Cum se programează
— un set de reguli pentru scrierea eficientă de programe —

Mihai BUDIU – mihaib@pub.ro

februarie 1996

Subiect: reguli pentru scrierea programelor.

Cuvinte cheie: comentariu, indentare, modul, variabilă, procedură.

Cunoştinţe necesare: cel puţin un limbaj de programare.

Contents

1 Introducere

Voi începe cu un citat:

> As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on

> was going to be spent in finding mistakes in my own programs.
> — Maurice Wilkes discovers debugging, 1949

În traducere asta ar suna cam aşa:

> Imediat ce am început să programăm am descoperit spre surprinderea noastră ca nu era aşa de uşor să facem programe bune pe cît crezusem. Depanarea (debugging) trebuia să fie inventată. Îmi pot aminti cu precizie momentul exact cînd am realizat că o mare parte din viaţa mea de atunci încolo urma să fie petrecută în a găsi greşeli în propriile mele programe.

De asemenea — pentru contrast — îmi place să citez următorul fragment din manualul care însoţeşte programul MetaFont, scris de marele informatician Donald Knuth:

> On January 4, 1986 the "final" bug in Metafont was discovered and removed. If an error still lurks in the code, Donald E. Knuth promises to pay a finder's fee which doubles every year to the first person who finds it. Happy hunting.

Adică:

> Pe 4 ianuarie 1986 "ultimul" bug (defect) din Metafont a fost descoperit şi scos. Dacă o eroare mai zace prin cod, Donald E. Knuth promite să plătească o sumă care se dublează în fiecare an celui care o găseşte. Vînătoare plăcută.

Cîţi dintre dumneavoastră au reuşit să scrie un program mai lung de 2000 de linii sursă (într-un limbaj de nivel înalt) care să exhibe acelaşi îngrijorător simptom al perfecţiunii?

Cu riscul de a vă agasa voi mai cita un monstru sacru al informaticii, de data asta însă din memorie. Edsger Dijkstra spunea în cuvîntarea sa făcută cu prilejul decernării premiului "Turing" al prestigioasei societăţi americane ACM (Association for Computing Machinery), cuvîntare care dealtfel se numea "The Humble Programmer" (Programatorul Umil), că un bun programator trebuie să recunoască faptul că sarcina ce şi-a asumat-o îl depăşeşte cu mult datorită complexităţii sale, şi ca atare trebuie să încerce cît mai mult să o reducă la bucăţi mai mici, cît mai uşor de manevrat.

Deşi aparent legăturile dintre cele trei citate de mai sus sunt cam şubrede, sper că textul care urmează va arăta că de fapt acestea sunt trei feţe ale aceluiaşi poliedru.

În rezolvarea unei probleme cu calculatorul se pot distinge mai multe etape (chiar dacă ele nu sunt distincte temporal): analiza problemei, elaborarea algoritmului, implementarea, documentarea, testarea, etc. Acest articol se ocupă de una singură dintre ele, care dealtfel este foarte puţin dependentă de problema care trebuie rezolvată: presupun că ştiu algoritmii, tot ce trebuie să fac este să-i implementez într-un limbaj de programare oarecare. Aparent banală, etapa aceasta se dovedeşte adesea foarte intricată. De fapt acest lucru îl arată şi primul citat din introducere.

Cele mai multe consideraţii care urmează sunt independente de limbajul ales.

2 Regula de aur

Un program trebuie să fie scris în aşa fel încît să poată fi citit de către oricine.

Programele sunt făcute în aşa fel încît părţile sunt strîns dependente. Trebuie ca cele scrise într-un loc să fie folosite într-altul (şi eventual refolosite). Mintea unui om normal nu poate reţine toate amănuntele din textul unui program, de aceea trebuie să-i dăm un ajutor.

Sfatul acesta este clar obligatoriu cînd se dezvoltă soft în echipă, pentru că ceea ce scrie unul trebuie să fie folosit de altul. Paradoxal este că el este extrem de oportun chiar cînd întreg procesul de scriere este făcut de un singur programator. Nu vi s-a întîmplat niciodată să vă uitaţi la o funcţie scrisă în urmă cu o lună şi să vă miraţi că aţi putut scrie aşa ceva? Sau să vă întrebaţi ce face? Acel "oricine" din sfat poate fi deci un "alter ego" al dumneavoastră.

