

Organizarea cursului

- 2.5 ore de curs
- 2 ore de laborator: prep. ing. Anca Pop, ing. Daniel Voina

Evaluare

- 60% examen
 - 1/2 parțial (30%), 1/2 final (30%)
- 40% activitate pe parcurs

Consultații: la birou (B 531)

- o oră fixă pe săptămână (liberă în orar): Marți 9 - 11 ?
- sau stabiliți o altă ora prin e-mail (marius@cs.utt.ro)

Pagina de curs: la <http://www.cs.utt.ro/~marius/curs/pc2>

Programarea calculatoarelor 2

Introducere

Marius Minea

6 octombrie 2003

Important: Onestitate

Scopul cursului: *fiecare* din voi să programați bine în C
⇒ cursul va evalua rezultatele *fiecăruia dintre voi*

DA:

- consultați cadrele didactice în caz de nelămuriri
- învățați împreună

NU:

- prezentați soluțiile altora (modificate sau nu) ca ale voastre

Principiu de bază: orice sursă folosită trebuie citată

Limbaje de nivel înalt: scurt istoric

- conceptul de *compiler*: descris prima dată de Grace Hopper (1952)
- 1954-1957: limbajul și compilatorul FORTRAN (John Backus, IBM)
- 1958: LISP (LIST Processing, John McCarthy, la MIT)
(Lots of Idiotic, Silly Parentheses :))
- 1959: COBOL (Common Business Oriented Language)
dezvoltat de CODASYL: Committee on Data Systems Languages
- 1960: ALGOL 60: limbaj structurat, stă la baza multor limbaje
- 1964: BASIC (John Kemeny, Tom Kurtz; la Dartmouth)
- 1967: SIMULA (Ole-Johan Dahl, Kristen Nygaard):
primul limbaj orientat pe obiecte !
- 1968: Edsger W. Dijkstra: "GO TO Considered Harmful"
- principiile programării structurate
- 1971: PASCAL (Niklaus Wirth); ulterior MODULA-2

Istoricul limbajului C

- dezvoltat și implementat în 1972 la AT&T Bell Laboratories de Dennis Ritchie <http://cm.bell-labs.com/cm/cs/who/dmr/chist.html>
- contextul: evoluția conceptului de programare structurată (C inspirat de ALGOL 68, via BCPL și B)
- necesitatea unui limbaj pentru programe de sistem (legătură strânsă cu sistemul de operare UNIX dezvoltat la Bell Labs)
- C dezvoltat inițial sub UNIX; în 1973, UNIX rescris în totalitate în C
- cartea de referință: Brian Kernighan, Dennis Ritchie:
The C Programming Language (1978)
- în 1988 (vezi K&R ediția II) limbajul a fost standardizat de ANSI (American National Standards Institute)
- dezvoltări ulterioare: C99 (standard ISO 9899)

Caracteristici ale limbajului C

- limbaj de nivel *mediu*
oferă tipuri, operații, instrucțiuni simple
fără facilitățile complexe ale limbajelor de nivel (foarte) înalt
- limbaj de programare *structurat*
- permite programarea la nivel scăzut, apropiat de hardware
acces la reprezentarea binară a datelor
mare libertate în lucrul cu memoria
- foarte folosit în programarea de sistem, interfața cu hardware
- produce un cod *eficient* (compact în dimensiune, rapid la rulare)
apropiat de eficiența limbajului de asamblare
datorită caracteristicilor limbajului, și maturității compilatoarelor
- slab tipizat (spre deosebire de PASCAL)

Comparatie PASCAL - C

Pascal	C
	Lexic
litere mari și mici: la fel	se face diferența !!
	Structura programului
declarații (în ordine: const, type)	declarații (în orice ordine);
subprograme, program principal	prog. principal = funcția main
proceduri și funcții	funcții (pot să nu returneze nimic)
	Tipuri
integer	int
real	float, double (precizii diferite)
boolean	se folosește int (valori 0 și 1)
	Declarații (inclusiv pt. parametri de funcții)
nume_var : tip;	tip nume_var;
	Tablouri
nume: array[min..max] of tip;	tip nume[lung]; indici de la 0 la lung - 1

Comparatie PASCAL - C (cont.)

