  Vector(int=0);
  T& operator [](int);
  const T& operator [](int) const;
private:
  T* tab;
  int n;
};
```

- Specializări:

```
Vector<int> vi;
Vector<Punct*> vpp;
Vector<string> vs;
Vector<char*> vpc;
```

Clase parametrizate - Specializări

- Specializare a clasei **`Vector<T>`** pentru pointeri la **`void:`**

```
template<> class Vector<void*>{
// specializare fara parametri template
  void** p; // …
};
```

- Specializare a clasei **`Vector<T>`** pentru pointeri la **`T:`**

```
template<class T> class Vector<T*>{
// specializare cu parametri template
//…
};
```

Clase parametrizate - Specializări

```
template <class T>
void swap(T& x, T& y){
  T t = x; x = y; y = t;
}
```

- Specializare pentru **`Vector<T>`** :

```
template <class T>
void swap(Vector<T>& a, Vector<T>& b){
  a.swap(b);
}
```

- În clasa **`Vector<T>`**, metoda:

```
template <class T>
void Vector<T>:: swap(Vector<T>& a){
  swap(tab, a.tab);
  swap(n, a.n);
}
```

Clase parametrizate – relația friend

- Funcții(clase) **`friend`** în clase **`template`**:
  - Dacă funcția **`friend`** accesează un parametru **`template`**, aceasta trebuie să fie **`template`**. În acest caz o instanțiere este **`friend`** doar pentru instanțierile clasei **`template`** cu aceiaşi parametri (**`friend`** "legat").
  - Prieteni template "nelegați" - are alți parametri template
  - Dacă funcția **`friend`** nu accesează parametri **`template`** , este **`friend`** pentru toate instanțierile clasei

Clase parametrizate – relația friend

```
template <class T> class X
{
  //..
  friend void f1();
  // f1 friend pentru toate specializarile
  friend void f2(X<T>& );
  // f2(X<float>&) friend pentru X<float>
  friend class Y;
  friend class Z<T>;
  friend void A::f3();
  friend void C<T> :: f4(X<double>&);
  //…
};
```

Membri statici în template –uri

- Fiecare specializare a unei clase template are copia proprie a membrilor static, atât date cât şi funcţii
- Datele statice trebuiesc iniţializate, “global” sau pe specializări

```
template<class T, int n>
class ClasaN{
        static const int cod;
        //
};
template<class T, int n>
const int ClasaN<T, n>::cod = n;
template<>
const int ClasaN<int, 10>::cod =50
```

Conținutul STL

- Containere: obiecte ce conțin alte obiecte
- Iteratori: “pointeri” pentru parcurgerea containerelor
- Algoritmi generici: funcții ce se aplică diverselor tipuri de containere
- Clase adaptor: clase ce adaptează alte clase (variații ale altor containere)
- Alocatori: obiecte responsabile cu alocarea spațiului
- Toate componentele stl sunt template-uri

Containere secvențe

- clase template ce implementează structuri de date în care memoria este organizată secvențial
  - suportă accesul secvențial la componente
  - în unele cazuri acces aleator
- vector<T> : fișierul header <vector>
  - suportă acces aleator la elemente
  - timp constant pentru inserție și eliminarea componentelor de la sfârșit
  - timp liniar pentru inserarea și eliminarea elementelor de la început și interior
  - lungime variabilă

Containere secvențe

- deque<T>: fișierul header <deque>
  - suportă acces (inserare și eliminare) la ambele capete
  - lungime variabilă

- list<T> : fișierul header <deque>
  - suportă parcurgerea în ambele sensuri
  - timp constant pentru inserție și eliminare la început, la sfârșit sau în interior
  - lungime variabilă

Clase adaptor

- sunt simple variații ale containerelor secvențe
- clase care nu suportă iteratori
- stack<T> **<stack>**
  - LIFO (last in, first out)
- queue<T> **<queue>**
  - FIFO (first in, first out)
- priority_queue<T> **<queue>**
  - obiectul cu valoare maximă este totdeauna primul în coadă

Containere asociative

- **`map<T>` : `<map>`**
  - componentele sunt perechi <cheie, dată> cu cheie unică (nu există două date cu aceeași cheie)
- **`multimap<T>: <map>`**
  - componentele sunt perechi <cheie, dată> cu cheie multiplă(pot exista mai multe date cu aceeași cheie)
- **`set<T>: <set>`**
  - componentele sunt doar de tip cheie și NU pot exista în duplicat
- **`multiset<T>: <set>`**
  - componentele sunt doar de tip cheie și POT exista în duplicat

Tipuri asociate containerului X

`X::value_type`

tipul obiectului memorat in container

`X::allocator_type`

tipul managerului de memorie

`X::size_type`

tip pentru indici, număr de elemente etc.

`X::iterator`

tip pentru iterator cu care se parcurge containerul

Accesul la elementele containerului

`c.front()`

primul element

`c.back()`

ultimul element

`c[i]` **`//`**(nu pentru liste)

al i-lea element din secvență (NU se validează valoarea lui i)

`c.at(i)` **`//`**(nu pentru liste)

al i-lea element din secventă (SE validează valoarea lui i)

Operații de tip listă

`c.insert(p, x)`
inserează x inaintea lui p

`c.insert(p, n, x)`
inserează n copii ale lui x înaintea lui p

`c.insert(p, first, last)`
adaugă elementele din [first, last) înaintea lui p

`c.erase(p)`
elimină de la p

`c.erase(first, last)`
elimină elementele din [first, last)

`c.clear()`
elimină toate elementele

Operații de tip stivă și coadă

`c.push_back(x)`
inserează **`x`** la sfârșit

`c.pop_back()`
elimină de la sfârșit

`c.push_front(x)`
inserează **`x`** la început

`c.pop_front()`
elimină de la început

Funcții utile

