Facultatea de Matematică şi Informatică
Universitatea din Bucureşti

Programare orientată pe obiecte

- suport de curs -

Andrei Păun
Anca Dobrovăț

An universitar 2021 – 2022
Semestrul II
Seriile 13, 14 şi 15

Curs 10 & 11

Agenda cursului

- Pointeri şi referințe
- Const
- Volatile
- Static

Pointerii în C

- Recapitulare
- &, *, array
- Operații pe pointeri: =,++,--,+int, -
- Pointeri către pointeri, pointeri către functii
- Alocare dinamică: malloc, free
- Diferențe cu C++

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 şi 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
- în 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 în 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:

  -> în 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++

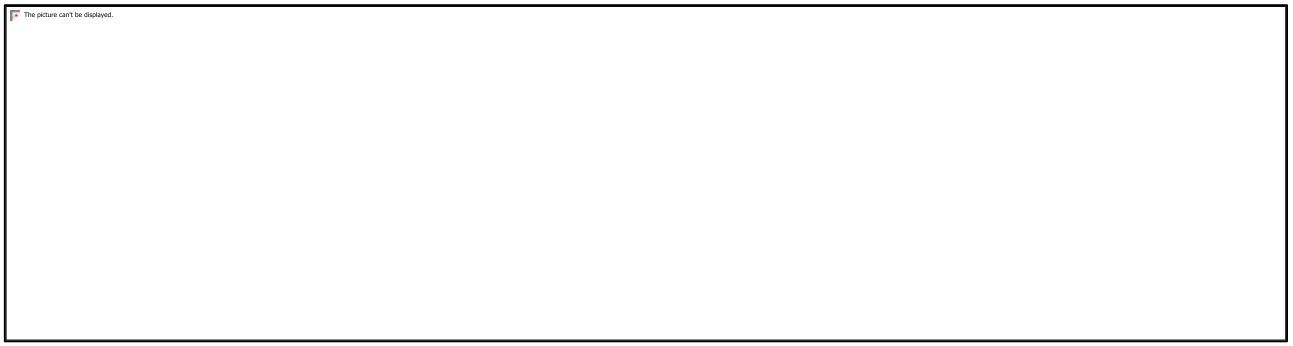
pointerul **this**

- 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;
}
```



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 { int val;
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 { int val;
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;
     cout << "\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 folosiți decât rar în 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;
}
```

referințe către obiecte

- dacă transmitem obiecte prin apel prin referință la funcții nu se mai creează noi obiecte temporare, se lucrează direct pe obiectul transmis ca parametru
- deci copy-constructorul şi destructorul nu mai sunt apelate
- la fel şi la întoarcerea din funcție a unei referințe

```
#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
```

întoarcere de referințe

```
#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 către apel de funcție
- replace(5) este un element din s care se schimbă
- e nevoie de atenție ca obiectul referit să nu iasă din scopul de vizibilitate

referințe independente

- nu e asociat cu apelurile de funcții
- se creează un alt nume pentru un obiect
- referințele independente trebuiesc inițializate la definire pentru că ele nu se schimbă în 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
```

referințe către clase derivate

- putem avea referințe definite către clasa de bază şi apelată funcția cu un obiect din clasa derivată
- exact ca la pointeri

Alocare dinamică în C++

- new, delete
- operatori nu funcții
- se pot folosi încă malloc() şi free() dar vor fi deprecated în viitor

operatorii new, delete

- new: alocă memorie şi întoarce un pointer la începutul zonei respective
- delete: sterge zona respectivă de memorie

p= new tip;

delete p;

la eroare se “aruncă” excepţia 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
- după creare, new întoarce un pointer către obiect
- după creare se execută constructorul obiectului
- când obiectul este șters din memorie (delete) se execută destructorul

