

JZD AGROKOMBINÁT
SLUŠOVICE



C - JAZYK

REFERENČNÍ PRÍRUČKA

ZÁVODY APLIKOVANÉ KYBERNETIKY



SOFTWARE
SLUŠOVICE

JZD AGROKOMBINÁT SLUŠOVICE
nositel Radu práce

ZÁVOD APLIKOVANÉ KYBERNETIKY 2
763 15 SLUŠOVICE
CZECHO-SLOVAKIA TELEX 67279

OBSAH

1. Úvod	1
2. Lexikální konvence	1
2.1 Komentáře	1
2.2 Identifikátory	1
2.3 Klíčová slova	2
2.4 Konstanty	2
2.4.1 Celočíselné konstanty	2
2.4.2 Explicitní konstanty typu long	3
2.4.3 Znakové konstanty	3
2.4.4 Konstanty v pohyblivé řadové čárce	3
2.5 Řetězce	4
2.6 Charakteristiky hardware	4
2.7 Základní datové typy	5
3. Syntaktický zápis	5
4. Co je to jméno?	5
5. Objekty a l-hodnoty	7
6. Konverze	7
6.1 Znaky a celá čísla	8
6.2 Pohyblivá čárka a dvojnásobná přesnost	8
6.3 Floating a integrální typ	8
6.4 Ukazovátka a celá čísla	9
6.5 Čísla bez znaménka	9
6.6 Aritmetické konverze	9
7. Výrazy	10
7.1 Základní výrazy	10
7.2 Unární operátory	13
7.3 Multiplikativní operátory	15
7.4 Aditivní operátory	16
7.5 Operátory posuvu	17
7.6 Relační operátory	17
7.7 Operátory porovnání	18
7.8 Bitový operátor AND	18

7.9	Bitový operátor XOR	19
7.10	Bitový operátor OR	19
7.11	Logický operátor AND	19
7.12	Logický operátor OR	20
7.13	Operátor podmínky	20
7.14	Operátory přiřazení	20
7.15	Operátor čárky	21
8.	Deklarace	22
8.1	Specifikace třídy uložení	22
8.2	Specifikace typu	23
8.3	Deklarátory	24
8.4	Význam deklarátoru	25
8.5	Deklarace struktur a unionu	27
8.6	Inicializace	31
8.7	Názvy typu	33
8.8	Typedef	34
9.	Příkazy	35
9.1	Výrazové příkazy	35
9.2	Složené příkazy nebo bloky	36
9.3	Podmínkové příkazy	36
9.4	Příkaz while	37
9.5	Příkaz do	37
9.6	Příkaz for	37
9.7	Příkaz switch	38
9.8	Příkaz break	39
9.9	Příkaz continue	39
9.10	Příkaz return	40
9.11	Příkaz goto	40
9.12	Návěští příkazu	40
9.13	Prázdný příkaz	41
10.	Externí definice	41
10.1	Definice externích funkcí	41
10.2	Externí definice dat	42

11. Pravidla rozsahu platnosti	43
11.1 Lexikální rozsah platnosti	43
11.2 Rozsah platnosti externích proměnných	44
12. Řídicí řádky překladače	45
12.1 Záměna syntaktické jednotky (znaková náhrada) .	45
12.2 Vkládání souboru	46
12.3 Podmíněná kompilace	47
12.4 Řízení řádkování	47
12.5 Připojení assembleru	47
13. Implicitní deklarace	48
14. Další informace o typech	48
14.1 Struktury a uniony	49
14.2 Funkce	49
14.3 Pole, ukazovátka a indexy	50
14.4 Explicitní konverze ukazovátek	51
15. Konstantní výrazy	52
16. Úvahy o přenositelnosti	53
17. Anachronismy	54

REFERENČNÍ PŘÍRUČKA C JAZYKA

1. Úvod

Tato příručka je určena pro rychlou orientaci v syntaxi jazyka C. Je rozdělena do 17 kapitol.

