Introducere in C++

Clase si obiecte

- *Structurile de Date* sunt o forma de a stoca si organiza datele

- *Clasele* sunt un concept extins al *Structurilor de Date*

- Spre deosebire de *Structurile de Date*, clasele pot avea ca membrii inclusiv functii

- *Obiectele* sunt instante ale *claselor*

Exemplu

```
//  01_01_ClassesAndObjects

#include <iostream>

class Point{ // Point is a class
public:
    int x, y;

    void print(){
      std::cout << "(" << x << "," << y << ")\n";
    }

    void multiply(int factor){
      x *= factor;
      y *= factor;
    }
};

int main() {
    Point p; // p is an object of class Point
    p.print();
    p.x = 2;
    p.y = 3;
    p.print();
    p.multiply(3);
    p.print();
     return 0;
}
```

Output:

```
(1606416240,32767)
(2,3)
(6,9)
```

Exemplu (folosind un constructor)

```
//  01_01a_ClassesAndObjects

#include <iostream>

class Point{ // Point is a class
public:
    int x, y;

//  Point():x(0),y(0){}

    Point(){
      x = 0;
      y = 0;
    }

    void print(){
      std::cout << "(" << x << "," << y << ")\n";
    }

    void multiply(int factor){
      x *= factor;
      y *= factor;
    }
};

int main() {
    Point p; // p is an object of class Point
    p.print();
    p.x = 2;
    p.y = 3;
    p.print();
    p.multiply(3);
    p.print();
    return 0;
}
```

Output:

```
(0,0)
(2,3)
(6,9)
```

Exemplu (folosind doi constructori)

```
//  01_01b_ClassesAndObjects

#include <iostream>

class Point{ // Point is a class
public:
    int x, y;

    Point():x(0),y(0){}
    Point(int x, int y):x(x), y(y){}

    void print(){
      std::cout << "(" << x << "," << y << ")\n";
    }

    void multiply(int factor){
      x *= factor;
      y *= factor;
    }
};

int main() {
    Point p; // p is a object of class Point
    p.print();
    Point q(1,2); // q is an object of class Point
    q.print();
    q.multiply(10);
    q.print();
    return 0;
}
```

Output:

```
(0,0)
(1,2)
(10,20)
```

Abstractizarea

- *Abstractizarea* este o tehnica ce se bazeaza pe separarea interfetei de implementare

- In C++ clasele ne ofera un nivel foarte bun de abstractizare

- In C++ putem implementa tehnici de abstractizare folosind *modificatorii de acces*

Modificatorii de acces

- Membrii declarati *public* sunt accesibili de oriunde si oricine din afara clasei in care sunt declarati

- Membrii declarati *private* nu pot fi accesati din afara clasei. Acestia sunt accesibili numai in interiorul clasei sau din clase si functii declarate *friend*

- Membrii declarati *protected* nu pot fi accesati din afara clasei, dar, spre deosebire de cei declarati *private*, acestia sunt accesibile de clase *derivate*

Exemplu

```
//  01_02_Abstraction

#include <iostream>


class Point{ // Point is a class
    friend void ShowPointContents(Point * p);
private:
    int x, y;
public:
    Point():x(0),y(0){}
    Point(int x, int y):x(x), y(y){}

    void print(){
      std::cout << "(" << x << "," << y << ")\n";
    }

    void multiply(int factor){
      x *= factor;
      y *= factor;
    }
};

void ShowPointContents(Point * p){
    std::cout << "(" << p->x << "," << p->y << ")\n";
}

int main() {
    Point p;
    p.print();
    // p.x = 2; // this would not compile
    Point q(1,2);
    q.multiply(10);
    q.print();

    ShowPointContents(&q);
    return 0;
}
```

Output:

```
(0,0)
(10,20)
(10,20)
```

Encapsularea

- *Encapsularea* este includerea intr-un obiect al tuturor resurselor (functii si date) necesare pentru buna functionare a acestuia

- De obicei (good practice), publicam interfata si ascundem datele interne

- Impreuna cu *abstractizarea*, *encapsularea* este folosita pentru a implementa notiunea de *data hiding*

Exemplu - Ecuatia de gradul II

```
//  01_03_Encapsulation

#include <iostream>
#include <math.h>

class QuadraticEquation{

public:
    QuadraticEquation(int a, int b, int c): a(a), b(b), c(c){
     delta = b*b - 4*a*c;
     isComplex = false;
     if (delta > 0){
          x1 = ((-b) - sqrt(delta)) / (2*a);
          x2 = ((-b) + sqrt(delta)) / (2*a);
     }
     else if (delta == 0){
          x1 = x2 = (-b)/(2*a);
     }
     else {
          isComplex = true;
     }
    }

    bool getSolutions(int &x1, int &x2){
     if (isComplex){
          return false;
     }
     x1 = this->x1;
     x2 = this->x2;

     return true;
    }
    void printEquation(){
     std::cout << "Equation: " << a << "*x^2" << ((b > 0)?"+" :
"" )<< b << "x" << ((c > 0)?"+" : "")<< c << "\n";
    }
private:
    QuadraticEquation (): a(0), b(0), c(0) { };
    int a, b, c, delta;
    float x1, x2;
    bool isComplex;
};

void PrintSolutions(QuadraticEquation &eq){
    eq.printEquation();
    int x1, x2;
    if (eq.getSolutions(x1, x2)){
     std::cout << "x1 = " << x1 << "\nx2 = " << x2 << "\n";
    }
    else {
     std::cout << "The solutions to this equation are complex
numbers\n";
    }
}

int main(int argc, const char * argv[]) {
    QuadraticEquation eq1(1, -4, 1);
    QuadraticEquation eq2(1, -2, 1);
    QuadraticEquation eq3(1, 1, 1);
    PrintSolutions(eq1);
    PrintSolutions(eq2);
    PrintSolutions(eq3);
    return 0;
}
```

```
Equation: 1*x^2-4x+1
x1 = 0
x2 = 3
Equation: 1*x^2-2x+1
x1 = 1
x2 = 1
Equation: 1*x^2+1x+1
The solutions to this equation are complex numbers
```

