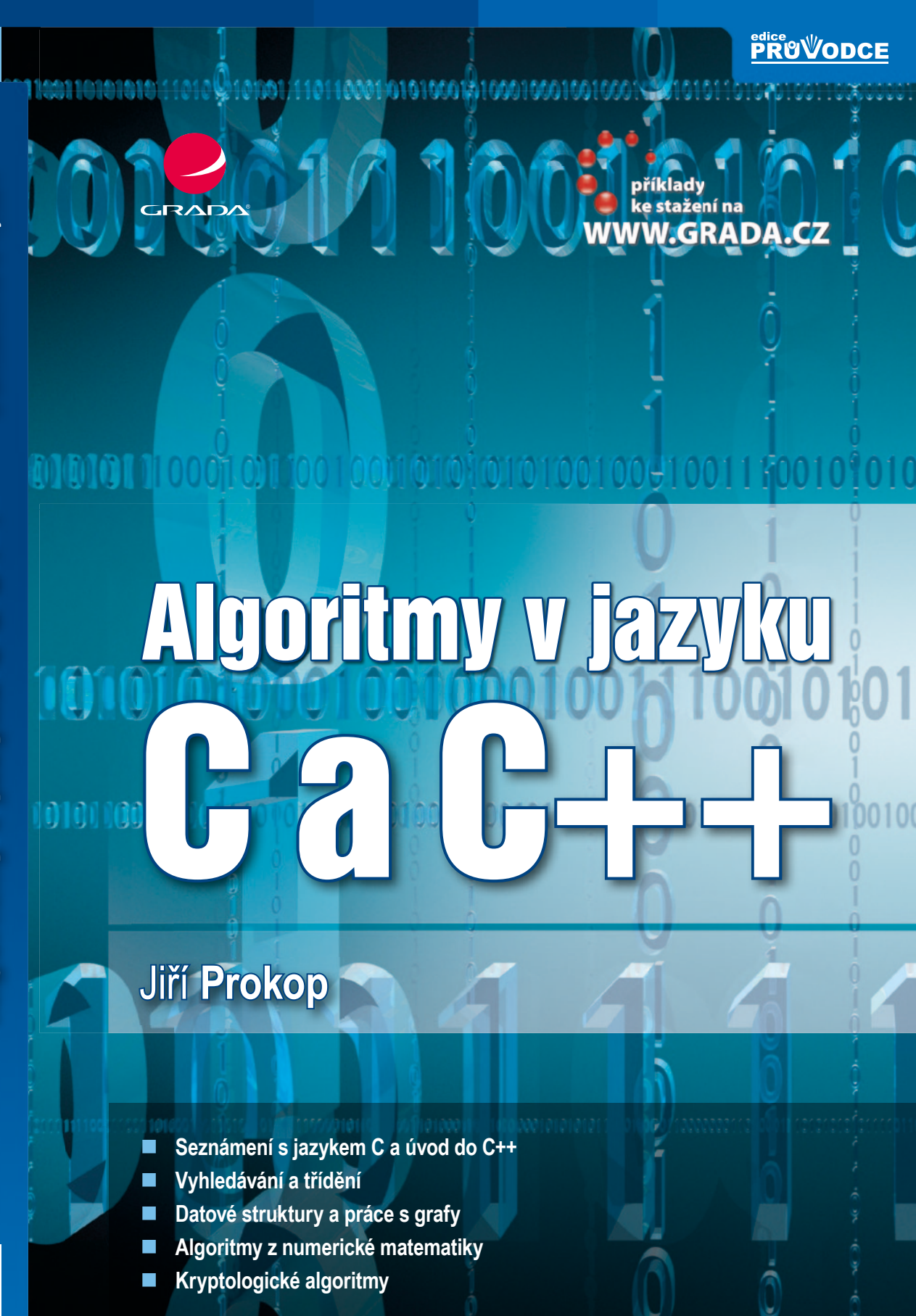
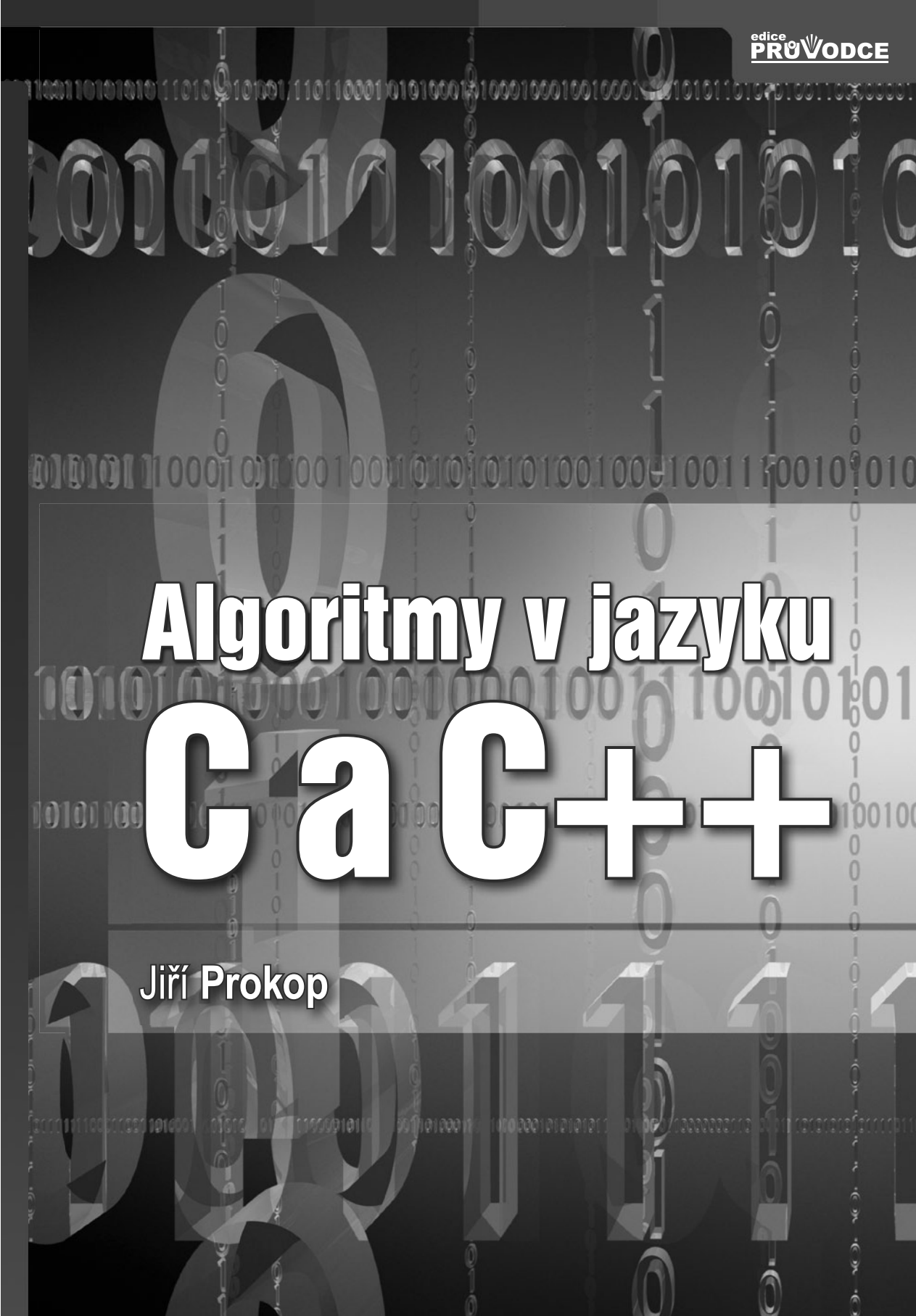