2. Lexikální konvence

Existuje šest kategorií lexikálních položek:

- identifikátory
- klíčová slova
- konstanty
- řetězce
- operátory
- jiné oddělovače

Mezery, tabulátory, znaky pro nový řádek apod., komentáře (souhrnně nazývané nevýznamné znaky) jsou při kompilaci ignorovány, pokud neslouží k oddělení lexikálních položek. Některé nevýznamné znaky jsou požadovány pro oddělení sousedních identifikátorů, klíčových slov a konstant.

2.1 Komentáře

Komentář je uveden dvojicí znaků "/*" a ukončen dvojicí znaků "*/". Komentáře nesmějí být do sebe navzájem vnořené.

2.2 Identifikátory

Identifikátor je řada písmen a čísel. První znak identifikátoru musí být vždy písmeno. Znak "_" se řadí k písmenům. Jsou rozlišována malá a velká písmena. Pouze prvních 6 znaků identifikátoru je významných, ačkoliv identifikátor může být

utvořen z více znaků. Pro externí identifikátory, které jsou používány různými překladači jazyka symbolických adres a zaváděcími programy, platí přísnější pravidla.

2.3 Klíčová slova

Následující identifikátory jsou rezervovány jako klíčová slova a nesmějí být použity v jiném významu:

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

2.4 Konstanty

Existuje několik typů konstant, které budou popsány dále. Jejich charakteristiky závislé na technických prostředcích jsou popsány v odstavci 2.6.

2.4.1 Celočíselné konstanty

Celočíselná konstanta se skládá z řady číslic. Jestliže tato řada začíná znakem \, předpokládá se, že se jedná o oktálovou reprezentaci určitého čísla. Konstanta v hexadecimálním tvaru musí začínat 0x nebo 0X a kromě číslic může též obsahovat znaky a až f nebo A až F.

Dekadické číslo 31 lze tedy v oktálové reprezentaci zapsat jako \37 nebo \037. Hexadecimální zápis tohoto čísla má tvar 0x1f nebo 0X1F.

2.4.2 Explicitní konstanty typu long

Dekadická, oktálová nebo hexadecimální celočíselná konstanta, která je bezprostředně následována znakem l nebo L je chápána jako konstanta typu long. Na některých počítačích mohou být konstanty typu celé číslo a typu long považovány za totožné.

2.4.3 Znakové konstanty

Znaková konstanta je znak uzavřený mezi jednoduché uvozovky, např. 'x'. Hodnota znakové konstanty je číselná hodnota znaku v použitém znakovém systému.

Některé nezobrazitelné znaky mohou být reprezentovány pomocnými sekvencemi:

<u>Název znaku:</u>	<u>označení:</u>	<u>vkládáný řetězec znaků:</u>
nový řádek	LF	\n
tabelátor	HT	\t
zpětný výmaz	BS	\b
návrat vozíku	CR	\r
nová stránka	FF	\f
zpětné lomítko	\	\\
jednoduché uvoz.	'	\'
řetěz bitu	ddd	\ddd

Řetězec "\ddd" se skládá ze znaku zpětné lomítko, které je následováno 1, 2 nebo 3 oktalogými číslicemi, které specifikují hodnotu požadovaného znaku. Speciálním případem použití této konvence je řetězec znaků "\0", který představuje znak NULL.

Všechny pomocné sekvence, i když jsou tvořeny dvěma nebo i více znaky, reprezentují jednoduchý znak.

2.4.4 Konstanty v pohyblivé řadové čárce

Konstanta v pohyblivé řadové čárce se skládá z celočíselné části, desetinné tečky a desetinné části, znaku e nebo E

a případně celočíselného exponentu se znaménkem. Celočíselná a desetinná část se skládají z řady číslic.

př. 65.1321654e-7

0.523E3

Buď celočíselná, nebo desetinná část může být vynechána, ale nikoliv obě dvě zároveň. Buď desetinná tečka, nebo znak e (E) a exponent může být vynechán, ale nikoliv oba současně. Každá konstanta v pohyblivé řadové čárce je chápána s dvojnásobnou přesností.

