Programarea calculatoarelor numerice

Descrierea CIP a Bibliotecii Naționale a României
MORARU, FLORIAN
Programarea calculatoarelor numerice / Florian Moraru, Carmen Odubasteanu. - Bucureşti : Editura Academiei Oamenilor de Ştiinţă din România, 2011
Bibliogr.
Index
ISBN 978-606-8371-14-6

I. Odubăşteanu, Carmen

004.42

Editura Academiei Oamenilor de Ştiinţă din România

Adresa: Splaiul Independenţei, nr. 54, sectorul 5, cod 050094 Bucureşti, România

Redactor: ing. Mihail CĂRUŢAŞU

Documentarist: ing. Ioan BALINT

Coperta: ing. sist. Adrian Nicolae STAN

Florian Moraru

Carmen Odubasteanu

Programarea calculatoarelor numerice

Editura Academiei Oamenilor de Ştiinţă din România

Bucureşti

2011

Prefață

Un program pentru calculator este cea mai precisă şi cea mai concisă descriere a unui algoritm, descriere care foloseste un limbaj de programare. Un algoritm este o metodă de rezolvare a unei probleme.

Învățarea şi utilizarea unui limbaj de programare oferă mai multe avantaje:

- Pune la dispoziție un instrument cu care se pot crea aplicații utile;
- Permite utilizarea calculatorului într-un mod superior față de utilizarea unor programe existente (scrise de alte persoane)
- Permite formarea şi exersarea unei gândiri algoritmice, utile şi în afara activității de programare.
- Dezvoltă calități intelectuale şi pofesionale cum ar fi gândirea logică şi analitică, atenția la detalii, precizia şi concizia exprimării, s.a.

De aceea, în toată lumea, învățarea unui limbaj de programare şi a deprinderilor de gândire algoritmică începe tot mai devreme în cadrul procesului educational, chiar dacă limbajul de programare folosit poate fi subiect de discuții şi poate evolua în timp, pe măsură ce utilizarea calculatoarelor şi tehnologia informației progresează.

Înca de mai multi ani limbajul C şi extensia sa C++ s-au impus ca prim limbaj de programare, cu toate că alte limbaje sunt mai simple sau mai sigure în utilizare. Principalele argumente în favoarea limbajelor C, C++ sunt: existența unui standard care nu a mai necesitat extinderi ulterioare cu facilități nestandard şi posibilitatea de a scrie programe eficiente pentru aproape orice tip de aplicație. Ca dovadă a alegerii limbajelor C şi C++ în şcolile şi facultățile din lume este şi faptul că la concursurile internaționale de programare pentru elevi sau studenți C şi C++ sunt printre puținele limbaje admise oficial şi sunt pe departe cele mai folosite de concurenți în rezolvarea problemelor.

Limbajele de programare pot fi de uz general sau specializate pe anumite tipuri de aplicatii. Limbajul de programare folosit este important pentru că el determină efortul şi timpul de scriere şi de punere la punct a programului. Pentru anumite programe folosite frecvent este important şi timpul de execuție (de rulare) al programului. Limbajul C este un limbaj de uz general care, prin compilare, produce programe compacte şi cu timp mic de execuție.

Limbajele care au apărut după limbajul C (inclusiv C++) au preluat practic toate instrucțiunile limbajului, modul de comentare a programelor şi alte elemente de limbaj; dintre limbajele procedurale menționăm PHP, iar dintre limbajele orientate pe obiecte C++, Java, C#, PHP5, ş.a.

În mod uzual se foloseşte limbajul C cu câteva inovații preluate din C++ pentru simplificarea unor operații de citire-scriere, de alocare a memoriei, de transmitere a unor argument modificabile la funcții ş.a. Facilitățile existente în C++ pentru programarea cu obiecte şi definirea de noi clase nu sunt folosite de începători pentru că ar presupune o abordare diferită a scrierii de programe.

Paradigma programării cu obiecte este superioară programării procedurale (structurate) în cazul unor aplicaţii mari şi complexe, dar pentru programe mici şi pentru început este preferabilă abordarea clasică (procedurală). Totuşi, programele C tratate ca fiind scrise în C++ sunt verificate mai strict şi astfel mai multe erori de programare sunt detectate la compilare şi nu se mai manifestă ca erori la execuţie. De aceea recomandăm salvarea surselor programelor în fisiere cu extensia cpp chiar dacă nu se foloşeste nimic din ceea ce aduce nou C++ faţă de C.

Limbajul C permite mai multe stiluri de scriere a programelor şi se poate ajunge uşor la programe foarte compacte dar aproape criptice şi dificil de înţeles. Nu recomandăm acest stil de programare care face dificilă înţelegerea de către oameni a programelor, pentru că orice program evoluează în timp; el este modificat fie de către autorii lui fie de alţi programatori, pentru a răspunde cerinţelor beneficiarilor şi pentru optimizarea performanţelor sale.

Pe de altă parte, trebuie utilizate (fără a abuza) facilităţile oferite de fiecare limbaj în parte, iar limbajul C a adus multe inovaţii fată de limbajele folosite anterior (Basic, Fortran, Pascal). Exemplele din textul care urmează încearcă să păstreze acest echilibru între claritatea programelor şi utilizarea posibilităţilor oferite de limbajele C/C++.

Acest support de curs urmează programele analitice şi manualele folosite în scolile profesionale şi liceele din Romania, în sensul că sunt prezente toate elementele din nucleul limbajului C predate la acest nivel şi că nu se preia din limbajul C++ mai mult decât prevăd aceste programe.