edice
PRŮVODCE
GRADA
příklady
ke stažení na
WWW.GRADA.CZ
Algoritmy v jazyku
C a C++
Jiří Prokop
Seznámení s jazykem C a úvod do C++
Vyhledávání a třídění
Datové struktury a práce s grafy
Algoritmy z numerické matematiky
Kryptologické algoritmy


GRADA


edice
PRŮVODCE
Algoritmy v jazyku
C a C++
Jiří Prokop

Algoritmy v jazyku C a C++
2., rozšířené a aktualizované vydání

Jiří Prokop

Vydala Grada Publishing, a.s.
U Průhonu 22, Praha 7
jako svou 4705. publikaci

Odpovědný redaktor Pavel Němeček
Sazba Tomáš Brejcha
Počet stran 176
První vydání, Praha 2012

Vytiskly Tiskárny Havlíčkův Brod, a. s.
Husova ulice 1881, Havlíčkův Brod

ISBN 978-80-247-3929-8 (tištěná verze)
ISBN 978-80-247-7715-3 (elektronická verze ve formátu PDF)
ISBN 978-80-247-7716-0 (elektronická verze ve formátu EPUB)

Obsah

Úvod

V roce 2002, kdy jsem začal na Gymnáziu Christiana Dopplera vést seminář „Programování v jazyku C", neexistovala na našem knižním trhu učebnice, která by se věnovala algoritmům a používala jazyk C. Algoritmy byly po řadu let prezentovány téměř výlučně v jazyku Pascal, např. [Wir89] a [Top95]. Musel jsem tedy během šesti let algoritmy pro účely výuky naprogramovat, a tak vznikl základ této knihy.

Kniha nechce být učebnicí jazyka C, i když může být k užitku všem, kteří jazyk právě studují. Dobrých učebnic jazyka je dostatek, doporučit lze např. [Her04] nebo [Ka01], pro C++ [Vi02], [Vi97]. Jestliže jsem přesto zařadil do knihy alespoň stručný přehled jazyka C a také úvod do C++, je to proto, aby čtenář měl při studiu knihy vše potřebné pro porozumění zdrojovým textům algoritmů a nemusel hledat informace jinde.