Aparent scrierea îngrijită a unui program consumă prea mult timp.

Pentru programe scurta acest lucru poate fi adevărat. Pentru programe la care lucraţi mai mult de o săptămînă timpul cheltuit cu o scriere îngrijită este recuperat înzecit în fazele ulterioare ale dezvoltării (depanare, extindere, întreţinere). Dacă nu credeţi, încercaţi măcar o dată.

3 Împărţirea programului

Fă bucăţile în aşa fel încît să înţelegi uşor *ce* face fiecare fără a trebui să ştii *cum* face.

Domnul Dijkstra (în olandeză se citeşte aproximativ "daicstra") recomanda o tactică veche de pe vremea lui Cezar: "divide et impera": "împarte şi stăpîneşte". Aşa e bine să scrieţi şi programele: în bucăţi cît mai independente unele de altele, care interacţionează intre ele în moduri cît se poate de clare şi simple, şi care nu se influenţează "pe la spate".

Din fericire toate limbajele moderne de programare aduc un suport extrem de eficace acestei metode de scriere: funcţiile, procedurile şi modulele.

Împărţiţi deci programul dumneavoastră în astfel de părţi cît mai des cu putinţă. Faceţi părţile logic independente; funcţiunea fiecăreia trebuie să fie extrem de clară şi cît se poate de limpede enunţată. De exemplu, o funcţie care caută ceva într-un şir nu trebuie să facă şi ordonarea şirului. Faceţi două funcţii independente pentru asta, una de căutare şi una de sortare.

Dacă limbajul permite module, atunci faceţi-le în fişiere separate, ale căror nume să fie suficient de clare. Grupaţi în fiecare modul numai funcţii şi variabile înrudite. Dacă programaţi într-un limbaj orientat pe obiecte, implementaţi fiecare clasă într-un modul separat, cu toate metodele ei. O să ştiţi apoi unde să le căutaţi, şi apoi o să le puteţi reutiliza cu uşurinţă în alte programe.

Cheia economiei de timp este asta: cînd vrei să foloseşti o funcţie trebuie să ştii numai ceea *ce* face; nu trebuie să o citeşti în întregime ca să vezi *cum* face acest lucru. De aceea funcţia este mult mai uşor de folosit, şi apoi poate fi la nevoie rescrisă în cu totul alt fel, fără a influenţa cîtuşi de puţin restul programului. (De exemplu nu contează dacă sortarea este bubble sort sau quick sort, totul este să fie tot o sortare. Funcţiile care folosesc funcţia de sortare nu sunt interesate de modul în care aceasta se face). Un alt uriaş beneficiu al acestei relative independenţe între feluritele părţi este că în momentul în care un bug este descoperit şi scos dintr-una, celelalte vor rămîne neschimbate, pentru că nu se bazau pe felul în care lucra acea parte, ci doar pe ceea ce ea făcea.

Cîte funcţii sau linii de program trebuie să fie într-un modul? Asta este destul de mult o chestie de gust, dar editorul de texte pe care îl folosiţi poate influenţa mult alegerea. Dacă aveţi 24 de linii de ecran (din care 3 folosite la borduri şi meniuri) este greu să scrieţi fişiere mari, pentru că vă plimbaţi cu greu prin ele. Alta este situaţia dacă aveţi 42 sau 80 de linii.

Un editor inteligent reduce handicapul unor fişiere mari, pentru că vă permite să va mutaţi căutînd automat apariţiile funcţiilor şi variabilelor.

Eu personal mă descurc cel mai bine cu fişiere între 300 şi 700 de linii, conţinînd pînă în 15 funcţii.

4 Minima vizibilitate

Principiul minimei vizibilităţi cere ca un obiect să nu fie vizibil decît părţilor din program care au cu adevărat nevoie de el. Să vedem nişte aplicări ale lui:

> Datele trebuie separate cît se poate de mult de corpul programului.