Am încercat să reducem cât mai mult partea de prezentare a sintaxei limbajului pentru a face loc unor observaţii asupra erorilor uzuale la scrierea programelor C şi asupra unor tehnici de programare, rezultate dintr-o experienţă îndelungată de predare şi de îndrumare în laborator a studentilor anilor întâi din facultatea de Automatica şi Calculatoare din UPB.

Pentru o descriere completă a unor elemente de limbaj mai rar utilizate se pot folosi lucrările recomandate ca bibliografie, iar pentru mai multe exemple se pot folosi culegeri de probleme de programare rezolvate .

Programarea calculatoarelor nu se învaţă doar citind cărti şi programe scrise de alţii, fără a lucra efectiv la calculator. Cea mai bună abordare este îmbinarea experienţei altora cu experienţa proprie de programare şi examinarea a cât mai multor variante pentru o aceeasi problemă. În plus este necesară formarea unei tehnici de găsire a erorilor din programe (erori ce se produc la rulare) cu sau fără utilizarea unor instrumente software pentru depanare. De asemenea este bine să se folosească un mediu integrat pentru dezvoltarea de programe (IDE = Integrated Development Environment), care poate fi unul mai simplu (Dev-Cpp, CodeBlocks) sau unul mai elaborat (Netbeans, Eclipse).

Cuprins

Capitolul 1: Introducere 9
Introducere în programare. Limbajele C şi C++ 9
Elemente de bază ale limbajului 13

Capitolul 2: Operații de intrare-ieşire 21
Funcții standard de intrare-ieşire 21
Funcții de citire-scriere caractere şi şiruri de caractere 21
Funcții de citire-scriere cu format 22
Observații 25
Fluxuri de intrare-ieşire în C++ 26

Capitolul 3: Instrucțiuni 27
Instrucțiunea expresie 27
Instrucțiunea compusă (bloc) 27
Instrucțiunea *if* 27
Instrucțiunea *switch* 29
Instrucțiunea *while* 29
Instrucțiunea *for* 30
Instrucțiunea *do* 31
Instrucțiunile *break* şi *continue* 32

Capitolul 4: Funcții 33
Importanța funcțiilor în programare 33
Definirea şi utilizarea funcțiilor 33
Domeniu de vizibilitate (scope) 36
Funcții recursive 37
Funcții în C++ 38

Capitolul 5: Tablouri 39
Tablouri unidimensionale: vectori 39
Tablouri bidimensionale: matrice 40
Funcții cu argumente vectori 42

Capitolul 6: Pointeri 43
Variabile pointer 43
Operații cu pointeri la date 44
Vectori şi pointeri 45

Pointeri în funcţii 47
Pointeri la funcţii 48
Tipul referinţă în C++ 49

Capitolul 7: Structuri 51
Definirea de tipuri şi variabile structură 51
Utilizarea tipurilor structură 53
Funcţii cu argumente şi rezultat structură 54
Structuri cu conţinut variabil (uniuni) 55
Enumerări 56

Capitolul 8: Şiruri de caractere 57
Memorarea şirurilor de caractere în C 57
Funcţii standard pentru operaţii cu şiruri 58
Erori uzuale la operaţii cu şiruri de caractere 60
Definirea de noi funcţii pe şiruri de caractere 61
Argumente în linia de comandă 62

Capitolul 9: Alocarea dinamică a memoriei 63
Funcţii de alocare şi eliberare a memoriei 63
Vectori alocaţi dinamic 64
Matrice alocate dinamic 65
Structuri alocate dinamic 66
Operatori pentru alocare dinamică în C++ 66

Capitolul **10**: Fişiere de date 69
Tipuri de fişiere 69
Funcţii pentru deschidere şi închidere fişiere 70
Operaţii uzuale cu fişiere text 71
Citire-scriere cu format 72
Funcţii de acces secvenţial la fişiere binare 72
Funcţii pentru acces direct la date 74
Fişiere în C++ 75
Redirectarea fişierelor standard 76
Bibliografie 77

Capitolul 1

Introducere

Introducere în programare. Limbajele C şi C++

Algoritmi şi programe

Un algoritm este o succesiune de etape care se pot aplica mecanic pentru rezolvarea unei clase de probleme. Cerințele pe care trebuie să le îndeplinească un algoritm sunt următoarele:

- Claritate: să nu existe ambiguități
- Generalitate: să poată fi aplicat pentru o întreagă clasă de probleme
- Finitudine: să furnizeze rezultatul în timp finit

În general un algoritm nu se poate reduce la aplicarea unor formule de calcul; un algoritm conține operații executate condiționat, numai pentru anumite date, şi operații repetate de un număr de ori, în funcție de datele problemei.

Algoritmii mai simpli pot fi exprimați direct într-un limbaj de programare, dar pentru un algoritm mai complex se practică descrierea algoritmului fie sub formă grafică (organigrame sau scheme logice), fie folosind un "pseudocod", ca un limbaj intermediar între limbajul natural şi un limbaj de programare. Un pseudocod are reguli mai puține şi descrie numai operațiile de prelucrare (nu şi variabilele folosite). Nu există un pseudocod standardizat sau unanim acceptat. Exemplu de pseudocod pentru algoritmul lui Euclid care determină cel mai mare divizor comun a doi întregi, prin împărțiri repetate:

```
citeşte a şi b
r = rest împărțire a prin b
repeta cât timp r > 0
    a=b
    b=r
    r=rest impartire a prin b
scrie b
```

Prin deplasarea spre dreapta a celor trei linii s-a arătat că acele operații fac obiectul repetării (nu însă şi operația de scriere), punându-se astfel în evidență structura de blocuri, adică ce operații fac obiectul unei comenzi de repetare (*repetă*) sau de selecție (*dacă*).