Mostenirea

- Unul dintre cele mai importante concepte in POO
- In POO, o noua clasa poate mosteni membrii unei clase existente
- Numim clasa existenta ***clasa de baza*** iar clasa noua o numim ***clasa derivate***

Mostenirea si controlul accesului la membrii

ACCES	public	protected	private
Clasa de baza	DA	DA	DA
Clasa derivata	DA	DA	NU
Alte clase	DA	NU	NU

Tipuri de mostenire

- *mostenire public*
    - membrii *public* ai clasei de baza devin membri *public* ai clasei derivate
    - membrii *protected* ai clasei de baza devin membri *protected* ai clasei derivate

- *mosternire protected*
    - membrii *public* si *protected* ai clasei de baza devin membri *protected* ai clasei derivate

- *mostenire private*
    - membrii *public* si *protected* ai clasei de baza devin membri *private* ai clasei derivate

ATENTIE!

Membrii *private* ai unei clase nu vor fi **niciodata** accesibili clasei derivate. Acestia pot fi accesibili din afara doar claselor sau functiilor *friend*

Mostenire - Exemplu

```
//  01_05_Inheritance

#include <iostream>

class Phone{
public:
    Phone(const char* manufacturer) :
manufacturer(manufacturer){};
    std::string const & getManufacturer (){
      return manufacturer;
    }
protected:
    std::string manufacturer;
private:
    Phone() : manufacturer("") {};
};


class Smartphone : public Phone {
public:
    Smartphone (const char *manufacturer, const char *os) :
Phone(manufacturer), os(os){};
    std::string const & getOS(){
      return os;
    }
    void printManufacturerAndOs(){
      std::cout << "Manufacturer: " << manufacturer << "; OS: "
<< os << "\n";
    }
private:
    std::string os;
};

int main(int argc, const char * argv[]) {
    Phone simplePhone("Nokia");
    Smartphone smartphone("Samsung", "Android");

    std::cout << "Simple phone manufacturer:" <<
simplePhone.getManufacturer() << "\n";
    std::cout << "Smartphone manufacturer: " <<
smartphone.getManufacturer() << "\n";
    std::cout << "Smartphone os: " << smartphone.getOS() <<
"\n";

    smartphone.printManufacturerAndOs();

    return 0;
}
```

```
Simple phone manufacturer:Nokia
Smartphone manufacturer: Samsung
Smartphone os: Android
Manufacturer: Samsung; OS: Android
```

Mostenirea multipla

- Mostenirea multipla permite claselor noi sa deriveze mai multe clase
- De exemplu, o clasa *Button* poate mosteni clasa *Shape* dar si clasa *Drawable*

Mostenire multipla - Exemplu

```
//  01_06_Multiple_Inheritance

#include <iostream>

class Phone{
public:
    Phone(const char *manufacturer) :
manufacturer(manufacturer){};
    std::string const & getManufacturer (){
      return manufacturer;
    }
private:
    std::string manufacturer;
    Phone() : manufacturer("") {};
};


class Computer {
public:
    Computer (const char *os) : os(os){};
    std::string const & getOS(){
      return os;
    }
private:
    std::string os;
    Computer() : os("") {};
};

class Smartphone: public Phone, public Computer {
public:
    Smartphone (const char *manufacturer, const char *os) :
Phone(manufacturer), Computer(os) {};
    void printDetails(){
      std::cout << "I am made by " << getManufacturer() << "
and I run " << getOS() << "\n";
    }
};

int main(int argc, const char * argv[]) {
    Smartphone galaxyPhone("Samsung", "Android");
    Smartphone iPhone("Apple", "iOS");

    galaxyPhone.printDetails();
    iPhone.printDetails();
    return 0;
}
```

```
I am made by Samsung and I run Android
I am made by Apple and I run iOS
```

Polimorfism

- Este baza programarii orientata pe obiecte
- Polimorfismul este abilitatea obiectelor sau metodelor de a se comporta diferit in contexte diferite
- In C++, pointerii la clase de baza si la clase derivate au tipuri compatibile (spunem ca sunt *type compatible*)
- O metoda *virtuala* (keyword *virtual*) este o metoda ce poate fi redefinita intr-o clasa derivata
- Spre deosebire de o metoda *virtuala,* o metoda redefinita intr-o clasa derivate nu poate fi accesata printr-un pointer la clasa de baza

Polimorfism

- Numim o *clasa polimorfica* o clasa care declara sau mosteneste o metoda *virtuala*

- O metoda *virtuala pura* este o metoda care nu are implementare in clasa de baza

- Clasele care au cel putin o metoda *virtuala pura* nu pot fi instantiate in obiecte

- **In C++ polimorfismul functioneaza numai cu tipuri non-valorice, adica pointeri si referinte!**

Exemplu

```
//  01_07_Polymorphism

#include <iostream>

class Computer {
public:
    virtual std::string manufacturer(){
      return std::string("Dell");
    }
    virtual std::string whatAmI(){
      return std::string("I'm a computer");
    }

    virtual int getStorage(){
      return storage;
    }

    int storage = 10;
};

class Smartphone: public Computer{
public:
    virtual std::string manufacturer(){
      return std::string("Apple");
    }
    virtual std::string whatAmI(){
      return std::string("I'm a smartphone");
    }
    virtual std::string getNetwork(){
      return std::string("Orange");
    }
    virtual ~Smartphone(){};
};

class IPhone: public Smartphone {
public:
    virtual std::string whatAmI(){
      return std::string("I'm an iPhone");
    }
};

class Tablet: public Computer {
public:
    virtual std::string whatAmI(){
      return std::string("I'm a tablet");
    }
};
int main(int argc, const char * argv[]) {
    Computer *computer = new Computer();
    Computer *smartphone = new Smartphone();
    Computer *iPhone = new IPhone();
    Computer *tablet = new Tablet();

    std::cout << computer->whatAmI() << " made by " << computer-
>manufacturer() << "\n";
    std::cout << smartphone->whatAmI() << " made by " <<
smartphone->manufacturer() << "\n";
    std::cout << iPhone->whatAmI() << " made by " << iPhone-
>manufacturer() << "\n";
    std::cout << tablet->whatAmI() << " made by " << tablet-
>manufacturer() << "\n";

    Smartphone *iPhone5 = new IPhone();
    std::cout << "My network is " << iPhone5->getNetwork() << "\n";

    delete computer; delete smartphone; delete iPhone; delete
tablet; delete iPhone5;
    return 0;
```

