Fundamentele C++

Dorin Mancu
dmancu@memiq.ro

memIQ
Progress by Education

www.memiq.ro

Cuprins

- Limbajul C
- Elemente C++ extra clase & obiecte
- Paradigme de programare
- Clase și obiecte
- Membrii dată
- Membrii funcție
- Constructori & destructori
- Relația friend
- Moștenire simplă

- Polimorfism
- dynamic_cast; cast-uri C++
- Moștenire multiplă
- Supraîncărcare
- Supraîncărcarea operatorilor, operatori
- Tratarea excepțiilor
- Spații de nume
- Templates
- STL – Standard Template Library

Bibliografie

- The C Programming Language, Brian W. Kernighan, Dennis M. Ritchie, ed. 2, 1988
- The C++ Programming Language – Bjarne Stroustrup, ed. 4, 2013

Limbajul C

- C - 1972
- C++ - 1983
- Exemplu:

```
#include <stdio.h>
main()
{
        printf("hello, world\n");
}
```

- Ciclul de dezvoltare al programelor C

- Tipuri de bază: char (ASCI), short, int, long, float, double
- Tipurile întregi – spațiu ocupat
- Comenzi preprocesor

```
#define MAX 100
```

- Exemplu:

```
#include <stdio.h>
main() {
        int c;
        c = getchar();
        while (c != EOF) {
                putchar(c);
                c = getchar();
        }
}
```

- Exercițiu:
  - scrieți un program C care elimină comentariile din sursele C.
  - program pentru verificarea că parantezele sunt în pereche și ordine corectă - () {} []
- Tablouri

  int t[10];
- Exercițiu: histogramă a lungimii cuvintelor
- Funcții
- Șiruri de caractere – terminare prin null
- Exercițiu: program care inversează un șir de caractere

- Variabile globale, locale
- Vizibilitatea variabilelor
- Declararea variabilelor
- bool – convenție pentru evaluarea expresiilor logice
- Exerciții:

```
int htoi(char *nrHexa);
void itoa(char *buff, int cine);
```

- Expresii, operatori
- Instrucțiuni – goto
- Surse multiple, fișiere header
- Exemplu: implementare stivă

- Static – variabile și funcții
- Variabile register, volatile
- Preprocesor C
  - #include <nume> “nume”
  - #define forever for(;;)
  - #define max(A, B) ((A) > (B) ? (A) : (B))
  - header:

```
#ifndef HDR
#define HDR
/* content of hdr.h */
#endif
```

- Pointeri și tablouri - * &
- t[i] *(t+i)

- Aritmetica pointerilor: adunare întreg, diferență, comparare

```
main(int argc, char *argv[])
```

- Pointeri la funcții

```
void qsort(void *v[], int left, int right, int (*comp)(void *, void *))
```

- Structuri

```
struct point {
        int x;
        int y;
};
struct point pt;
struct { ... } x, y, z;
```

- Exemplu: afișați descrescător, după frecvența apariției lor, cuvintele de la stdin

- typedef
- Union

```
union u_tag {
        int ival;
        float fval;
        char *sval;
} u;
```

- Biblioteca standard C
- Tratarea erorilor în C:
  - variabilă globală (ex. errno)
  - returnare tip mai mare pentru a include cod eroare (ex. int getchar())
  - void f(..., &err); // pasare param. Out
- Temă: implementare stivă în C

Elemente C++ extra clase/obiecte

- Proiecte C++

  - cpp, hpp, evitarea incluziunii multiple
  - include cu “” <>
- Comentarii // ...
- Variabile definite oriunde
- const
- bool
- Referinte &: int &x = y;
- Name mangling, supraîncărcarea numelor de funcții

- extern “C” { }
- Funcții cu argumente implicite
- enum
- Heap: operatori new și delete

Paradigme de programare

- Programare procedurală – funcții
- Programare modulară - date
- Programare orientată pe obiecte:

  - abstractizarea datelor
  - moștenire (relații între clase)
  - polimorfism
  - programare generică