Un program are un caracter general şi de aceea are nevoie de date inițiale care particularizează programul pentru o situație concretă. Rezultatele produse de un program pe baza datelor inițiale sunt de obicei afişate pe ecran. Datele de intrare se introduc manual de la tastatură sau se citesc din fişiere disc.

Operațiile uzuale din limbajele de programare sunt operații de prelucrare (calcule, comparații etc) şi operații de intrare-ieşire (de citire-scriere). Aceste

operaţii sunt exprimate prin instrucţiuni ale limbajului sau prin apelarea unor funcţii standard predefinite (de bibliotecă). Desfăşurarea în timp a instrucţiunilor de prelucrare şi de intrare-ieşire este controlată prin instrucţiuni de repetare (de ciclare) şi de selecţie (de comparaţie).

Fiecare limbaj de programare are reguli gramaticale precise, a căror respectare este verificată de programul compilator (translator).

Dezvoltarea de programe

Un program scris într-un limbaj independent de maşină trebuie mai întâi tradus de către un program translator sau compilator. Compilatorul citeşte şi analizează un text sursă (în limbaj C, de ex.) şi produce un modul obiect (scris într-un fişier), dacă nu s-au găsit erori în textul sursă. Pentru programele mari este uzual ca textul sursă să fie format din mai multe fişiere sursă, care să poată fi scrise, compilate, verificate şi modificate separat de celelalte fişiere sursă.

Mai multe module obiect, rezultate din compilări separate sunt legate împreună şi cu alte module extrase din biblioteci de funcţii standard într-un program executabil de către un program numit editor de legături (*Linker*). Execuţia unui program poate pune în evidentă erori de logică sau chiar erori de programare care au trecut de compilare (mai ales în limbajul C).

Cauzele erorilor la execuţie sau ale unor rezultate greşite nu sunt de obicei evidente din cauză că ele sunt efectul unui număr mare de operaţii efectuate de calculator. Pentru descoperirea cauzelor erorilor se poate folosi un program depanator (*Debugger*) sau se pot insera instrucţiuni de afişare a unor rezultate intermediare în programul sursă, pentru trasarea evoluţiei programului.

Fazele de modificare (editare) a textului sursă, de compilare, linkeditare şi execuţie sunt repetate de câte ori este necesar pentru a obţine un program corect. Testarea unui program cu diverse date iniţiale poate arăta prezenţa unor erori şi nu absenţa erorilor, iar efectuarea tuturor testelor necesare nu este posibilă pentru programe mai complexe (pentru un compilator sau un editor de texte, de exemplu).

Limbajul C

Limbajul C s-a impus în principal datorită existenţei unui standard care conţine toate facilităţile necesare unui limbaj pentru a putea fi folosit într-o mare diversitate de aplicaţii, fără a fi necesare abateri sau extinderi faţă de standard. În plus, există un număr relativ mare de funcţii uzuale care fac parte din standardul limbajului şi care contribuie la portabilitatea programelor C pe diferite platforme (sisteme de operare).

Unii programatori apreciază faptul că limbajul C permite un control total asupra operaţiilor realizate de procesor şi asupra funcţiilor sistemului de operare gazdă, aproape la fel ca şi limbajele de asamblare. Astfel se explică de ce majoritatea programelor de sistem, utilitarele şi multe programe de aplicaţii sunt scrise de mai mulţi ani în limbajul C. Comparativ cu limbajele mai noi (Java, C#)

limbajul C permite generarea unui cod maşină foarte eficient, aspect important pentru programele executate frecvent (compilatoare, sisteme de operare, ş.a.).

Limbajul C permite scrierea unor programe foarte compacte, ceea ce poate fi un avantaj dar şi un dezavantaj, atunci când programele devin criptice şi greu de înțeles. Scurtarea programelor C s-a obținut prin reducerea numărului de cuvinte cheie, prin existența unui număr mare de operatori exprimați prin unul sau prin două caractere speciale dar şi prin posibilitatea de a combina mai mulți operatori şi expresii într-o singură instrucțiune.

Utilizarea directă de pointeri (adrese de memorie) de către programatorii C corespunde lucrului cu adrese de memorie din limbajele de asamblare şi permite operații imposibile în alte limbaje, dar în timp s-a dovedit a fi o sursă importantă de erori la execuție, greu de depistat.

Programarea în C este mai puțin sigură ca în alte limbaje (Pascal, Java) şi necesită mai multă atenție. Limbajul C permite o mare diversitate de construcții corecte sintactic (care trec de compilare), dar multe din ele trădează intențiile programatorului şi produc erori greu de găsit la execuție. Un exemplu este utilizarea greşită a operatorului de atribuire '=' în locul operatorului de comparare la egalitate '=='.

Limbajul C++

Limbajul C++ este o extindere a limbajului C în două directii:

- Imbunătățiri rezultate din experiența de utilizare a limbajului C: alt fel de comentarii, operatori pentru a simplifica alocarea şi eliberarea de memorie, alte utilizări pentru operatorii existenti, argumente de tip referință mai usor de folosit, operații de citire-scriere mai simple (fără pointeri), variabile declarate oriunde ş.a.

- Introducerea tipului „clasă" şi alte facilități pentru programare orientată pe obiecte, plus o bibliotecă standard de clase predefinite (STL), care simplifică mult scrierea unor programe cu diverse structuri de date.