Output:

```
I'm a computer made by Dell
I'm a smartphone made by Apple
I'm an iPhone made by Apple
I'm a tablet made by Dell
My network is Orange
```

Constructorul de copiere si operatorul de asignare

- Constructorul de copiere este un constructor special al unei clase care este folosit pentru a crea o copie a unei instante

- Operatorul de asignare, la fel ca si constructorul de copiere, ne permite sa cream copii ale obiectelor

Regula celor trei

- Se aplica inainte de standardul C++11
- Daca o clasa implementeaza una din urmatoarele, atunci ar trebui sa le implementeze pe toate trei:
  - Destructor
  - Constructor de copiere
  - Operatorul de asignare

Regula celor cinci

- Se aplica dupa C++11, datorita noilor notiuni de *move*

- Daca o clasa implementeaza una din urmatoarele, atunci ar trebui sa le implementeze pe toate trei:
  - Destructor
  - Constructor de copiere
  - Constructor de mutare
  - Operator de atribuire prin copiere
  - Operator de atribuire prin mutare

Exemplu

```
//  02_10_Rule_Of_Five

#include <iostream>

class Student{
public:
    Student(const char *_name){
        name = strdup(_name);
    }
    virtual ~Student(){
        free(name);
    }
    void printName(){
        std::cout << "My name is " << name << "\n";
    }
private:
    Student(): name(nullptr){};
    char *name;
};

class SmartStudent{
public:
    SmartStudent(const char *_name){
        name = strdup(_name);
    }
    SmartStudent(const SmartStudent &stud){
        name = strdup(stud.name);
    }
    SmartStudent(SmartStudent &&stud){
        name = stud.name;
        stud.name = nullptr;
    }
    SmartStudent & operator=(const SmartStudent &stud){
        name = strdup(stud.name);
        return *this;
    }
    SmartStudent & operator=(SmartStudent &&stud){
        name = stud.name;
        stud.name = nullptr;
        return *this;
    }
    virtual ~SmartStudent(){
        free(name);
    }
    void printName(){
        std::cout << "My name is " << name << "\n";
    }
private:
    SmartStudent(): name(nullptr){};
    char *name;
};

int main(int argc, const char * argv[]) {
    Student fred("Fred");
    SmartStudent barney("Barney");

/* This will cause the app to crash:
    {
        Student anotherFred = fred;
        anotherFred.printName();
    }
*/
    fred.printName();

    {
        SmartStudent barn = barney;
        barn.printName();
    }

    barney.printName();

    return 0;
}
```

Output:

```
x = 0
y = 2
p3 = (3,9)
p4 = (3,9)
t is 0
bool val is true
```

Membri statici

- Clasele pot contine membri statici si functii membre declarate *static*

- In POO, daca o clasa contine un membru sau o functie declarate static, acestea pot fi accesate fara a avea o instanta a unui obiect de acea clasa. Membrii statici exista fara ca clasele sa fie instantiate in obiecte

- In C++, membrii declarati statici nu sunt parte din obiecte

- The declaration of a static data member is not considered a definition. Static data members are declared at class scope, but defined at file scope. They have external linkage.

- Declararea unui membru static nu este considerata o definitie. Membrii statici sunt declarati in clase, dar definiti in fisierul care contine clasa. Acestia au linkare externa (sunt accesibili oriunde in program, nu doar in fisierul obiect care ii contine)

Exemplu

```
//  03_01_Static_Members

#include <iostream>

class Animal{
public:
    Animal() {nrOfInstances++;}
    static int nrOfInstances;
};

int Animal::nrOfInstances = 0;
int main(int argc, const char * argv[]) {
    std::cout << "Nr of Animal instances:" <<
Animal::nrOfInstances << "\n";
    Animal wolf;
    Animal zebra, lion;
    std::cout << "Nr of Animal instances:" <<
zebra.nrOfInstances << "\n";
    Animal monkey((Animal(lion)));
    std::cout << "Nr of Animal instances:" <<
Animal().nrOfInstances << "\n";
    return 0;
}
```

Output:

```
Nr of Animal instances:0
Nr of Animal instances:3
Nr of Animal instances:4
```