Exerciții

- Implementare Stiva în C
- Implementare mai multe stive in C
- Clasa Stiva

  - constructori, destructori

  - constructorul implicit
- Template Stiva

Clase și obiecte

- Noțiunea de clasă, obiecte
- Definire clasă
  - clasa Contor, seter, getter, inline
  - vizibilitate: public, protected, private
  - struct
- Crearea obiectelor – instanțiere
- Clase goale, clase definite în alte clase
- Declararea incompletă a unei clase

Membrii dată

- Membrii dată nestatici
- Membrii dată statici – exemplu

  - operatorul scope resolution ::
- Obiecte ca și date membru
- Membrii pointeri la obiecte
- Pointeri la membrii dată a unui obiect

```
p = & ob.x;
```

Membrii funcție

- Membrii funcție nestatici
- Membrii funcție statici
- A const *this
- Membrii funcție cu this constant

  int f() const;
- Membrii funcție cu this volatile

  int f() volatile;

Constructori, destructor

- Constructor implicit A()
- Constructori cu argumente
- Constructori pentru clase cu obiecte incluse
- Constructori privați; ex. singleton
- Constructor de copiere A(A& a)
  - pasarea obiecte ca parametru la funcții
  - returnare obiecte ca valoare
- Destructor ~A()

Relația friend

- Clasă sau funcție

```
class A {
    friend class B;
    friend void C::f();
 }
```

- Proprietăți:
  - nu e tranzitivă
  - nu e comutativă
  - nu se moștenește
  - nu contează în ce secțiune apare

Exerciții

- Studiu tablou de obiecte, tablou de pointeri la obiecte; aplicație Stiva/Vector ce ține obiecte de tip Contor.
- Clonarea unui arbore binar; destructor
- Implementare container Vector si iterator asociat
- Proiectarea unui sistem informatic pentru o bibliotecă publică.
- Sistem informatic folosit într-un hotel pentru a gestiona camerele

Moștenire

- Definiție; reutilizare
- Moștenire simplă și multiplă
- Relație statică

Moștenire simplă

- Modul moștenirii: public, protected, private

  Ex: class A: public B { …. };

- Structura obiectului de clasă fiu
- Controlul creării părții moștenite;
  - ordinea invocării constructorilor – întâi clasa de bază
  - ordinea invocării destructorilor - invers

Comportare membrii clasă în contextul moștenirii:

- Membrii dată nestatice
- Membrii dată statice
- Membrii funcție nestatice
- Membrii funcție statice

Conversii de tip cu clase derivate – upcasting și downcasting

```
class B: public A { … };
A a;
B b;
A* pa = new A();
B* pb = new B();
a = b;              // obiecte;    b = a;
pa = pb;            // pointeri
pb = (A*) pa;       // doar daca pa este NULL sau tine un obiect B
A& ra = b;          // referințe;   B& rb = a;
```

Polimorfism (late binding)

- Exemplu: editor de forme în plan
  - forme: Dreptunghi, Cerc, Triunghi, Grup
  - funcții: calcul arie totală
- Mecanismul apelării de funcții polimorfice
- Sintaxa – funcții virtuale
- Termen: overriding
- Rezolvare exemplu cu polimorfism

- Comparație funcții virtuale și nevirtuale
- Lanț de funcții virtuale
- Operatorul scope resolution dezactivează polimorfismul
- Implementare polimorfism – vptr, vtab
- Clase abstracte, funcții virtuale pure
- Destructori virtuali
- Exemple: observator, strategy

Operatorul dynamic_cast<>

- dynamic_cast<B&> (a) – se încearcă conversia obiectului la o referință de tip B
- Se folosește pentru downcasting (tip părinte → tip fiu) sau cross-casting (vezi ex.)
- Se convertesc:
  - pointeri; NULL dacă nu e posibil
  - referințe; excepție std::bad_cast dacă nu e posibil
- Durata depinde de complexitatea lanțului de moștenire

Exemplu de folosire dynamic_cast<>:

```
struct A
{
    int i;
    virtual ~A () {} //enforce polymorphism; needed for dynamic_cast
};
struct B
{
    bool b;
};
struct D: public A, public B
{
    int k;
    D() { b = true; i = k = 0; }
};
A *pa = new D;
B *pb = dynamic_cast<B*> (pa); //cross cast; access the second base
                               //of a multiply-derived object
```

static_cast<>

- nu este așa sigur ca dynamic_cast, se bazează doar pe informațiile de la compilare