Compilatorul C++ face mai multe verificări asupra textului sursă ceea ce permite descoperirea unor erori de programare la compilare şi nu la executie.

Programarea în C++ este în general mai simplă şi mai sigură decât în C şi produce un cod executabil aproape la fel de eficient (ca memorie şi ca timp).

Cu mici excepții, există compatibilitate între cele două limbaje, în sensul că un program C este acceptat de compilatorul C++ şi produce aceleaşi rezultate la execuție. Multe compilatoare acceptă ambele limbaje şi tratează conținutul unui fişier sursă în funcție de extensia fişierului: fişierele cu extensia CPP conțin programe C++, iar fişierele cu orice altă extensie se consideră a fi scrise în C.

Structura unui program C

Un program C este compus din mai multe funcții şi trebuie să conțină cel puțin funcția *main*, cu care începe executia programului. Funcțiile pot face parte dintr-un singur fişier sursă sau din mai multe fişiere sursă.

Un fişier sursă C este un fişier text care conţine definiţii de funcţii şi, eventual, directive preprocesor, definiţii de tipuri,declaraţii de variabile în afara funcţiilor şi declaratii de funcţii (prototipuri).

O definitie de funcţie C are un antet şi un bloc de instrucţiuni (prin care se execută anumite acţiuni) încadrat de acolade. În interiorul unei funcţii există de obicei declaraţii de variabile precum şi alte blocuri de instrucţiuni.

Regula generală este că o funcţie trebuie definită sau declarată înainte de a fi folosită (apelată), astfel încât compilatorul să poată verifica utilizarea sa corectă. O declaraţie de funcţie menţionează tipul şi numele funcţiei precum şi tipul argumentelor funcţiei. Dacă este posibil să definim orice funcţie înainte de a o folosi nu mai este necesară declararea ei.

Declaratiile funcţiilor definite într-un fisier (bibliotecă) şi folosite într-un alt fisier sursă sunt de obicei grupate în fisiere antet, incluse în compilare.

Urmează un exemplu de program C minimal, cu o funcţie "main" ce conţine o singură instrucţiune (apelul funcţiei *printf*) şi nu conţine declaraţii:

```
#include <stdio.h>
int main ( ) {
    printf ("Primul program!\n");
    return 0;
}
```

Execuţia programului începe cu prima linie din *main*. Cuvântul din faţa funcţiei *main* reprezintă tipul funcţiei (*int* arată că trimite un rezultat de tip întreg, prin instrucţiunea *return*). Parantezele care urmează cuvântului "main" arată că numele *main* este numele unei funcţii (şi nu este numele unei variabile), dar o funcţie fără parametri (*main* poate avea şi argumente). Acoladele sunt necesare pentru a delimita corpul unei funcţii, corp care este un bloc de instrucţiuni şi declaraţii. Funcţia "printf" realizează afişarea pe ecran a textului *Primul program!*. Directiva *#include* este necesară pentru folosirea funcţiei *printf*: fisierul inclus „stdio.h” contine declaratii pentru funcţiile standard de citire-scriere.

Alte observaţii: cuvintele cheie sunt scrise cu litere mici, instrucţiunile se termină cu '; ', şirurile de caractere sunt incluse între ghilimele, caracterul *'\n'* („new line”) poziţionează cursorul la începutul liniei următoare, instructiunea „return” poate lipsi din funcţia „main” deoarece *main* nu este apelată din altă funcţie (rezultatul lui „main” este un cod de terminare corectă sau incorectă a programului şi poate fi folosit de către sistemul de operare).

Directive preprocesor

Un program C conţine una sau mai multe linii iniţiale, care încep toate cu caracterul ‘#’. Acestea sunt directive pentru preprocesorul C şi sunt interpretate înainte de a se analiza programul propriu-zis. Cele mai folosite directive sunt *#include* şi *#define*.

- #define ident text

înlocuieşte toate aparitiile identificatorului *ident* prin şirul *text*

- #include <fişier> sau #include ”fisier”

cere includerea în compilare a unor fişiere sursă C, care sunt de obicei fişiere *antet* (*header*), ce reunesc declaraţii de funcţii şi de constante.

De exemplu, în C nu există instrucţiuni de citire şi de scriere, dar există mai multe funcţii standard destinate acestor operaţii. Declaraţiile funcţiilor standard de I/E sunt reunite în fişierul antet *stdio.h* (*Standard Input-Output*), care trebuie inclus în compilare:

```
#include <stdio.h>          // sau  #include <STDIO.H>
```

De obicei toate directivele *include* şi *define* se scriu la începutul unui fişier sursă, în afara funcţiilor, dar pot fi scrise şi între funcţii dacă respectă regula generală că “definiţia trebuie să preceadă utilizarea”.

Din motive de spatiu în exemplele care urmează pot lipsi directivele de includere necesare pentru funcţiile folosite, dar ele sunt necesare pentru o compilare fără erori.

Elemente de bază ale limbajului

Alfabetul şi atomii lexicali

Caracterele se codifică conform *codului ASCII* (American Standard Code for Information Interchange) pe 8 biţi (un octet). Sunt 256 (0 - 255) de caractere, unele afisabile (litere, cifre, alte semne) şi unele neafisabile .

Limbajul C este case-sensitive, adică se face diferenţă între litere mici şi litere mari, iar cuvintele cheie ale limbajului trebuie scrise cu litere mici.