Kdo je s jazykem C seznámen do té míry, že zná nejdůležitější operátory, výrazy a přiřazení, příkazy pro řízení programu, příkazy vstupu a výstupu, funkce a vedle jednoduchých datových typů ještě pole, stačí mu to už ke studiu jednoduchých algoritmů. Takový přehled jazyka obsahuje právě první kapitola, a pak lze studovat druhou, věnovanou rekurzi, a třetí, která se zabývá třídicími algoritmy. Teprve pro studium datových struktur je nutno v kapitole 4 rozšířit zatím popsanou podmnožinu jazyka o struktury a dynamické přidělování paměti. Tyto znalosti jsou pak potřebné i pro pochopení algoritmů na grafech a pro vyhledávání pomocí binárních stromů. Stromy se využívají také k reprezentaci aritmetických výrazů a pro počítačové řešení hlavolamů a her. Popsaná podmnožina jazyka je v těchto kapitolách dále rozšiřována podle potřeby. Algoritmy z kapitol 1 až 8 jsou napsány v jazyku C, teprve kapitola 9 je úvodním popisem C++, a algoritmy v dalších kapitolách jsou v C++.

Z tohoto stručného průvodce obsahem knihy vyplývá samozřejmé doporučení studovat jednotlivé kapitoly postupně, bez přeskakování, protože v každé kapitole se už počítá s tím, co si čtenář osvojil z kapitol předchozích. Výjimkou jsou odstavce přidané ve druhém vydání této knihy, ty je možno při prvním čtení přeskočit. Jedná se o odstavec 1.3 Permutace, odstavec 6.5 AVL stromy, kapitolu 9 Kryptologické algoritmy a odstavec 11.4 Numerické integrování. Dalším doporučením je studium aktivní, usnadňuji ho tím, že všechny algoritmy rozdělené podle kapitol knihy lze najít na webových stránkách *www. grada.cz*. Zdrojové texty tedy nemusí nikdo pracně vkládat, čtenář může provádět v programech úpravy, mnohde k tomu zdrojový text přímo vybízí tím, že části zdrojového textu jsou „ukryty" v komentářích. Často lze algoritmus snáze pochopit, zobrazíme-li si některé mezivýsledky. Aktivní způsob studia je kromě toho určitě mnohem zajímavější. Algoritmy jsou ověřeny s použitím kompilátoru Dev C++, některé i kompilátoru Microsoft Visual C++. Čtenář, který by měl ke knize jakékoli připomínky, může je sdělit na emailovou adresu *Jiri.Prokop40@seznam.cz*. Na závěr přeji svým čtenářům mnoho úspěchů v jejich studiu.

1. Jazyk C

1.1 Stručný přehled jazyka C

Jazyk C rozlišuje velká a malá písmena. „Prog“, „prog“ a „PROG“ jsou tedy tři různé identifikátory. Identifikátory sestávají z písmen, číslic a podtržítka, číslice nesmí být na prvním místě. Pro oddělování klíčových slov, identifikátorů a konstant slouží oddělovače (tzv. „bílé znaky“). Všude tam, kde mohou být oddělovače, může být komentář.

```
/* toto je komentar */
```

Struktura programu: direktivy preprocesoru, deklarace, definice, funkce. V každém programu je právě jedna funkce hlavní (main), která se začne po spuštění programu vykonávat.

1.1.1 Deklarace

Deklarace jsou povinné. Deklaraci jednoduché proměnné tvoří specifikátor typu a jméno (identifikátor proměnné)

```
int a;        /* deklarace celočíselné proměnné a */
int b=1;      /* definice proměnné b */
```

Podle umístění dělíme deklarace na globální (na začátku programu) a lokální (v těle funkce). Lokální proměnné nejsou implicitně inicializovány a obsahují náhodné hodnoty.
Specifikátory typu pro celá čísla: `int`, `char`, `short int` (nebo jen `short`), `long int` (nebo jen `long`).
Každý z nich může být `signed` (se znaménkem) nebo `unsigned` (bez znaménka), implicitně je `signed`.
Specifikátory typu pro racionální proměnné: `float` (32 bytů), `double` (64), `long double` (80).
U konstant je typ dán způsobem zápisu. Pomocí klíčového slova `const` můžeme deklarovat konstantní proměnnou, jejíž obsah nelze později měnit:

```
const float pi=3.14159;
```

