Facultatea de Matematică şi Informatică
Universitatea din Bucureşti

Programare orientată pe obiecte

- suport de curs -

Andrei Păun
Anca Dobrovăț

An universitar 2019 – 2020
Semestrul II
Seriile 13, 14 şi 21

Curs 12 & 13

13-20/05/2020

Agenda cursului

1. Recapitulare Șabloane în C++ (Templates)
2. Pointeri şi referințe
3. Const
4. Volatile
5. Static

0. Polimorfismul la execuţie prin funcţii virtuale

Funcţii virtuale

Funcţiile virtuale şi felul lor de folosire: componentă IMPORTANTĂ a limbajului OOP.

Folosit pentru polimorfism la execuţie ---> cod mai bine organizat cu polimorfism.

Codul poate "creşte" fără schimbări semnificative: programe extensibile.

Funcţiile virtuale sunt definite în bază şi redefinite în clasa derivată.

Pointer de tip bază care arată către obiect de tip derivatşi cheamă o funcţie virtuala în bază şi redefinite în clasa derivată executa ***Funcţia din clasa derivată***.

Poate fi vazuta ca exemplu de separare dintre interfata si implementare.

1. Şabloane (Templates) în C++

Funcții generice

Mulți algoritmi sunt generici (nu contează pe ce tip de date operează).

Înlăturam bug-uri şi mărim viteza implementării dacă reușim să refolosim aceeași implementare pentru un algoritm care trebuie folosit cu mai mute tipuri de date.

O singură implementare, mai multe folosiri.
O funcție generică face auto overload (pentru diverse tipuri de date).

Sintaxa:

```
template <class Ttype> tip_returnat nume_funcţie(listă_de_argumente) {
// corpul funcției
}
```

Ttype este un nume pentru tipul de date folosit (încă indecis), compilatorul îl va înlocui cu tipul de date folosit.

1. Şabloane (Templates) în C++

Funcții generice

```
template <class Ttype> // e ok şi template <typename Ttype>
Ttype maxim (Ttype V[ ], int n) {
        Ttype max = V[0];
        for (int i = 1; i < n; i++) {
           if (max < V[i]) max = V[i];
        }
        return max;
}

int main ()
{
        int VI[] = {1, 5, 3, 7, 3};
        float VF[] = {(float)1.1, (float)5.1, (float)3.1, (float)4.1};
        cout << "maxim (VI): " << maxim<int> (VI, sizeof (VI)/sizeof (int))<< endl;
        cout << "maxim (VF): " << maxim<float> (VF, sizeof (VF)/ sizeof (double)) << endl;
}
```

1. Şabloane (Templates) în C++

Funcții generice

Putem avea funcții cu mai mult de un tip generic.
-compilatorul creează atâtea funcții cu același nume câte sunt necesare (d.p.d.v. al parametrilor folosiți).

```
template <class type1, class type2>
void myfunc(type1 x, type2 y)
 {
        cout << x << ' ' << y << '\n';
}

int main()
{
        myfunc(10, "I like C++");
        myfunc(98.6, 19L);
        return 0;
}
```

1. Şabloane (Templates) în C++

Funcții generice

Overload pe șabloane - Specializare explicită

```
template <class T> T maxim( T a, T b)
{
        cout<<"template"<<endl;
        if (a>b) return a; // operatorul < trebuie să fie definit pentru tipul T
        return b;
}
```

```
template < > char * maxim ( char* a, char* b)
{
        cout<<"supraîncărcare neconst"<<endl;
        if (strcmp(a,b)>0) return a;
        return b;
}
```

1. Şabloane (Templates) în C++

Funcții generice

Overload pe șabloane - Specializare explicită

```
template <class T> T maxim( T a, T b)
{       cout<<"template"<<endl;
        if (a>b) return a; // operatorul < trebuie să fie definit pentru tipul T
        return b;
}

template <> const char * maxim(const char* a,const char* b)
{       cout<<"supraîncărcare const"<<endl;
        if (strcmp(a,b)>0) return a;
return b; }

template <> char * maxim ( char* a, char* b)
{       cout<<"supraîncărcare neconst"<<endl;
        if (strcmp(a,b)>0) return a;
        return b; }
```

1. Şabloane (Templates) în C++

Funcții generice

Overload pe șabloane

Diferită de specializare explicită

Similar cu overload pe funcții (doar că acum sunt funcții generice)

Simplu: la fel ca la funcțiile normale

1. Şabloane (Templates) în C++

Funcții generice

Overload pe șabloane

```
// First version of f() template.
template <class X> void f(X a) {
        cout << "Inside f(X a)\n";
}

// Second version of f() template.
template <class X, class Y> void f(X a, Y b) {
        cout << "Inside f(X a, Y b)\n";
}

int main() {
        f(10); // calls f(X)
        f(10, 20); // calls f(X, Y)
        return 0;
}
```

1. Şabloane (Templates) în C++

Funcții generice

Overload pe șabloane - ce funcție se apelează (ordinea de alegere)

pas 1 potrivire FĂRĂ CONVERSIE

-prioritate varianta non-template,
-apoi template fără parametri,
-apoi template cu 1 parametru ,
-apoi template cu mai mulți parametrii

pas 2 dacă nu există potrivire exactă

- conversie DOAR la varianta non-template

1. Şabloane (Templates) în C++

Clase generice

Șabloane pentru clase nu pentru funcții.

Clasa conține toți algoritmii necesari să lucreze pe un anumit tip de date.

Din nou algoritmii pot fi generalizați, șabloane.

Specificăm tipul de date pe care lucrăm când obiectele din clasa respectivă sunt create.