```
c.size()        // numărul de elemente
c.empty()       // este c vid?
c.capacity()    // (numai pentru vector) spațiul alocat
c.reserve(n)    // (numai pentru vector) alocă spațiu
c.resize()      // (numai pentru vector) adaugă elemente la
                //  sfârșit
max_size()      // (numai pentru vector) dim celui mai mare
                // vector posibil
swap(c1, c2)    // intershimbă conținuturile a 2 containere
==              // testul de egalitate
!=              // testul diferit
<               // relația de ordine lexicografică
```

Iteratori

- Iterator: abstracție a noțiunii de pointer la un element din secvență
- obiect ce poate naviga printr-un container
  - operatorii de dereferențiere * și -> permit accesul la obiectul curent
  - operatorul ++ (--) permite accesul la obiectul următor (precedent)
  - operatorul ==
- Un iterator se declară în asociație cu un tip anume de container, este implementat relativ la acel tip dar acest lucru nu este relevant pentru utilizator
- Fiecare container include funcțiile membru **`begin()`**, **`end()`** pentru specificarea valorilor iterator corespunzătoare primului, respectiv primului după ultimul obiect din container
- În mod analog pentru explorare reverse: **`rbegin()`**, **`rend()`**

Clasificarea iteratorilor

Există 5 categorii de iteratori iar pentru fiecare categorie sunt definiți operatori specifici pentru citire (atribuirea obiect = *iterator), scriere(atribuirea *iterator= obiect), acces, iterație, comparare.

Categoria	Output	Input	Forward	Bidirectional	Random
Abreviere	Out	In	For	Bi	Ran
Citire		=*p	=*p	=*p	=*p
Acces		->	->	->	-> [ ]
Scriere	*p=		*p=	*p=	*p=
Iterație	++	++	++	++ --	++ -- + - += -=
Comparare		== !=	== !=	== !=	== != < > <= >=

Containerul vector

- se comportă ca un tablou alocat dinamic, suportă realocare dinamică la execuție
- Constructori:

```
vector<T> V; //empty vector
vector<T> V(n,value); //n copii din value
vector<T> V(n); //n copii din default pt. T
```

- clasa are definiți constructorul de copiere și operatorul de atribuire

Containerul vector

- Declarare:

```
vector<int> iVector;
vector<int> jVector(100);
cin >> Size;
vector<int> kVector(Size);
vector<int> v = vector<int>(100);
vector<int> vv = v;
```

- Acces la elemente:

```
jVector[23] = 71;
int temp = jVector[41];
cout << jVector.at(23) << endl;
int jFront = jVector.front();
int jBack = jVector.back();
```

- Informații:

```
cout << jVector.size();
cout << jVector.capacity();
cout << jVector.max_capacity();
if ( jVector.empty() ) //. . .
```

[ ] vs. at()

```
int MaxCount = 100;
vector<int> iVector(MaxCount);
for (int Count = 0; Count < 2*MaxCount; Count++) {
   cout << iVector[Count];
}
```

```
int MaxCount = 100;
vector<int> iVector(MaxCount);
for (int Count = 0; Count < 2*MaxCount; Count++) {
   cout << iVector.at(Count);
}
```

Utilizare iteratori

```
string DigitString = "456582284587205012 89";
vector<int> BigInt;
for (int i = 0; i < DigitString.length(); i++) {
  BigInt.push_back(DigitString.at(i) - '0');
  // push_back asigura redimensionarea vectorului
}

vector<int> Copy;
vector<int>::iterator It = BigInt.begin();
while ( It != BigInt.end() ) {
  Copy.push_back(*It);
  It++;
}
```

Utilizare iteratori

```
vector<int>::iterator It;
It = BigInt.begin(); // primul element
int FirstElement = *It; // copiere primul element
It++; // al doilea element
It = BigInt.end();
// It pointează în afara vectorului
// dereferențierea !!
It--; // inapoi, la ultimul element
int LastElement = *It; // ok
```

Utilizare iteratori

```
// vector::rbegin/rend
#include <iostream>
#include <vector>
using namespace std;
int main ()
{
  vector<int> myvector;
  for (int i=1; i<=5; i++) myvector.push_back(i);
  cout << "myvector:";
  vector<int>::reverse_iterator rit;
  for ( rit=myvector.rbegin() ; rit < myvector.rend(); ++rit )
    cout << " " << *rit;
  cout << endl;
  return 0;
}

//myvector: 5 4 3 2 1
```

Inserare în vector

- Se poate seta dimensiunea minimă n a vectorului v prin **`v.reserve(n)`**
- Inserarea la sfârșit folosind **`push_back()`** este eficientă
  - dacă inserția se face când vectorul este plin, se realocă spațiu
- **`pop_back()`** – elimină ultimul element
- Inserarea într-o poziție intermediară necesită deplasarea elementelor următoare

Inserare la o anume poziție

- Se utilizează un iterator și funcția **`insert()`**

```
vector<int> Y;
   for (int m = 0; m < 20; m++) {
        Y.insert(Y.begin(), m);
        cout << setw(3) << m
             << setw(5) << Y.capacity()
             << endl;
   }

   Y.insert(Y.begin()+4, 40);
   Y.insert(Y.end()-3, 30);
   for (int m = 0; m < 20; m++) {
        cout << setw(3) << m
             << setw(5) << Y.at(m)
             << endl;
   }
```

0	1	0	19
1	2	1	18
2	3	2	17
3	4	3	16
4	6	4	40
5	6	5	15
6	9	6	14
7	9	7	13
8	9	8	12
9	13	9	11
10	13	10	
11	13	10	
12	13	11	9
13	19	12	8
14	19	13	7
15	19	14	6
16	19	15	5
17	19	16	4
18	19	17	3
19	28	18	
		30	
		19	2
		20	1
		21	2

Eliminarea elementelor

- de la sfârșit: `V.pop_back()`
- de la o poziție It (cu șiftare): `V.erase(It)`
- Iteratorii care referă elemente ce urmează punctului în care s-a eliminat sunt invalidați:

```
vector<int> V=Y;
vector<int>::iterator j;
j = V.begin();
while (j != V.end())
V.erase(j++);
```

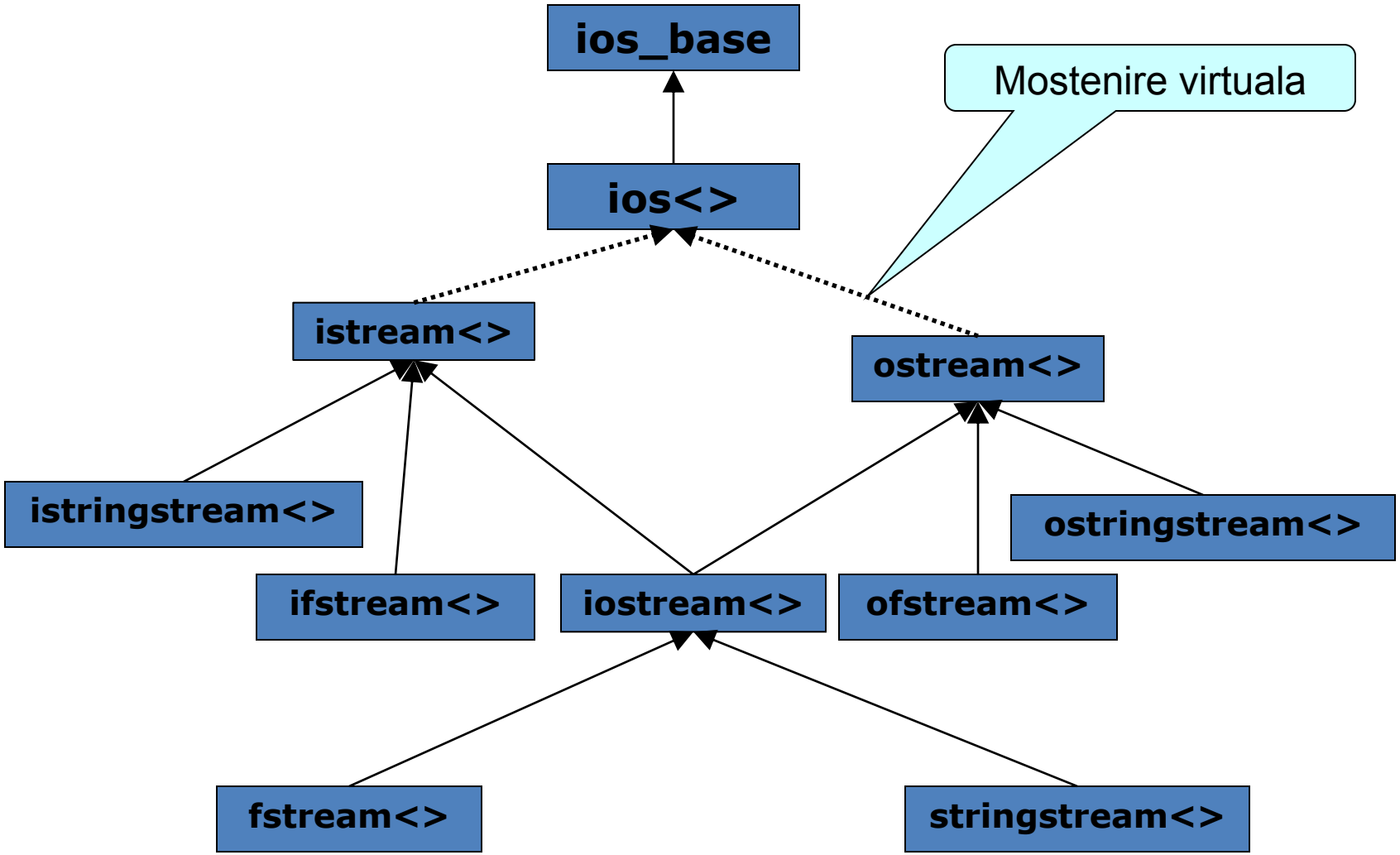
- Expression: vector iterator incompatible