Pascal	C
	Operatori
:=	=
=	==
<>	!=
	Instrucțiuni
begin ... end	{ ... }
; e separator de instrucțiuni	; e terminator de instrucțiuni
if condiție then instr ...	if (condiție) instr ...
while condiție do instr	while (condiție) instr
repeat instr until cond	do instr while (neg_cond);
for cnt := min to max do instr	for (exp_init;exp_test;exp_incr) instr
nume_fct := expr	return expr ;
	Comentarii
{ ... } sau (* ... *)	/* ... */

Un prim program C

```
void main(void)
{
}
```

- cel mai mic program: nu face nimic !
- pornind de la el, scriem orice program, adăugând cod între { și }
- orice program conține funcția *main* și e executat prin apelarea ei (programul poate conține și alte funcții)
- în acest caz: funcția nu returnează nimic (primul void), nu are parametri (al doilea void)
- Vom discuta: *main* poate lua și argumente, și returna un int

Un program comentat

```
/* Acesta este un comentariu */
void main(void)
{
    /* Acesta e un comentariu pe mai multe linii
       obisnuit, aici vine codul programului */
}
```

- programele pot conține *comentarii*, înscrise între /* și */
- orice conținut între aceste caractere nu are nici un efect asupra generării codului și execuției programului
- programele *trebuie* comentate
 - pentru ca un cititor să le înțeleagă (alții, sau noi, mai târziu)
 - ca documentație și specificație: funcționalitate, restricții, etc.

Să scriem ceva!

```
#include <stdio.h>

void main(void)
{
    printf("hello, world!\n"); /* tipăreste un text */
}
```

- prima linie: obligatorie pentru orice program care citește sau scrie (o *directivă de preprocesare*, include fișierul *stdio.h* care conține declarațiile funcțiilor standard de intrare/ieșire)
- *printf* ("print formatted"): o *funcție standard* implementată într-o bibliotecă care e inclusă (linkeditată) la compilare
- N.B.: *printf* *nu* este o instrucțiune sau cuvânt cheie
- e apelată aici cu un parametru șir de caractere
- șirurile de caractere: incluse între ghilimelele duble "
- \n este notația pentru caracterul de linie nouă.

Un prim calcul

```
void main(void)
{
    int sum; /* declarăm o variabilă întregă */
    int a = 2, b; /* o variabilă inițializată, alta nu */

    b = 3;
    sum = a + b; /* semnul de atribuire în C este = */
}
```

- o variabilă trebuie *declarată* (cu tipul ei) înainte de folosire
- poate fi opțional inițializată la declarare
- câteva tipuri standard: caracter *char*, întreg *int*, real *float*
- corpul unei funcții formează un *bloc*, între { și }
- conține *declarații*, urmate de o *secvență* de instrucțiuni (nu se pot amesteca între ele, ca în C++)

```
#include <stdio.h>
void main(void)
{
    int x;

    x = 5;
    printf("Numarul x are valoarea: ");
    printf("%d", x);
}
```

Pentru a tipări valoarea unei expresii, `printf` ia două argumente:

- un șir de caractere (specificator de format):
`%c` (caracter), `%d` (întreg), `%f` (float), `%s` (șir), etc.
- expresia, al cărei tip trebuie să fie compatibil cu cel indicat
 (verificarea cade în sarcina programatorului !!!)

```
#include <stdio.h>
void main(void)
{
    int x;

    scanf("%d", &x);
    printf("%d", x);
}
```

- `scanf`: funcție de citire formatată, perechea lui `printf`
- primul argument (șirul de format) la fel ca la `printf`
- deosebirea: înainte de numele variabilei apare operatorul `&` (adresă)
 (adresa variabilei = locul unde e memorată valoarea; detalii ulterior)

```
#include <stdio.h>
void main(void)
{
    int a, b, sum;

    printf("Introduceți un număr: ");
    scanf("%d", &a); /* numărul se citește în variabila a */
    printf("Introduceți alt număr: ");
    scanf("%d", &b);
    sum = a + b;
    printf("Suma este %d\n", sum);
}
```

În Pascal, `read/write(ln)` ia oricâte argumente, de orice tip; compilatorul tratează detaliile de formatare specifice fiecărui tip.

În C, `printf/scanf` iau tot un număr arbitrar de argumente:

- primul este un șir de caractere (care indică formatul)
- restul: *expresii* (`printf`) sau *adrese* (`scanf`) cu tipuri corespunzătoare celor indicate în șirul de format

```
int x, y;
printf ("Suma lui %d și %d este %d\n", x, y, x + y);
```

```
#include <stdio.h>
void main(void)
{
    int x;

    printf("Introduceți un număr: ");
    scanf("%d", &x);
    if (x < 0) {
        printf("x este negativ\n");
    } else {
        printf("x este nenegativ\n");
    }
    if (x == 0) printf("x este zero\n");
}
```

Formatul:

```
if ( expresie logică )
    instrucțiune
else
    instrucțiune
```

- ramura `else` este opțională
- instrucțiunile din ramuri pot fi compuse (blocuri `{ }`)
- N.B.: NU CONFUNDAȚI în limbajul C
`=` este operatorul de atribuire
`==` este operatorul test de egalitate
- operatori logici: `==`, `!=`, `<`, `>`, `<=`, `>=`

