Utilizarea și programarea calculatoarelor

Introducere

Marius Minea

5 octombrie 2004

Scopul cursului

Utilizarea calculatoarelor

– familiaritate cu sisteme PC, lucru cu fișiere, medii de programare
– în principal la laborator

Programarea calculatoarelor

– de la exemple la programe pentru situații reale
– programarea = dezvoltarea unui produs software, de la A la Z
– buna cunoaștere a unui limbaj de programare
și un punct de plecare pentru altele
– principii și stil de programare

Ce este un program

= o secvență de instrucțiuni care comandă execuția calculatorului

– program executabil: cod mașină interpretabil direct de calculator
– program sursă: în limbaj inteligibil de programatorul uman

Traducerea din format sursă în format executabil:
– *compilare*: anterior rulării programului (pt. C, C++, PASCAL)
– *interpretare*: direct la rulare: (pt. variante de BASIC, LISP)

În mod tipic, un program generic:
– citește *datele de intrare*
– efectuează *prelucrări* (calcule) asupra lor
– produce niște *rezultate* la ieșire

Arhitectura unui calculator

– *unitate centrală* de prelucrare (CPU)
  - unitate de control
  - unități aritmetice și logice pentru calcul
– *memorie*
  primară: circuite integrate
  secundară: medii magnetice (disc fix, floppy), optice (CD)
– *echipamente periferice* (dispozitive de intrare/ieșire)
  tastatură, mouse, ecran, imprimantă, joystick

Funcţionarea arhitecturii von Neumann

John von Neumann (1945)
– propune arhitectura menţionată (control, UAL, memorie, I/O)
– şi conceptul de *program memorat*
(deosebirea esenţială, de ex. faţă de calculatorul de buzunar, acţionat direct de utilizator)
Acestea stau la baza tuturor calculatoarelor convenţionale.

Funcţionarea:
1. citeşte instrucţiunea de la adresa din numărătorul de program
2. înaintează numărătorul de program la următoarea instrucţiune
3. unitatea de control decodifică instrucţiunea şi comandă operaţia
(care poate modifica regiştri, memoria, numărătorul de program)
4. se reia ciclul de la punctul 1.

Sisteme de operare

– *gestionează resursele* unui sistem de calcul (timpul de procesare al unității centrale, memoria, perifericele, sistemul de fișiere)
– creează o interfață *independentă de hardware*
– oferă *apeluri sistem* utilizate din limbaje de programare
(alocare de memorie, citire, tipărire)

Vom programa: sub Windows în semestrul I, sub Linux în semestrul II
Urmărim: scrierea de programe *portabile*, independent de sistemul de operare și mediul de programare folosit.

Ciclul de dezvoltare al unui program

– Definirea și analiza specificaţiilor
– Proiectare
– Implementare (Codare) ← doar o parte !!!
– Testare
– Mentenanţă

Dezvoltarea programului

Cerinţe şi specificaţii

La curs: reprezentate de enunţul problemei. În realitate însă:
– adesea cea mai dificilă parte
– trebuiesc eliminate ambiguităţile
– neînţelegerile au efecte pentru tot restul proiectului
– important: nu "ce ştiu eu să fac" ci "ceea ce se cere" (de client)

Proiectarea soluţiei

– arhitectura programului
– împărţirea în componente şi interfaţa între ele
– proiectarea structurilor de date
– proiectarea algoritmilor
– interfaţa cu utilizatorul

Corectitudinea programului

Raţionament logic

– ce face programul ? pot defini în mod precis ? (absolut necesar!)
– pot găsi o relaţie matematică ?
– pot urmări pas cu pas transformarea pe care o efectuează programul şi să demonstrez (să mă conving) astfel că rezultatul final e corect ?
– ce se schimbă pe parcursul programului ? ce rămâne neschimbat ? (exemplu: folosirea invarianţilor în raţionamentul despre cicluri)
– urmărirea raţionamentului în faza de implementare reduce erorile

Testare

– ce presupuneri/garanţii există despre intrare ?
– ce presupuneri/garanţii există despre alte module de program ?
– cum se comportă programul: pentru date normale, limită, eronate care e performaţa pt. date de dimensiuni mari ?

Dincolo de program

Documentarea

– complexitatea sistemului creşte pe măsura realizării

– documentaţie necesară pentru:

- descriere exactă a funcţionalităţii sistemului, împreună cu toate cerinţele, restricţiile, presupunerile
- comunicarea dintre programatori (chiar pt. programatorul iniţial!)
- proiectarea de teste, evoluţia şi mentenanţa ulterioară

Reutilizare. Portabilitate

– cât de generală este soluţia ? poate fi reutilizată ?

– se pot folosi elemente existente ?
(funcţii de bibliotecă, module de program, obiecte, etc.)

Modificare şi mentenanţă

– programul e proiectat pentru a fi întreţinut uşor ?