1.1.2 Výrazy a přiřazení

Výrazy jsou v jazyce C tvořeny posloupností operandů a operátorů. Operátory dělíme podle arity (počet operandů) na unární, binární a ternární, podle funkce na aritmetické: `+`, `-`, `*`, `/`, `%` pro zbytek po dělení (operátor / má význam reálného nebo celočíselného dělení podle typů operandů), relační: >, <, >=, <=, == (rovnost), `!=` (nerovnost), logické: `||` (log. součet), `&&` (log. součin), `!` (negace).
Jazyk C nezná logický typ, nenulová hodnota představuje `true`, nulová `false`.
Podmíněný operátor `?` (jediný ternární operátor)

```
x=(a<b) ? a:b;
```

má stejný význam jako

```
if  (a<b)      x=a; else x=b;
```

Obecně:

```
v1 ? v2 : v3
```

`v1` je výraz, jehož hodnota je po vyhodnocení považována za logickou. Je-li true, vyhodnotí se výraz `v2` a vrátí se jeho hodnota, je-li false, pak se vyhodnotí `v3` a vrátí se jeho hodnota. `v2` a `v3` jsou jednoduché výrazy.
Operátory přiřazení:

```
a=a+b;
a+=b; /* má význam a=a+b; */
```

na místě + může být –, *, /, %, & a další, o nichž zatím nebyla řeč.
Operátory inkrementace a dekrementace

```
a++;       /* postfixová verze */
--a;       /* prefixová verze */
```

Příklad:

```
a=10;
x=++a;     /* x bude mít hodnotu 11, a taky */
y=a--;     /* y=11, a=10 */
```

Unární operátory: adresní operátor &, operátor dereference *, unární +, unární –, logická negace ! a prefixová inkrementace ++ a dekrementace --. K postfixovým operátorům patří operátor přístupu k prvkům pole [ ], operátor volání funkce ( ), postfixová inkrementace ++ a dekrementace -- a operátory přístupu ke členům struktury, jimž se budu věnovat později.
Operátor přetypování ukáži na příkladu (`i1` a `i2` jsou celočíselné proměnné, ale chci reálné dělení):

```
f=(float) i1/i2;
```

Operátor `sizeof` pro zjištění velikosti: argumentem operátoru může být jak název typu, tak identifikátor proměnné.

1.1.3 Priorita a asociativita operátorů

Priorita	Operátory		Vyhodnocuje se
1	( ) [ ] -> postfix. ++ --		zleva doprava
2	! - pref. ++ -- + - (typ) * & sizeof		zprava doleva
3	* / %	(multiplikativní oper.)	zleva doprava
4	+ -	(aditivní operátory)	zleva doprava
5	<< >>	(operátory posunů)	zleva doprava
6	< <= > >=	(relační operátory)	zleva doprava
7	== !=	(rovnost, nerovnost)	zleva doprava
8	&	(operátor bitového součinu)	zleva doprava
9	^	(exklusivní nebo)	zleva doprava
10	|	(operátor bitového součtu)	zleva doprava
11	&&	(operátor logického součinu)	zleva doprava
12	||	(operátor logického součtu)	zleva doprava
13	?:	(ternární podmínkový operátor)	zprava doleva
14	= += -= *= /=	%= >>= &= |= ^=	zprava doleva
15	,	(operátor čárky)	zleva doprava

1.1.4 Příkazy a bloky

Napíšeme-li za výraz středník, stává se z něj příkaz, jako je tomu v následujících příkladech:

```
float x,y,z;
x=0;
a++;
x=y=z;
y=z=(f(x)+3); /* k hodnotě vrácené funkcí f je přičtena
              hodnota 3. */
              /* Součet je přiřazen jak proměnné z, tak y. */
```

Příkazy v jazyce C můžeme sdružovat do tzv. bloků nebo složených příkazů. Blok může obsahovat deklarace proměnných na svém počátku a dále pak jednotlivé příkazy. Začátek a konec bloku je vymezen složenými závorkami.
Složené příkazy používáme tam, kde smí být použit pouze jeden příkaz, ale potřebujeme jich více. Za uzavírací složenou závorku se nepíše středník.

Příkaz if

má dvě podoby:

```
if (výraz) příkaz
```

nebo

```
if výraz příkaz1 else příkaz2;
```

Složitější rozhodovací postup můžeme realizovat konstrukcí `if else if`.
Každé `else` se váže vždy k nejbližšímu předchozímu `if`.

Příkaz switch a break

```
switch(výraz)
{
   case konst_výraz1:
      /* příkazy, které se provedou, když výraz=výraz1 */
               break;
   case konst_výraz2:
      /* příkazy, které se provedou, když výraz=výraz2 */
      ….
      break;
   default: /* příkazy, které se provedou, není-li výraz roven
žádnému z předchozích konstantních výrazů */
}
```

Příkaz `break` říká, že tok programu nemá pokračovat následujícím řádkem, nýbrž prvním příkazem za uzavírající složenou závorkou příkazu `case`.
V těle příkazu `switch` budou provedeny všechny vnořené příkazy počínaje tím, na který bylo předáno řízení, až do konce bloku (pokud některý z příkazů nezpůsobí něco jiného – např. `break`). Tím se `switch` značně liší od pascalského `case`.

Příkaz while

```
while (výraz) příkaz;
```

Výraz za `while` představuje podmínku pro opakování příkazu. Není-li podmínka splněna už na začátku, nemusí se příkaz provést ani jednou. Je-li splněna, příkaz se provede a po jeho provedení se znovu testuje podmínka pro opakování cyklu.

