CURS 1

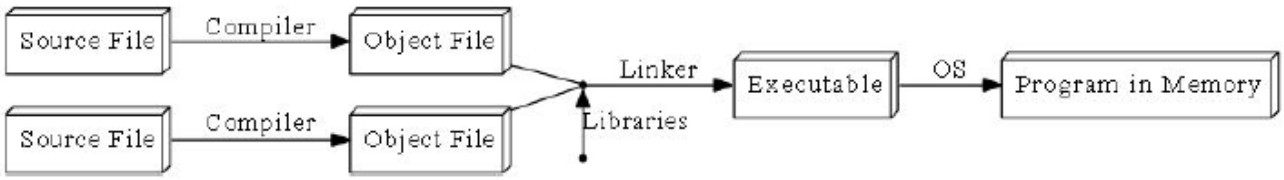
Paradigme de programare

• **Programarea structurată** este o paradigmă a programării apărută după anul 1970 datorită complicării crescânde a programelor de calculatoare. A apărut ca un model nou de programare, în scopul de a crea noi tehnici de programare apte de a produce programe care să fie sigure în funcţionare, pe o durată mai lungă.
• **Programarea imperativă**, în contrast cu programarea declarativă, este o paradigmă de programare care descrie calculul ca instrucţiuni ce modifică starea unui program. În aproape acelaşi fel în care modul imperativ din limbajele naturale exprimă comenzi pentru acţiuni, programele imperative sunt o secvenţă de comenzi pentru acţionarea calculatorului. **Programarea procedurală** este o metodă obişnuită de executare a programării imperative şi de aceea cei doi termeni sunt folosiţi deseori ca sinonime.
• **Programarea declarativă** este un stil de programare în care se descrie logica unui calcul, fără să se prezinte modul de execuţie; programarea declarativă răspunde la întrebarea ce trebuie calculat şi nu la întrebarea cum se face acest lucru. Programarea logică şi programarea bazată pe reguli se înscriu în paradigma programării declarative.
• **Programarea orientată pe obiecte** (sau programarea orientată obiect este o paradigmă de programare, axată pe ideea încapsulării, adică grupării datelor şi codului care operează asupra lor, într-o singură structură. Un alt concept important asociat programării orientate obiect este polimorfismul.
• **Programarea funcţională** este o paradigmă de programare care tratează calculul ca evaluare de funcţii matematice şi evită starea şi datele muabile. Se pune accent pe aplicarea de funcţii, spre deosebire de programarea imperativă, care foloseşte în principal schimbările de stare.

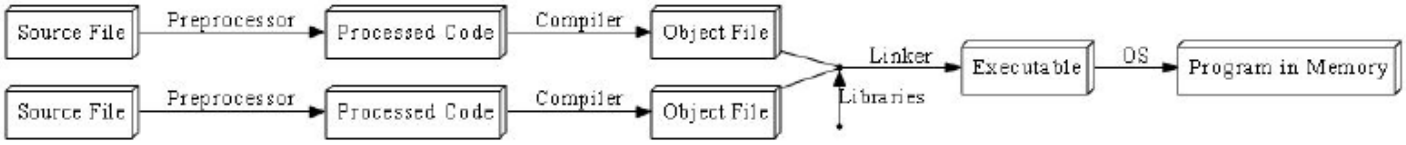
Depanarea

• Bug – eroare de programare, greu de sesizat
• Erori de compilare:
  - Probleme surprinse de compilator, in general rezultand din incalcarea regurilor de sintaxa
  - Sau prin folosirea incorecta a tipurilor de date, alocari de memorie
  - Sau erori in etapa de link-editare
• Erori din timpul rularii: programul nu face ceea ce te astepti sa faca.