La compararea de caractere sau de siruri trebuie stiut că spaţiul (blanc) are codul mai mic decât toate simbolurile grafice, iar cifrele (în ordine crescătoare), literele mari şi literele mici (în ordine alfabetică) ocupă câte trei zone compacte.

Atomii lexicali pot fi: identificatori, constante (numerice, caracter, şir), operatori şi semne de punctuaţie.

Un atom lexical trebuie scris integral pe o linie şi nu se poate extinde pe mai multe linii. În cadrul unui atom lexical nu se pot folosi spaţii albe (excepţie fac spaţiile dintr-o constantă şir). Respectarea acestei reguli poate fi mai dificilă în cazul unor şiruri constante lungi, dar există posibilitatea prelungirii unui şir constant de pe o linie pe alta folosind caracterul '\'.

Aceşti atomi sunt separaţi prin s*eparatori*, care pot fi: spaţiul, caracterul de tabulare orizontală *t*, terminatorul de linie *n*, comentariul.

Un program este destinat unui calculator, dar programul trebuie citit şi înţeles şi de către oameni; de aceea se folosesc comentarii care explică de ce se fac anumite acţiuni. Iniţial în limbajul C a fost un singur tip de comentariu, care începea cu secvenţa /* şi se termina cu secvenţa */. Ulterior s-au adoptat şi comentariile din C++, care încep cu secvenţa // şi se termină la sfârşitul liniei care conţine acest comentariu.

Comentariile /* ... */ se pot folosi şi pentru inactivarea unor secvente de cod la depanarea programului (se elimină din compilare anumite funcţii).

Identificatorii şi cuvintele cheie

Identificatorii pot fi nume de variabile, constante sau funcţii sau cuvinte cheie. Convenţia pentru identificatori este ca primul caracter să fie o literă sau _. Exemple: *suma, _produs, x2, X5a, PI, functia_gauss, etc.*

Cuvintele cheie se folosesc în declaraţii şi instrucţiuni şi nu pot fi folosite ca nume de variabile sau de funcţii (sunt cuvinte rezervate ale limbajului). Exemple de cuvinte cheie:

int	typedef	goto	short
extern	static	switch	break
double	do	union	if
char	else	return	long
register	for	case	
float	while	sizeof	continue
unsigned	struct	default	auto

Standardul ANSI C a mai adăugat:

enum const signed void volatile

Numele de funcţii standard (scanf, printf, sqrt, etc.) nu sunt cuvinte cheie, dar nu se recomandă utilizarea lor în alte scopuri, ceea ce ar produce schimbarea sensului iniţial, folosit în majoritatea programelor.

Tipuri de date

În C există tipuri de date fundamentale şi tipuri de date derivate (tipuri structurate - tablouri, structuri - şi tipul pointer). Tipurile fundamentale sunt: caracter, întreg, virgulă mobilă, virgulă mobilă dublă precizie, nedefinit (*void*).

Tipul *void* nu poate fi folosit pentru declararea de variabile sau tablouri; el este folosit pentru funcţiile fără rezultat asociat cu numele funcţiei, pentru pointeri la un tip de date necunoscut (void*) şi pentru a marca absenta argumentelor.

În C există mai multe tipuri de întregi şi respectiv de reali, ce diferă prin memoria alocată, deci prin numărul de cifre memorate şi prin domeniul de valori. Implicit toate numerele întregi sunt numere cu semn, dar prin folosirea cuvântului cheie *unsigned* la declararea lor se poate cere interpretarea ca numere fără semn.

Tipurile întregi sunt: *char, short, int, long, long long.* Tipuri neîntregi: *float double, long double* (numai cu semn).

Standardul C din 1999 prevede şi tipul boolean _Bool (sau bool) pe un octet.

Reprezentarea internă şi numărul de octeţi necesari pentru fiecare tip nu sunt reglementate de standardul limbajului C, dar limitele fiecărui tip pentru o anumită implementare a limbajului pot fi aflate din fişierul antet *limits.h* (care conţine şi nume simbolice pentru aceste limite - *INT_MAX* şi *INT_MIN* .

De obicei tipul *int* ocupă 4 octeţi, iar valoarea maximă este de cca. 10 cifre zecimale pentru tipul int. Depăşirile la operaţii cu întregi de orice lungime nu sunt semnalate deşi rezultatele sunt incorecte în caz de depăşire.

Reprezentarea numerelor reale în diferite versiuni ale limbajului C este mai uniformă deoarece urmează un standard IEEE de reprezentare în virgulă mobilă.

Pentru tipul float domeniul de valori este între 10E-38 şi 10E+38 iar precizia este de 6 cifre zecimale exacte. Pentru tipul double domeniul de valori este între 10E-308 şi 10E+308 iar precizia este de 15 cifre zecimale.

Constante

Constante există doar pentru tipurile primitive şi pentru siruri de caractere, iar tipul constantelor rezultă din forma lor de scriere.

Exemple de constante de tip întreg (implicit int): 123, +16, -5, 045 (octal), 0x2A5 (hexa), 1234L, 12345ul.

Exemple de constante de tip real (implicit double): -1.5e-5, 2.e-4, 12.4, -12.3f (*float*), 7.5E-14F (*float*).

Constantele de tip caracter se reprezintă pe un octet şi au ca valoare codul ASCII al caracterului respectiv. În reprezentarea lor se foloseşte caracterul apostrof: 'A' (cod ASCII 65), 'b', '+'. Pentru caractere speciale se foloseşte caracterul \. Exemple: '\'' - pentru apostrof, '\\' - pentru backslash.