- Exemplu:

```
A* pa = new A;      // A & B nu au legătură!
B* pb;

pb = static_cast<B*> (pa);
```

```
class A {
public:
        virtual ~A(){}
};
class B: public A { };

int main()
 {
        A* pa = new B;    // new A;
        B* pb1 = static_cast<B*>(pa);
        B* pb2 = dynamic_cast<B*>(pa);
}
```

Cast C

```
class A { };
class B: public A { };
int main()
{
        B* pb = new B;
        A* pa1 = (A*) pb;
        A* pa2 = static_cast<A*>(pb);
        return 0;
}
```

reinterpret_cast<>

- Convertește un tip pointer la orice alt tip pointer fără nici o verificare
- Convertește orice tip întreg la pointer și invers!

const_cast<>

- Este folosit pentru a elimina atributele const, volatile, __unaligned
- Exemplu:

```
class CCTest {
public:
   void setNumber( int );
    void printNumber() const;
 private:
    int number;
 };
void CCTest::printNumber() const {      // const CCTest* const this
   cout << "\nBefore: " << number;
   const_cast< CCTest * >( this )->number--;
   cout << "\nAfter: " << number;
}
```

Moștenire multiplă

- class C: public A, public B { … };
- Controlul construirii părții moștenite; de testat ordinea execuției constructorilor
- Ex:

  class B: public A { … };

  class C: public A { … };

  class D: public B, public C { … };
- Clase de bază virtuale

  class B: public virtual A { … };

  class C: public virtual A { … };

- Folosirea claselor de bază virtuale și non-virtuale împreună
- Conversii de tip în contextul moștenirii multiple

```
A *pa = (C*) new D();
```

- Nu sunt probleme pentru A clasă de bază virtuală
- Regula de dominanță – ce membru se folosește dacă sunt mai multe opțiuni; oricând avem opțiunea operator scope resolution

Supraîncărcarea (overloading)

- Name mangling
- Regula – semnătura funcțiilor; același domeniu de vizibilitate!
- Exemple în context moștenire:
  - A f(int)

    B: public A f(float)

    B b; b.f(2);
  - void f(A& a) { return a.f(3.14); }

    B b;

    int i = f(b);

- Potrivirea argumentelor: conversii implicite pentru tipurile predefinite (char, int, long..)
- Conversii utilizator prin supraîncărcare:
  - Constructori supraîncărcați: conversie de la orice la o clasă; ex. constructor Contor bazat pe diferite tipuri

    Conversii:
    - Implicite: creare obiecte, apel/return valori din funcție, asignări
    - Explicite: Contor c = (Contor) e; // constr. explicit

- Funcții speciale de conversie

  Conversii de la clasa la un tip

  Ex. pentru Contor

```
operator int*() { return &val; }
Contor c;
int *pi = (int*) c;
```

Supraîncărcarea operatorilor

- Operator = funcție
- Mod de supraîncărcare:
  - Membru clasă
  - Operator general friend
- Operator supraîncărcați ca și funcții membru:
  - @w → w.operator@()
  - w@ → w.operator@(int)
  - v@w → v.operator@(w)

- Operatori supraîncărcați ca și funcții friend
  - @w → operator@(w)
  - w@ → operator@(w, int)
  - v@w → operator@(v, w)
- Exercițiu: clasa Complex – implementare operatori (folosire referințe!)