Iteratori

- Un iterator este un obiect folosit pentru a traversa un container

- Este, de fapt, o abstractizare a unui pointer catre un membru din container

- Cel mai simplu exemplu de iterator, este un pointer

Exemplu

```
//  DoubleLinkedList.hpp
//  08_01_Need_For_Iterators

#include <iostream>

class DoubleLinkedList{
private:
    class Node;
    Node *first;
    Node *last;
public:
    DoubleLinkedList();
    virtual ~DoubleLinkedList();
    void addFront(int value);
    void addBack(int value);
    void printAll();

    class Iterator {
        friend class DoubleLinkedList;
    public:
        Iterator & operator ++();
        bool operator ==(Iterator);
        bool operator !=(Iterator);
        int operator*();
    private:
        Iterator();
        Iterator(DoubleLinkedList::Node *);
        DoubleLinkedList::Node *position;
    };
    Iterator begin();
    Iterator end();
};
//  DoubleLinkedList.cpp
DoubleLinkedList::Iterator DoubleLinkedList::begin(){
    return Iterator(first);
}
DoubleLinkedList::Iterator DoubleLinkedList::end(){
    return Iterator(nullptr);
}
DoubleLinkedList::Iterator::Iterator(DoubleLinkedList::Node *pos) {
    position = pos;
}
DoubleLinkedList::Iterator & DoubleLinkedList::Iterator::operator++(){
    position = position->next;
    return *this;
}
int DoubleLinkedList::Iterator::operator*(){
    return position->value;
}
bool DoubleLinkedList::Iterator::operator==(Iterator it){
    return (position == it.position);
}
bool DoubleLinkedList::Iterator::operator!=(Iterator it){
    return (position != it.position);
}
//  main.cpp
int SumOfList(DoubleLinkedList *list){
    int sum = 0;
    for (DoubleLinkedList::Iterator it = list->begin(); it != list->end();
++it){
        sum+=*it;
    }
    return sum;
}
int main(int argc, const char * argv[]) {
    DoubleLinkedList *list = new DoubleLinkedList();
    list->addBack(10); list->addFront(20);
    list->addBack(3); list->addFront(12);
    list->printAll();
    int sumOfAll = SumOfList(list);
    cout << "Sum of all elements is: " << sumOfAll << "\n";
    delete list;
    return 0;
}
```

Output:

```
All values:12 20 10 3
Sum of all elements is: 45
```

Iteratori in STL

- Colectiiile din STL ne ofera tot felul de forme de iteratori pentru accesarea membrilor acestora

- In functie de tipul lor, se pot face anumite operatii cu iteratorii (incrementare, adunare, comparare, dereferentiere etc)

Exemplu

```
//  08_02_STL_Iterators

#include <iostream>
#include <list>

using namespace std;

void printList(list<float> l){
    cout << "List is: ";
    for (list<float>::iterator it = l.begin(); it!=l.end(); +
+it){
        cout << *it << " ";
    }
    cout << "\n";
}

void printListReverse(list<float> l){
    cout << "List in reverse: ";
    for (list<float>::reverse_iterator it = l.rbegin(); it !=
l.rend(); ++it){
        cout << *it << " ";
    }
    cout << "\n";
}

int main(int argc, const char * argv[]) {
    list<float> l;
    l.push_back(10); l.push_back(14);l.push_front(24);
    l.push_back(3);l.push_back(7); l.push_back(9);
    printList(l);
    printListReverse(l);
    return 0;
}
```

Output:

```
List is: 24 10 14 3 7 9
List in reverse: 9 7 3 14 10 24
```

Tratarea exceptiilor

- Exceptiile ne ofera un mod de a reactiona la niste circumstante exceptionale prin oferirea unor functii speciale, numite *handler*-e

- O exceptie poate fi *aruncata* folosind cuvantul cheie *throw* din interiorul unui bloc *try*. *Handler*-ele sunt declarate imediat dupa blocul try, folosind cuvantul cheie *catch*

- Handler-ul unei exceptii este apelata *daca si numai daca* tipul parametrului ei este compatibil cu tipul parametrului obiectului folosit la *throw*

- Mai multe handler-e pot fi inlantuite, fiecare avand un parametru diferit. Va fi apelata numai cea al carei parametru este compatibil

- Daca, in locul parametrului, se foloseste …, atunci functia handler va fi apelata pentru orice exceptie aruncata din blocul try (atat timp cat tipul exceptiei nu este compatibil cu al altei functii handler)

Tratarea exceptiilor - best practices

- Folosim *throw* numai pentru a semnala o eroare (adica functia nu a putut sa faca ceea ce a promis ca face)

- Folosim *catch* numai pentru actiuni specifice tratarii unei eror, si ***numai atunci cand stim sigur ca putem trata eroarea***

- Nu vom folosi *throw* pentru a indica o eroare logica. Vom folosi, eventual, un *assert* sau un alt mecanism de a trimite procesul intr-un debugger sau pentru a termina procesul si a colecta un crash dump (numai in medii de test, niciodata in productie)

- Nu folosim niciodata exceptii pentru controlul flow-ului de executie!

Exemplu

```
//  09_01_Exception_Handling

#include <iostream>

using namespace std;

double Divide(double a, double b){
    if (b == 0){
        throw string("Cannot divide by 0");
    }
    return a/b;
}

int main(int argc, const char * argv[]) {
    try {
        cout << "1/2 = " << Divide(1,2) << "\n";
        cout << "1/0 = " << Divide(1,0) << "\n";
    } catch (std::string ex) {
        cout << "EXEPTION: " << ex << "\n";
    } catch (int ex){
        cout << "EXEPTION Nr: " << ex << "\n";
    } catch (...) {
        cout << "Unknown exception!\n";
    }
    return 0;
}
```

Output:

```
1/2 = 0.5
1/0 = EXCEPTION: Cannot divide by 0
```

Exceptii in constructor

- Daca o exceptie este aruncata in constructor, obiectul in constructie nu va fi creat, deci destructorul corespunzator nu va fi apelat

- Este garantata chemarea destructorului bazei, in cazul in care clasa este derivata

- Toti membrii obiectului care au fost creati inainte de exceptie vor fi distrusi

Exemplu

```
//  09_03_Exception_In_Constructors

#include <iostream>

using namespace std;

class Person {
public:
    Person(string name_):name(name_){
        cout << "Creating A person named " << name << "\n";
    }
    virtual ~Person(){
        cout << "Destroying THE person named " << name << "\n";
    }
protected:
    Person(){}
    string name;
};

class SpecialisationList{
public:
    SpecialisationList(){
        cout << "Creating A specialisation list\n";
    }
    virtual ~SpecialisationList(){
        cout << "Distroying THE specialisation list\n";
    }
};

class Student: public Person{
public:
    Student(string name_, int year_): Person(name_){
        cout << "Creating a student named " << name_ << " in
year " << year_ << "\n";
        if (year < 1 || year > 4){
            cout << "The student must be in year 1 to 4!
Throwing exception!\n";
            throw string("The student must be in year 1 to
4!");
        }
        year = year_;
    }
    virtual ~Student(){
        cout << "Destroying the student named " << name << " in
year " << year << "\n";
    }
private:
    Student() {}
    SpecialisationList specialisationList;
    int year;
};

int main(int argc, const char * argv[]) {
    try {
        Student john("John", 2);
        Student jack("Jack", 0);
    } catch (string ex) {
        cout << "EXCEPTION: " << ex << "\n";
    }
    return 0;
}
```