Funcțiile membru ale unei clase generice sunt şi ele generice (în mod automat).

Nu e necesar să le specificăm cu template.

1. Şabloane (Templates) în C++

Clase generice

```
template <class T>
class vector
{
        int dim;
        T v[100];
  public:
        void citire();
        void afisare();
};

template <class T>
void vector<T>::citire()
{
        cin>>dim;
        for(int i = 0; i<dim; i++)
            cin>>v[i];
}

template <class T>
void vector<T>::afisare()
{
        for(int i = 0; i<dim; i++)
            cout<<v[i]<<" ";
        cout<<"\n";
}

int main()
{
        vector<int> ob1;
        ob1.citire();
        ob1.afisare();
        vector<float> ob2;
        ob2.citire();
        ob2.afisare();
        return 0;
}
```

2. Şabloane (Templates) în C++

Clase generice

Șabloanele se folosesc cu operatorii suprascriși

```
class student {
        string nume;
        float vârstă;
} x,y;

template <class T> void maxim(T a, T b) {
        if (a > b) cout<<"Primul este mai mare\n";
        else cout<<"Al doilea este mai mare\n";
}

int main()
{       int a = 3, b = 7;
        maxim(a,b); // ok
        maxim(x,y); // operatorul > ar trebui definit în clasa student
}
```

1. Şabloane (Templates) în C++

Clase generice

Specializări explicite pentru clase

La fel ca la șabloanele pentru funcții

Se folosește **template** <>

1. Şabloane (Templates) în C++

Clase generice

Specializare de clasă

```
template <class T> // sau template <typename T>
class Nume {
        T x;
public:
        void set_x(T a){x = a;}
        void afis(){cout<<x;}
};

template <>
class Nume<unsigned> {
        unsigned x;
public:
        void set_x(unsigned a){x = a;}
        void afis(){cout<<"\nUnsigned "<<x;}
};
```

```
int main()
{
        Nume<int> m;
        m.set_x(7);
        m.afis();
        Nume<unsigned> n;
        n.set_x(100);
        n.afis();
        return 0;
}
```

1. Şabloane (Templates) în C++

Clase generice

Argumente default şi şabloane

```
template <class AType=int, int size=10>
class atype {
        AType a[size]; // size of array is passed in size
public:
        atype();
};

template <class AType, int size>
atype<AType,size>::atype() {                for(int i=0; i<size; i++) a[i] = i;    }

int main()
{
        atype<int, 100> intarray; // integer array, size 100
        atype<double> doublearray; // double array, default size 10
        atype<> defarray; // default to int array of size 10
        return 0;
}
```

2. Pointerii în C

- Recapitulare
- &, *, array
- Operatii pe pointeri: =,++,--,+int, -
- Pointeri catre pointeri, pointeri catre functii
- Alocare dinamica: malloc, free
- Diferente cu C++

2. Pointerii în C/C++

- O variabilă care ţine o adresă din memorie
- Are un tip, compilatorul știe tipul de date către care se pointează
- Operațiile aritmetice țin cont de tipul de date din memorie
- Pointer ++ == pointer+sizeof(tip)
- Definiție: tip *nume_pointer;
  - Merge si tip* nume_pointer;

Operatori pe pointeri

- *, &, schimbare de tip
- *== “la adresa”
- &==“adresa lui”

```
int i=7, *j;
j=&i;
*j=9;
```

```c
#include <stdio.h>
int main(void)
{
    double x = 100.1, y;
    int *p;
    /* The next statement causes p (which
    is an    integer pointer) to point to a
    double. */
    p = (int *)&x;
    /* The next statement does not operate
    as    expected. */ y = *p;
    printf("%f", y); /* won't output 100.1 */
    return 0;
}
```

- Schimbarea de tip nu e controlată de compilator
- In C++ conversiile trebuiesc făcute cu schimbarea de tip

Aritmetica pe pointeri

- pointer++; pointer--;
- pointer+7;
- pointer-4;
- pointer1-pointer2; întoarce un întreg
- comparații: <,>,==, etc.

pointeri şi array-uri

- numele array-ului este pointer
- lista[5]==*(lista+5)

- array de pointeri, numele listei este un pointer către pointeri (dublă indirectare)
- int **p; (dublă indirectare)

alocare dinamică

- void *malloc(size_t bytes);
  - alocă în memorie dinamic bytes şi întoarce pointer către zona respectivă

```
char *p;
p=malloc(100);
```

întoarce null dacă alocarea nu s-a putut face

pointer void* este convertit AUTOMAT la orice tip

- diferența la C++: trebuie să se facă schimbare de tip dintre void* in tip*

p=(char *) malloc(100);

sizeof: a se folosi pentru portabilitate

a se verifica dacă alocarea a fost fără eroare (dacă se întoarce null sau nu)

if (!p) …

eliberarea de memorie alocată dinamic

void free(void *p);

unde p a fost alocat dinamic cu malloc()

a nu se folosi cu argumentul p invalid pentru că rezultă probleme cu lista de alocare dinamică

C++: Array-uri de obiecte

- o clasă de un tip
- putem crea array-uri cu date de orice tip (inclusiv obiecte)
- se pot defini neinițializate sau inițializate

clasa lista[10];

sau

clasa lista[10]={1,2,3,4,5,6,7,8,9,0};

pentru cazul inițializat dat avem nevoie de constructor care primește un parametru întreg.