Tot aici intră şi secvențele escape, pentru caractere neafisabile, scrise sub forma unei secvențe ce începe cu \, urmat de o literă: '\n' = new line, '\t' =tab, '\b' = backspace etc), sau urmat de codul numeric al caracterului în octal sau în hexazecimal (\012 = \0x0a = 10 este codul pentru caracterul de linie nouă '\n').

Constantele şir de caractere sunt succesiuni de octeți ce conțin codurile ASCII ale caracterelor din şir şi un octet nul la sfârşit (terminator de şir). Pentru valori de tip şiruri de caractere se folosesc caracterele *""*: "abc", "A", "abc01".

Tipul constantei "a" este char* şi este deci diferit de tipul constantei 'a'; un argument formal de tip sir nu poate fi înlocuit cu constanta 'r', chiar daca sirul are un singur caracter (în funcția „fopen", de exemplu).

Se recomandă definirea de constante simbolice, prin asocierea unui nume sugestiv constantei; în felul acesta se poate modifica mai usor şi mai sigur valoarea unei constante (cum ar fi dimensiunea unui vector).

Numele unor constante simbolice predefinite folosesc litere mari: `EOF`, `M_PI`, `INT_MAX`, `INT_MIN`, NULL. Aceste constante simbolice sunt definite în fişiere de tip "h": EOF şi NULL în "stdio.h", M_PI în "math.h".

Definirea unei constante simbolice se face utilizând directiva *#define* astfel:

```
#define identificator [text]
```

Exemplu:

```
#define M 100
```

În C++ se preferă altă definire pentru constante simbolice, utilizând cuvântul cheie *const*, dar constantele *const* nu sunt identice cu cele din *define*. Exemplu

```
const double M=100 ;
```

Variabile

Orice program prelucrează date inițiale şi produce o serie de rezultate. Pe lângă acestea pot fi necesare date de lucru, pentru păstrarea unor valori folosite în

prelucrare. Toate aceste date sunt memorate la anumite adrese, dar programatorul se referă la ele prin nume simbolice: nume de „variabile”.

Pentru a preciza tipul unei variabile este necesară declararea acesteia. O declarație trebuie să specifice numele variabilei (ales de programator), tipul variabilei şi, eventual, alte atribute. O definiție de variabilă poate fi însoțită de inițializarea ei cu o valoare, valoare care poate fi ulterior modificată.

O definiție de variabilă alocă şi memorie iar o declarație nu alocă memorie ci doar anunță folosirea unei variabile definite în altă parte. Această diferență între definiție şi declarație apare la variabile externe funcțiilor, definite într-un fisier sursă şi folosite în alt fisier sursă. În cazul variabilelor definite în funcții nu există necesitatea unor declaratii separate şi cele două cuvinte se pot folosi alternativ.

O declarație (definiție) de variabile începe cu un nume de tip urmat (după unul sau mai multe spații) de lista variabilelor care au acel tip şi se termină cu ’;’.

Exemple de declarații de variabile:

```
char a,b,c;
float f=1.2; double x,y;
```

Tipul de date folosit în declarații poate fi un tip predefinit (un cuvânt cheie al limbajului) sau un tip definit de utilizator, prin declarația *typedef* sau *struct*.

În C declarațiile trebuie să preceadă prima instrucțiune executabilă dintr-un bloc, dar în C++ declarațiile de variabile sunt tratate la fel cu instrucțiunile şi deci pot să apară oriunde într-un bloc. În C++ este uzual ca o variabilă sa fie declarată acolo unde este necesară prima dată. Exemplu:

```
for (int i=0;i<n;i++) s=s+a[i];
```

Clase de memorare în C

Clasa de memorare arată când, cum şi unde se alocă memorie pentru o variabilă. Orice variabilă are o clasă de memorare care rezultă fie din declarația ei, fie implicit din locul unde este definită variabila.

Există trei moduri de alocare a memoriei şi două clase de memorare:

- *Static*: memoria este alocată la compilare în segmentul de date din cadrul programului şi nu se mai poate modifica în cursul execuției. Variabilele externe, definite în afara funcțiilor, sunt implicit statice, dar pot fi declarate explicit *static* şi variabile locale, definite în cadrul funcțiilor.
- *Automat*: memoria este alocată automat, la activarea unei funcții, în zona stivă alocată unui program şi este eliberată automat la terminarea funcției. Variabilele locale unui bloc (unei funcții) şi argumentele formale sunt implicit din clasa *auto*. Memoria se alocă în stiva ataşată programului.
- *Dinamic*: memoria se alocă la execuție în zona *heap* ataşată programului, dar numai la cererea explicită a programatorului, prin apelarea unor funcții de bibliotecă (*malloc, calloc, realloc*).

Variabilele statice pot fi inițializate numai cu valori constante (pentru că inițializarea are loc la compilare), dar variabilele *auto* pot fi inițializate cu

rezultatul unor expresii (pentru că inițializarea are loc la execuție). Toate variabilele externe statice sunt automat inițializate cu valori zero.

O variabilă statică declarată într-o funcție își păstrează valoarea între apeluri succesive ale funcției, spre deosebire de variabilele *auto* care sunt realocate pe stivă la fiecare apel al funcției și pornesc de fiecare dată cu valoarea primită la inițializarea lor (sau cu o valoare imprevizibilă, dacă nu sunt inițializate). Variabilele locale statice se folosesc foarte rar în practica programării.