2.5 Řetězce

Řetězec je řada znaků uzavřená mezi dvojité uvozovky, např. "...". Řetězec je objekt typu pole znaků a je inicializován uvedením znaku. Překladač umístí na konec každého řetězce znak \0, aby program mohl najít konec tohoto řetězce. Uvozovku uvnitř řetězce musí bezprostředně předcházet znak \. Dále mohou být použity stejné pomocné posloupnosti, které byly popsány v odstavci 2.4.3. Znak zpětné lomítka, který je bezprostředně následován znakem nový řádek je spolu s ním ignorován.

Pozor! 'x' není totéž co "x". V prvním případě se jedná o užití znaku ve formě jeho numerické hodnoty a v druhém případě jde o řetězec, který obsahuje jeden znak x.

2.6 Charakteristiky hardware

Znak:	ASCII
char	8 bitu
int	16
short	16
long	32
float	32
double	64
rozsah	(+10)**
	(+38)

2.7 Základní datové typy

char	jeden byte schopný uchovat jeden znak ASCII množiny znaků
int	celé číslo z intervalu -32768 až +32767
float	číslo v pohyblivé řadové čárce s jednoduchou přesností
double	číslo v pohyblivé řadové čárce s dvojnásobnou přesností

Další typy celých čísel jsou short, long a unsigned.

3. Syntaktický zápis

Syntaktické kategorie a literální slova jsou v textu zřejmá. Pokud by mělo dojít k nejasnostem, jsou od textu oddělena. Znaky jsou zřejmé.

Nepovinný terminální nebo neterminální symbol je označen znaky "???".

Například:

(expression???)

označuje nepovinný výraz uzavřený v závorkách.

4. Co je to jméno?

V C jazyku má každý identifikátor přiděleny dva atributy: paměťovou třídu a typ. Paměťová třída určuje umístění a dobu existence paměti přidělené identifikátoru. Typ určuje význam hodnoty uložené paměti přidělené identifikátoru.

Existují čtyři deklarovatelné paměťové třídy:

- automatická (auto)
- externí (extern)
- statická (static)
- registrová (register)

Proměnné třídy auto jsou lokální při každém vyvolání bloku, viz odstavec 9.2 a jsou zrušeny po ukončení bloku.

Proměnné třídy static jsou lokální v bloku, ale uchovávají svoji hodnotu až do opětovného návratu do bloku.

Externí proměnné existují a uchovávají svoji hodnotu po celou dobu běhu programu. Mohou být použity pro komunikaci mezi různými funkcemi, které mohou být dokonce odděleně překládány.

Registrové proměnné jsou ukládány do stálých registrů počítače, jinak se chovají stejně jako automatické proměnné.

V jazyku C je definováno několik základních typů objektu:

znaky - char

celá čísla - int

celá čísla bez znaménka - unsigned

čísla v pohyblivé řadové čárce - float

s dvojnásobnou přesností - double

Objekty deklarované jako char jsou tak veliké, aby do nich mohl být uložen libovolný člen použitého systému znaků. Jestliže je nějaký znak uložen do znakové proměnné, jeho hodnota odpovídá celočíselnému kódu tohoto znaku. Do proměnné typu char mohou být uloženy libovolné hodnoty, ale jejich význam je závislý na technických prostředcích.

Existují tři různé proměnné typu int, které jsou deklarovány jako short int, int a long int. Proměnné typu long nebo int neposkytují pro uložení proměnných méně paměti než proměnné typu int nebo short. Implementace však může být taková, že proměnné typu short int nebo long int nebo obě dvě mohou být ekvivalentní základní proměnné typu int. Základní proměnná typu int má velikost odpovídající přirozené architektuře počítače. Ostatní typy proměnných typu int jsou definovány pro speciální účely.