Output:

```
Creating A person named John
Creating A specialisation list
Creating a student named John in year 2
Creating A person named Jack
Creating A specialisation list
Creating a student named Jack in year 0
The student must be in year 1 to 4! Throwing exception!
Distroying THE specialisation list
Destroying THE person named Jack
Destroying the student named John in year 2
Distroying THE specialisation list
Destroying THE person named John
EXCEPTION: The student must be in year 1 to 4!
```

Exceptii in destructor

- Destructorii nu trebuie sa arunce niciodata exceptii!

- If a destructor handles something that might throw an exception, it must do it in a try block and catch the exception

- Daca un destructor lucreza cu ceva ce ar putea arunca exceptie, va trebui sa o faca intr-un block try si sa prinda acea exceptie

- In timpul tratarii unei exceptii, foarte proabil mai multi destructori vor fi apelati. Dar, daca controlul iese din destructor (de exemplu, in urma unui throw) si avem deja alta exceptie activa, C++ va apela functia *terminate* (care va opri aplicatie si va transfera controlul catre SO). Cand este apelata functia *terminate* programul se termina **imediat.** Nu sunt distruse nici macar obiectele locale!

Exemplu

```
//  09_04_Exception_In_Destructors

#include <iostream>

using namespace std;

class TheWorstClassEver{
public:
    ~TheWorstClassEver(){
        throw string("Exception in the destructor of class
'~TheWorstClassEver'");
    }
};

int main(int argc, const char * argv[]) {
    try {
        TheWorstClassEver worstClassEver;
        throw string("OOPS!");
    } catch (string ex) {
        cout << "EXCEPTION: " << ex << "\n";
    }
    return 0;
}
```

```
(lldb) bt
* thread #1: tid = 0x609015, 0x00007fff8e16f0ae
libsystem_kernel.dylib`__pthread_kill + 10, queue =
'com.apple.main-thread', stop reason = signal SIGABRT
    frame #0: 0x00007fff8e16f0ae
libsystem_kernel.dylib`__pthread_kill + 10
    frame #1: 0x000000010007d43d
libsystem_pthread.dylib`pthread_kill + 90
    frame #2: 0x00007fffa026037b libsystem_c.dylib`abort + 129
    frame #3: 0x00007fff90b12f81 libc++abi.dylib`abort_message
+ 257
    frame #4: 0x00007fff90b38a47 libc+
+abi.dylib`default_terminate_handler() + 267
    frame #5: 0x00007fff8f6e5173
libobjc.A.dylib`_objc_terminate() + 124
    frame #6: 0x00007fff90b3619e libc+
+abi.dylib`std::__terminate(void (*)()) + 8
    frame #7: 0x00007fff90b36213 libc+
+abi.dylib`std::terminate() + 51
  * frame #8: 0x0000000100000ec6
09_04_Exception_In_Destructors`__clang_call_terminate + 22
```

Output:

```
libc++abi.dylib: terminating with uncaught exception of type
std::__1::basic_string<char, std::__1::char_traits<char>,
std::__1::allocator<char> >
```

Esuarea alocarii de memorie

- Operatorul *new* este definit in 3 moduri:

```
1. void* operator new (std::size_t size) throw (std::bad_alloc);
2. void* operator new (std::size_t size, const std::nothrow_t&
   nothrow_value) throw();
3. void* operator new (std::size_t size, void* ptr) throw();
```

Exemplu

```
//  10_02_Allocation_Failure

#include <iostream>

using namespace std;

int main(int argc, const char * argv[]) {
    int *elements = new(std::nothrow) int[231424324324234];
    if (elements == nullptr){
      cout << "ALLOCATION ERROR elements: Cannot alloc so much\n";
    }

    int *elements1 = nullptr;

    try {
      elements1 = new int[231424324324234];
    } catch (std::bad_alloc ex) {
        cout << "Cought bad_alloc exception: " << ex.what() << "\n";
    }

    if (elements1 == nullptr){
        cout << "ALLOCATION ERROR elements1: Cannot alloc so much\n";
    }

    int *elements2 = new int[231424324324234];
    if (elements2 == nullptr){
        cout << "ALLOCATION ERROR elements2: Cannot alloc so much\n";
    }
    return 0;
}
```

Output:

```
10_02_Allocation_Failure(9798,0x100081000) malloc: ***
mach_vm_map(size=925697297297408) failed (error code=3)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
ALLOCATION ERROR elements: Cannot alloc so much
10_02_Allocation_Failure(9798,0x100081000) malloc: ***
mach_vm_map(size=925697297297408) failed (error code=3)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
Cought bad_alloc exception: std::bad_alloc
ALLOCATION ERROR elements1: Cannot alloc so much
10_02_Allocation_Failure(9798,0x100081000) malloc: ***
mach_vm_map(size=925697297297408) failed (error code=3)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
libc++abi.dylib: terminating with uncaught exception of type
std::bad_alloc: std::bad_alloc
Program ended with exit code: 9
```

Controlarea (particularizarea) alocarii de memorie

- Este necesara in cazurile in care se face foarte multe alocari de mici dimensiuni intr-un timp scurt

- Functiile de alocare a memoriei (pe HEAP) din C si C++ sunt optimizate pentru alocari mari

- Prin controlarea alocarii, putem aloca o zona mare de memorie o singura data, iar apoi, prin functiile noastre (sau prin supraincarcarea operatorilor new si delete) putem numai sa asociem blocuri de memorie deja alocata