O a treia clasă de memorare este clasa *register* pentru variabile cărora li se alocă registre ale procesorului și nu locații de memorie, pentru a obține un timp de acces mai bun. Această clasă nu se va folosi deoarece se lasă compilatorului decizia de alocare a registrelor mașinii pentru anumite variabile *auto* din funcții.

Operatori, operanzi, expresii

Operatorii sunt simboluri utilizate pentru precizarea operațiilor care trebuie executate asupra operanzilor. Operanzii pot fi: constante, variabile, nume de funcții, expresii.

Expresiile sunt entități construite cu ajutorul operanzilor și operatorilor, respectând sintaxa (regulile de scriere) și semantica (sensul, înțelesul) limbajului. Cea mai simplă expresie este cea formată dintr-un singur operand.

Clasificarea operatorilor se poate face după:

- numărul operanzilor prelucrați: unari, binari, ternari - cel condițional;
- poziția operatorilor față de operanzi: prefixați, infixați, postfixați
- tipul operanzilor și al prelucrării: aritmetici, relaționali, logici, pe biți.

Operatorii se împart în *clase de precedență,* fiecare clasă având o *regulă de asociativitate*, care indică ordinea aplicării operatorilor consecutivi de aceeași precedență (prioritate).

Operatorii aritmetici sunt:

++	incrementare (adună 1)	/	împărțire
--	decrementare (scade 1)	%	rest împărțire întregi
-	minus unar	+	adunare
*	înmulțire	-	scădere

Se pot folosi cu operanzi numerici întregi sau reali. Operatorul / cu operanzi întregi are rezultat întreg (partea întreagă a câtului) și operatorul % are ca rezultat restul împărțirii întregi a doi întregi. Semnul restului este același cu semnul deîmpărțitului, adică restul poate fi negativ.

Operatorii relaționali sunt:

>	mai mare	<=	mai mic sau egal
>=	mai mare sau egal	==	comparație la egalitate
<	mai mic	!=	diferit de

Operatorii de relație se folosesc de obicei între operanzi numerici și, mai rar, între variabile pointer.

În C orice valoare diferită de 0 este considerată ca având valoarea adevărat!

Toți operatorii de relație au rezultat zero (0) dacă relația nu este adevărată şi unu (1) dacă relația este adevărată.

Operatorii logici se folosesc de obicei între expresii de relație pentru a exprima condiții compuse din două sau mai multe relații. Operatorii logici au rezultat 1 sau 0 după cum rezultatul expresiei logice este adevărat sau fals. Operatorii logici în C sunt:

```
!    not
&&   şi-logic    (a && b ==1 dacă şi a==1 şi b==1)
||   sau-logic (a || b ==1 dacă sau a==1 sau b==1 sau a==b==1)
```

Operatorul *&&* se foloseşte pentru a verifica îndeplinirea simultană a două sau mai multe condiții, iar operatorul || se foloseşte pentru a verifica dacă cel puțin una dintre două (sau mai multe) condiții este adevărată.

Exemplu de condiție compusă:

```
x >= a && x <= b          // daca x mai mare ca a şi x mai mic ca b
```

Dacă primul operand dintr-o expresie logică determină rezultatul expresiei prin valoarea sa, nu se mai evaluează şi ceilalți operanzi; deci ordinea a două conditii într-o expresie compusă poate fi importantă. Un exemplu este verificarea existentei unei valori corecte (indice, pointer) înainte de a o folosi într-o conditie.

În limbajul C, **operatorul de atribuire** '=' poate avea diferite utilizări:

- var = expr // atribuire simplă
- a = b = … = x = expr // atribuire multiplă
- var op = expr // echivalent cu var = var op expr

În partea stângă a unei atribuiri se poate afla o variabilă sau o expresie de indirectare printr-un pointer; în partea dreaptă a operatorului de atribuire poate sta orice expresie. Exemple: `k=1; i=j=k=0; d = b*b-4*a*c; x1=(-b +sqrt(d))/(2*a);`

Operatorii compusi dintr-un operator binar (aritmetic, logic, pe biti) şi o atribuire permit o scriere mai compactă atunci când se modifică valoarea unei variabile cu nume lung printr-o operatie cu altă valoare. Exemplu:

```
capacity *= 2  // capacity=capacity*2
```

La atribuire, dacă tipul părții stânga diferă de tipul părții dreapta atunci se face automat conversia de tip (la tipul din stânga), chiar dacă ea necesită trunchiere sau pierdere de precizie. Exemplu: `int a; a= sqrt(3.); // a=1`

Conversiile automate pot fi o sursă de erori (la execuție) şi de aceea se preferă conversii explicite prin **operatorul de conversie explicită** (*cast*), care se aplică unei expresii şi are următoarea formă: *(tip) operand*

Exemplu:

```
float x; int a,b;     x= (float)a/b;
```

Operatorul dimensiune are forma *sizeof(tip/variabila)* şi are ca rezultat numărul de octeți pe care este reprezentat tipul sau variabila respectivă.