obiecte create dinamic cu constructori parametrizaţi

```
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 inițializa
  - trebuie să existe un constructor fără parametri
  - delete poate fi apelat pentru fiecare element din array

- new şi delete sunt operatori
- pot fi suprascrișI pentru o anumită clasă
- pentru argumente suplimentare există o formă specială
  - p_var = new (lista_argumente) tip;
- există forma nothrow pentru new: similar cu malloc:

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

const şi volatile

- idee: să se elimine comenzile de preprocesor #define
- #define făceau substituție de valoare
- se poate aplica la pointeri, argumente de funcții, param de întoarcere din funcții, obiecte, funcții membru
- fiecare dintre aceste elemente are o aplicare diferită pentru const, dar sunt în aceeași idee/filosofie

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

- acum compilatorul poate face calculele la început: “constant folding”: important pt. array: o expresie complicată e calculată la compilare

- char buf[bufsize];

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

- se poate face const şi pe obiecte

- const implică "***internal linkage***" adică e vizibilă numai în fișierul respectiv (la linkare)

- trebuie dată o valoare pentru elementul constant la declarare, singura excepție:

```
extern const int bufsize;
```

- în mod normal compilatorul nu alocă spațiu pentru constante, dacă e declarat ca extern alocă spațiu (să poată fi accesat şi din alte părți ale programului)

- pentru structuri complicate folosite cu const se alocă spațiu: nu se știe dacă se alocă sau nu spațiu şi atunci const impune localizare (să 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 //valoarea
nu e cunoscuta la compile time si necesita storage
  const char c2 = c + 'a';
  cout << c2;
  // ...
}
```

- dacă știm că variabila nu se schimbă să o declaram cu const
- dacă încercam să o schimbăm primim eroare de compilare

- const poate elimina memorie şi acces la memorie
- const pentru agregate: aproape sigur compilatorul alocă 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() {}
```

diferenţe cu C

- const în C: o variabilă globală care nu se schimbă

- deci nu se poate considera ca valoare la compilare

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

eroare în C

- în C se poate declara cu

const int bufsize;

- în C++ nu se poate așa, trebuie extern:

extern const int bufsize;

- diferența:
  - C external linkage
  - C++ internal linkage

- în C++ compilatorul încearcă să nu creeze spațiu pentru const-uri, dacă totuși se transmite către o funcție prin referință, extern etc. atunci se creează spațiu

- C++: const în afara tuturor funcțiilor: scopul ei este doar în fișierul respectiv: internal linkage,

- alți identificatori declarați în același loc (fara const) EXTERNAL LINKAGE

pointeri const

- const poate fi aplicat valorii pointerului sau elementului pointat

- const se alatură elementului cel mai apropiat

const int* u;

- u este pointer către un int care este const

int const* v; la fel ca mai sus

pointeri constanţi

- pentru pointeri care nu își schimbă adresa din memorie

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

- w e un pointer constant care arată către întregi+inițializare

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 adresă pentru obiect non-const către un pointer const

- nu se poate face atribuire pe adresă de obiect const către 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";

dacă se încearcă schimbarea caracterelor din "howdy" compilatorul ar trebui să genereze eroare; nu se întâmplă în mod uzual (compatibilitate cu C)

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

şi atunci nu ar mai trebui să fie probleme

argumente de funcții, param de întoarcere

- apel prin valoare cu const: param formal nu se schimbă în functie
- const la întoarcere: valoarea returnată nu se poate schimba
- dacă se transmite o adresă: promisiune că nu se schimbă valoarea la adresa respectivă

```
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() creează obiecte temporare, de aceea nu compilează
- aceste obiecte au constructor şi destructor dar pentru că nu putem să le “atingem” sunt definite sub forma de obiecte constante
- f7(f5()); se creează ob. temporar pentru rezultatul lui f5(); si apoi apel prin referință la f7
- ca să compileze (dar cu erori mai târziu) trebuie apel prin referință const