CURS 2



- În C++:



Tip de date:
Standard
Structurate de nivel jos: operatiile la nivel de componenta
Structurate de nivel inalt: operatiile implementate de algoritmi utilizator

sizeof():
char = 1
int = 4
short = 2
long = 4
float = 4
double = 8
long double = 8

signed short int ≡ short
unsigned short int ≡ unsigned short
signed int ≡ int
unsigned int ≡ unsigned
signed long int ≡ long
unsigned long int ≡ unsigned long

ALL_CAPS pentru constante
lowerToUpper pentru variabile, cu nume cât mai sugestive

```
int a, b;
a = b= 5;
```

Constante intregi:

Octale: prefix 0 - 032 = 26
Hexazecimale: au prefix 0x - 0x32 =50
Intregi „long”: au sufix l/L - 24352L sau 0xaf9Fl = 44959
Intregi „unsigned” : au sufix u/U
Caractere între apostrof: ‘A’, ‘+’, ‘n’
Caractere în zecimal: 65, 42
Caractere în octal: ’\101’, ‘\52’
Caractere în hexazecimal: ‘\x41’, ‘\x2A’

CURS 3

```
typedef char litera_mare;
typedef char varsta;
typedef int vector[20];
typedef char string[30];
typedef float matrice[10][10];
typedef struct { double re, im; } complex;
```

```
litera_mare u, v='a';
varsta v1, v2;
vector x; string s;
matrice a;
complex z;
```

```
complex suma(complex z1, complex z2) {
 complex z;
 z.re=z1.re+z2.re; z.im=z1.im+z2.im;
 return z;
}
```

Operatorul conditional ?:

exp1 ? exp2 : exp3

Operatorul ?: este drept asociativ

Exemple:

```
int a=1, b=2, c=3;
int x, y, z;
x = a?b:c?a:b;
y = (a?b:c)?a:b; /* asociere stanga */
z = a?b:(c?a:b); /* asociere dreapta */
```

Operatorul sizeof()

```
sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)
sizeof(signed)=sizeof(unsigned) = sizeof(int)
sizeof(float)<=sizeof(double)<=sizeof(long double)
```

Exemple:

```
int x = 1; double y = 9; long z = 0;
cout << "sizeof(x +y + z) = " << sizeof(x+y+z) << endl;
      sizeof(x +y + z) = 8 _
```

Forţarea tipului - cast

(numetip) expresie

Exemple:

```
(long)('A' + 1.0)
(int)(b*b-4*a*c)
(double)(x+y)/z
(float)x*y/z
x / (float)2
```

Instrucţiuni

- Expresii: ; *expresie;*
- Compuse (bloc): {*declaraţii instrucţiuni*}
- Condiţionale: **if if-else switch-case**
- Iterative: **for while do-while**
- Întreruperea secvenţei: **continue; break; return *expr;***
- Salt necondiţionat: **goto**

Comparatori: ==, !=, <, <=, >, >=
Conectori logici: &&, ||, !

Regula este: else este ataşat celui mai apropiat if.

Instrucţiunea **switch**

Diferenta dintre switch si if este ca switch: În caz de egalitate se execută instrucţiunea corespunzătoare si toate cele ce urmează daca nu ai break.

Instrucţiunea **for**

Una, doua sau toate trei dintre expresii pot lipsi, dar cei doi separatori (;) sunt obligatorii.

```
i = 1;
suma = 0;
for(;i <= N;++i) suma += i;
i = 1;
suma = 0;
for(;i <= N;) suma += i++;
i = 1;
suma = 0;
for(;;) suma += i++; // Bucla infinita
```