O variantă a acestei reguli este "în sursa programului nu trebuie să apară *nici* un număr cu excepţia lui 0, 1, sau -1". Constantele (în particular cele numerice) se mai numesc şi "magic numbers" (numere magice). Ele trebuie evitate întotdeauna.

Metoda comună este de a da *nume* simbolice constantelor undeva la începutul fiecărui modul, sau în fişiere speciale (cu `const` în Pascal şi C, cu `#define` în C, etc.). Numele dat unei constante are multe avantaje, care sunt expuse în mai toate cursurile de programare. Noi vom reaminti decît pe unul: o constantă face programul *mult* mai uşor de citit. Aceasta este o aplicare a principiului minimei vizibilităţi: ştii numai numele constantei (care trebuie să arate la ce foloseşte ea), şi nu valoarea ei!

Iată un exemplu:

```
for i:=1 to 128 do persoana[i].nume := '';
```

şi tradus:

```
for p:=1 to NumarDePersoane do persoana[p].nume := '';
```

Nu e mai clar a doua oară?

Cum spuneam părţile din program interacţionează te miri cum. Cel mai adesea interacţiunile se fac prin variabile globale, a căror valori pot fi modificate de o funcţie şi citite de alta.

De aceea este bine să transmitem funcţiilor (procedurilor) toate valorile de care au nevoie prin argumente şi să nu le lăsăm să modifice variabile globale.

Dacă vi se pare greu să programaţi fără variabile globale, aflaţi că există limbaje de programare în care nu există variabile de loc, şi totuşi se pot face foarte multe lucruri (un exemplu este Prolog; despre Prolog se crede îndeobşte că are variabile, însă o analiză atentă va arăta că ceea ce în Prolog se numeşte variabilă nu foloseşte de loc la acelaşi lucru ca o variabilă în sensul uzual al limbajelor imperative ca Pascalul).

În realitate este foarte greu adesea să ne descurcăm fără variabile globale, pentru că unele funcţii fac modificări atît de masive încît ar trebui să aibă zeci de parametri şi să returneze tot atîtea valori (ceea ce multe limbaje nici nu permit, dealtfel!). Tocmai pentru a circumveni aceste dezavantaje au fost inventate limbajele orientate pe obiecte (alde C++), în care în loc să dai unei funcţii argumente, grupezi argumentele la un loc într-un obiect (un fel de `record` din Pascal); funcţia însăşi devine un cîmp al obiectului (se va numi "funcţie membru"), şi în loc să chemi funcţia cu cîmpurile obiectului ca argumente, rogi obiectul să-şi aplice singur funcţia.

Această reducere la maximum a vizibilităţii din limbajele orientate pe obiecte (căci cîmpurile unui obiect sunt în mod normal accesibile numai funcţiilor care-i sunt membre) se numeşte "încapsulare" şi constituie unul din avantajele majore ale programării orientate pe obiecte faţă de cea ordinară.

Cu disciplină însă se pot aplica beneficiile încapsulării şi în programe scrise în limbaje fără astfel de construcţii (C, Pascal), cu foarte mult succes.

5 Spaţiile albe; i(n)dentarea

Unii separă părţile din programe prin comentarii baroce ca:

```
{******************************}
{========intrare/iesire========}
{******************************}
```

Ei bine, puţine spaţii albe judicios plasate fac mult mai mult bine lizibilităţii programului. Iată care sunt recomandările mele în ceea ce le priveşte:

1. Separaţi fiecare funcţie de vecinele ei printr-o linie albă.

2. Între titlul funcţiei şi corpul ei eu las mereu un rînd în care pun de obicei comentarii.

3. Între declaraţiile variabilelor locale şi corpul funcţiei las mereu o linie liberă.