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

- compilează fără probleme, dar procesarea se face pe obiectul temporar (modificările se pierd imediat, deci aproape sigur este bug)

parametrii de intrare şi iesire: adrese

- e preferabil să fie definiți ca const
- în felul acesta pointerii şi referințele nu pot fi modificați/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 schimbă adresa întoarsă din w (pointerul contant care arată spre constantă) deci e ok; următoarea linie schimbă valoarea deci compilatorul intervine

comparații cu C

- în C daca vrem param. o adresa: se face pointer la pointer
- în C++ nu se încurajează acest lucru: const referință
- pentru apelant e la fel ca apel prin valoare
  - nici nu trebuie să se gândească la pointeri
  - trimiterea unei adrese e mult mai eficientă decât transmiterea obiectului prin stivă, se face const deci nici nu se modifică

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());
} ///:~
```

- în C avem pointeri deci e OK

Const în clase

- const pentru variabile de instanță şi
- funcții de instanță de tip const

- să construim un vector pentru clasa respectivă, în C folosim #define
- problemă în C: coliziune pe nume

- în C++: punem o variabilă de instanță const
- problemă: toate obiectele au această variabilă, şi putem avea chiar valori diferite (depinde de inițializare)
- când se creează un const într-o clasă nu se poate inițializa (constructorul inițializează)
- în constructor trebuie să fie deja inițializat (altfel am putea să îl schimbăm în constructor)

- inițializare de variabile const în obiecte: lista de inițializare 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 inițiale

- cu static
  - inseamnă că nu e decât un singur asemenea element în clasă
  - îl facem static const şi devine similar ca un const la compilare
  - static const trebuie inițializat la declarare (nu în 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;
}
```

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

obiecte const şi funcții membru const

- obiecte const: nu se schimbă
- pentru a se asigura că starea obiectului nu se schimbă funcțiile de instanță apelabile trebuie definite cu const
- declararea unei funcții cu const nu garantează că nu modifică starea obiectului!

functii membru const

- compilatorul şi linkerul cunosc faptul că funcția este const
- se verifică acest lucru la compilare
- nu se pot modifica parți ale obiectului în aceste funcții
- nu se pot apela funcții 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 funcțiile care nu modifică date să fie declarate cu const

- ar trebui ca “default” funcțiile membru să 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;
} ///:~
```

schimbări în obiect din funcții const

- “casting away constness”
- se face castare a pointerului this la pointer către tipul de obiect
- pentru că în funcții const este de tip clasa const*
- după această 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 în cod vechi
- nu e ok pentru că funcția modifică şi noi credem că nu modifică
- o metodă mai bună: în 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, întreruperi
- nu se fac optimizări de cod
- avem obiecte volatile, funcții volatile, etc.

static

- ceva care își tine poziția neschimbată
- alocare statică pentru variabile
- vizibilitate locală a unui nume

- variabile locale statice
- își mențin valorile intre apelări
- inițializare 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

- când se termină main se distrug obiectele
- De obicei se cheamă exit() la ieșirea din main
- daca se cheamă exit() din destructor e posibil să avem ciclu infinit de apeluri la exit()
- destructorii statici nu sunt executați dacă se iese prin abort()

- dacă avem o funcție cu obiect local static
- şi funcția nu a fost apelată, nu vrem să apelam destructorul pentru obiect neconstruit
- C++ ţine minte care obiecte au fost construite şi 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 într-o clasă sau funcție este vizibil in celelalte parți ale programului (external linkage)

- daca e definit ca static are internal linkage: vizibil doar în fișierul respectiv

- linkarea e valabilă pentru elemente care au adresă (clase, var. locale nu au)

- int a=0;
- în afara claselor, funcțiilor: este var globală, vizibilă pretutindeni
- similar cu: extern int a=0;
- static int a=0; // internal linkage
- nu mai e vizibilă pretutindeni, doar local în fișierul respectiv

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

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

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

funcții extern şi static

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

- restrictiv:
  - static void f();

- alți specificatori:
  - auto: aproape nefolosit; spune ca e var. locală
  - register: să se pună într-un registru

variabile de instanță statice

- când vrem să 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 dată
- e obligatoriu să fie făcut 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(); } ///:~
```

funcții 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 12:

Biblioteca Standard Template Library - STL

- Containere, iteratori şi algoritmi.
- Clasele vector, list, map / multimap.
- Elemente avansate