```
vector<int>::iterator j=V.begin();
V.erase(j+10);
V.erase(j+2,j+5);
```

Alte funcții

- ==
  - doi vectori v1 și v2 sunt egali dacă v1.size() = v2.size() și v1[k] = v2[k] pentru orice index valid k.
- <
  - v1<v2 dacă
    - primul element v1[i] care nu este egal cu v2[i] este mai mic decât v2[i], sau
    - v1[size] < v2[size] și fiecare v1[i] este egal cu v2[i]
- analog sunt definiți operatorii !=, <=, >, >=
- v1.swap(v2)

Ierarhia de clase STL:: IOS



Ierarhia de clase iostream

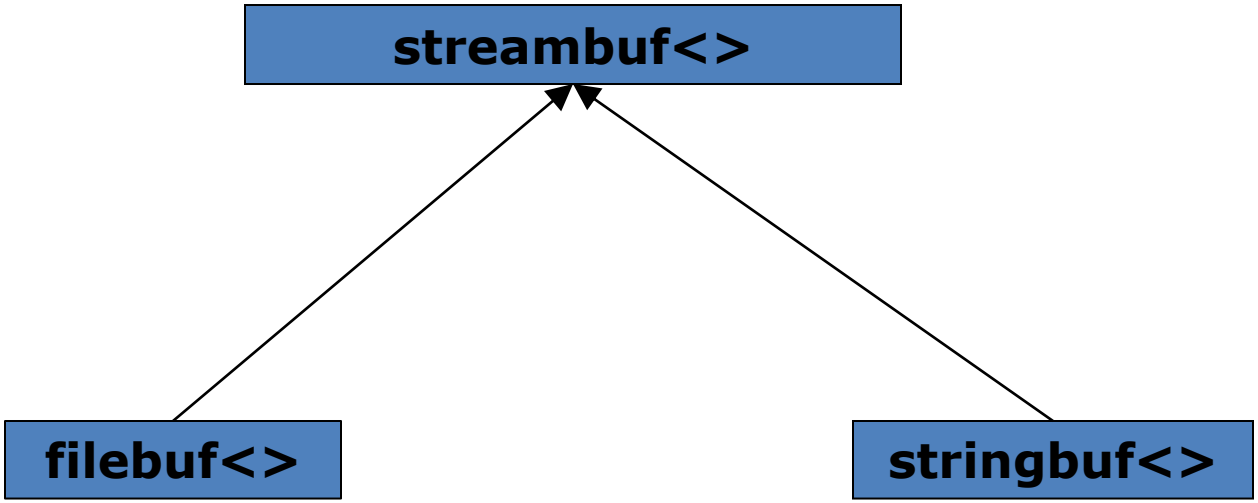
- Clasele cu sufixul <> sunt template - uri parametrizate cu un tip caracter iar numele lor au prefixul *basic_ :*

```
template <class E, class T = char_traits<E> >
class basic_ios : public ios_base {
//…
};
```

- **`char_traits<E>`** conține informații pentru manipularea elementelor de tip E

```
template <class E, class T = char_traits<E> >
class basic_istream : virtual public basic_ios<E, T>
  {
  //…
  };
```

Ierarhia de clase streambuf



Clasa basic_iostream

```
template <class E, class T = char_traits<E>>
class basic_iostream :
   public basic_istream<E, T>,
   public basic_ostream<E, T> {
public:
   explicit basic_iostream(basic_streambuf<E, T>* sb);
   virtual ~basic_iostream();
};
```

Fisierul header <iosfwd>

```
typedef basic_ios<char, char_traits<char> > ios;
typedef basic_istream<char, char_traits<char> >  istream;
typedef basic_ostream<char, char_traits<char> >  ostream;

typedef basic_iostream<char, char_traits<char> > iostream;
typedef basic_ifstream<char, char_traits<char> > ifstream;
typedef basic_ofstream<char, char_traits<char> > ofstream;
typedef basic_fstream<char, char_traits<char> >  fstream;
```

Clasa ostream

- Un obiect din ostream (flux de ieșire) este un mecanism pentru conversia valorilor de diverse tipuri în secvențe de caractere
- În <iostream>:

```
ostream cout; //fluxul standard de iesire
ostream cerr; // mesaje eroare,fara buffer
ostream clog; // mesaje eroare,cu buffer
```

`operator<<` (în ostream)