Operatori

- Operatorul de asignare – doar ca funcție membru; generare unul implicit
- Operatorul apel de funcție
  - Operator n-ar
  - ex: a(i,j) = a(j,i);
  - … operator()(……)
- Operatorul de subscriere [ ]
  - ex. asociere nume – număr telefon

- Operatori unari prefix și postfix
  - ... operator++(); // prefix
  - ... operator++(int);// postfix
- Operatorul →
  - x → m (x.operator→()) → m
  - ex: acces la un membru data pointer
- Operatorii new si delete
  - void* operator new(size_t);
  - void operator delete(void *);
  - ex: folosire singleton automat

Operatori – note finale

- Se pot supraîncărca doar operatorii existenți, nu se pot defini noi operatori
- Se păstrează precedența, asociativitatea, numărul de operanzi
- Nu există restricții pentru tipul returnat; excepții – new delete →
- Unii operatori se pot supraîncărca doar ca funcții membru: = → new delete apel fct.
- Unii operatori nu se pot supraîncărca deloc: . .* :: ?::

Operatori – aplicație

- Implementare smart pointer
- Transformare în template

Tratarea excepțiilor

- Problema mai generală de tratare a erorilor; metode:
  - Valoare returnată de o funcție
  - Variabilă globală – errno
  - Terminare program cu:
    - exit() – inchide handle-rele de fisiere, flush
    - abort() – terminare imediata
    - aceste funcții nu știu obiecte!
  - Excepții – mecanism oferit de limbaje
- Caracteristici:
  - Separarea mai clară a sursei erorii de tratarea ei (izolare)

- Mecanism de tratare a excepțiilor
- Elemente:
  - Bloc try
  - Handlere catch (valoare sau referință)
  - Operator throw (nu folosește new!)
  - Obiect excepție
- catch(...) – catch all!
- Ordinea handlerelor
- Stack unwinding – pericolul de memory leak (nu se mai executa delete)!
- Blocuri try imbricate

- throw; - rearuncarea excepției
- Eficiență run time

Excepții standard:

- std::bad_alloc // operator new
- std::bad_cast // operator dynamic_cast < >
- std::bad_typeid // operator typeid (*p, pointer nul)
- std::bad_exception // aruncată când o specificație de
  // excepție a unei funcții este încălcată
  // deprecated în C++11

Excepții în destructor – OK doar dacă nu este o altă excepție în curs de desfășurare

```
class FileException{};
File::~File() throw (FileException)
{
        if ( close(file_handle) != success) // failed to close current file?
        {
                if (uncaught_exception() == true ) // is there any uncaught exception
                                                   //being processed currently?
                        return; // if so, do not throw an exception
                throw FileException(); // otherwise, it is safe to throw an exception
                                       // to signal an error
        }
        return; // success
}
```

- Exerciții:
  - Adăugați două excepții la o stivă pentru cazul în care se face pop() pe stiva ce nu conține nimic, respectiv push() pe stiva “plină”
  - Verificarea dinamică a corectitudinii indexului într-un vector: se va genera o excepție dacă indexul este în afara valorilor corecte.

Spații de nume (namespaces)

- Grupare logică de clase, funcții, variabile
- Ascunde numele
- Definiție:

  namespace NUME {clase, functii, var };
- Utilizare:
  - Operator scope resolution: X::A
  - using namespace X;
  - using X::A;
- Imbricare

- Alias de spațiu de nume:

  namespace LIB = ORACLE_LIB;

  LIB::init();

  - control versiuni

```
namespace vista {
  class Winsock{/*..*/};
  class FileSystem{/*..*/};
};
namespace w7 {
  class Winsock{/*..*/};
  class FileSystem{/*..*/};
}
namespace current = vista;
```

- Exercițiu: implementati următorul scenariu: spațiul de nume X conține clasa A, Y conține B și clasa C este în spațiul de nume implicit. O funcție f din X folosește B și C, o funcție f din spațiul implicit folosește A și C.

Templates

- Mecanismul de generare de clase la compilare
- Template-uri de clase

```
template <class T> class Vector;          // declarație
template <class T> class Vector {… };     // definiție
template <typename T> class Vector;       // orice tip, nu doar clase
```

- Instanțierea unui template – procesul de creare a unei clase din template + tipuri actuale; se instanțiază doar funcțiile membru necesare (eficiență + flexibilitate)!
- Parametru tip obișnuit:

```
template <class T, int n> class Array
```

- Argumente tip implicite

```
template <class T, class S = size_t > class Vector
```

- Membrii dată statici

```
template<class T> class C {
public:
        static T stat;
};
template<class T> T C<T>::stat = 5;       // definiție

void f() {
        int n = C<int>::stat;
}
```