4. Puneţi mai multe linii, şi eventual un comentariu de o linie de steluţe între părţi majore ale programului.

5. Am observat că o expresie este mult mai uşor de citit dacă separ cu spaţii unii operatori de argumentele lor. Observaţi:

   ```
   for(i=0;i<ultim;i++) if(!a[i]) total++;
   ```

   faţă de:

   ```
   for (i = 0; i < ultim; i++) if ( ! a[i] ) total++;
   ```

6. Pentru a uşura depanarea este bine să puneţi o singură instrucţiune pe linie! Pentru că linia de mai sus la depanare va executa întreg ciclul `for` la o singură comandă "step", iar liniile de mai jos vor permite să vedeţi ce se întîmplă la fiecare pas:

   ```
   for (i = 0; i < ultim; i++)
     if ( ! a[i] ) total++;
   ```

7. Dacă aveţi prea multe argumente la o funcţie scrieţi aşa:

```
venit(persoana[pozitie_curenta].nume,
      departament[persoana[pozitie_curenta].cod_departament],
      indexare_salarii);
```

8. Dacă trebuie să spargeţi o expresie pe două linii lăsaţi operatorul la începutul celei de-a doua linii.

```
venit_max = persoana[pozitie_curenta].venit
  + crestere_salariu;
```

9. Logica expresiei rupte pe mai multe linii trebuie să fie clară:

```
copii := numar_copii( indice[disc_curent, compartiment,
                             raft_curent],
                      eticheta);
```

10. Nu calculaţi prea multe lucruri într-o singură expresie complicată, pentru că după aceea n-o să înţelegeţi nici dumneavoastră ce aţi vrut:

```
i := 1;
c := 'y';
while (i < maxN) and not (a[i] < 0) and not (c = 'n') do begin
  i := i + 1;
  read(f, c)
end
```

ci:

```
i := 1;
gata := false;
c := 'y';
while not gata do begin
  if i >= maxN then gata := true;
  if a[i] < 0  then gata := true;
  if c = 'n'   then gata := true;
  if not gata  then begin
     i := i + 1;
     read(f, c)
  end
end
```

11. În engleză "to indent" înseamnă "a zimţui". În româna s-a format barbarismul "a identa" sau "a indenta". care înseamnă a aranja frumos în pagină un program, în aşa fel încît structura lui să fie limpede. Mai toate limbajele moderne sunt indiferente la numărul de spaţii, aşa că folosiţi-le ca să indicaţi structura programului. Exemple se găsesc şi mai sus.

    Regula de bază este ca o instrucţiune care depinde de altă instrucţiune să fie scrisă puţin mai la dreapta. De pildă `if` arată aşa: `if <expresie> then <instructiune1> else <instructiune2>`. `<instructiune1,2>` depind de `if`. Un `if` se scrie deci aşa:

```
if TrebuieSters then
   StergeJucator(BazaDeDate, NumarulDeOrdine)
else
   calificari(BazaDeDate, NumarulDeOrdine);
```

12. Dacă instrucţiunea dependentă este o instrucţiune bloc, aveţi mai multe posibilităţi de a o aranja în pagină. Totul e să alegeţi una şi să o respectaţi mereu.

```
for i := 1 to ultim do begin
   CalculeazaNota(i);
   AlegePostura(i)
end;
```

    sau

```
for i := 1 to ultim do
   begin
   CalculeazaNota(i);
   AlegePostura(i)
   end;
```

    sau

```
for i := 1 to ultim do
   begin
   CalculeazaNota(i);
   AlegePostura(i)
end;
```

    sau

```
for ( i = 1; i <= ultim; i++)
  {
    calculeaza_nota(i);
    alege_postura(i);
  }
```

13. Cît de multe spaţii trebuie să se folosească pentru a indenta? 1 este prea puţin, şi structura programului nu apare suficient de clar, 8 este prea mult, pentru că la o indentare de 4 nivele nu mai ajunge ecranul. Cel mai bine este între 2 şi 6.

14. Etichetele de la `goto` sau `case` se indentează puţin mai în stînga:

```
  repeta:
    switch(raspuns) {
    case 'd':
stiut = DA;
break;
    case 'n':
stiut = NU;
break;
    case '?':
interogare();
goto repeta;
    default:  /* alt raspuns */
return;
    }
```

15. Construcţiile de genul `else if` sunt o excepţie de la regula de indentare. Ele pot fi scrise în loc de:

```
if e1 then cutare1
else
   if e2 then cutare2
   else
      if e3 then cutare3
      else cutare3
```

aşa:

```
if e1 then cutare1
else if e2 then cutare2
else if e3 then cutare3
else cutare3
```

pentru că sunt echivalente cu un `case` din Pascal (`switch` din C) — adică un `if` cu mai multe condiţii, care au aceeaşi importanţă.