```
basic_ostream& operator<< (const char *s);
basic_ostream& operator<< (char c);
basic_ostream& operator<< (bool n);
basic_ostream& operator<< (short n);
basic_ostream& operator<< (unsigned short n);
basic_ostream& operator<< (int n);
basic_ostream& operator<< (unsigned int n);
basic_ostream& operator<< (long n);
basic_ostream& operator<< (unsigned long n);
basic_ostream& operator<< (float n);
basic_ostream& operator<< (double n);
basic_ostream& operator<< (long double n);
basic_ostream& operator<< (void * n);
basic_ostream& put(E c);      // scrie c
basic_ostream& write(E *p, streamsize n); // scrie p[0],…,p[n-1]
basic_ostream& flush(); // goleste bufferul
pos_type tellp(); // pozitia curenta
basic_ostream& seekp(pos_type pos); // noua positie pos
basic_ostream& seekp(off_type off, ios_base::seek_dir way);
       // pozitia way (= beg, cur, end) + off
```

Formatare, fișierul **<iomanip>**

```
void main()
{
     int i = 10, j = 16, k = 24;
     cout << i << '\t' << j << '\t' << k << endl;
     cout << oct << i << '\t' << j << '\t' << k << endl;
     cout << hex << i << '\t' << j << '\t' << k << endl;
     cout << "Introdu 3 intregi: " << endl;
     cin >> i >> hex >> j >> k;
     cout << dec << i << '\t' << j << '\t' << k << endl;
}

/*
10       16       24
12       20       30
a        10       18
Introdu 3 intregi:
42 11 12a
42   17       298
*/
```

Manipulatori

MANIPULATOR	EFECTUL	FISIERUL
`endl`	**Scrie newline**	**iostream**
`ends`	**Scrie NULL in string**	**iostream**
`flush`	**Goleste**	**iostream**
`dec`	**Baza 10**	**iostream**
`hex`	**Baza 16**	**iostream**
`oct`	**Baza 8**	**iostream**
`ws`	**Ignoră spaţiile la intrare**	**iostream**
`skipws`	**Ignoră spaţiile**	**iostream**
`noskipws`	**Nu ignoră spaţiile**	**iostream**
`showpoint`	**Se scrie punctul zecimal şi zerourile**	**iostream**
`noshowpoint`	**Nu se scriu zerourile, nici punctul**	**iostream**
`showpos`	**Scrie + la numerele nenegative**	**iostream**
`noshowpos`	**Nu scrie + la numerele nenegative**	**iostream**
`boolalpha`	**Scrie true si false pentru bool**	**iostream**
`noboolalpha`	**Scrie 1 si 0 pentru bool**	**iostream**

Manipulatori

MANIPULATOR	EFECTUL	FISIERUL
`scientific`	**Notaţia ştiinţifică pentru numere reale**	**iostream**
`fixed`	**Notaţia punct fix pentru numere reale**	**iostream**
`left`	**Aliniere stânga**	**iostream**
`right`	**Aliniere dreapta**	**iostream**
`internal`	**Caract. de umplere intre semn si valoare**	**iostream**
`setw(int)`	**Seteaza dimensiunea câmpului**	**iomanip**
`setfill(char c)`	**Caracterul c înlocueşte blancurile**	**iomanip**
`setbase(int)`	**Setarea bazei**	**iomanip**
`setprecision(int)`	**Seteaza precizia in flotant**	**iomanip**
`setiosflags(long)`	**Seteaza bitii pentru format**	**iomanip**
`resetiosflags(long)`	**Reseteaza bitii pentru format**	**iomanip**

```
double a = 12.05, b = 11.25, c = -200.89;
cout << a << ',' << b << ',' << c << endl;
cout << setfill('*') << setprecision(3);
cout << setw(10) << a << endl;
cout << setw(10) << b << endl;
cout << setw(10) << c << endl;
cout << setw(10) << showpoint << c << endl;
cout << setfill(' ') << right << showpoint;
cout << setw(15) <<  setprecision(5)
                     << c << endl;
cout << scientific << c <<  endl;
char d;
cin >> noskipws;
while(cin >>d)
      cout << d;
```