Celá čísla bez znaménka, deklarovaná jako unsigned, používají aritmetiku modulo 2^{*n} , kde n je počet bitů v dané reprezentaci celého čísla.

Proměnné typu floating point jednoduché i dvojnásobné přesnosti mohou být v některých implementacích totožné.

Typy char a int všech velikostí budou souhrnně označovány jako proměnné integrálního typu. Proměnné typu float a double budou souhrnně označovány jako proměnné typu floating.

Vedle těchto základních typů existuje v podstatě nekonečný počet odvozených typů, které mohou být od základních odvozeny následujícími způsoby:

pole (array)	objektů z různých typů,
funkce	která vrací objekt daného typu
ukazovátka (pointer)	na objekt daného typu
struktura (struct)	obsahující seznam objektů různých typů
union (union)	obsahující současně několik objektů různých typů

Všechny tyto způsoby mohou být použity rekurzivně. Všechny objekty musí být před použitím deklarovány.

5. Objekty a 1-hodnoty

Pojem objekt znamená určitou část paměti, která slouží k uložení nějaké hodnoty a se kterou je možné manipulovat. 1-hodnota je výraz, který ukazuje na nějaký objekt. Nejčastějším příkladem 1-hodnotového výrazu je identifikátor. Existují operátory, které poskytují 1-hodnoty, např., jestliže R je výraz typu ukazovátka, potom *R je 1-hodnotový výraz ukazující na objekt, na který ukazuje R.

6. Konverze

Většina operátorů může v závislosti na svých operandech způsobit konverzi hodnoty operandu daného typu na jiný typ.

V této části příručky bude objasněno, jaké mohou být výsledky takových konverzí. V odstavci 6.6 jsou souhrnně vysvětlena pravidla pro aritmetickou konverzi.

6.1 Znaky a celá čísla

Znakové objekty a objekty typu short integer mohou být použity všude tam, kde je dovoleno použít objekty typu integer. Ve všech případech se však provádí konverze na typ integer. Při konverzi objektu typu short integer na objekt typu integer se vždy provádí znaménkové rozšíření, protože objekty všech typů integer jsou hodnoty se znaménkem. U objektů znakových se znaménkové rozšíření provádí v závislosti na technickém vybavení počítače. Vždy je však zaručeno, že znak ze standardního souboru znaků nemá zápornou hodnotu.

Kdykoliv je delší objekt typu integer konvertován na objekt typu short integer nebo char, je zkrácen zleva, bity navíc jsou jednoduše ignorovány.

6.2 Pohyblivá čárka a dvojnásobná přesnost

Veškerá aritmetika v pohyblivé čárce je v C jazyku prováděna v dvojnásobné přesnosti. Kdykoliv se objekt floating typu vyskytne ve výrazu, je konvertován na objekt typu dvojnásobné přesnosti tak, je doplněn nulami. Kdykoliv musí být objekt dvojnásobné přesnosti konvertován na objekt typu float, je double před zkrácením zaokrouhleno na float délku.

6.3 Floating a integrální typ

Konverze floating typu na typy celočíselné jsou závislé na použitém technickém vybavení, především zkrácení záporných čísel je na různých počítačích odlišné. Výsledek je nedefinovaný, jestliže konvertovaná hodnota není v rozsahu celých čísel.

Konverze objektu celočíselných na floating objekty jsou bez problému. Může dojít ke ztrátě přesnosti, jestliže floating objekt nemá dostatečný počet bitu.

6.4 Ukazovátka a celá čísla

Objekty typu integer nebo long integer mohou být sčítány a odčítány od objektu typu ukazovátka. V těchto případech je první operand konvertován tak, jak je popsáno v části popisující operátor sčítání.

Mohou být také odečtena dvě ukazovátka, která ukazují na objekty téhož typu. Potom je výsledek konvertován na celočíselný typ, tak jak je popsáno v části popisující operátor rozdílů.