Instrucţiuni de întrerupere a secvenţei

- **break;**
  - se referă la bucla sau instrucţiunea switch cea mai apropiată.
  - produce ieşirea din buclă sau din switch şi trece controlul la instrucţiunea următoare.
- **continue;**
  - se referă la bucla (for, while, do..while) cea mai apropiată.
  - întrerupe execuţia iteraţiei curente şi trece controlul la iteraţia următoare.
- **goto;**
  - Permite saltul la o anumită secţiune din program, identificată prin punctul de începere.
- **return *expr*; *sau* return;**
  - în funcţii, întrerupe execuţia şi transferă controlul apelantului, eventual cu transmiterea valorii expresiei *expr*.

```
int n;
 for (n=10; n>0; n--)
 {
   cout << n << ", ";
   if (n==3)
   {
     cout << "countdown aborted!";
     break;
   }
 }
```

```
for (int n=10; n>0; n--) {
   if (n==5) continue;
   cout << n << ", ";
 }
 cout << "FIRE!\n";
```

```
int n=10;
 loop:
 cout << n << ", ";
 n--;
 if (n>0) goto loop;
 cout << "FIRE!\n";
 return 0;
```

```
int i, suma=0;
  for(i = 1; i<=N; i++){
if(i%3 != 0) continue;
suma+=i;
  }
  cout << "suma = " << suma; /* suma multiplilor de 3 până la N */
```

```
int main(void)
{
    float x, y, rezultat;
    char op, c;
    int ERROR;
    cout << "Calculator pentru expresii de forma \n numar operator numar\n";
    cout << "Folositi operatorii + - * / \n";
    do
    {
        ERROR = 0;
        cout << "Expresia: ";
        cin >> x >> op >> y;
        switch(op)
        {
        case '+':
            rezultat = x+y;
            break;
        case '-':
            rezultat = x-y;
            break;
        case '*':
            rezultat = x*y;
            break;
        case '/':
            if(y != 0) rezultat = x/y;
            else
            {
                cout << "Impartire prin zero!\n";
                ERROR = 1;
            }
            break;
        default :
        {
            cout << "Operator necunoscut!\n";
            ERROR = 1;
        }
        }
        if(!ERROR)
            cout << x << " " << op << " " << y << " = " << rezultat;
        cin.sync();
        do
        {
            cout << "\n Continuati (d/n)?";
            c = getchar();
        }
        while (c != 'd' && c != 'n');
    }
    while (c != 'n');
    cout << "See you later!\n";
    return 0;
}
```

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right

2	`a++` `a--`
	`type``()` *`type`*`{}`
	`a()`
	`a[]`
	`.` `->`
		Suffix/postfix increment and decrement
		Functional cast
		Function call
		Subscript
		Member access	

3	`++a` `--a`
	`+a` `-a`
	`!` `~`
	(*`type`*)
	`*a`
	`&a`
	`sizeof`
	`new` `new[]`
	`delete` `delete[]`
		Prefix increment and decrement
		Unary plus and minus
		Logical NOT and bitwise NOT
		C-style cast
		Indirection (dereference)
		Address-of
		Size-of[note 1]
		Dynamic memory allocation
		Dynamic memory deallocation
			Right-to-left

4	`.*` `->*`	Pointer-to-member	Left-to-right
5	`a*b` `a/b` `a%b`	Multiplication, division, and remainder	
6	`a+b` `a-b`	Addition and subtraction	
7	`<<` `>>`	Bitwise left shift and right shift	
8	`<` `<=` `>` `>=`	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
9	`==` `!=`	For relational operators = and ≠ respectively	
10	`a&b`	Bitwise AND	
11	`^`	Bitwise XOR (exclusive or)	
12	`|`	Bitwise OR (inclusive or)	
13	`&&`	Logical AND	
14	`||`	Logical OR	

15	`a?b:c`
	`throw`
	`=`
	`+=` `-=`
	`*=` `/=` `%=`
	`<<=` `>>=`
	`&=` `^=` `|=`
		Ternary conditional[note 2]
		throw operator
		Direct assignment (provided by default for C++ classes)
		Compound assignment by sum and difference
		Compound assignment by product, quotient, and remainder
		Compound assignment by bitwise left shift and right shift
		Compound assignment by bitwise AND, XOR, and OR
			Right-to-left

