Programarea calculatoarelor 2

Introducere

Marius Minea

4 octombrie 2004

Organizarea cursului

– 2.5 ore de curs
– 2 ore de laborator:
prep.ing. Gabriela Bobu, drd.ing. Dan Cireşan, ing. Gabriel Fischmann

Evaluare
–60% examen
1/2 parţial (30%), 1/2 final (30%)
–40% activitate pe parcurs

Consultaţii: la birou (B 531)
– o oră fixă pe săptămână (liberă în orar): Miercuri 8-10 ?
– sau stabiliţi o altă ora prin e-mail (marius@cs.utt.ro)

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

Important: Onestitate

Scopul cursului: *fiecare* din voi să programaţi bine în C
⇒ laboratorul şi examenul evaluează rezultatele *fiecăruia dintre voi* (nu colectiv!)

DA:
– consultaţi cadrele didactice în caz de nelămuriri
– învăţaţi împreună

NU:
– prezentaţi soluţiile altora (modificate sau nu) ca ale voastre

Principiu (nu numai la acest curs): o*rice sursă folosită trebuie citată* (cărţi, articole, pagini de web, idei ale altora)

Limbaje de nivel înalt: scurt istoric

– conceptul de *compilator*: descris prima dată de Grace Hopper (1952)
– 1954-1957: limbajul si compilatorul FORTRAN (John Backus, IBM)
– 1958: LISP (LISt Processing, John McCarthy, la MIT)
(Lots of Idiotic, Silly Parantheses :))
– 1959: COBOL (Common Business Oriented Language)
dezvoltat de CODASYL: Committee on Data Systems Languages
– 1960: ALGOL 60: limbaj structurat, a inspirat multe altele
– 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 şi implementat în 1972 la AT&T Bell Laboratories de Dennis Ritchie `http://cm.bell-labs.com/cm/cs/who/dmr/chist.html`
– contextul: evoluţia conceptului de *programare structurată*
(ALGOL 68 → BCPL → B → C)
– necesitatea unui limbaj pentru *programe de sistem*
(legătură strânsă cu *sistemul de operare UNIX* dezvoltat la Bell Labs)
– C dezvoltat iniţial sub UNIX; în 1973, UNIX rescris în totalitate în C
– cartea de referinţă: Brian Kernighan, Dennis Ritchie:
The C Programming Language (1978)
– în 1988 (vezi K&R ediţia 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, operaţii, instrucţiuni simple
fără facilităţile complexe ale limbajelor de nivel (foarte) înalt
(nu: tipuri mulţime, concatenare de şiruri, etc.)
– limbaj de programare *structurat* (funcţii, blocuri)
– 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, interfaţa cu hardware
– produce un cod *eficient* (compact în dimensiune, rapid la rulare)
apropiat de eficienţa limbajului de asamblare
datorită caracteristicilor limbajului, şi maturităţii compilatoarelor
– *slab tipizat* (spre deosebire de PASCAL)
conversii implicite şi explicite între tipuri, `char` e tip întreg, etc.

Comparatie PASCAL - C

Pascal	C
Lexic	
litere mari și mici: la fel	diferite!! (*case sensitive*
Structura programului	
declaratii în ordine: const, type,	declarații în orice ordine
subprograme, program principal	prog. principal = funcția `main`
proceduri și funcții	funcții (pot returna și nimic)
Tipuri	
`integer`	`int`
`real`	`float`, `double` (precizii diferite)
`boolean`	se folosește `int` (valori 0 și 1)
Declaratii	
`var1, var2 : tip;`	`int var1, var2;`
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`),
și 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) // comentariu până la capăt de linie
{
  /* Acesta e un comentariu pe mai multe linii
     obisnuit, aici vine codul programului */
}
```

– programele pot conține *comentarii*, înscrise între `/*` și `*/`
sau începând cu `//` și terminându-se la capătul liniei (ca în C++)
– orice conținut între aceste caractere nu are nici un efect asupra
generării codului și execuției programului
– programele *trebuie* comentate
– pentru ca un cititor sa 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 – adică informațiile
(nume, parametri) necesare compilatorului pt. a le folosi corect
– `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 ghilimele duble "
– `\n` este notația pentru caracterul de linie nouă.

Un prim calcul

