Cursul 2

- De la C la C++, elemente de programare structurată
  - comentarii cu `//`
  - conversii de tip (type casting)
  - structuri
  - intrări/ieşiri cu **`cin >>`** respectiv **`cout <<`**
  - fişiere
  - declarații de variabile, referințe
  - apel prin referință
  - funcții care întorc variabile
  - supraîncarcarea funcțiilor
  - parametri impliciți
  - funcții inline
  - excepții

Comentarii, Conversii de tip (type casting)

In C :

- /*sir de caractere..
  .....*/
- (*tip*)*expresie*

In C++:

- //.....
- *tip*(*expresie*)
- operatorul de conv. pentru tipuri nepolimorfe:
  - **`static_cast`**<*tip*>(*expresie*)
- operatorul de conversie pentru tipuri polimorfe:
  - **`dinamic_cast`**<*tip*>(*expresie*)

Comentarii, Conversii de tip (type casting)

```
/* in limbajul C */
int x = 33;
double y;
y = (double) x;

// in limbajul C++
int x = 33;
double y, z;
y = double(x);
z = static_cast<double>(x);
```

Comentarii, Conversii de tip (type casting)

```
// in limbajul C++
class complex
{
public:
    complex():re(0), im(0){} // constructor implicit
    complex(double x):re(x), im(0){} // constr. de conv.

    …
private:
    double re, im;
};

complex u; // u este obiect al clasei complex
double y, z;
complex v(z); // constructor de conversie
u = complex(y);//apel constructor de conversie
```

Structuri în C++

- Tipul **`struct`** poate conține atât date cât şi funcții :

```
struct Data
{
  int zi, ln, an;
  void init(int o_zi, int o_luna, int un_an);
  void aduna_an(int n);
  void aduna_luna(int n);
  void aduna_zi(int n);
  void afiseaza();
};
```

Structuri in C++

- Definiția funcțiilor membru trebuie să conțină calificarea:

```
void Data::init(int o_zi, int o_luna, int un_an)
{
  zi = o_zi; ln = o_luna; an = un_an;
}

void Data::aduna_an(int n)
{
   an += n;
}
```

Structuri in C++

- Utilizarea structurilor:
  - la declarare nu se mai foloseşte `struct`
  - membrii, date şi funcții se accesează prin operatorii . şi ->

```
void main(void)
{
  Data azi; // echivalent cu: struct Data azi;
  azi.init(4,3,2003);
  azi.afiseaza();
  Data ieri;
  ieri = azi;
  ieri.aduna_zi(-1);
  ieri.afiseaza();
}
```

Structuri in C++

- O structură C++ este o clasă în care toți membrii sunt implicit **`public`**; declarația de mai sus este echivalentă cu:

```
struct Data
{
public:
  int zi, ln, an;
  void init(int o_zi, int o_luna, int un_an);
  void aduna_an(int n);
  void aduna_luna(int n);
  void aduna_zi(int n);
  void afiseaza();
};
```

Structuri in C++

- Se poate folosi **`struct`** pentru a declara clase în C++:

```
struct Data
{
public:
  Data();
  Data(int o_zi, int o_luna, int un_an);
  void aduna_an(int n);
  void aduna_luna(int n);
  void aduna_zi(int n);
  void afiseaza();
private:
  int zi, ln, an;
};
```

Intrări/ieşiri: **`cin >>.. cout <<..`**

- Flux(stream): tip de date ce descrie la nivel abstract fişiere
- În limbajul C++ pachetul (ierarhia) de clase **`iostream`** descrie şi implementează fluxurile de intrare ieşire cu toate funcționalitățile
  - fluxuri de intrare **`istream`**
    - obiect: **`cin`** – stream-ul standard de intrare
    - operatorul **`>>`**
  - fluxuri de ieşire **`ostream`**
    - obiect: **`cout`** – stream-ul standard de ieşire
    - operatorul **`<<`**

Intrări/Ieşiri: cin >>.. cout <<..

```
cin >> a1 >> a2 >> … >> an;

cout << exp1 << exp2 << … << expn;

int n; char* sir = new char[20];
cout << "Introdu un intreg > ";
cin >> n;
cout << "Introdu un sir de caractere> ";
cin >> sir;
//…
delete []sir;
```