Příkaz do-while

Zajistí aspoň jedno provedení těla cyklu, protože podmínka opakování se testuje na konci cyklu.

```
do příkaz while (výraz);
```

Příkaz for

Nejčastější podoba příkazu je `for (i=0;i<n;i++)`, kde i je proměnná cyklu, inicializační výraz jí přiřadí počáteční hodnotu 0, opakování cyklu bude probíhat s hodnotou proměnné zvýšenou o 1 tak dlouho, dokud bude i < n. Obecný tvar příkazu for vypadá následovně:

```
for(inicializační_výraz;podmíněný výraz;opakovaný výraz) příkaz
je ekvivalentní zápisu:
inicializační výraz;
while (podmíněný výraz)
{
      příkaz
      opakovaný výraz
}
```

Inicializační výraz může být vypuštěn, zůstane po něm však středník. Stejně může být vynechán i podmíněný výraz a opakovaný výraz. Příkaz `continue` je možno použít ve spolupráci se všemi uvedenými typy cyklů. Ukončí právě prováděný průchod cyklem a pokračuje novým průchodem. Podobně i příkaz `break` může být použit ve všech typech cyklů k jejich ukončení.

Příkaz goto a návěští

Příkaz `goto` přenese běh programu na místo označené návěštím (identifikátor ukončený dvojtečkou). Jsou situace, kdy může být výhodný, např. chceme-li vyskočit z vnitřního cyklu z více vnořených cyklů.

Prázdný příkaz

```
;
```

Použití všude tam, kde je prázdné tělo.

1.1.5 Preprocesor

Preprocesor zpracuje zdrojový text programu před překladačem, vypustí komentáře, provede záměnu textů, např. identifikátorů konstant za odpovídající číselné hodnoty a vloží texty ze specifikovaných souborů. Příkazy pro preprocesor začínají znakem # a nejsou ukončeny středníkem. Nejdůležitějšími příkazy jsou `#define` a `#include`.

```
#define ID hodnota
```

Říká, že preprocesor nahradí všude v textu identifikátor ID specifikovanou hodnotou, např.

```
#define PI 3.14159
#include <stdio.h>
```

znamená příkaz vložit do zdrojového textu funkce vstupu a výstupu ze systémového adresáře.

```
#include "filename"
```

znamená, že preprocesor vloží text ze specifikovaného souboru v adresáři uživatele.
Některé standardní knihovny:

`stdio.h`	funkce pro vstup a výstup
`stdlib.h`	obecně užitečné funkce
`string.h`	práce s řetězci
`math.h`	matematické funkce v přesnosti double
`time.h`	práce s datem a časem

1.1.6 Funkce

Každá funkce musí mít definici a:

- má určeno jméno, pomocí kterého se volá;
- může mít parametry, v nichž předáme data, na kterých se budou vykonávat operace;
- může mít návratovou hodnotu poskytující výsledek;
- má tělo složené z příkazů, které po svém vyvolání vykoná. Příkazy jsou uzavřeny ve složených závorkách `{}`.

Příkaz `return vyraz;` vypočte hodnotu `vyraz`, přiřadí ji jako návratovou hodnotu funkce a funkci ukončí.

Příklad:

```
int max(int a, int b)        /* hlavička */
{
      if (a>b)       return a;
      return b;
}
```

Nevrací-li funkce žádnou hodnotu, píšeme v místě typu návratové hodnoty `void`. Nepředáváme-li data, uvádíme jen kulaté závorky nebo `void`. Neznáme-li definici funkce, a přesto ji chceme použít, musíme mít deklaraci funkce (prototyp), která určuje jméno funkce, paměťovou třídu a typy jejích parametrů. Na rozdíl od definice funkce již neobsahuje tělo a je vždy ukončena středníkem.

```
int max(int a, int b);
```

nebo jen

```
int max(int,int);
```

Pokud neuvedeme paměťovou třídu, je automaticky přiřazena třída `extern`. Je-li funkce definována v paměťové třídě `extern` (explicitně nebo implicitně), můžeme definici funkce umístit do jiného zdrojového souboru. Funkce je společná pro všechny moduly, ze kterých se výsledný program skládá a může být v libovolném modulu volána. Je-li deklarována ve třídě static, musí její definice následovat ve stejné překladové jednotce a je dostupná pouze v jednotce, ve které je deklarována a definována.
Volání funkcí:

```
výraz(seznam skutečných parametrů);
```

Nemá-li funkce žádné parametry, musíme napsat (). Parametry se vždy předávají hodnotou, jsou tedy následně překopírovány do formálních parametrů funkce. Rekurzivní funkce jsou v C dovoleny.

1.1.7 Vstup a výstup

Standardní vstup a výstup: `stdin`, `stdout`
Standardní vstup a výstup znaků

```
int getchar(void);        /* načte 1 znak */
int putchar(int znak); /* výstup 1 znaku */
```

Pro načtení a výstup celého řádku znaků

```
char *gets(char *radek);
int puts(const char *radek);
```

Funkce `gets` načte znaky ze standardního vstupu, dokud není přechod na nový řádek. Ten už není do pole zapsán. Návratovou hodnotou je ukazatel předaný funkci jako parametr. Když došlo k nějaké chybě, má hodnotu `NULL`. Na řádku nesmíme zadat více znaků než je velikost pole. Funkce `puts` vypíše 1 řádek textu, za který automaticky přidá přechod na nový řádek. Řetězec samotný nemusí tento znak obsahovat. V případě, že výstup dopadl dobře, vrátí funkce nezápornou hodnotu, jinak `EOF`.