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

class cl {
    int i;
public:
    cl(int j) { i=j; } // constructor
    int get_i() { return i; }
};

int main()
{
    cl ob[3] = {1, 2, 3}; // initializers
    int i;

    for(i=0; i<3; i++)
        cout << ob[i].get_i() << "\n";
    return 0;
}
```

- inițializare pentru constructori cu mai mulți parametri

clasa lista[3]={clasa(1,5), clasa(2,4), clasa(3,3)};

- pentru definirea listelor de obiecte neinițializate: constructor fără parametri

- dacă in program vrem şi inițializare şi neiniţializare: overload pe constructor (cu şi fără parametri)

pointeri către obiecte

- obiectele sunt în memorie
- putem avea pointeri către obiecte
- &obiect;
- accesarea membrilor unei clase:

  -> in loc de .

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

class cl {
    int i;
public:
    cl(int j) { i=j; }
    int get_i() { return i; }
};

int main()
{
    cl ob(88), *p;
    p = &ob; // get address of ob
    cout << p->get_i(); // use -> to call
    get_i()
    return 0;
}
```

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

class cl {
public:
    int i;
    cl(int j) { i=j; }
};

int main()
{
    cl ob(1);
    int *p;
    p = &ob.i; // get address of ob.i
    cout << *p; // access ob.i via p
    return 0;
}
```

- în C++ tipurile pointerilor trebuie să fie la fel

```
int *p;
float *q;
p=q; //eroare
```

se poate face cu schimbarea de tip (type casting) dar ieșim din verificările automate făcute de C++

pointerul **this**

- orice funcție membru are pointerul **this** (definit ca argument implicit) care arată către obiectul asociat cu funcția respectivă
- (pointer către obiecte de tipul clasei)
- funcțiile prieten nu au pointerul this
- funcțiile statice nu au pointerul this

```
#include <iostream>
using namespace std;
class pwr {
   double b;
   int e;
   double val;
public:
   pwr(double base, int exp);
   double get_pwr() { return this->val; }
};

pwr::pwr(double base, int exp)
{
   this->b = base;
   this->e = exp;
   this->val = 1;
   if(exp==0) return;
   for( ; exp>0; exp--) this->val = this->val * this->b;
}
```

```
int main()
{
    pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);
    cout << x.get_pwr() << " ";
    cout << y.get_pwr() << " ";
    cout << z.get_pwr() << "\n";
    return 0;
}
```

```
pwr::pwr(double base, int exp)
{
    b = base;
    e = exp;
    val = 1;
   if(exp==0) return;
   for( ; exp>0; exp--) val = val * b;
}
```

pointeri către clase derivate

- clasa de bază B şi clasa derivată D
- un pointer către B poate fi folosit şi cu D;

```
B *p, o(1);
D oo(2);
p=&o;
p=&oo;
```

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

class base {
   int i;
public:
   void set_i(int num) { i=num; }
   int get_i() { return i; }
};

class derived: public base {
   int j;
public:
   void set_j(int num) { j=num; }
   int get_j() { return j; }
};
```

```
int main()
{
   base *bp;
   derived d;
   bp = &d; // base pointer points to derived object
            // access derived object using base
pointer
   bp->set_i(10);
   cout << bp->get_i() << " ";
/* The following won't work. You can't access
elements of a derived class using a base class
pointer.

   bp->set_j(88); // error
   cout << bp->get_j(); // error
*/
((derived *)bp)->set_j(88);
cout << ((derived *)bp)->get_j();

return 0;
}
```

pointeri către clase derivate

- de ce merge şi pentru clase derivate?
  - pentru că acea clasă derivată funcționează ca şi clasa de bază plus alte detalii
- aritmetica pe pointeri: nu funcționează dacă incrementăm un pointer către bază şi suntem în clasa derivată
- se folosesc pentru polimorfism la execuție (funcții virtuale)

```
// This program contains an error.
#include <iostream>
using namespace std;

class base {
  int i;
public:
  void set_i(int num) { i=num; }
  int get_i() { return i; }
};

class derived: public base {
  int j;
public:
  void set_j(int num) {j=num;}
  int get_j() {return j;}
};
```

```
int main()
{
  base *bp;
  derived d[2];
  bp = d;
  d[0].set_i(1);
  d[1].set_i(2);
  cout << bp->get_i() << " ";
  bp++; // relative to base, not derived
  cout << bp->get_i(); // garbage value
displayed
  return 0;
}
```

pointeri către membri în clase

- pointer către membru
- nu sunt pointeri normali (către un membru dintr-un obiect) ci specifică un offset în clasă
- nu putem să aplicăm . şi ->
- se folosesc .* şi ->*

```
#include <iostream>
using namespace std;
class cl {
public:
   cl(int i) { val=i; }
   int val;
   int double_val() { return val+val; }
};

int main()
{
   int cl::*data; // data member pointer
   int (cl::*func)(); // function member pointer
   cl ob1(1), ob2(2); // create objects
   data = &cl::val; // get offset of val
   func = &cl::double_val; // get offset of
double_val()
   cout << "Here are values: ";
   cout << ob1.*data << " " << ob2.*data << "\n";
   cout << "Here they are doubled: ";
   cout << (ob1.*func)() << " ";
   cout << (ob2.*func)() << "\n";
   return 0;
}
```

```
#include <iostream>
using namespace std;
class cl {
public: cl(int i) { val=i; }
        int val;
        int double_val() { return val+val;
}};

int main(){
     int cl::*data; // data member pointer
     int (cl::*func)(); // function member
pointer
     cl ob1(1), ob2(2), *p1, *p2;
     p1 = &ob1; // access objects through a
pointer
     p2 = &ob2;
     data = &cl::val; // get offset of val
     func = &cl::double_val;
     cout << "Here are values: ";
     cout << p1->*data << " " << p2->*data <<
"\n";
     cout << "Here they are doubled: ";
     cout << (p1->*func)() << " ";
    cout << (p2->*func)() << "\n";
    return 0;
}
```

```
int cl::*d;
int *p;
cl o;

p = &o.val // this is address of a
specific val
d = &cl::val // this is offset of generic
val
```