Întrebare: ce face fragmentul următor pentru `x = -1, y = -2` ?

```
if (x > 0) if (y > 0) printf("unu"); else printf("doi");
```

```
#include <stdio.h>
void main(void)
{
    char c;
    int words = 0;

    c = getchar(); /* citește un caracter de la intrare */
    while (c == ' ') c = getchar(); /* spatii la început */
    while (c != '\n') {
        words = words + 1;
        while (c != ' ' && c != '\n') c = getchar(); /* cuvânt */
        while (c == ' ') c = getchar(); /* spatii */
    }
    printf("%d", words);
    printf(" cuvinte\n");
}

```

Programarea calculatoarelor. Curs 2

Marius Minea

Comentarii: structuri repetitive și prelucrări de texte

O reprezentare schematică a structurii textului de prelucrat ne poate conduce aproape direct la implementarea în program. În acest caz:

linie = (spațiu)* ((altceva)* (spațiu))* \n unde:
altceva = orice caracter în afară de spațiu și \n
notația (ceva)* = zero sau mai multe apariții ale lui "ceva"

Cu această schematizare, (ceva)* se traduce direct într-o buclă while care încearcă să recunoască secvența respectivă

Programarea calculatoarelor. Curs 2

Marius Minea

Expresii regulate

O expresie regulată descrie o familie de șiruri dintr-o mulțime Σ de simboluri (alfabet). Fie $\mathcal{L}(E)$ limbajul definit de expresia regulată E (mulțimea șirurilor reprezentate de E). Definim:

- $\mathcal{L}(a) = \{a\} \forall a \in \Sigma$ (șir dintr-un singur simbol) și operațiile:
- concatenare: $\mathcal{L}(E_1E_2) = \{s_1s_2 \mid s_1 \in \mathcal{L}(E_1), s_2 \in \mathcal{L}(E_2)\}$
- repetiție: $\mathcal{L}(E^*) = \{\epsilon\} \cup \mathcal{L}(E) \cup \mathcal{L}(EE) \cup \dots$ (ϵ = șirul vid)
- alternativă: $\mathcal{L}(E_1|E_2) = \mathcal{L}(E_1) \cup \mathcal{L}(E_2)$

Exemple: $\mathcal{L}(a b^* c) = \{ac, abc, abbc, abbbc, \dots\}$
 $\mathcal{L}(((a^*)b)^*) = \mathcal{L}((a|b)^*)$ = orice secvență de a și b
 identificator = (litera | _) (litera | cifra | _)^*

Expresiile regulate au importanță practică deosebită în analiza lexicală, prelucrări de texte, limbaje pentru "script-uri", automate finite, etc.

Programarea calculatoarelor. Curs 2

Marius Minea

Să raționăm despre programele cu bucle

Principiul: stabilim un invariant care se păstrează la fiecare iterație

```
#include <stdio.h>
void main(void)
{
    int m, lo = 0, hi = 1023;
    printf("Gândiți-vă la un număr întreg între 0 și ");
    printf("%d\n", hi);
    do { /* invariant: lo <= N <= hi, N fiind numărul cautat */
        m = (lo + hi) / 2;
        printf("Numărul e mai mare decât %d ? (d/n) ", m);
        if (getchar() == 'd') lo = m+1;
        else hi = m; /* getchar() citește un caracter */
        /* dacă da, N > m, deci N >= m + 1, deci facem lo = m + 1;
        * dacă nu, atunci N <= m, deci facem hi = m */
        while (getchar() != '\n'); /* ignora restul până la '\n' */
    } while (lo < hi); /* hi <= lo <= N <= hi --> lo = N = hi */
    printf("Numărul este %d !\n", lo);
}

```

Programarea calculatoarelor. Curs 2

Marius Minea

Să ne amintim: recursivitate

Șirul lui Fibonacci: $F_0 = F_1 = 1, F_n = F_{n-1} + F_{n-2} (n \geq 2)$

```
#include <stdio.h>
int fib(int n)
{
    if (n <= 1) return 1;
    else return fib(n-1) + fib(n-2);
}
void main(void)
{
    int n;

    printf("Introduceți numărul n: ");
    scanf("%d", &n);
    printf("Fibonacci(%d) = %d\n", n, fib(n));
}

```

Programul e eficient? Câte apeluri se fac pentru fib(4)?

Programarea calculatoarelor. Curs 2

Marius Minea

Transformarea recursivității în iterație

```
#include <stdio.h>
void main(void)
{
    int n, f, f1, f2;
    printf("Introduceți numărul n: ");
    scanf("%d", &n);
    printf("Fibonacci(%d) = ", n);
    f = 1; f1 = 1; /* f = fib(k); f1 = fib(k-1); cu k = 1 */
    n = n - 1;
    while (n > 0) { /* invariant: k+n = N (val. data pt. n) */
        f2 = f1; /* f2 = fib(k-1) */
        f1 = f; /* f1 = fib(k) */
        f = f1 + f2; /* f = fib(k+1), deci k crește cu 1 */
        n = n - 1; /* n scade cu 1 */
    }
    printf("%d\n", f);
}

```

Programarea calculatoarelor. Curs 2

Marius Minea