Exemplu

```
//  10_03_Customising_Memory_Allocation

#include <iostream>
#include <vector>
#include <chrono>

using namespace std;

class MyAllocator{
public:
    MyAllocator():currentBufferPos(0){
        buffer = calloc(10000000, sizeof(int));
    }
    ~MyAllocator(){
        free(buffer);
    }
    typedef int value_type;
    int* allocate(size_t count){
        int *retPtr = (int *)buffer + currentBufferPos;
        currentBufferPos += count;
        return retPtr;
    }
    void deallocate(int *mem, size_t size){
        currentBufferPos-= size * sizeof(int);
    }
private:
    void *buffer;
    int currentBufferPos;
};

int main(int argc, const char * argv[]) {
    vector<int> *v1 = new vector<int>();
    std::chrono::milliseconds ms1 =
std::chrono::duration_cast< std::chrono::milliseconds
>(std::chrono::system_clock::now().time_since_epoch());
    for (int i = 0; i < 15000000; i++) {
        v1->push_back(i);
    }
     std::chrono::milliseconds ms2 =
std::chrono::duration_cast< std::chrono::milliseconds
>(std::chrono::system_clock::now().time_since_epoch());
    cout << "Finished adding elements in " << (ms2.count() -
ms1.count()) << "\n";
    delete v1;
    vector<int, MyAllocator> *v2 = new vector<int,
MyAllocator>();
    ms1 = std::chrono::duration_cast<
std::chrono::milliseconds
>(std::chrono::system_clock::now().time_since_epoch());
    for (int i = 0; i < 15000000; i++) {
        v2->push_back(i);
    }
    ms2 = std::chrono::duration_cast<
std::chrono::milliseconds
>(std::chrono::system_clock::now().time_since_epoch());
    cout << "Finished adding elements in " << (ms2.count() -
ms1.count()) << "\n";
    delete v2;
    return 0;
}
```

Output:

```
Finished adding elements in 420
Finished adding elements in 306
```

Modificari aduse in standardul C++11

Cuvintele cheie *auto* si *decltype*

- In C++03 trebuie sa specificam tipul unei variabile atunci cand o declaram

- C++11 takes advantage of the fact that a variable is also initialised and allows us to declare an object without specifying its type, deducing it from the initialisation

- C++11 se foloseste de faptul ca, la declarare, o variabila este si initializata. Astfel, putem declara un obiect fara sa-i specificam tipul, acesta fiind dedus din intializare

- Este util atunci cand tipul este foarte lung sau generat automat (din template-uri)

- Noul operator *decltype* ne spune tipul unui obiect sau al unei expresii

Exemplu

```
//  14_01_Auto_Decltype

#include <iostream>
#include <vector>

using namespace std;

auto Sum(vector<int> & numbers) -> int{
    int sum = 0;
    auto it = numbers.begin();
    while(it != numbers.end()){
      sum += (*it++);
    }
    return sum;
}

int main(int argc, const char * argv[]) {
    auto intNumbers = new vector<int>();
    intNumbers->push_back(1); intNumbers->push_back(23);
    intNumbers->push_back(2); intNumbers->push_back(34);
    intNumbers->push_back(12); intNumbers->push_back(12);
    intNumbers->push_back(45); intNumbers->push_back(4);

    auto sumOfAll = Sum(*intNumbers);
    cout << "The sum of all elements is: " << sumOfAll <<
"\n";

    typedef decltype(intNumbers->begin()) VectorIterator;

    auto max = (*intNumbers)[0];

    for (VectorIterator vecIt = intNumbers->begin(); vecIt  !=
intNumbers->end(); ++vecIt) {
      if ((*vecIt) > max){
            max = (*vecIt);
      }
    }

    cout << "Max element in vector is: " << max << "\n";
    delete intNumbers;
    return 0;
}
```

Output:

```
The sum of all elements is: 133
Max element in vector is: 45
```

Sintaxa pentru initializare uniforma

- C++11 ne ofera o notatie {} pentru intializare
- Poate fi folosita si pentru initializarea containerelor
- In C++11 este posibila si initializarea membrilor clasei la declarare

Exemplu

```
//  14_02_Uniform_Initialization

#include <iostream>
#include <vector>
#include <map>

using namespace std;

template <typename T>
class Numbers{
public:
    Numbers(initializer_list<T> initList): numbers(initList)
{}
    auto getCount() -> int{
      return count;
    }

    auto Sum() -> T{
      T sum = 0;
      auto it = numbers.begin();
      while(it != numbers.end()){
            sum += (*it++);
      }
      return sum;
    }

private:
    int count = 0;
    vector<T> numbers;
};
```

```
auto Sum(vector<int> & numbers) -> int{
    int sum = 0;
    auto it = numbers.begin();
    while(it != numbers.end()){
      sum += (*it++);
    }
    return sum;
}

int main(int argc, const char * argv[]) {
    Numbers<int> intNumbers = {1, 23, 2, 34, 12, 12, 45, 4};
    map<string, int> ages = {
      {"Jane", 15},
      {"Mark", 4},
      {"James", 12}
    };

    auto sumOfAll = intNumbers.Sum();
    cout << "The sum of all elements is: " << sumOfAll <<
    "\n";

    return 0;
}
```

Output:

```
The sum of all elements is: 133
```

delete si *default*

- O functie default (=default la declarare) ii spune compilatorului sa genereze implementarea default pentru acea functie (folosit de obicei pentru contstructori, destructori etc)

- O functie deleted (=delete la declarare) face operatia opusa. Ii spune compilatorului sa stearga acea functie din obiect. Este foarte utila pentru a preveni constructia sau copierea unui obiect intr-un mod nedorit