Formátovaný vstup a výstup

Funkce `printf` a `scanf` s následujícími deklaracemi:

```
int printf(const char *format,…);
int scanf(const char *format,…);
```

Obě funkce mají proměnný počet parametrů, který je určen prvním parametrem – formátovacím řetězcem. Formátovací řetězec funkce `printf` může obsahovat dva typy informací. Jednak jde o běžné znaky, které budou vytištěny, dále pak speciální formátovací sekvence znaků začínající `%` (má-li být `%` jako obyčejný znak, musím jej zdvojit). K tisknutelným znakům patří také `escape sekvence`, např. `\n`.
`scanf` se liší tím, že formátovací řetězec smí obsahovat jen formátovací sekvence, a tím, že druhým a dalším parametrem je vždy ukazatel na proměnnou (adresa proměnné).

Formátovací sekvence (printf)

```
%[příznak] [šířka] [přesnost] [F] [N] [h] [l] [L] typ
```

typ

d,i	znaménkové decimální číslo typu `int`
o	neznam. oktalové číslo typu `int`
u	neznam. decimální číslo typu `int`
x,X	neznam.hexadecimální číslo typu `int`, pro x tištěno a, b, c, d, e, f, pro X pak A, B, C…
f	znam.racionální číslo formátu `[-]dddd.dddd`
e,E	znam. rac. č. v exp.formátu `[-d]d.ddde[+|-]ddd`
g,G	znam. rac. č. ve formátu bez exponentu nebo s exponentem (v závislosti na velikosti čísla)
c	jednoduchý znak
s	ukazatel na pole znaků ukončené nulovým znakem
p	tiskne argument jako ukazatel
n	ukazatel na číslo typu `int`, do kterého se uloží počet vytištěných znaků

příznak

-	výstup zarovnán zleva a doplněn zprava mezerami
+	u čísel vždy znaménko
mezera	kladné číslo mezera, záporné minus
#	závisí na typu

šířka

n	alespoň n znaků se vytiskne doplněno mezerami
0n	je vytištěno alespoň n znaků doplněných zleva nulami
*	šířka dána následujícím parametrem

přesnost

(nic)	je různá podle části typ
.0	`stand`
.n	n des. míst
*	přesnost dána následujícím parametrem
h	argument funkce chápán jako `short int` – pouze pro d, i, o, u, x, X
l	long int
L	long double

Formátovací sekvence (scanf)

`%[*][šířka][F|A][h|l|L]typ`

typ

d	celé číslo
u	celé číslo bez znaménka
o	oktalové
x	hexadecimální
i	celé číslo (s předponou o – oktalové, 0x hexadecimální
a	počet přečtených znaků do aktuální chvíle
e, f, g	racionální čísla typu `float`, lze modifikovat pomocí l, L
s	řetězec znaků na vstupu oddělený mezerou od ostatních znaků
c	jeden znak
*	přeskočení dané položky vstupu
šířka	max. počet znaků vstupu pro danou proměnnou

`sprintf` a `sscanf` realizují formátovaný vstup a výstup z paměti. Potřebují textový řetězec, který se bude chovat jako standardní vstup / výstup

```
int sprintf(char *buffer,const char *format,…);
int sscanf(char *buffer,const char *format,…);
```

1.1.8 Ukazatele

Ukazatel je proměnná, jejíž hodnota je adresa jiné proměnné nebo funkce. Deklarace ukazatele se skládá z uvedení typu, na který ukazujeme, a jména ukazatele, doplněného zleva hvězdičkou.

```
int *pCeleCis;           /* může ukazovat na libovolné místo, kde
                            je uložena proměnná  typu int */
float *pReal1, *pReal2;  /* ukazatelé na libovolné proměnné typu
                            float */
```

Ukazatel po svém založení neukazuje na žádnou platnou proměnnou a označujeme jej neinicializovaný ukazatel. S hodnotou neinicializovaného ukazatele nesmíme nikdy pracovat. Inicializaci ukazatele můžeme provést např. pomocí operátoru `&`, který slouží k získání adresy objektu.

```
int Cislo=7;
int *pCislo;
pCislo=&Cislo;
```

Jakmile ukazatel odkazuje na smysluplné místo v paměti, můžeme s ním pracovat. K tomu potřebujeme ještě operátor *, kterému říkáme operátor dereference.

```
int x, y=8;
int *pInt;
pInt=&y;
x=*pInt; /* v x je 8 */
y=*pInt+20; /* do y se uloží součet obsahu proměnné, na kterou
ukazuje pInt, a konstanty 20 */
```