```
12.05,11.25,-200.89
******12.1
******11.3
******-201
*****-201.
        -200.89
-2.00890e+002
text pentru test
text pentru test
```

Clasa istream

- Un obiect din **`istream`** (flux de intrare) este un mecanism pentru conversia caracterelor în valori de diverse tipuri
- În <iostream>:

```
istream cin; //fluxul standard de intrare
```

- În **`basic_istream`** este definit **`operator>>`** pentru tipurile fundamentale

Funcții utile în istream

```
streamsize gcount() const;
int_type get();
basic_istream& get(E& c);
basic_istream& get(E *s, streamsize n);
basic_istream& get(E *s, streamsize n, E delim);
basic_istream& get(basic_streambuf<E, T> *sb);
basic_istream& get(basic_streambuf<E, T> *sb, E delim);
basic_istream& getline(E *s, streamsize n);
basic_istream& getline(E *s, streamsize n, E delim);
basic_istream& ignore(streamsize n = 1,
                                  int_type delim = T::eof());
int_type peek(); // citeste fara a-l extrage
basic_istream& read(E *s, streamsize n);
streamsize readsome(E *s, streamsize n); // peek cel mult n
basic_istream& putback(E c); // pune c in buferul de intrare
basic_istream& unget(); // pune inapoi ultimul citit
pos_type tellg();
basic_istream& seekg(pos_type pos);
basic_istream& seekg(off_type off, ios_base::seek_dir way);
int sync();  // flush input
```

Fișiere

- Într-un program C++ fluxurile standard **`cin, cout, cerr, clog`** sunt disponibile; corespondența lor cu dispozitivele (fișierele) standard este făcută de sistem

- Programatorul poate crea fluxuri proprii – obiecte ale clasei **`fstream`** (**`ifstream`**, **`ofstream`**). Atașarea fluxului creat unui fișier sau unui string se face de programator:
  - la declarare cu un constructor cu parametri
  - după declarare prin mesajul **`open()`**

Fișiere

- Constructorii clasei fstream:

```
explicit basic_fstream();
explicit basic_fstream(const char *s,
  ios_base::openmode mode =
   ios_base::in | ios_base::out);
```

- Metode:

```
bool is_open() const;
void open(const char *s,
  ios_base::openmode mode =
   ios_base::in | ios_base::out);
void close();
```

Fișiere de intrare

- Constructorii clasei ifstream:

```
explicit basic_ifstream();
explicit basic_ifstream(const char *s,
  ios_base::openmode mode = ios_base::in);
```

- Metode:

```
bool is_open() const;
void open(const char *s,
  ios_base::openmode mode = ios_base::in);
void close();
```

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("file.cpp");
  if (myfile.is_open())
  {
    while (! myfile.eof() )
    {
      getline (myfile,line);
      cout << line << endl;
    }
    myfile.close();
  }else
    cout << "Unable to open file";

  return 0;
}
```

Fișiere de ieșire

- Constructorii clasei ofstream:

```
explicit basic_ofstream();
explicit basic_ofstream(const char *s,
  ios_base::openmode which =
  ios_base::out | ios_base::trunc);
```

- Metode:

```
bool is_open() const;
void open(const char *s,
  ios_base::openmode mode =
  ios_base::out | ios_base::trunc);
void close();
```

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

int main () {
  ofstream myfile ("example.txt");
  if (myfile.is_open()){
    myfile << "This is a line.\n";
    myfile << "This is another line.\n";
    myfile.close();
  }
  else
    cout << "Unable to open file";
  return 0;
}
```

ios_base::openmode

- **app**, poziționare la sfârșit înainte de fiecare inserție
- **ate**, poziționare la sfârșit la deschiderea fișierului
- **binary**, citirea fișierului în mod binar și nu în mod text
- **in**, permite extragerea (fișier de intrare)
- **out**, permite inserția (fișier de ieșire)
- **trunc**, trunchiază un fișier existent la prima creare a fluxului ce-l controlează