Exemplu

```
//  14_03_Delete_And_Default


#include <iostream>

using namespace std;

class Person{
public:
    Person()=delete;
    Person(const string &name_): name(name_){}
    Person(const Person &) = default;
    Person &operator=(const Person&) = delete;
    void Heigth(int h) {heigth = h;}
    int Heigth() {return heigth;}
    const string &Name(){
      return name;
    }
private:
    string name="";
    int heigth = 0;
};

class Student: public Person{
public:
    Student()=delete;
    Student(const string &name_): Person(name_){}
    Student(const Student &) = default;
    Student& operator=(const Student&)=default;
    void Heigth(int h)=delete;
    int Heigth()=delete;
};

int main(int argc, const char * argv[]) {
    //Student s; // ERROR:  Call to deleted constructor of
'Student'
    Student jane("Jane");
    Student jenny(jane);
    Student jena = jane;
    //jena.Heigth(2); // ERROR: Call to deleted member
function 'Heigth'
    cout << "Jena's name is: " << jena.Name() << "\n";
    return 0;
}
```

Output:

```
Jena's name is: Jane
```

nullptr

- nullptr inlocuieste macro-ul NULL si constanta 0 care erau folosite ca inlocuitori pentru pointeri null
- Este aplicabil tuturor poonterilor si pointerilor catre membrii

Exemplu

```
//  14_04_nullptr

#include <iostream>

using namespace std;

void f(int x){
    cout << "f(int x) was called\n";
}

void f(char *){
    cout << "f(char *) was called\n";
}

int main(int argc, const char * argv[]) {
    //f(NULL); // ERROR: Call to 'f' is ambiguous
    f(nullptr);
    f(0);
    return 0;
}
```

Output:

```
f(char *) was called
f(int x) was called
```

Delegarea constructorilor

- In C++11, we can call a constructor from another constructor of the same class

Exemplu

```
//  14_05_Delegating_Constructors

#include <iostream>
#include <string>

using namespace std;

class Student{
public:
    Student()=delete;
    Student(string name_, int year_): name(name_), year(year_)
{}
    Student(string name_): Student(name_, 1){}
    Student(int year_):Student("", year_){};
    string &&toString(){
      return string("Name: ") + name + "; year: " +
to_string(year);
    }
private:
    string name = "";
    int year = 1;
};

int main(int argc, const char * argv[]) {
    Student jane("Jane", 2);
    Student july("July");
    cout << jane.toString() << "\n";
    cout << july.toString() << "\n";
    return 0;
}
```

Output:

```
Name: Jane; year: 2
Name: July; year: 1
```

Expresii lambda

- O expresie lambda ne permite sa definim local o functie, char la momentul apelarii

- Are forma: `[capture](parameters)->return_type{body}`

- Constructia [] din interiorul unei liste de argumente ne indica inceputul unui expresii lambda

- Expresiile lambda pot captura variabile (sau obiecte) si sa le foloseasca prin referinta sau prin valoare

Exemplu

```
//  14_06_Lambdas

#include <iostream>
#include <vector>

using namespace std;

int main(int argc, const char * argv[]) {
    vector<int> numbers = {2, 12, -3, 4, 4, 2, -5};
    int max = numbers[0];
    for_each(numbers.begin(), numbers.end(), [&max](int nr){
      if (nr > max){
            max = nr;
      }
    });
    auto printOutput = [max](){
      cout << "The max is " << max << "\n";
    };
    printOutput();
    [](){cout << "Another message\n";}();
    int x = 2;
    auto y = [&r = x, x = x + 1]()->int { // C++14
      r += 2;
      return x + 2;
    }();

    [](int a, int b){
      cout << "(" << a << ", " << b << ")\n";
    }(x, y);
    return 0;
}
```

Output:

```
The max is 12
Another message
(4, 5)
```

Referinte rvalue

- O rvalue este o valoare temporara ce nu persista in afara expresiei ce o foloseste

- O referinta rvalue (&&) poate fi legata de o rvalue

- Motivul principal al adaugarii acestui fel de referinte este introducerea noilor semantici move

- Folosind o referinta catre o rvalue putem, de exemplu, sa facem o functie sa se comporte diferit atunci cand este apelata cu o astfel de valoare

Exemplu

```
//  14_07_RValue_References


#include <iostream>

using namespace std;

void f(int& x)
{
    cout << "lvalue reference overload f(" << x << ")\n";
}

void f(const int& x)
{
    cout << "lvalue reference to const overload f(" << x << ")
\n";
}

void f(int&& x)
{
    cout << "rvalue reference overload f(" << x << ")\n";
}

int main(int argc, const char * argv[]) {
    int i = 1;
    const int ci = 2;
    f(i);  // calls f(int&)
    f(ci); // calls f(const int&)
    f(3);  // calls f(int&&)
     // would call f(const int&) if f(int&&) overload wasn't
provided    return 0;
    int &&x = 2 + 3;
    x+=2;

    cout << "X = " << x << "\n";
}
```

Output:

```
lvalue reference overload f(1)
lvalue reference to const overload f(2)
rvalue reference overload f(3)
X = 7
```

Constructorul de mutare

- Este ca un constructor de copiere, dar primeste ca parametru o referinta catre o rvalue

- In unele cazuri, in loc sa copiem memorie, preferam mutarea doarece este mai rapid sa schimbam pointerii catre anumite zone decat sa copiem cu totul acele zone

- Aceste noi semantici sunt folosite foarte mult in noile clase din STL

Exemplu

```
//  14_08_Move_Constructor

#include <iostream>

using namespace std;

class Person{
public:
    Person()=delete;
    Person(const char *name_){
      cout << "PersonPerson(const char *name_) constructor
called!\n";
      if (name != nullptr){
            name = strdup(name_);
      }
      else {
            name = strdup("");
      }
    }
    Person(const Person &p){
      cout << "Person(const Person &p) constructor called!\n";
      name = strdup(p.name);
    }
    Person & operator=(Person &p){
      cout << "Person & operator=(const Person &p) called!\n";
      name = strdup(p.name);
      return *this;
    }
    Person(Person&& p){
      cout << "Person(Person&& p) constructor called!\n";
      name = strdup(p.name);
      free(p.name);
    }

    Person & operator=(Person &&p){
      cout << "Person & operator=(Person &&p) called!\n";
      name = strdup(p.name);
      free(p.name);
      return *this;
    }

private:
    char *name;
};

Person f(Person p){
    return p;
}

int main(int argc, const char * argv[]) {
    Person john("John");
    Person johnny(john);
    johnny = john;
    Person jane(f(Person("Jane")));
    jane = f(Person("Jill"));
    return 0;
}
```