6 Comentariile

Comentariile sunt într-adevăr extrem de utile pentru a uşura înţelegerea funcţionării unui program. Rolurile lor sunt multiple, şi modul în care se pot folosi extrem de variat.

În primul rînd există două feluri de comentarii:

- comentarii de linie
- comentarii de sfîrşit de linie

Primele ocupă o linie (sau mai multe) de program în întregime şi explică ceva din funcţionarea lui. Iată unde sunt extrem de utile:

1. Orice fişier (modul) trebuie să conţină pe prima sau a doua linie un comentariu arătînd numele fişierului şi explicînd pe scurt la ce foloseşte. Cînd fişierul este scos la imprimantă se acest comentariu indică provenienţa sursei.
2. Orice fişier este bine să conţină la început comentarii descriind autorul, data şi versiunea ultimei modificări ale textului. Există programe speciale ("Source Code Control System" — control al codului sursă) care ajută păstrarea şi folosirea tuturor versiunilor unui program. În cazul în care mai multă lume lucrează la un proiect sunt extrem de utile.
3. Fiecare funcţie trebuie sa aibă o descriere a comportării ei. Vedeţi mai jos secţiunea despre funcţii. Aceste comentarii se pot întinde pe mai multe linii.
4. Algoritmii mai complecşi trebuie să fie descrişi printr-un comentariu care explică sumar comportarea lor. Acest comentariu în mod normal se plasează înaintea zonei de cod care implementează algoritmul.
5. Comentariile se indentează la fel cu liniile de cod de care ţin:

```
for i := 1 to sfirsit do
  { verific toate pozitiile }
  verifica( pozitie[i] );
```

Comentariile de sfîrşit de linie în general explică operaţia făcută pe linia curentă. Ele sunt mult mai greu de întreţinut, pentru că schimbarea uneia din linii strică întreaga lor aliniere.

1. Ele trebuie să însoţească obligatoriu declaraţiile variabilelor globale:

```
var
  persoane,                   { numar de participanti      }
  locuri,                     { numarul de scaune la masa  }
  LocCurent: integer;         { locul pe care incerc sa pun }
```

2. Orice linie care face o acţiune mai deşucheată (al cărei sens nu este limpede) trebuie să fie comentată:

```
/* elementele nu pot fi 0 */
for ( i = 0; i < nr_elem; i++ ) {
  if ( a[i] == 0 ) break;   /* eroare! opresc procesarea */
  if ( a[i] < a[i+1] ) swap (&a[i], &a[i+1]);
}
```

3. Variabilele locale ale unei funcţii trebuie să fie cîteodată comentate.

4. Mai rar se comentează argumentele unei funcţii în declaraţia ei:

```
function arata(x, y, {coordonatele din stinga-sus}
               lungime, latime: integer): boolean;
```

5. Blocurile la mare distanţă de testele care le controlează trebuie să fie comentate astfel (vedeţi la else):

```
if ( functionare == INTERACTIV ) {
   int alege;

   printf("\a\a\a");           /* un pic de galagie */
   alege = meniu_principal();  /* selectia          */
   switch (alege) {
     case FISIER:
        fisier();              /* toata intrarea/iesirea */
        break;
     case CALCUL:
        calcul();
        break;
     default:
        return 0;               /* nimic valid -> termin */
   }
}
else {                 /* aici functionare != INTERACTIV */
        ...
}
```

7 Variabilele, constantele, funcţiile, procedurile

> Dacă nu poţi să-i dai un nume unei variabile înseamnă că acea variabilă nu este bună. Reconsideră arhitectura întregului program.

Într-adevăr, nu trebuie să faci nici un efort ca să-ţi aduci aminte ce reprezintă o variabilă; numele ei trebuie să-ţi spună tot ce te interesează.