- pointeri la membri nu sunt folositi decat rar in cazuri speciale

parametri referință

- nou la C++
- la apel prin valoare se adaugă şi apel prin referință la C++
- nu mai e nevoie să folosim pointeri pentru a simula apel prin referință, limbajul ne dă acest lucru
- sintaxa: în funcție & înaintea parametrului formal

```
// Manually: call-by-reference using a
pointer.

#include <iostream>
using namespace std;

void neg(int *i);

int main()
{
   int x;
   x = 10;
   cout << x << " negated is ";
   neg(&x);
   cout << x << "\n";
   return 0;
}

void neg(int *i)
{
   *i = -*i;
}
```

```
// Use a reference parameter.
#include <iostream>
using namespace std;

void neg(int &i); // i now a reference

int main()
{
   int x;
   x = 10;
   cout << x << " negated is ";
   neg(x); // no longer need the &
operator
   cout << x << "\n";
   return 0;
}

void neg(int &i)
{
   i = -i; // i is now a reference,
don't need *
}
```

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

void swap(int &i, int &j);

int main()
{    int a, b, c, d;
      a = 1;       b = 2;       c = 3;       d = 4;
     cout << "a and b: " << a << " " << b << "\n";
     swap(a, b); // no & operator needed
     cout << "a and b: " << a << " " << b << "\n";
    cout << "c and d: " << c << " " << d << "\n";
    swap(c, d);
    cout << "c and d: " << c << " " << d << "\n";
   return 0;
}

void swap(int &i, int &j)
{     int t;
      t = i; // no * operator needed
      i = j;
      j = t;
}
```

referinte catre obiecte

- daca transmitem obiecte prin apel prin referinta la functii nu se mai creeaza noi obiecte temporare, se lucreaza direct pe obiectul transmis ca parametru
- deci copy-constructorul si destructorul nu mai sunt apelate
- la fel si la intoarcerea din functie a unei referinte

```
#include <iostream>
using namespace std;
class cl {
   int id;
public:
   int i;
   cl(int i);
   ~cl(){ cout << "Destructing " << id << "\n"; }
   void neg(cl &o) { o.i = -o.i; } // no temporary created
};

cl::cl(int num)
{       cout << "Constructing " << num << "\n";
         id = num;
}

int main()
{    cl o(1);
     o.i = 10;
     o.neg(o);
   cout << o.i << "\n";
   return 0;
}
```

```
Constructing 1
-10
Destructing 1
```

intoarcere de referi

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

char &replace(int i); // return a reference

char s[80] = "Hello There";

int main()
{
    replace(5) = 'X'; // assign X to space
after Hello
    cout << s;
    return 0;
}

char &replace(int i)
{
    return s[i];
}
```

- putem face atribuiri catre apel de functie
- replace(5) este un element din s care se schimba
- e nevoie de atentie ca obiectul referit sa nu iasa din scopul de vizibilitate

referinte independente

- nu e asociat cu apelurile de functii
- se creeaza un alt nume pentru un obiect
- referintele independente trebuiesc initializate la definire pentru ca ele nu se schimba in timpul programului

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

int main()
{
    int a;
    int &ref = a; // independent reference
    a = 10;
    cout << a << " " << ref << "\n";
    ref = 100;
    cout << a << " " << ref << "\n";
    int b = 19;
    ref = b; // this puts b's value into a
    cout << a << " " << ref << "\n";
    ref--; // this decrements a
    // it does not affect what ref refers to
    cout << a << " " << ref << "\n";
    return 0;
}
```

```
10 10
100 100
19 19
18 18
```

referinte catre clase derivate

- putem avea referinte definite catre clasa de baza si apelata functia cu un obiect din clasa derivata
- exact la la pointeri

Alocare dinamica in C++

- new, delete
- operatori nu functii
- se pot folosi inca malloc() si free() dar vor fi deprecated in viitor

operatorii new, delete

- new: aloca memorie si intoarce un pointer la inceputul zonei respective
- delete: sterge zona respectiva de memorie

p= new tip;

delete p;

la eroare se “arunca” exceptia bad_alloc din <new>

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

int main()
{
   int *p;
   try {
        p = new int; // allocate space
 for an int
   } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
   }
   *p = 100;
   cout << "At " << p << " ";
   cout << "is the value " << *p << "\n";
   delete p;
   return 0;
}
```

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

int main()
{
   int *p;
   try {
        p = new int(100); // initialize
with 100
   } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
   }

   cout << "At " << p << " ";
   cout << "is the value " << *p <<
"\n";
   delete p;
   return 0;
}
```

alocare de array-uri

p_var = new *array_type [size];*

delete [ ] *p_var*;

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

int main()
{
    int *p, i;
    try {
        p = new int [10]; // allocate 10 integer array
    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
    }
    for(i=0; i<10; i++ )
        p[i] = i;
    for(i=0; i<10; i++)
        cout << p[i] << " ";
    delete [] p; // release the array
    return 0;
}
```

alocare de obiecte

- cu new
- dupa creare, new intoarce un pointer catre obiect
- dupa creare se executa constructorul obiectului
- cand obiectul este sters din memorie (delete) se executa destructorul

obiecte create dinamic cu constructori parametrizati

```
class balance {…}
…

balance *p;
// this version uses an initializer
try {
    p = new balance (12387.87, "Ralph Wilson");
} catch (bad_alloc xa) {
   cout << "Allocation Failure\n";
   return 1;
}
```