Output:

```
PersonPerson(const char *name_) constructor called!
Person(const Person &p) constructor called!
Person & operator=(const Person &p) called!
PersonPerson(const char *name_) constructor called!
Person(Person&& p) constructor called!
PersonPerson(const char *name_) constructor called!
Person(Person&& p) constructor called!
Person & operator=(Person &&p) called!
```

Adaugari la STL

- Noi clase de containere: unordered_set, unordered_map, unordered_multiset si unordered_multimap

- Noi librarii pentru expresii regulate, tupluri

- O noua librarie de threading (cu promises, futures, functia async() pentru lansare dea taskuri concurente etc)

- Algoritmi noi: all_of(), any_of() and none_of()

- C++11 still lacks a garbage collector, a very useful XML API (or a JSON API), sockets, GUI, reflection

- C++11 inca nu are un garbage collector, un API de pentru fisierele XML (sau JSON), socketi, GUI, reflectie

Exemplu

```
//  14_09_Additions_To_STL

#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

class Person{
public:
    Person(string name_):name(name_) {}
    const string & Name(){
      return name;
    }
private:
    string name = "";
};

int main(int argc, const char * argv[]) {
    auto john = unique_ptr<Person>(new Person("John"));
    auto jack = shared_ptr<Person>(new Person("Jack"));
    cout << "Name: " << john->Name() << "\n";
    cout << "Name: " << jack->Name() << "\n";
    vector<int> numbers = {1,2, 3, 4, -1};
    bool anyNrPositive = any_of(numbers.begin(),
numbers.end(), [](int nr)->bool{
      return (nr >= 0);
    });
    bool allNrPositive = all_of(numbers.begin(),
numbers.end(), [](int nr)->bool{
      return (nr >= 0);
    });
    bool noNrPositive = none_of(numbers.begin(),
numbers.end(), [](int nr)->bool{
      return (nr >= 0);
    });

    auto bToStr = [](bool b) -> string{
      if (b) return "YES";
      return "NO";
    };

    cout << "Are any numbers positive? " <<
bToStr(anyNrPositive) << "\n";
    cout << "Are all numbers positive? " <<
bToStr(allNrPositive) << "\n";
    cout << "Are no numbers positive? " <<
bToStr(noNrPositive) << "\n";

    return 0;
}
```

Output:

```
Name: John
Name: Jack
Are any numbers positive? YES
Are all numbers positive? NO
Are no numbers positive? NO
```

Cuvantul cheie *final*

- Specifica faptul ca o functie virtuala nu poate fi supraincarcata intr-o clasa derivata, sau faptul ca o clasa nu poate fi mostenita

- Este un identificator mai special; este folosit la declararea unei functii membru sau al unei clase, dar in alte contexte nu este cuvant rezervat si poate fi folosit pentru numirea obiectelor sau a functiilor

Exemplu

```
//  14_10_Keyword_final

#include <iostream>
#include <math.h>

using namespace std;

class Point final{
public:
    int x, y;

    void print(){
      std::cout << "(" << x << "," << y << ")\n";
    }
};

class Circle{
public:
    Circle()=delete;
    Circle(double radius_): radius(radius_) {}
    virtual double Area() final {return radius * radius *
M_PI;}
    string getInfo(){
      return string("Circle with radius: ") + to_string(radius)
+ " and area of " + to_string(Area());
    }
private:
    double radius;
    int final = 2;
};
```

```
int main(int argc, const char * argv[]) {
    return 0;
}
```

Output:

Exercitiu

Un ecran color poate avea desenate oricate forme de oricate culori. Formele sunt de mai multe feluri: triunghi, patrat, cerc etc. Controllerul ecranului are nevoie sa stie suprafetele formelor pentru a le putea desena.

Utilizatorul poate crea forme si le poate trimite ecranului pentru desenare.

Scrieti un program care defineste si implementeaza cele definite mai sus.

Exercitiu

Un sistem de fisiere poate contine fisiere sau directoare. Fisierele pot fi de mai multe tipuri: executabile, imagini, audio, video, documente).

Implementati operatiile de copy, move, delete.

Implementati o metoda care enumera toate fisierele dintr-un director.

Implementati o metoda care intoarce dimensiunea unui director.

Implementati o metoda care listeaza in ordine alfabetica continutul unui director. Implementati si afisarea in ordinea dimensiunii.

Implementati o metoda care sa stearga toate fisierele care indeplinesc o anumita conditie. (de exemplu, size < 1KB, numele contine .txt etc)

Implementati metoda Open(), care functioneaza in felul urmator: daca este apelata pe un obiect de tip fisier, in functie de tipul lui va scrie la output: “Se reda meloadia <nume_melodie>”, “Se reda filmul <nume_film>”, “Se afiseaza imaginea <img_name>” etc. Daca este apelata pe un obiect de tip director, acesta va lista tot continutul lui, recursiv.

Bibliografie

Bibliografie

- http://en.cppreference.com/w/
- http://www.cplusplus.com/
- http://www.stroustrup.com/except.pdf
- http://www.stroustrup.com/3rd_safe.pdf
- http://www.cs.princeton.edu/courses/archive/fall98/cs441/mainus/node12.html
- http://www.parashift.com/c++-faq/
- https://sourcemaking.com/design_patterns
- https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms
- https://www.wikipedia.org/
- http://www.stroustrup.com/bs_faq.html

De citit

- *Thinking in C++ (Vol I & II)*, Bruce Eckel

- *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson and John Glissades (The Gang of Four)

- *Modern C++ Design: Generic Programming and Design Patterns Applied*, Andrei Alexandrescu

- *Inside the C++ Object Model*, Stanley B. Lippman