- Specializare parțială – parametru tip de o anumită formă:

```
template <class T> class Vector  {};        // template
template <class T> class Vector <T*> {}  // specializare parțială
```

```
template <class T> class Vector <T*> // specializare
{
private:
        size_t size;
        void * p;
public:
        Vector();
        ~Vector();
//...member functions
        size_t size() const;
};
```

Alte exemple:

```
template<class T, class U, int i> class A { };        // template
template<class T, int i> class A<T, T*, i> { };       // specializare parțială
template<class T> class A<int, T*, 8> { };
```

- Specializare explicită

```
template <> class Vector <bool>          // specializare explicită
{
private:
        size_t sz;
        unsigned char * buff;
public:
        explicit Vector(size_t s = 1) : sz(s),
                buff (new unsigned char [(sz+7U)/8U] ) {}
        Vector<bool> (const Vector <bool> & v);
        Vector<bool>& operator= (const Vector<bool>& v);
        ~Vector<bool>();
        bool& operator [] (unsigned int index);
        const bool& operator [] (unsigned int index) const;
        size_t size() const;
};
```

- Template-uri funcție – algoritmi independenți de tipuri
- Alternative:
  - macro-uri

```
#define min(x,y) ((x)<(y))?(x):(y)
```

  - pointeri la void pentru a indica spre orice tip

```
void qsort( void *, size_t, size_t, int (*) (const void *, const void *));
```

  - o clasă rădăcină comună (Java Object)

```
const Object& min(const Object &x, const Object& y) {
        return x.operator<(y) ? x : y; // x & y pot fi de orice tip
}
```

- Exemplu – tipul se deduce din parametrii

```
template <class T> T max( T t1, T t2) {
        return (t1 > t2) ? t1 : t2;
}
```

STL – Standard Template Library

- Headere: vector, algorithm, iterator, etc.
- std – spațiu de nume folosit de STL

  ex: `#include <string>`

  `using namespace std;`

  `string s = “casa”;`
- iostream – bibliotecă I/O orientată pe stream-uri; cout, cerr, cin
  - Ieșire `cout << “casa” << ‘\n’;`
  - Intrare `int i;`

    `cin >> i;`

- string

```
#include <string>
string s = “casa”;
```

  - +, +=, substr(), replace(), c_str()

- Containere – specializate să țină obiecte, manipularea lor
- vector – [i], at(i), size(), resize()

```
vector<Complex> v(100);
v[0] = Complex(1,3);
```

- list – push_front(), push_back()
  - Iteratori – const_iterator, iterator

```
list<string> lista;
lista.push_back("casa"); ….
typedef list<string>::const_iterator CI;
for(CI i = lista.begin(); i != lista.end(); ++i) {
        string & s = *i;
        cout << s << '\n';
}
```

- ```
  map<string, int> phoneBook;
  string n = "Popescu";
  int i = phoneBook[n];
  ```
  ```

STL - Containere

- vector<T>
- list<T> - lista dublu inlantuita
- queue<T>
- stack<T>
- deque<T>
- priority_queue<T>
- set<T>
- multiset<T>
- map<key, value>
- multimap<key, value>

- Operații pe containere – inserție, ștergere, sortare, căutare

```
void f(vector<string>& ve, list<string>& li)
{
        sort(ve.begin(), ve.end());
        unique_copy(ve.begin(), ve.end(), back_inserter(li));
}
```

- Iteratori - ++, *

STL - Algoritmi

```
int count(const string& s, char c)
{
	string::const_iterator i = find(s.begin(), s.end(), c);
	int n = 0;
	while(i != s.end()) {
		++n;
		i = find(i+1, s.end(), c);
	}
	return n;
}
```

STL – Traversatori si Predicate

```
for_each(s.begin(), s.end(), f);
i = find_if(s.begin(), s.end(), predicat);
n = count_if(s.begin(), s.end(), predicat);
```

Temă: proiectați și implementați clasele necesare unui sistem de informatizare a unei facultăți care oferă suport pentru construirea orarului. Entități implicate: an, grupă, student, profesor, curs, sală, interval orar.