Programarea calculatoarelor 2

Introducere

Marius Minea

5 octombrie 2005

Organizarea cursului

– 2.5 ore de curs: miercuri 8-9:30, marti 12:30-13:30 (R611)
– 2 ore de laborator (B426)
prep.ing. Gabriela Bobu, drd.ing. Dan Cireşan, prep.ing. Elena Doandeş

Evaluare

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

Consultaţii: la birou (B 531)
– o oră fixă pe săptămână (liberă în orar): Luni 11-12 ?
– sau stabiliţi o altă ora prin e-mail (marius@cs.utt.ro)

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

Despre curs (cont.)

Scopul cursului: *fiecare* din voi să programaţi bine în C
– laborator cu probleme realiste
– experienţa *individuală* de la laborator esenţială pentru examen

DA:
– învăţaţi materia după fiecare curs
– spuneţi ce e dificil de înţeles (şi cum aţi învăţa mai bine)
– veniţi la consultaţii în caz de nelămuriri
– învăţaţi împreună

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

Principiu de bază: orice sursă folosită trebuie citată
– cărţi, articole, pagini de web, idei ale altora
– oriunde: în teme, proiecte, prezentări, lucrarea de diploma ...

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)
– 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 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* (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 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

De ce folosim C ?

– produce un cod *eficient* (compact în dimensiune, rapid la rulare)
  apropiat de eficienţa limbajului de asamblare
  (fiind un limbaj relativ simplu, cu compilatoare mature)
– 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

Generalităţi şi comparaţii

– limbaj de programare *structurat* (funcţii, blocuri)
– limbaj de nivel *mediu*: tipuri, operaţii, instrucţiuni simple
  fără facilităţile complexe ale limbajelor de nivel (foarte) înalt
(nu: tipuri mulţime, concatenare de şiruri, etc.)
– *slab tipizat* $\Rightarrow$ pericol mai mare de erori
  conversii implicite şi explicite între tipuri, ex. `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, subprograme, program principal	declarații în orice ordine 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*`;`	*tip* `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	
`:=`	`=`
`=`	`==`
`<>`	`!=`
Instrucţiuni	
`begin ... end`	`{ ... }`
`;` e *separator* de instrucţiuni	`;` e *terminator* de instrucţiuni
`if` *condiţie* `then` *instr* ...	`if (` *condiţie* `)` *instr* ...
`while` *condiţie* `do` *instr*	`while (` *condiţie* `)` *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 `(* ... *)`	`/* ... */` sau `// ...`

Un prim program C

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

– cel mai mic program: nu face nimic !
– orice program conţine funcţia *main* şi e executat prin apelarea ei
(programul poate conţine şi alte funcţii)
– în acest caz: funcţia nu returnează nimic (primul `void`),
şi nu are parametri (al doilea `void`)
Cf. standard: `main` returneaza un cod intreg catre sistemul de operare
(conventie: 0 == terminare cu succes, != 0: cod de eroare)

```
int main(void)
{
  return 0;
}
```

Discutam ulterior: `main` poate avea parametri (argumentele liniei de comanda)

Un program comentat

```
/* Acesta este un comentariu */
int main(void) // comentariu până la capăt de linie
{
  /* Acesta e un comentariu pe mai multe linii
     obisnuit, aici vine codul programului */
  return 0;
}
```

– programele pot conţine *comentarii*, înscrise între /* şi */
sau începând cu // şi terminându-se la capătul liniei (ca în C++)
– orice conţinut între aceste caractere nu are nici un efect asupra generării codului şi execuţiei programului
– programele *trebuie* comentate
  – pentru ca un cititor sa le înţeleagă (alţii, sau noi, mai târziu)
  – ca documentaţie şi specificaţie: funcţionalitate, restricţii, etc.
  – ce reprezinta variabilele, parametrii functiilor, rezultatul, ce conditii sunt necesare, cum se comporta la eroare, etc.

Să scriem ceva!

```
#include <stdio.h>

int main(void)
{
  printf("hello, world!\n"); // tipăreste un text
  return 0;
}
```

– prima linie: obligatorie pentru orice program care citește sau scrie
= o *directivă de preprocesare*, include fișierul `stdio.h` care conține *declarațiile* (NU implementarea) funcțiilor standard de intrare/ieșire
= informațiile (nume, parametri) necesare compilatorului pt. a le folosi
– implementarea (cod obiect, compilat): într-o bibliotecă inclusă (linkeditată) la compilarea programului utilizator
– `printf` ("print formatted"): o *funcție standard*
– 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

```
int 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 =
  return 0;
}
```

– 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

Tipărirea

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

  x = 5;
  printf("Numarul x are valoarea: ");
  printf("%d", x);
  return 0;
}
```

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 !!!)

Citirea de la intrare

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

  scanf("%d", &x);
  printf("%d", x);
  return 0;
}
```

– `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*
primind *adresa* lui `x` (prin valoare!), `scanf` știe unde să pună valoarea

Citirea unui caracter: cu functia `getchar()`

```
char c;                          // mai bine: int, discutam mai tarziu
c = getchar();
```

O combinaţie: citire, calcul, tipărire

```
#include <stdio.h>
int 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);
  return 0;
}
```

`printf/scanf`: formatul mai general

În Pascal, `read/write(ln)` ia oricăte 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>
int 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");
  return 0;
}
```

Instrucţiunea de decizie `if`

Formatul:

`if` (*expresie logică*)
  instrucţiune
`else`
  instrucţiune

– ramura `else` este opţională
– instrucţiunile din ramuri pot fi compuse (blocuri `{ }`)
– N.B.: NU CONFUNDAŢI 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` aparţine de cel mai apropiat `if` (precedent).

Exemplu: câte cuvinte sunt într-o linie citită ?

```
#include <stdio.h>
int 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 cuvinte\n", words);
  return 0;
}
```

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:
– ce stim la inceputul ciclului ?
– ce stim dupa fiecare iteratie ? se pastreaza o anumita proprietate ?
– ce dorim sa deducem la sfarsit ?
$\Rightarrow$ 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>
int 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 e numarul cautat
    m = (lo + hi) / 2;
    printf("Numărul e mai mare decât %d ? (d/n) ", m);
    if (getchar() == 'd') lo = m+1;             // raspuns afirmativ
    else hi = m;
    /* 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 caractere pana la '\n'
  } while (lo < hi);          // hi <= lo <= N <= hi --> lo = N = hi
  printf("Numărul este %d !\n", lo);
  return 0;
}
```

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);
}
int main(void)
{
  int n;

  printf("Introduceti numarul n: ");
  scanf("%d", &n);
  printf("Fibonacci(%d) = %d\n", n, fib(n));
  return 0;
}
```

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

Transformarea recursivităţii în iteraţie


```
#include <stdio.h>
int 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);
  return 0;
}
```