Operatorul condițional, cu forma exp1 ? exp2: exp3; poate fi privit ca o expresie concentrată a unei instrucțiuni de decizie: dacă exp1 e adevărată rezultatul este exp2, altfel rezultatul este exp3, unde exp1, exp2, exp3 sunt expresii. Exemplu de calcul al elementului minim dintre 2 variabile a şi b:

int minim = a<b ? a:b;

Operatorul virgulă este utilizat pentru calculul mai multor expresii. Are forma: expr1, expr2, …, exprn iar rezultatul este cel al ultimei expresii. Este folositor în instructiunea *for*: for (i=0,k=0; i<n; i++,k++) {...}

Operatorii pe biți sunt aplicabili numai unor operanzi de tip întreg (char, int, long). Putem deosebi două categorii de operatori pe biți:

- Operatori logici bit cu bit:
  & şi ^ XOR (sau exclusiv)
  | sau ~ complement față de 1
- Operatori pentru deplasare cu un număr de biți
  << deplasare la stânga >> deplasare la dreapta

Aceşti operatori se aplică fiecărui bit din reprezentarea *operanzilor întregi*, şi nu valorilor operanzilor! Exemplu:

7	0000000000000111	7 & 8	0000000000000000 → fals
8	0000000000001000	7 && 8 = 1 (adevărat)	

Operatorul ~ transformă fiecare bit din reprezentarea operandului în complementul său (biții 1 în 0 şi cei 0 în 1) şi poate fi util în crearea unor configurații binare cu mulți biți egali cu 1, pe orice lungime.

Exemplu: ~0x8000 // este 0x7FFF

Operatorul pentru produs logic bit cu bit, &, se foloseşte pentru forțarea pe zero a unor biți selectați printr-o mască şi pentru extragerea unor grupuri de biți dintr-un şir de biți. Pentru a extrage cei 4 biți din dreapta (mai puțini semnificativi) dintr-un octet memorat în variabila *c* vom scrie: c & 0x0F, unde constanta 0x0F (în baza 16) reprezintă un octet cu primii 4 biți zero şi ultimii 4 biți egali cu 1.

Operatorul pentru sumă logică bit cu bit | se foloseşte pentru a forța selectiv pe 1 anumiți biți şi pentru a reuni două configurații binare într-un singur şir de biți. Exemplu: a | 0x8000 // pune semn minus la numărul din a.

Operatorul pentru sumă modulo 2 (sau exclusiv) poate fi folosit pentru inversarea logică sau pentru anularea unei configurații binare.

Pentru operatorii de deplasare, primul operand este cel al cărui biți sunt deplasați iar al doilea indică numărul de biți cu care se face deplasarea: a<<n, a>>n. Efectul deplasării depinde de tipul primului operand: dacă este un număr cu semn (*signed*) se face deplasare aritmetică cu extindere semn, dacă este un număr fără semn (*unsigned*) atunci se face deplasare logică (nu există bit de semn).

La *deplasarea la stânga* cu o poziție, bitul cel mai semnificativ se pierde, iar în dreapta se completează cu bitul 0. La *deplasarea la dreapta* cu o poziție, bitul cel mai puțin semnificativ se pierde, iar în stânga se completează cu un bit identic cu cel de semn. Operatorii pentru deplasare stânga << sau dreapta >> se folosesc pentru modificarea unor configurații binare. Deplasarea are acelaşi efect cu înmulțirea şi respectiv împărțirea cu puteri ale lui 2 (dar într-un timp mai scurt). Exemplu: a >>10 // echivalent cu a / 1024

Expresiile sunt construite cu constante, variabile, operatori şi pot fi de mai

multe tipuri, ca şi variabilele: 3*x+7, x<y, etc. La evaluarea expresiilor se ține cont de precedență şi asociativitatea operatorilor!

Precedența operatorilor C/C++ este dată în următorul tabel de priorități:

Prioritate	Operator
1	Paranteze şi acces la structuri: ( ) [ ] -> .
2	Operatori unari: ! ~ + - ++ -- & * sizeof (tip)
3	Inmulțire, împărțire, rest: * / %
4	Adunare şi scădere: + -
5	Deplasări: << >>
6	Relaționali: <= < > >=
7	Egalitate: == !=
8	Produs logic bit cu bit: &
9	Sau exclusiv bit cu bit: ^
10	Sumă logică bit cu bit:|
11	Produs logic: &&
12	Sumă logică: ||
13	Operator condițional:? :
14	Atribuiri: = *= /= %= += -= &= ^= |= <<= >>=
15	Operator virgula: ,

Două recomandări utile sunt evitarea expresiilor complexe (prin folosirea de variabile pentru rezultatul unor subexpresii) şi utilizarea de paranteze pentru specificarea ordinii de calcul (chiar şi atunci când ele nu sunt necesare).

Operatorii de aceeaşi prioritate se evaluează în general de la stânga la dreapta, cu excepția unor operatori care acționează de la dreapta la stânga (atribuire, operatorii unari şi cel condițional).

Exemple:

```
int a,v=0;
a=++v;      // v=1 şi a=1
a=v++;      //a=1 şi v=2
a=v--;      //a=2 şi v=1
a=--v;       //a=0 şi v=0;
```

```
float b=5;
b>5 && b<10
int a, b,c,d;
a=b=c=d=1;
```

Dacă operanzii expresiei diferă ca tip atunci tipul "inferior" este automat promovat la tipul "superior" înainte de efectuarea operației. Un tip T1 este superior unui tip T2 dacă toate valorile de tipul T2 pot fi reprezentate în tipul T1 fără trunchiere sau pierdere de precizie. Ierarhia tipurilor aritmetice din C este următoarea:

char < short < int < long < long long < float < double < long double

Dacă o expresie are doar operanzi întregi, ei se convertesc la *int*. Dacă o expresie are doar operanzi reali şi întregi, ei se convertesc la *double*.

La evaluarea expresiilor în forma *variabila = expresie*, se evaluează prima dată expresia, fără a ține cont de tipul variabilei; dacă tipul rezultatului obținut este diferit de cel al variabilei, se realizează conversia implicită la tipul variabilei. Pentru conversii explicite de tip se foloseşte operatorul *cast*.