- array-uri de obiecte alocate dinamic
  - nu se pot initializa
  - trebuie sa existe un constructor fara parametri
  - delete poate fi apelat pentru fiecare element din array

- new si delete sunt operatori
- pot fi suprascrisi pentru o anumita clasa
- pentru argumente suplimentare exista o forma speciala
  - p_var = new (lista_argumente) tip;
- exista forma nothrow pentru new: similar cu malloc:

  p=new(nothrow) int[20]; // intoarce null la eroare

const si volatile

- idee: sa se elimine comenzile de preprocesor #define
- #define faceau substitutie de valoare
- se poate aplica la pointeri, argumente de functii, param de intoarcere din functii, obiecte, functii membru
- fiecare dintre aceste elemente are o aplicare diferita pentru const, dar sunt in aceeasi idee/filosofie

- #define BUFSIZE 100 (tipic in C)
- erori subtile datorita substituirii de text
- BUFSIZE e mult mai bun decat “valori magice”
- nu are tip, se comporta ca o variabila
- mai bine: const int bufsize = 100;

- acum compilatorul poate face calculele la inceput: “constant folding”: important pt. array: o expresie complicata e calculata la compilare

- char buf[bufsize];

- se poate face const pe: **char**, **int**, **float**,**double** si variantele lor

- se poate face const si pe obiecte

- const implica "***internal linkage***" adica e vizibila numai in fisierul respectiv (la linkare)

- trebuie data o valoare pentru elementul constant la declarare, singura exceptie:

```
extern const int bufsize;
```

- in mod normal compilatorul nu aloca spatiu pentru constante, daca e declarat ca extern aloca spatiu (sa poata fi accesat si din alte parti ale programului)

- pentru structuri complicate folosite cu const se aloca spatiu: nu se stie daca se aloca sau nu spatiu si atunci const impune localizare (sa nu existe coliziuni de nume)

- de aceea avem "***internal linkage***"

```
// Using const for safety
#include <iostream>
using namespace std;

const int i = 100;  // Typical constant
const int j = i + 10; // Value from const expr
long address = (long)&j; // Forces storage
char buf[j + 10]; // Still a const expression

int main() {
  cout << "type a character & CR:";
  const char c = cin.get(); // Can't change
  const char c2 = c + 'a';
  cout << c2;
  // ...
}
```

- daca stim ca variabila nu se schimba sa o declaram cu const
- daca incercam sa o schimbam primim eroare de compilare

- const poate elimina memorie si acces la memorie
- const pentru agregate: aproape sigur compilatorul aloca memorie
- nu se pot folosi valorile la compilare

```
// Constants and aggregates
const int i[] = { 1, 2, 3, 4 };

//! float f[i[3]]; // Illegal

struct S { int i, j; };
const S s[] = { { 1, 2 }, { 3, 4 } };

//! double d[s[1].j]; // Illegal

int main() {}
```

diferente cu C

- const in C: o variabila globala care nu se schimba

- deci nu se poate considera ca valoare la compilare

```
const int bufsize = 100;
char buf[bufsize];
```

eroare in C

- in C se poate declara cu

const int bufsize;

- in C++ nu se poate asa, trebuie extern:

extern const int bufsize;

- diferenta:
  - C external linkage
  - C++ internal linkage

- in C++ compilatorul incearca sa nu creeze spatiu pentru const-uri, daca totusi se transmite catre o functie prin referinta, extern etc atunci se creeaza spatiu

- C++: const in afara tuturor functiilor: scopul ei este doar in fisierul respectiv: internal linkage,

- alti identificatori declarati in acelasi loc (fara const) EXTERNAL LINKAGE

pointeri const

- const poate fi aplicat valorii pointerului sau elementului pointat

- const se alatura elementului cel mai apropiat

const int* u;

- u este pointer catre un int care este const

int const* v; la fel ca mai sus

pointeri constanti

- pentru pointeri care nu isi schimba adresa din memorie

```
int d = 1;
int* const w = &d;
```

- w e un pointer constant care arata catre intregi+initializare

const pointer catre const element

```
int d = 1;
const int* const x = &d; // (1)
int const* const x2 = &d; // (2)
```

```
//: C08:ConstPointers.cpp
const int* u;
int const* v;

int d = 1;

int* const w = &d;

const int* const x = &d; // (1)
int const* const x2 = &d; // (2)

int main() {} ///:~
```

- se poate face atribuire de adresa pentru obiect non-const catre un pointer const

- nu se poate face atribuire pe adresa de obiect const catre pointer non-const

```
int d = 1;

const int e = 2;

int* u = &d; // OK -- d not const

//! int* v = &e; // Illegal -- e const

int* w = (int*)&e; // Legal but bad practice

int main() {} ///:~
```

constante caractere

char* cp = "howdy";

daca se inceraca schimbarea caracterelor din “howdy” compilatorul ar trebui sa genereze eroare; nu se intampla in mod uzual (compatibilitate cu C)

mai bine: char cp[] = "howdy";

si atunci nu ar mai trebui sa fie probleme

argumente de functii, param de intoarcere

- apel prin valoare cu const: param formal nu se schimba in functie
- const la intoarcere: valoarea returnata nu se poate schimba
- daca se transmite o adresa: promisiune ca nu se schimba valoarea la adresa

```
void f1(const int i) {
i++; // Illegal -- compile-time error
}
```