```
void main(void)
{
  int sum; /* declarăm o variabilă întreagă */
  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*
în ANSI C, instrucțiunile vin după declarații (nu se pot amesteca)
în C++și C99, se pot intercala oricum

Să tipărim un număr

```
#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 !!!)

Să citim un număr

```
#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)
  în C, parametrii se pot transmite *doar prin valoare*
  transmițând explicit *adresa* lui `x`, `scanf` știe unde să pună valoarea

O combinație: citire, calcul, tipărire

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

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


`printf/scanf`: formatul mai general

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;
scanf ("%d%d'', &x, &y);
printf ("Suma lui %d şi %d este %d\n", x, y, x + y);
```


Să luăm o primă decizie

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

  printf("Introduceţi 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");
}
```


Instrucțiunea de decizie `if`

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");`
Răspuns: `else` aparține de cel mai apropiat `if` (precedent).

Să gândim programele: structuri repetitive

Exemplu: câte cuvinte sunt într-o linie de text ?
Soluție: reprezentăm schematic structura textului într-o linie:
Notații (folosite în informatică pentru a descrie *expresii regulate*):
(ceva)* = zero sau mai multe apariții ale lui "ceva"
(ceva)+= una sau mai multe apariții ale lui "ceva"

linie = (spațiu)* ((altceva)+(spațiu)*)* \n sau
linie = ((spațiu)* (altceva)*)* \n unde:
altceva = orice caracter în afară de spațiu și \n

Putem transforma acum schema direct în program:
– fiecare grup de forma (ceva)* corespunde unui ciclu `while`
– condiția din ciclu: ce fel de caractere fac parte din "ceva"

Exemplu: câte cuvinte sunt într-o linie citită ?

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

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


Să raționăm despre programele cu cicluri

Multe programe "interesante" au cicluri (sau recursivitate). Trebuie:
– să proiectăm programul așa încât *să nu cicleze infinit*
– să fim siguri că la ieșirea din ciclu dă rezultatul dorit
Cum ? Nu prin încercări, ci raționând după o anumită schemă:
căutăm un *invariant* (proprietate) adevărat(ă) la fiecare iterație

Fie programul `while ( E ) do S;`.
Vrem să demonstrăm că după terminare e adevărată proprietatea Q.
Căutăm un *invariant* I cu următoarele proprietăți:
– I e adevărat înainte de a începe ciclul `while`
– dacă I și E sunt adevărate (se intră în ciclu), după execuția corpului S, e din nou adevărat I
– dacă I e adevărat și E e fals (ciclul s-a terminat), putem deduce Q

Exemplu: căutare binară într-un interval

```
#include <stdio.h>
void main(void)
{
  int m, lo = 0, hi = 1023;
  printf("Gândiţi-vă la un număr întreg între 0 şi ");
  printf("%d\n", hi);
  do { /* invariant: lo <= N <= hi, N fiind numarul 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() citeste un caracter */
    /* daca da, N > m, deci N >= m + 1, deci facem lo = m + 1;
     * daca nu, atunci N <= m, deci facem hi = m */
    while (getchar() != '\n'); /* ignora restul pana la '\n' */
  } while (lo < hi); /* hi <= lo <= N <= hi --> lo = N = hi */
  printf("Numărul este %d !\n", lo);
}
```


Să ne amintim: recursivitate

Sirul 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("Introduceti numarul n: ");
  scanf("%d", &n);
  printf("Fibonacci(%d) = %d\n", n, fib(n));
}
```

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

Transformarea recursivității în iterație

```
#include <stdio.h>
void main(void)
{
  int n, f, f1, f2;
  printf("Introduceti numarul 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 creste cu 1 */
    n = n - 1;          /* n scade cu 1 */
  }
  printf("%d\n", f);
}
```