1.1.9 Adresní aritmetika

Význam aritmetických operací s ukazateli spočívá ve zvýšení přehlednosti a zrychlení chodu programu. Aritmetika ukazatelů je omezena na operace sčítání, odčítání, porovnání a unární operace inkrementace a dekrementace. Jestliže p je ukazatel, p++ inkrementuje p tak, že zvýší jeho hodnotu nikoli o jedničku, nýbrž o počet bytů představující velikost typu, na který ukazatel p ukazuje.

```
y=*(pInt+50);      /* tady zvětšuji hodnotu ukazatele
                   o 50*sizeof(int) */
```

1.1.10 Ukazatele a funkce

Má-li funkce vrátit více než jednu hodnotu, použijeme ukazatele:

```
void vymen(int *px, int *py)
{
  int pom; pom=*px; *px=*py; *py=pom;
}
int a=7,b=4;
vymen(&a, &b);          /* tím vlastně dosáhnu předání odkazem */
```

Ukazatel na funkci a funkce jako parametry funkcí

Definice `double (*pf)();` definuje `pf` jako ukazatel na funkci vracející hodnotu typu double. Dvojice prázdných závorek je nezbytná, jinak by `pf` byl ukazatel na `double`. Závorky kolem jména proměnné jsou také nutné, protože `double *pf()` znamená deklaraci funkce `pf`, která vrací ukazatel na `double`. Přiřadíme-li ukazateli `pf` jméno funkce, můžeme tuto funkci vyvolat příkazem `pf(); i (*pf)();`. Jméno funkce je tedy adresou funkce podobně jako jméno pole je adresou pole. Ukazatel, jemuž jsme přiřadili jméno funkce, může být také předán jako parametr jiné funkci. Příklad užitečného využití této možnosti uvidíme v odstavci 3.9.

1.1.11 Pole

Pole je datová struktura složená z prvků stejného datového typu. Deklarace pole vypadá obecně takto:

```
typ id_pole [pocet];
```

V hranatých závorkách musí být konstantní výraz, který udává počet prvků pole. Pole v jazyku C začíná vždy prvkem s indexem nula a nejvyšší hodnota indexu je počet-1. Jazyk C zásadně nekontroluje meze polí! K prvkům pole přistupujeme pomocí indexu, např. `id_pole[0]` pro první prvek pole. Indexem může být výraz. Pole můžeme při jeho deklaraci inicializovat konstantami, uvedenými mezi složenými závorkami a oddělovanými čárkou:

```
int pole[5]={6,7,8,9,10}
```

Počet inicializátorů by měl být menší nebo roven počtu prvků pole. Má-li pole být parametrem funkce, bude formální parametr tvořen typem a identifikátorem pole následovaným prázdnými hranatými závorkami, např. `double pole[]`. Jako skutečný parametr stačí jméno pole, tedy adresa začátku pole. Pole se tedy předává na rozdíl od jednoduchých proměnných odkazem. Pole nemůže být typem návratové hodnoty funkce (i když struktura obsahující pole jím být může). S polem jako celkem není možné provádět žádné operace s výjimkou určení velikosti pole operátorem `sizeof` a určení adresy pole operátorem `&`.

```
int b[8]; int i=sizeof(b);     /*  8*sizeof(int)  */
```

1.1.12 Ukazatele a pole

```
int x[12];                     /* deklarace pole o 12 prvcích,
indexy jsou 0 až 11 */
&x[i] = adresa pole x + i * sizeof(typ)
int *pData;
pData=&data[0];                /* není totéž jako pData=&data */
for(i=0;i<12;i++)
    (pData+i)=0;               /* nulování pole - přičítá se
i-násobek délky typu -adresní aritmetika */
```

Inicializaci ukazatele `pData` můžeme zapsat i takto: `pData=data;`
což je stejné jako `pData=&data[0];`

Máme-li deklaraci

```
int i, *pi, a[N];  /* a[0] je totéž jako &a[0], a,  anebo a+0  */
a+i je totéž jako &a[i], *(a+i) je totéž jako a[i]
```

Je-li `N=100` a přiřadíme-li `pi=a;` mají výrazy uvedené níže stejný význam:

```
a[i], *(a+i), pi[i], *(pi+i)
```

1.1.13 Řetězce znaků

Řetězec je jednorozměrné pole znaků ukončené specielním znakem `'\0'`, který má funkci zarážky. Řetězcové konstanty píšeme mezi dvojici uvozovek, uvozovky v řetězcové konstantě musíme uvést zpětným lomítkem.
`"abc"` je konstanta typu řetězec délky 3+1 znak, `"a"` je rovněž řetězcovou konstantou délky 1+1, `'a'` je znaková konstanta délky 1.
Překopírování textového řetězce:

```
void strcpy(char cil[ ],char zdroj[ ])      /* pro funkci strcpy
je potřebné #include <string.h> */
{
 int i;
for (i=0; zdroj[i]!='\0'; i++)
      cil[i]=zdroj[i];
cil[i]='\0';
}
```

nebo:

```
void strcpy(char *cil, char *zdroj)
{
while(*cil++ = *zdroj++);
}
```

-

Nejdříve dojde k přiřazení odpovídajících buněk polí, oba ukazatele jsou pak posunuty a výsledek přiřazení je také chápán jako logická hodnota. Nastal-li konec řetězce, cyklus dále nepokračuje.