cod mai clar echivalent mai jos:

```
void f2(int ic) {
const int& i = ic;
i++; // Illegal -- compile-time error
}
```

```
// Returning consts by value
// has no meaning for built-in types

int f3() { return 1; }
const int f4() { return 1; }

int main() {
  const int j = f3(); // Works fine
  int k = f4(); // But this works fine too!
}
```

```
// Constant return by value
// Result cannot be used as an lvalue

class X {   int i;
public:   X(int ii = 0);
void modify();
};

X::X(int ii) { i = ii; }
void X::modify() { i++; }

X f5() {   return X(); }
const X f6() {   return X(); }

void f7(X& x) { // Pass by non-const
reference
  x.modify();
}

int main() {
  f5() = X(1); // OK -- non-const return
value
  f5().modify(); // OK
// Causes compile-time errors:
//!  f7(f5());
//!  f6() = X(1);
//!  f6().modify();
//!  f7(f6());
} ///:~
```

- f7() creeaza obiecte temporare, de accea nu compileaza
- aceste obiecte au constructor si destructor dar pentru ca nu putem sa le “atingem” sunt definite ca si obiecte constante
- f7(f5()); se creeaza ob. temporar pentru rezultatul lui f5(); si apoi apel prin referinta la f7
- ca sa compileze (dar cu erori mai tarziu) trebuie apel prin referinta const

- f5() = X(1);
- f5().modify();

- compileaza fara probleme, dar procesarea se face pe obiectul temporar (modificarile se pierd imediat, deci aproape sigur este bug)

parametrii de intrare si iesire: adrese

- e preferabil sa fie definiti ca const
- in felul asta pointerii si referintele nu pot fi modificate

```
// Constant pointer arg/return

void t(int*) {}

void u(const int* cip) {
//!  *cip = 2; // Illegal -- modifies value
  int i = *cip; // OK -- copies value
//!  int* ip2 = cip; // Illegal: non-const
}

const char* v() {
  // Returns address of static character
array:
  return "result of function v()";
}

const int* const w() {
  static int i;
  return &i;
}
```

```
int main() {
  int x = 0;
  int* ip = &x;
  const int* cip = &x;
  t(ip);  // OK
//!  t(cip); // Not OK
  u(ip);  // OK
  u(cip); // Also OK
//!  char* cp = v(); // Not OK
  const char* ccp = v(); // OK
//!  int* ip2 = w(); // Not OK
  const int* const ccip = w(); // OK
  const int* cip2 = w(); // OK
//!  *w() = 1; // Not OK
} ///:~
```

- cip2 nu schimba adresa intoarsa din w (pointerul contant care arata spre constanta) deci e ok; urmatoarea linie schimba valoarea deci compilatorul intervine

comparatii cu C

- in C daca vrem param. o adresa: se face pointer la pointer
- in C++ nu se incurajeaza acest lucru: const referinta
- pentru apelant e la fel ca apel prin valoare
  - nici nu trebuie sa se gandeasca la pointeri
  - trimiterea unei adrese e mult mai eficienta decat transmiterea obiectului prin stiva, se face const deci nici nu se modifica

Ob. temporare sunt const

```
//: C08:ConstTemporary.cpp
// Temporaries are const

class X {};

X f() { return X(); } // Return by value

void g1(X&) {} // Pass by non-const reference
void g2(const X&) {} // Pass by const reference

int main() {
  // Error: const temporary created by f():
//!  g1(f());
  // OK: g2 takes a const reference:
  g2(f());
} ///:~
```

- In C avem pointeri deci e OK

Const in clase

- const pentru variabile de instanta si
- functii de instanta de tip const

- sa construim un vector pentru clasa respectiva, in C folosim #define
- problema in C: coliziune pe nume

- in C++: punem o variabila de instanta const
- problema: toate obiectele au aceasta variabila, si putem avea chiar valori diferite (depinde de initializare)
- cand se creeaza un const intr-o clasa nu se poate initializa (constructorul initializeaza)
- in constructor trebuie sa fie deja initializat (altfel am putea sa il schimbam in constructor)

- initializare de variabile const in obiecte: lista de initializare a constructorilor

```
// Initializing const in classes
#include <iostream>
using namespace std;

class Fred {
  const int size;
public:
  Fred(int sz);
  void print();
};

Fred::Fred(int sz) : size(sz) {}
void Fred::print() { cout << size << endl; }

int main() {
  Fred a(1), b(2), c(3);
  a.print(), b.print(), c.print();
} ///:~
```

rezolvarea problemei initiale

- cu static
  - insemna ca nu e decat un singur asemenea element in clasa
  - il facem static const si devine similar ca un const la compilare
  - static const trebuie initializat la declarare (nu in constructor)

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

class StringStack {
  static const int size = 100;
  const string* stack[size];
  int index;
public:
  StringStack();
  void push(const string* s);
  const string* pop();
};

StringStack::StringStack() : index(0) {
  memset(stack, 0, size * sizeof(string*));}

void StringStack::push(const string* s) {
  if(index < size)
    stack[index++] = s;}

const string* StringStack::pop() {
  if(index > 0) {
    const string* rv = stack[--index];
    stack[index] = 0;
    return rv;   }
  return 0;}
```