Formatarea informației

MANIPULATOR	EFECTUL
`endl`	Scrie newline
`fixed`	Notația punct fix pentru numere reale
`left`	Aliniere stânga
`right`	Aliniere dreapta
`scientific`	Notația ştiințifică pentru numere reale
`setfill(c)`	Caracterul c înlocueşte blancurile

FORMATAREA INFORMAȚIEI

MANIPULATOR	EFECTUL
`setprecision(n)`	Precisia pentru numere reale setată la n
`setw(n)`	Dimensiunea câmpului setată la n
`showpoint`	Se scrie punctul zecimal şi zerourile
`noshowpoint`	Nu se scriu zerourile, nici punctul
`showpos`	Scrie + la numerele nenegative
`noshowpos`	Nu scrie + la numerele nenegative
`skipws`	Ignoră spațiile de dinaintea intrării
`noskipws`	Nu ignoră spațiile de dinaintea intrării

Exemplu

```
#include <iostream>
#include <iomanip>
using namespace std;
main(){
     double a = 12.05, b = 11.25, c = -200.89;
     cout << a << ',' << b << ',' << c << endl;
     cout << setfill('*') << setprecision(2);
     cout << setw(10) << a << endl;
     cout << setw(10) << b << endl;
     cout << setw(10) << c << endl;
     cout << setw(10) << showpoint << c << endl;
     cout << setfill(' ') << right << showpoint;
     cout << setw(15) <<  setprecision(5)
                          << c << endl;
     cout << scientific << c <<  endl;
     char d;
     cin >> noskipws;
     while(cin >>d)
         cout << d;
     return 0;
}
```

```
12.05,11.25,-200.89
********12
********11
***-2e+002
*-2.0e+002
        -200.89
-2.00890e+002
text pentru test
text pentru test
```

Fişiere

- Declarare fişier

```
ifstream inp_file(nume);  // ifstream inp_file;
ofstream out_file(nume);  // ofstream out_file;
fstream file(nume);       // fstream file;
```

- Funcţii :

```
file.open(nume);
file.close();
file >> var;
file << var;
file.get(c);
file.get();
file.put(c);
```

Fişiere

```
// copierea unui fisier de caractere
{
   char c;
   ifstream f_inp(“input.txt");
   ofstream f_out(“output.txt");
   if (f_out && f_inp)
      while (f_inp.get(c))
            f_out.put(c);
   else cout << "Eroare la deschidere/creare de
            fisiere." << endl;
}
```

Fişiere

```
// citirea de intregi si copierea in alt
// fisier cite unul pe linie
{
  int x;
  ifstream f_inp("inp.int");
  ofstream f_out("out.int");
  if (f_out &&  f_inp) // Expresia va avea valoarea
                       // zero in caz de eroare.
    while (f_inp >> x)
      f_out << x << endl;
  else
    cout << "Eroare la deschidere/creare de fisiere."
                          << endl;
 }
```

Variabile: declarații, alias-uri

- În C++ variabilele pot fi declarate oriunde în program

```
double x[10], s=0;
for(int i = 0; i < 10; i++) s += x[i];
int u = int(s);
cout << u;
```

- Variabilele globale pot fi accesate folosind operatorul de rezoluție `::`

```
int a = 3;
void main(){
   double a = 0.5;
   cout << a << ::a ;
}
```

Variabile: declarații, alias-uri

- În C++ pot fi declarate spații de nume:

```
namespace A{
    int x;
    namespace AA{
    char c;
    double u;
    }
    int y;
}
namespace B{
    char x;
    float u;
}

int main(){
    A::x = 4;
    A::AA::c = 'a';
    B::x = A::AA::c;
    return 0;
}
```

Variabile: declarații, alias-uri

- Tipul referință: alias pentru variabile existente:

```
int a = 5;
int& b = a;
cout << a;
b = 7;
cout << a;
double& d = 2; // Eroare
const double& d = 2; // OK
```

Variabile: declarații, alias-uri

- Referințele pot evita pointerii:

```
double *p;
p = new double;
*p = 5.0;
double& x = *p;
x *= 2.0;
cout << *p << " este acelasi cu " << x << endl
```