Nu faceţi niciodată economie înghesuind în aceeaşi variabilă mai multe valori, depinzînd de context! Fiecare variabilă trebuie să reprezinte un singur lucru în fiecare clipă. Compilatoarele moderne sunt suficient de inteligente ca să transforme cele două variabile în una singură dacă asta se poate! (De fapt le chiar uşuraţi munca neînghesuind mai multe lucruri laolaltă).

Numele unei variabile trebuie să arate exact ce reprezintă valoarea ei.

Numele variabilelor, funcţiilor, procedurilor, argumentelor lor, trebuie să fie de asemenea uşor de reamintit. Dacă scrii un program şi trebuie mereu să te duci la prima pagină ca să vezi dacă o variabilă se cheamă `max_n`. sau `max_numere` sau `maximum_numere` treaba merge destul de greu. În principiu nu se folosesc prescurtări. Un nume de variabilă ca `x` sau `total` este absolut inutil, căci orice poate fi `x`. Scrieţi `x_fereastra` sau `total_venit` de pildă.

Nu trebuie să va fie lene să tastaţi. De altfel un editor foarte bun (ca de exemplu GNU Emacs) vă ajută foarte mult (Emacs la apăsarea tastei `M-/` încearcă să completeze cuvîntul curent cu un sufix care se regăseşte undeva prin text. De pildă dacă am definit variabila `complexitate_totala`. atunci probabil ajunge sa tastez `com` şi apoi `M-/` ca Emacs să completeze întreg numele).

Nu scrieţi funcţii care în funcţie de un argument fac două lucruri diferite:

```
int operatie(FILE * f, struct persoana * p, int actiune)
/* daca actiune = 0 -> cauta de cite ori apare persoana p in fisier
 * daca actiune = 1 -> sterge toate aparitiile p din f
 */
```

Scrieţi două funcţii, pe care le chemaţi dintr-un if sau switch (case).

Este util să folosiţi notaţii diferite pentru constantele numite pentru a le distinge de variabile. În C, în care contează literele mari/mici, aceste constante au de obicei numele scrise numai cu majuscule.

Un program cu nume de variabile eficient alese are nevoie de mult mai puţine comentarii, şi este mult mai uşor de citit şi întreţinut. Cineva spunea că pot fi citite la gura sobei, ca o carte bună. Am avut ocazia să citesc astfel de programe şi trebuie să recunosc că am învăţat extrem de mult din ele. Pentru curioşi recomand doi monştri sacrii ai programării cristaline, Andrew S. Tanenbaum (autorul sistemelor de operare MINIX şi Amoeba) şi Richard Stallman (autorul compilatorului GCC şi al lui GNU Emacs).

Dacă scrieţi module care exportă multe variabile sau funcţii puteţi prefixa toate obiectele unui modul cu un text care le arată provenienţa şi semnificaţia. De exemplu toate procedurile care lucrează cu ferestre se vor numi ceva de genul `Win_xxx` (win de la "window".. Aceasta complică şi simplifică deopotrivă unele lucruri. E mai simplu pentru că poţi avea o procedură `make` şi pentru ferestre, şi pentru stive, numai că una se cheamă `Win_make` iar alta `Stk_make`.

Dacă vă decideţi pentru prefixe, în general ele sunt preferabil sufixelor. Adică e mai bine `Win_make` decît `make_win`. pentru că prefixele se văd mai uşor. În nici un caz nu amestecaţi într-un program cele două feluri de numire.

Apropos de amestecare, faceţi la început nişte decizii de numire omogenă a obiectelor cu care operaţi. De exemplu: "toate comentariile sunt în engleză", sau "toate numele de proceduri sunt cuvinte englezeşti", ş.a.m.d. În program coexistă greu o procedură `numar_persoane` cu una `first_person`. (Dacă nu ştiţi engleză lucrurile sunt ceva mai simple.)

8 Fiabilitatea

“Nu avea încredere în nimeni, nici măcar în tine însuţi!” Respectarea acestei reguli aduce nişte beneficii greu de imaginat. Ce înseamnă ea pentru un programator?