```
string iceCream[] = {
  "pralines & cream",
  "fudge ripple",
  "jamocha almond fudge",
  "wild mountain blackberry",
  "raspberry sorbet",
  "lemon swirl",
  "rocky road",
  "deep chocolate fudge"
};

const int iCsz =
  sizeof iceCream / sizeof *iceCream;

int main() {
  StringStack ss;
  for(int i = 0; i < iCsz; i++)
    ss.push(&iceCream[i]);
  const string* cp;
  while((cp = ss.pop()) != 0)
    cout << *cp << endl;
}
```

enum hack

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

class Bunch {
  enum { size = 1000 };
  int i[size];
};

int main() {
  cout << "sizeof(Bunch) = " << sizeof(Bunch)
       << ", sizeof(i[1000]) = "
       << sizeof(int[1000]) << endl;
}
```

- in cod vechi
- a nu se folosi cu C++ modern
- static const int size=1000;

obiecte const si functii membru const

- obiecte const: nu se schimba
- pentru a se asigura ca starea obiectului nu se schimba functiile de instanta apelabile trebuie definite cu const
- declararea unei functii cu const nu garanteaza ca nu modifica starea obiectului!

functii membru const

- compilatorul si linkerul cunosc faptul ca functia este const
- se verifica acest lucru la compilare
- nu se pot modifica parti ale obiectului in aceste functii
- nu se pot apela functii non-const

```
//: C08:ConstMember.cpp
class X {
  int i;
public:
  X(int ii);
  int f() const;
};

X::X(int ii) : i(ii) {}
int X::f() const { return i; }

int main() {
  X x1(10);
  const X x2(20);
  x1.f();
  x2.f();
} ///:~
```

- toate functiile care nu modifica date sa fie declarate cu const

- ar trebui ca “default” pentru functiile membru sa fie de tip const

```
#include <iostream>
#include <cstdlib> // Random number generator
#include <ctime> // To seed random generator

using namespace std;

class Quoter {
    int lastquote;
public:
    Quoter();
    int lastQuote() const;
    const char* quote();
};

Quoter::Quoter(){   lastquote = -1;
  srand(time(0)); // Seed random number
generator
}

int Quoter::lastQuote() const {  return
lastquote;}
```

```
const char* Quoter::quote() {
  static const char* quotes[] = {
    "Are we having fun yet?",
    "Doctors always know best",
    "Is it ... Atomic?",
    "Fear is obscene",
    "There is no scientific evidence "
    "to support the idea "
    "that life is serious",
    "Things that make us happy, make us wise",
  };
const int qsize = sizeof quotes/sizeof *quotes;
  int qnum = rand() % qsize;
  while(lastquote >= 0 && qnum == lastquote)
    qnum = rand() % qsize;
  return quotes[lastquote = qnum];
}

int main() {
  Quoter q;
  const Quoter cq;
  cq.lastQuote(); // OK
//!  cq.quote(); // Not OK; non const function
  for(int i = 0; i < 20; i++)
    cout << q.quote() << endl;
} ///:~
```

schimbari in obiect din functii const

- “casting away constness”
- se face castare a pointetrului this la pointer catre tipul de obiect
- pentru ca in functii const este de tip clasa const*
- dupa aceasta schimbare de tip se modifica prin pointerul this

```
// "Casting away" constness

class Y {
  int i;
public:
  Y();
  void f() const;
};

Y::Y() { i = 0; }

void Y::f() const {
//!  i++; // Error -- const member function
  ((Y*)this)->i++; // OK: cast away const-
ness
  // Better: use C++ explicit cast syntax:
  (const_cast<Y*>(this))->i++;
}

int main() {
  const Y yy;
  yy.f(); // Actually changes it!
} ///:~
```

- apare in cod vechi
- nu e ok pentru ca functia modifica si noi credem ca nu modifica
- o metoda mai buna: in continuare

```
// The "mutable" keyword

class Z {
  int i;
  mutable int j;
public:
  Z();
  void f() const;
};

Z::Z() : i(0), j(0) {}

void Z::f() const {
//! i++; // Error -- const member function
    j++; // OK: mutable
}

int main() {
  const Z zz;
  zz.f(); // Actually changes it!
} ///:~
```

volatile

- e similar cu const
- obiectul se poate schimba din afara programului
- multitasking, multithreading, intreruperi
- nu se fac optimizari de cod
- avem obiecte volatile, functii volatile, etc.

static

- ceva care isi tine pozitia neschimbata
- alocare statica pentru variabile
- vizibilitate locala a unui nume

- variabile locale statice
- isi mentin valorile intre apelari
- initializare la primul apel

```
#include "../require.h"
#include <iostream>
using namespace std;

char oneChar(const char* charArray = 0) {
  static const char* s;
  if(charArray) {
    s = charArray;
    return *s;
  }
  else
    require(s, "un-initialized s");
  if(*s == '\0')
    return 0;
  return *s++;
}

char* a = "abcdefghijklmnopqrstuvwxyz";

int main() {
  // oneChar(); // require() fails
  oneChar(a); // Initializes s to a
  char c;
  while((c = oneChar()) != 0)
    cout << c << endl;
} ///:~
```

obiecte statice

- la fel ca la tipurile predefinite
- avem nevoie de constructorul predefinit

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

class X {
  int i;
public:
  X(int ii = 0) : i(ii) {} // Default
  ~X() { cout << "X::~X()" << endl; }
};

void f() {
  static X x1(47);
  static X x2; // Default constructor
required
}

int main() {
  f();
} ///:~
```

destructori statici

- cand se termina main se distrug obiectele
- deobicei se cheama exit() la iesirea din main
- daca se cheama exit() din destructor e posibil sa avem ciclu infinit de apeluri la exit()
- destructorii statici nu sunt executati daca se iese prin abort()

- daca avem o functie cu obiect local static
- si functia nu a fost apelata, nu vrem sa apelam destructorul pentru obiect neconstruit
- C++ tine minte ce obiecte au fost construite si care nu