Funcții: apel prin referință

- Apel prin referință - parametrii funcției au tipul referință:

```
void swap(int &a, int &b){
   int c = a;
   a = b;
   b = c;
}
void main(){
   int x = 3, y = 4;
   cout << "x = " << x << " y = " << y << endl;
   swap(x, y);
   cout << "x = " << x << " y = " << y << endl;
}
```

Funcții: returnare referință

- O funcție poate returna o referință:

```
double& var_max(double &x, double &y){
   if(x > y) return x;
   else return y;
}
double u=5.5, v=3.3, w = var_max(u,v);
double &z = var_max(w, v);
var_max(w, v) += 2.5;
cout << z;
```

Funcții: supraîncarcare

- C++ dispune de un mecanism de verificare a tipului mult îmbunatățit față de C.
- Numele funcțiilor pot fi supraîncărcate: mai multe funcții, în care numărul şi/sau tipul parametrilor diferă, pot avea acelaşi nume:

```
int f(int x, int y){return x*y;}
int f(double x, int y){return x+y;}
void main(){
    cout << f(5.3, 3)<< ' ';
    cout << f(2.f,3)<< ' ';
    cout << f(2,3) << ' ';
}
// 8 5 6
```

- Funcțiile cu acelaşi nume nu pot diferi doar prin tipul returnat:

```
double f(int);
int f(int); // eroare pentru ca f(40) este ambiguu
```

Funcții: parametri impliciți

- Parametrii funcțiilor pot avea valori implicite:

```
void add3(int& s, int a, int b, int c = 0){
   s = a + b + c;
}
add3(s1, 3, 5, 9);
add3(s2, 5, 6);
```

- Atenție la ambiguități:

```
int f(int x = 0, int y = 0);
double f(int z);
// apelul f(7) este ambiguu!
```

Funcții: parametri impliciți

- Supraîncărcare şi valori implicite:

```
#include <iostream>
using namespace std;
int f(int x, int y=1){return x*y;}
int f(double x, int y){return x+y;}
int f(int x=9, int y);
void main(){
   cout << f(5.3, 3)<< ' ';
   cout << f(2.f,3)<< ' ';
   cout << f(2,3) << ' ';
   cout << f( );
}
// 8 5 6 9
```

Funcții implementate **`inline`**

- Implementare inline a funcțiilor:

```
#define max(a,b) ((a<b)?b:a); // macro in C
inline int max(int i, int j) {
    return (i<j)?i:j;
}
inline int fac(int n) {
    assign(n >= 0);
    return (n<2) ? 1 : n*fac(n-1);
}
```

- Nu orice funcție calificată **`inline`** este implementată **`inline`**; decizia este a compilatorului

Funcții implementate **`inline`**

- Dacă o funcție este definită într-un tip **`struct`** atunci implementarea sa este inline:

```
struct Data {
  int getZi() { return zi; }// implementare inline
  // ...
  int zi, ln, an;
};
```

- Dacă se implementează în afară şi se doreşte **`inline`** trebuie calificată:

```
inline int Data::getZi(){  return zi; }
```

Excepții

- Tratarea erorilor în programe compuse din module separate:
  - Raportarea condițiilor erorilor ce nu pot fi rezolvate local
  - Tratarea erorilor ce sunt detectate în altă parte
- Sintaxa:

<bloc_try>::= **`try`** *<instr_compusa> <secventa_handler>*

<secventa_handler> ::= <handler> <secventa_handler>$_{opt}$

<handler> ::= **`catch(`***<declaratie_exceptie>***`)`** *<instr_compusa>*

<declaratie_exceptie> ::= <tip> <declarator> | *<tip>*

<expresie_throw> ::= **`throw`** *<expresie>*

<specificare_exceptie> ::= **`throw(`***<lista-tip_id>***`)`**

- Instrucțiunea compusă din blocul **`try`** este o secțiune de cod pentru care *secventa_handler* asigură protecția

Excepții

- **`catch(/*…*/){/*…*/}`**

  se foloseşte imediat dupa blocul **`try`**:
  parantezele se folosesc la fel ca la funcții,
  argumentul specifică tipul obiectului ce poate fi "prins" si, opțional, numele acestui obiect

- **`catch(…){/*…*/}`**

  handler implicit (default): prinde orice exepție (… înseamna orice argument)

Excepții

- *O exceptie* este o eroare produsă la execuție si cauzată de o condiție; de exemplu un index de tablou în afara domeniului