1.1.14 Vícerozměrná pole

Jazyk C zná pouze jednorozměrné pole. Prvky pole mohou ovšem být libovolného typu, tedy např. opět pole, a to umožňuje pracovat s vícerozměrnými poli. Příklad deklarace dvojrozměrného pole:

```
int pole2d [10][20];
```

Uložení v paměti je po řádcích. Ve funkcích, kde je pole parametrem, nemusíme předat nejvyšší rozměr, všechny ostatní ano. Práci s vícerozměrným polem demonstrujme na příkladu: máme překlopit čtvercovou matici podle hlavní diagonály, tedy vyměnit vzájemně prvky a_{ij} a a_{ji} pro všechna `i` různá od `j`.

```
void Preklop(float m[][3])
/* preklopeni ctvercove matice 3 x 3 podle hlavni diagonaly */
{
   int i,j;
   float pom;
   for(i=0;i<2;i++)
      for(j=1;j<2;j++)
      {
         pom=m[i][j]; m[i][j]=m[j][i]; m[j][i]=pom;
      }
}
```

V hlavním programu jsou deklarace

```
float a[3][3];
int pocet = 3;
```

a funkce je volána příkazem

```
Preklop(a);
```

Nedostatkem je, že může být překlopena jen matice 3x3. Následující funkce je obecnější, využívá toho, že matice je uložena po řádcích, a dovoluje překlopení matice `nxn`:

```
void Preklop(float *m, int n)
/* preklopeni ctvercove matice n x n podle hlavni diagonaly */
{
      int i,j;
      float pom;
      for(i=0;i<n-1;i++)
            for(j=i+1;j<n;j++)
            {
                  pom=m[i*n+j]; m[i*n+j]=m[j*n+i]; m[j*n+i]=pom;
            }
}
```

V hlavním programu může být deklarace např.

```
float a[4][4];
int pocet=4;
a funkce se volá příkazem
Preklop(a,pocet);
```

1.2 Jednoduché algoritmy

Algoritmus je konečná posloupnost kroků, po jejichž provedení dojdeme k určitému předem vytčenému cíli. Musí splňovat následující vlastnosti: musí být konečný, tzn. skončit po konečném počtu kroků (to je sice požadavek velmi samozřejmý, ale všichni, kdo mají již zkušenost s praktickým programováním, dobře vědí, jak snadno lze udělat chybu, která způsobí uváznutí v nekonečné smyčce). Jednotlivé kroky algoritmu musí být definovány jednoznačně. Proto je nejlepším popisem algoritmu jeho zápis v programovacím jazyce, kde je význam příkazů přesně definován. Často se setkáváme s popisem v přirozeném jazyce, ale zde je nutno na jednoznačnost dávat větší pozor. Každý algoritmus má nějaké vstupy (hodnoty, z nichž vychází) a výstupy, které jsou jeho výsledkem. Požadavek konečnosti musíme z praktických důvodů zesílit a chtít, aby algoritmus skončil v „rozumně krátkém čase". Proto má velký význam efektivita algoritmu, které se budeme podrobněji věnovat v odstavci 3.4. Konečně, při zápisu algoritmu v programovacím jazyce bychom měli myslet na to, že „čtenářem" může být nejen počítač, ale i člověk, a dbát čitelnosti a srozumitelnosti zápisu. Dosáhneme toho jednak dodržováním určitých zvyklostí (např. každý příkaz na jednom řádku) charakteristických pro daný jazyk, a také vhodným využíváním komentářů.
Než se pustíme do složitějších algoritmů, procvičme znalosti jazyka C na jednoduchých příkladech:

1.2.1 Vyhledání minimálního prvku v nesetříděném poli

```
int hledej(p[ ],n]
{
   int i,min,imin;
   min=p[0]; imin=0;
   for(i=1;i<n,i++)
      if  (p[i]<min)
      {
          min=p[i]; imin=i;
      }
    return imin; /* vracím index prvku s minimální hodnotou */
}
```

1.2.2 Vyhledání zadaného prvku v nesetříděném poli

```
int najdi(p[ ],n,x)
   {
      int i;
      for(i=0;i<n;i++)
         if (x==p[i]) return i;
      return -1;                    /* hledaný prvek v poli není */
   }
```

1.2.3 Určení hodnoty Ludolfova čísla pomocí rozvoje pi=4(1-1/3+1/5-1/7+1/9+…)

Asi nás nejdříve napadne uchovávat v paměti poslední dvě aproximace, abychom porovnáním jejich rozdílu s požadovanou přesností 0,000 01 zjistili, máme-li pokračovat výpočtem další aproximace. Pak bude zdrojový text vypadat následovně:

```
#include <stdio.h>
#include <conio.h>
int main()
```