```
#include <fstream>
using namespace std;
ofstream out("statdest.out"); // Trace file

class Obj {
  char c; // Identifier
public:
  Obj(char cc) : c(cc) {
    out << "Obj::Obj() for " << c << endl;
  }
  ~Obj() {
    out << "Obj::~Obj() for " << c << endl;
  }
};

Obj a('a'); // Global (static storage)
// Constructor & destructor always called

void f() {
  static Obj b('b');
}

void g() {
  static Obj c('c');
}
```

```
int main() {
  out << "inside main()" << endl;
  f(); // Calls static constructor for b
  // g() not called
  out << "leaving main()" << endl;
} ///:~
```

Obj::Obj() for a

inside main()

Obj::Obj() for b

leaving main()

Obj::~Obj() for b

Obj::~Obj() for a

static pentru nume (la linkare)

- orice nume care nu este intr-o clasa sau functie este vizibil in celalalte parti ale programului (external linkage)

- daca e definit ca static are internal linkage: vizibil doar in fisierul respectiv

- linkarea e valabila pentru elemente care au adresa (clase, var. locale nu au)

- int a=0;
- in afara claselor, functiilor: este var globala, vizibila pretutindeni
- similar cu: extern int a=0;
- static int a=0; // internal linkage
- nu mai e vizibila pretutindeni, doar local in fisierul respectiv

```
//{L} LocalExtern2
#include <iostream>

int main() {
  extern int i;
  std::cout << i;
} ///:~

//: C10:LocalExtern2.cpp {O}
int i = 5;
///:~
```

functii extern si static

- schimba doar vizibilitatea
- void f(); similar cu extern void f();

- restrictiv:
  - static void f();

- alti specificatori:
  - auto: aproape nefolosit; spune ca e var. locala
  - register: sa se puna intr-un registru

variabile de instanta statice

- cand vrem sa avem valori comune pentru toate obiectele
- static

```
class A {
  static int i;
public:
  //...
};
int A::i = 1;
```

- int A::i = 1;
- se face o singura data
- e obligatoriu sa fie facut de creatorul clasei, deci e ok

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

int x = 100;

class WithStatic {
  static int x;
  static int y;
public:
  void print() const {
    cout << "WithStatic::x = " << x << endl;
    cout << "WithStatic::y = " << y << endl;
  }
};

int WithStatic::x = 1;
int WithStatic::y = x + 1;
// WithStatic::x NOT ::x

int main() {
  WithStatic ws;
  ws.print();
}
```

```
// Static members & local classes
#include <iostream>
using namespace std;

// Nested class CAN have static data members:
class Outer {
  class Inner {
    static int i; // OK
  };
};

int Outer::Inner::i = 47;

// Local class cannot have static data members:
void f() {
  class Local {
  public:
//! static int i;  // Error
    // (How would you define i?)
  } x;
}

int main() { Outer x; f(); } ///:~
```

functii membru statice

- nu sunt asociate cu un obiect, nu au this

```
class X {
public:
  static void f(){};
};

int main() {
  X::f();
} ///:~
```

Despre examen

- Descrieţi pe scurt funcţiile şablon (template).

```
#include <iostream.h>
class problema
{   int i;
   public: problema(int j=5){i=j;}
               void schimba(){i++;}
               void afiseaza(){cout<<"starea
curenta "<<i<<"\n";}
};
problema mister1() { return problema(6);}
void mister2(problema &o)
{   o.afiseaza();
   o.schimba();
   o.afiseaza();
}
int main()
{   mister2(mister1());
   return 0;
}
```

```
#include<iostream.h>
class B
{ int i;
  public: B() { i=1; }
          virtual int get_i() { return i; }
};
class D: virtual public B
{ int j;
  public: D() { j=2; }
          int get_i() {return B::get_i()+j; }
};
class D2: virtual public B
{ int j2;
  public: D2() { j2=3; }
  int get_i() {return B::get_i()+j2; } };

class MM: public D, public D2
{ int x;
  public: MM() { x=D::get_i()+D2::get_i(); }
          int get_i() {return x; } };

int main()
{ B *o= new MM();
  cout<<o->get_i()<<"\n";
  MM *p= dynamic_cast<MM*>(o);
  if (p) cout<<p->get_i()<<"\n";
  D *p2= dynamic_cast<D*>(o);
  if (p2) cout<<p2->get_i()<<"\n";
  return 0;
}
```

```
#include <iostream.h>
#include <typeinfo>
class B
{ int i;
  public: B() { i=1; }
          int get_i() { return i; }
};

class D: B
{ int j;
  public: D() { j=2; }
          int get_j() {return j; }
};

int main()
{ B *p=new D;
  cout<<p->get_i();
  if (typeid((B*)p).name()=="D*")
cout<<((D*)p)->get_j();
  return 0;
}
```

```
#include<iostream.h>
template<class T, class U>
T f(T x, U y)
{ return x+y;
}
int f(int x, int y)
{ return x-y;
}
int main()
{ int *a=new int(3), b(23);
  cout<<*f(a,b);
  return 0;
}
```

```
#include<iostream.h>
class A
{ int x;
  public: A(int i=0) { x=i; }
          A operator+(const A& a) { return
x+a.x; }
          template <class T> ostream&
operator<<(ostream&); };
template <class T>
ostream& A::operator<<(ostream& o) { o<<x;
return o; }
int main()
{ A a1(33), a2(-21);
  cout<<a1+a2;
  return 0;
}
```

Perspective

Cursul 13:

Design Pattern – introducere
- Singleton