16	`,`	Comma	Left-to-right

CURS 4

Tablouri unidimensionale

Numele unui tablou:
- nume de variabilă;
- pointer către primul element din tablou:

```
        a echivalent cu &a[0]              *a echivalent cu a[0]
      a+1 echivalent cu &a[1]          *(a+1) echivalent cu a[1]
      a+2 echivalent cu &a[2]          *(a+2) echivalent cu a[2]
      a+i echivalent cu &a[i]          *(a+i) echivalent cu a[i]
```

Parcurgerea unui tablou

```
Varianta 1: for(i=0; i<n; ++i) suma+= a[i];
Varianta 2: for(i=0; i<n; ++i) suma+= *(a+i);
Varianta 3: for(p=a; p<&a[n]; ++p) suma+= *p;
Varianta 4: p=a; for(i=0; i<n; ++i) suma+= p[i];
```

Tablouri bidimensionale

Expresii echivalente cu a[i][j]:

```
*(a[i] + j)
*(&a[0][0] + NMAX*i +j)
(*(a + i))[j]
*((*(a + i)) + j)
```

Pointeri

```
int i = 3, j = 5;
int *p = &i, *q = &j, *r;
double x;
```

Expresia	Echivalent	Valoare
p == &i	p == (&i)	002EFEA4
**&p	*(*(&p))	3
r = &x	r = (&x)	eroare! (cannot convert from double* to int*)
3**p/(*q)+2	(((3*(*p)))/(*q))+2	3
(r=&j)=*p	(*(r=(&j)))*=(*p)	15

Aritmetica pointerilor:

```
pi = 0x6aff94   p= 2
dupa *pi++:     pi = 0x6aff98   p= 2010058713
pi = 0x6aff98   p= 2
dupa (*pi)++:   pi = 0x6aff98   p= 3
pi = 0x6aff98   p= 2
dupa *++pi      pi = 0x6aff9c   p= 3407872
pi = 0x6aff9c   p= 2
dupa ++*pi      pi = 0x6aff9c   p= 3

Process returned 0 (0x0)   execution time : 0.037 s
Press any key to continue.
```

CURS 5

Structuri dinamice de date

- Liste simplu înlănțuite
- Liste dublu înlănțuite
- Stive
- Cozi
- Arbori binari

Liste simplu înlănțuite (LIFO - Last in, first out)

```
#define tipDate int
struct nod {
      tipDate info;
      nod* urm; };
typedef nod* listaSimpla;
```

Listă dublu înlănțuită (FIFO - First in, first out)

```
#define tipDate int
struct nod {
      tipDate info;
      nod* urm;
      nod* prec; };
struct listaDubla {
      nod* prim; nod* ultim;
      unsigned int lungime; };
```

Arbori binari de căutare

Fiecare nod conține un element care este mai mare decât elementul din oricare nod al subarborelui stâng (daca există) şi mai mic sau egal cu orice element din subarborele drept (dacă există).

Adăugarea recursivă a unui nou element într-un arbore binar de căutare

- dacă arborele este NULL, atunci se creează un nod în care se pune acest element;
- dacă arborele nu este NULL, atunci:
  - dacă elementul din rădăcină este > elementul nou sosit, acesta se adaugă în subarborele din stângă;
  - dacă nu, el se adaugă în subarborele din dreapta.

Preordine: rădăcina, subarborele stând, subarborele drept

```
void parcurgereInPreordine(arbore a) {
      if (!esteArboreNul(a)) {
            cout<<a->info<<", ";
            parcurgereInPreordine(a->st);
            parcurgereInPreordine(a->dr); }
}
```

Preordine: rădăcina, subarborele stând, subarborele drept

```
void parcurgereInPreordine(arbore a) {
      if (!esteArboreNul(a)) {
            cout<<a->info<<", ";
            parcurgereInPreordine(a->st);
            parcurgereInPreordine(a->dr); }
}
```

Inordine: subarborele stâng, rădăcina, subarborele drept

```
void parcurgereInInordine(arbore a) {
      if (!esteArboreNul(a)) {
            parcurgereInInordine(a->st);
            cout<<a->info<<", ";
            parcurgereInInordine(a->dr); }
}
```