Securitate

– robusteţe, rezistenţă la date de intrare invalide

Organizarea cursului

– 2 ore de curs
– 1 ora de seminar (2 la 2 săptămâni): prep. ing. Gabriela Bobu
– 2 ore de laborator: prep. ing. Elena Doandeș

Evaluare

–60% examen
1/2 parțial (30%), 1/2 final (30%)
–40% activitate pe parcurs (30% laborator, 10% seminar)

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

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

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)

Algoritmi și scheme logice

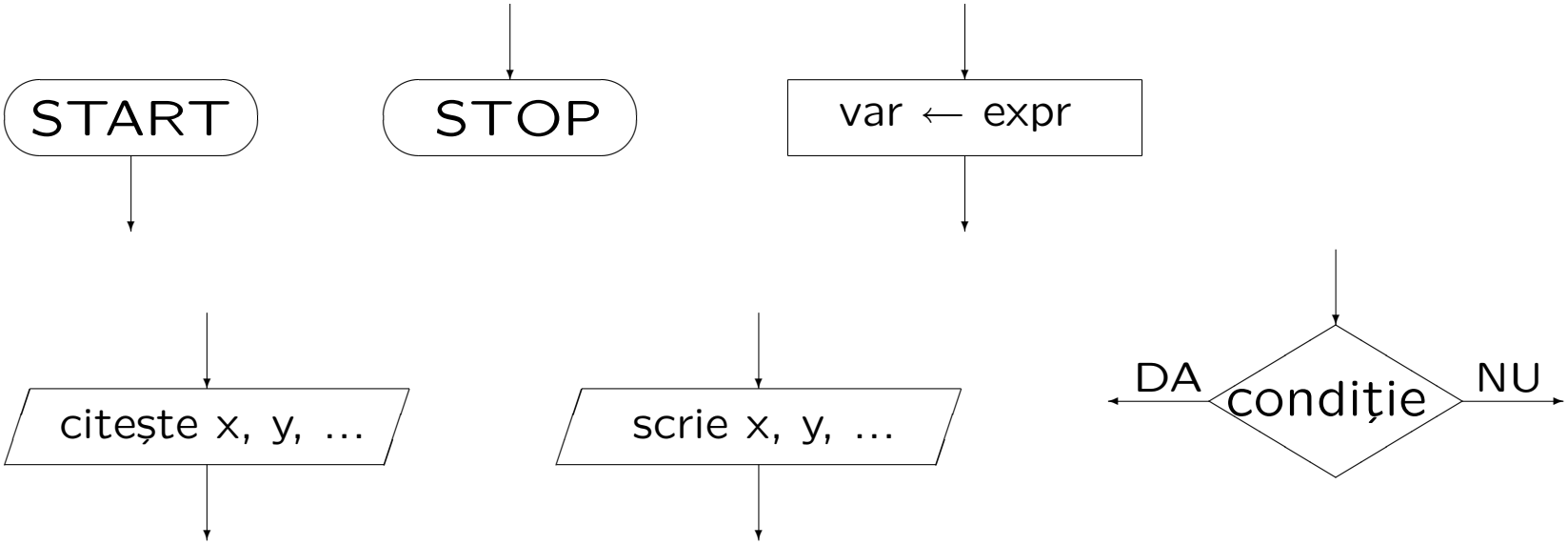
algoritm = secvență *finită* de pași pentru rezolvarea unei probleme

schemă logică: reprezentarea grafică a unui algoritm

– fără particularitățile unui anumit limbaj de programare, dar precis

Blocuri (instrucțiuni) componente în scheme logice:

start, stop, atribuire, citire, scriere, decizie



schemă logică: graf format din cele 6 tipuri de instrucțiuni, cu un singur nod de START și unul singur de STOP

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`
– limbaj de *programare structurată* (blocuri, cicluri, funcţii)
(concept apărut în ALGOL 60, apoi ALGOL 68, PASCAL, ...)
– 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* → necesită mare atenţie în programare
  conversii implicite şi explicite între tipuri, `char` e tip întreg, etc.

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 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.

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, b; /* declarăm încă două variabile întregi */

  a = 2;
  b = 3;
  sum = a + b; /* semnul de atribuire în C este = */
}
```

– pentru a memora data și calcula, avem nevoie de *variabile*
o variabilă are un *nume*, un *tip* și o *valoare*
– o variabilă trebuie *declarată* (cu tipul ei) înainte de folosire
– câteva tipuri standard: caracter `char`, întreg `int`, real `float`
– corpul unei funcţii formează un *bloc*, între `{` și `}`
– blocul poate conţine *declaraţii* și o *secvenţă* de instrucţiuni

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), %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`: funcţie de citire formatată, perechea lui `printf`
– primul argument (şirul de format) la fel ca la `printf`
– deosebirea: înainte de numele variabilei apare operatorul & (adresă)
  transmiţând explicit *adresa* lui `x`, `scanf` ştie unde să pună valoarea

O combinaţie: 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);
}
```

Obs.: `\n` este caracterul de linie nouă.

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");
  } else {
    printf("x este nenegativ");
  }
  if (x == 0) printf("x este zero");
}
```

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: `==`, `!=`, `<`, `>`, `<=`, `>=`