1. Funcţiile nu trebuie niciodată să se bazeze pe cel care le cheamă, cum că le furnizează argumente corecte. Verificaţi toate argumentele înainte de a le prelucra; dacă nu sunt corecte anunţaţi eroarea, fie scriind ceva pe ecran, fie returnînd o valoare indicînd eroare utilizatorului.

2. Funcţiile nu trebuie să se bazeze niciodată pe cele pe care le cheamă. De fiecare dată cînd aţi chemat o altă funcţie verificaţi dacă nu cumva a semnalat o eroare. Dacă da acţionaţi în consecinţă, şi opriţi procesarea.

3. Un caz particular al lui 2. este apelul unor funcţii din biblioteci. Mai ales cele care fac intrare/ieşire pot eşua din zeci de motive. Verificaţi *mereu* dacă au funcţionat înainte de a vă arunca cu capul înainte. Dacă vă e lene, scrieţi funcţii “înveliş” (“wrappers”) care le cheamă pe cele de bibliotecă şi care verifică erorile. Folosiţi apoi numai wrapper-ele. De exemplu:

   ```
   void * aloca_memorie(size_t marime)
           /* aloca memorie.  eroare fatala daca nu mai este memorie */
   {
           void * p;  /* aici alocam temporar */

           p = malloc(marime);
           if (p == NULL) {
                   /* standardul spune ca malloc intoarce NULL numai
                    * daca nu a reusit sa aloce cantitatea indicata */
                   fprintf(stderr, " Alocare nereusita!\n");
                   exit(1);
           }
           return p;
   }
   ```

   sau (în TURBO-Pascal):

   ```
   procedure deschide_fisier(var f:text; nume:string);
           { deschide un fisier; verifica erorile }
   begin
        {$i-}      { inhiba erori de executie pentru intrare-iesire }
        assign(f, nume);
        reset(fis);
        {$i+}      { inversul lui $i- }
        if ioresult <> 0 then begin
           writeln('deschide_fisier: Nu pot deschide fisierul ', nume);
   ```

```
        halt
    end
end;
```

4. Din loc în loc introduceţi verificări asupra datelor pe care le prelucraţi, chiar dacă vi se par inutile. În C există funcţia standard `assert` care este *extrem* de utilă, dar puteţi scrie şi în Pascal una asemănătoare. Această funcţie opreşte definitiv programul dacă este chemată cu un argument 0. Frumuseţea este că toate invocările lui `assert` sunt scoase definitiv din program doar definind macro-ul `NDEBUG`.

   Metoda universală este următoarea: definiţi constanta booleană (sau întreagă) `DEPANARE`, pe care o faceţi `true`. Apoi în program verificaţi mereu astfel:

   ```
   if DEPANARE then begin
       { verificam daca locasele sunt bine initializate }
       for i := 1 to spatii do
            if locas[i].urmator = nil then begin
                 writeln('Locas cu "urmator" nil la indicele ', i);
                 halt
            end
   end
   ```

   Testele pentru locaşe se vor efectua numai dacă `DEPANARE` este `true`. Dacă faceţi `DEPANARE` false, testele dispar cu desăvîrşire! Puteţi astfel să vă depanaţi programul, iar cînd merge să-l scurtaţi. Un compilator cu optimizări va observa că liniile de test nu se vor putea efectua niciodată (pentru că `DEPANARE` este o constantă), şi le va scoate cu totul din program. (Pe de altă parte unele teste este înţelept să rămînă pentru totdeauna în program; nu se ştie niciodată pentru ce date de intrare programul o ia razna. Este recomandabil să găseşti o eroare cît mai repede şi să urlii că ai găsit-o decît să o ignori o vreme şi ea să facă prăpăd în linişte. Mulţi utilizatori vor aprecia.)

5. Distingeţi în programele dumneavoastră (cel puţin) două tipuri de erori: fatale şi recuperabile. Scrieţi două funcţii care tratează aceste tipuri de erori. Toate erorile ar trebui să indice cît mai clar locul unde s-au produs: numele programului, numele modulului, numele funcţiei, tipul erorii. La depanare nu ajută prea tare dacă vezi ceva de genul: "eroare de intrare/iesire". Mai bine e "statistic: functia 'citeste_data': eroare: nu pot deschide fisierul 'info.txt'".