- O funcție f poate defini condițiile ce identifică excepțiile:

```
void f(/* parametri */) {
   //…
   throw expresie;
   //…
}

twrow "depasire superioara!" //un string
twrow n;     // un numar
throw Overflow(i); // un obiect
throw; // fara tip
```

Excepții

- O funcție g care apelează f poate testa daca excepția definită de f apare la execuția programului plasând f într-un bloc **`try`**; Funcția g poate să ofere propriul tratament al excepției: cod ce trebuie executat când apare excepția respectivă:

```
try{
    f(…);// sau cod ce trebuie sa arunce exceptii
}
catch(int x){
   // cod ce trateaza cazul unui intreg
}
catch(char* s){
   // cod ce trateaza cazul unui string
}
catch(void){
   // cod ce trateaza cazul throw fara tip
}
catch(...){/* orice tip */}
```

Excepții

- Aserțiunile din C exprimă, într-un fel, excepții:

`assert(conditie)`

pentru ca execuția să se producă normal **`conditie`** trebuie să fie satisfacută, altfel codul nu este corect si programul se termină cu un mesaj

- Aserțiunile nu permit programului să încerce să recupereze erorile şi să execute codul în aceste cazuri
- Este de preferat ca în loc de **`assert`** să se introducă blocuri de tratare a excepțiilor:

```
assert(virf >= -1 && virf < MAX_STIVA)

if(virf < -1 || virf >= MAX_STIVA)
   throw " Eroare la adaugare in stiva!";
```

Excepții - Exemple

```
// Exceptii 1
#include <iostream>
using namespace std;
void foo(){
   int  i = 140, j = 150;
   throw i;
}
void call_foo(){int i = 200; foo();i++;}

void main(){
   try {
      call_foo();  //s-a iesit din foo cu i si j distrusi
   }
   catch(int n) { cout << "Am prins: " << n << endl; }
}
/* Am prins: 140 */
```

Excepții - Exemple

```
// Exceptii 2
#include "vect.h"
#include <stdlib.h>

void g (int m){
  try {
      vect  a(m), b(m);
  }
  catch(int m)  {
     cerr << "SIZE ERROR " << m << endl;
     g(10); // se ia decizia de a crea 2
            // vectori cu m=10
  }
  catch(const char* error)  {
     cerr << error << endl;
     abort();
  }
}
void main(){
   int n = -5;
   g(n);
}
/*
SIZE ERROR -5
*/
```

```
/* vect.h contine si:
vect::vect(int n){
    size = n;
   if (n < 1)throw (n);
    //asertiune preconditie
   p = new int[n];
   if (p == 0)  //asertiune
                //postconditie
      throw ("Alocare
          imposibila! \n");
}
*/
```

Excepții - Exemple

```
// Exceptii 3
#include <iostream>
using namespace std;
int add(int a, int b){
      if((b>0)&&(a>INT_MAX - b))
         throw "Depasire superioara";
      if((b<0)&&(a<INT_MIN - b))
         throw "Depasire inferioara";
      return a+b;
}
main(){
      int x,y;
      cout << "max_int = " << INT_MAX;
      cout << "min_int = " << INT_MIN;
      cout <<"Introdu x, y -->";
      cin >> x >> y;
      try {
         cout << "Suma este :" << add (x, y)
      << endl;
      }
      catch(char *err){
         cout << "EROARE: " << err;
      }
      return 0;
}
```

```
/*
max_int = 2147483647
min_int = -2147483648
Introdu x, y -->2111111111 444444444
EROARE: Depasire superioara

Introdu x, y -->5454545 6363636
Suma este : 11818181

Introdu x, y -->-2111111111
    -33333333
EROARE: Depasire inferioara
*/
```

Excepții - Exemple

```
// Exceptii 4
#include <iostream>
#include <math.h>
using namespace std;

void main(){
      int nr;
      cout << "Numar> ";
      cin >> nr;
      cout << endl;
      try{
         if (nr == 0) throw "zero";
         if (nr == 1) throw "unu";
         if (nr % 2 == 0) throw "par";
         for (int i = 3; i < sqrt(nr); i++)
                  if (nr % i == 0) throw "neprim";
         throw "prim";
      }
      catch (char *concluzie) {
         cout << " Numarul introdus este " << concluzie;
         cout << endl;
      }
}
```