Tocmai din această cauză nu orice eroare trebuie semnalată ca fiind fatală. Nu poţi să faci o împărţire la 0? Anunţă doar pe funcţia care te-a chemat cu argumente greşite că ceva nu e în regulă. Cum ar fi un program de calculator de buzunar care s-ar opri la o astfel de eroare? Penibil! Eroarea trebuie doar să fie anunţată şi procesarea continuată.

Poate vom reveni cu un articol special despre tratamentul erorilor, pentru că este un subiect extrem de interesant, şi foarte prost tratat în literatură. O eroare se poate adesea propaga în sus pe un lanţ de funcţii care s-au chemat între ele (mai ales dacă una era recursivă) la distanţe extrem de mari de locul producerii. Cum se face asta cu grijă, cum se semnalează erorile cel mai simplu, asta merită discutat mai pe-ndelete.

9 Documentaţia

Donald Knuth (unul din cei cu citatele din introducere) este unul dintre adepţii a ceea ce se numeşte "litterate programming". Acesta este un sistem (numit Web) în care programul şi documentaţia sa se scriu dintr-un singur foc, apoi cu două compilatoare diferite se obţin executabilul şi cartea de descriere a produsului. Paradoxal, această metodă măreşte productivitatea şi fiabilitatea, pentru că îţi poţi introduce explicaţiile detaliate chiar în corpul programului în timp ce-l scrii, gîndind mai uşor atît asupra programului cît şi documentaţiei.

Dacă nu spuneţi *cum* face ceva un program de-al dumneavoastră, trebuie totuşi să spuneţi *ce* face. Chiar dacă nu scrieţi programe pentru vînzare este un obicei extrem de util să scrieţi scurte documentaţii pentru utilizatorii lor sau cei care vor să le modifice. Un model împămîntenit este pagina de manual interactivă din UNIX (sau comanda "help" din DOS, într-un fel). O descriere a programului dumneavoastră ar trebui să conţină următoarele informaţii:

1. numele programului;
2. modul de lansare (argumente, etc);
3. o descriere a comportării;
4. pentru programe interactive indicaţii sumare de folosire;
5. informaţii despre fişierele pe care le foloseşte şi structura lor;
6. versiunea curentă;
7. autorul programului cu adresa lui (nu se recomandă dacă programul este un virus);
8. defecţiuni cunoscute;
9. documentaţii înrudite.

10 Portabilitatea

Dacă trebuie să folosiţi trăsături ale limbajului care *nu* sunt standard (de exemplu în Turbo C `gotoxy()`, atunci nu le chemaţi niciodată direct; scrieţi pentru ele "wrappere". Astfel de wrappere ar trebui să asigure o comportare uniformă a funcţiilor, chiar dacă bibliotecile de care dispuneţi se schimbă.

De exemplu dacă faceţi un program pe PC care lucrează cu mai multe culori, dacă vreţi să meargă bine şi pe plăci monocrome şi color scrieţi o funcţie `culoare_text()` cam aşa:

```
void culoare_text(int culoare)
        /* alege culoarea textului in functie de numarul
           de culori disponibile (2 sau 16) */
{
        int cul_reala;

        /* MONOCROM este aflat de exemplu intrebind utilizatorul
```

```
           la inceputul programului */
        if (MONOCROM) cul_reala = culoare / 8;
          /* daca avem numai 2 culori pastram numai primul bit */
        else cul_reala = culoare;
        textcolor(cul_reala);
}
```

Dacă vă concepeţi programele într-un mod cît mai general vor deveni adesea şi ceva mai uşor de scris, şi cu siguranţă foarte uşor de întreţinut şi "portat" (mutat de la un compilator la altul).

Subiectul este foarte generos, iar tratarea pe care i-am dat-o aici este departe de a fi exhaustivă. Numai experienţa vă va convinge de utilitatea unei discipline în actul programării. Deci spor la treabă!