6.5 Čísla bez znaménka

Při jakékoliv kombinaci celého čísla bez znaménka a čísla typu int je číslo int konvertováno na číslo bez znaménka a výsledek je bez znaménka. Jeho hodnota je nejmenší celé číslo bez znaménka, jehož hodnota je souhlasná s celým číslem se znaménkem.

Při konverzi objektu typu unsigned na objekt typu long, bude hodnota výsledku numericky rovná hodnotě objektu typu unsigned. Při konverzi se pouze doplní potřebný počet nul zleva do čísla typu long.

6.6 Aritmetické konverze

Velký počet operátorů způsobuje konverze a vytváří výsledný typ podobným způsobem. Následují tzv. obvyklé aritmetické konverze.

- operandy typu char a short jsou konvertovány na typ int, operandy typu float jsou konvertovány na typ double.
- jestliže jeden z operandů je typ double, jsou na tento typ konvertovány i zbývající operandy a také typ výsledku je double.
- jinak, jestliže je jeden z operandů typu long, jsou na tento typ konvertovány i zbývající operandy a také typ výsledku je long.
- jinak, jestliže je jeden z operandů typu unsigned, jsou na

tento typ konvertovány i zbývající operandy a také typ výsledku je unsigned.

- jinak musí být operandy typu int a to je také typ výsledku.

7. Výrazy

Priorita operátorů ve výrazech je stejná, jako je pořadí odstavců v této kapitole. Odstavec, který předchází jinému, popisuje operátory, které mají vyšší prioritu. Například odstavec 7.4 popisuje operátory, které mají vyšší prioritu, než operátory, které jsou popisovány v odstavcích 7.5 a dalších. Operátory uvnitř každého odstavce mají stejnou prioritu. Asociativita operátoru zleva či zprava je uvedena pro každou skupinu operátorů zvlášť.

Mimo výše uvedená pravidla není pořadí vyhodnocení výrazu definováno. Překladač sám rozhodne o takovém vyhodnocení částečných výrazů, které považuje za efektivní, bez ohledu na vedlejší efekt takového vyhodnocení. Pořadí, ve kterém dojde k vedlejším efektům, není specifikováno. Výrazy, které obsahují komutativní a asociativní operátory (*, +, &, |, ^) mohou být překladačem přeskupeny, dokonce i když obsahují závorky. Při snaze dodržet zadané pořadí vyhodnocení výrazu, by musely být použity přechodné proměnné.

Ošetření situací aritmetického přetečení a dělení nulou při vyhodnocení výrazu je závislé na technickém vybavení. Všechny existující implementace jazyka C ignorují celočíselné přetečení. Ošetření dělení nulou a všech výjimečných situací při zpracování čísel v pohyblivé čárce je na různých počítačích odlišné a je obvykle umístěno v knihovně aritmetických funkcí jazyka C.

7.1 Základní výrazy

Základní výrazy včetně ., -, indexových výrazů a funkčních volání jsou následující:

základní_výraz:

identifikátor
konstanta
řetězec
(výraz)
základní_výraz[výraz]
základní_výraz(seznam_výrazu???)
základní_l-hodnota.identifikátor
základní_výraz->identifikátor

seznam_výrazů:

výraz
seznam_výrazů, výraz

Identifikátor je základním výrazem, jestliže je deklarován vhodným způsobem, jak je popsáno dále. Typ identifikátoru je dán jeho deklarací. Jestliže typ identifikátoru je "pole z ..", hodnota identifikátoru ve výrazu je hodnotou ukazovátka na první objekt pole a typ výrazu je "pointer na...". Navíc identifikátor pole není výraz typu l-hodnoty. Podobně identifikátor deklarovaný jako "funkce vracející..." použitý mimo funkční volání je chápán jako "pointer na funkci vracející...".

Konstanta je základní výraz. Její typ může být int, long nebo double. Znaková konstanta má typ int a konstanta v pohyblivé řadové čárce má typ double.

Řetězec je základním výrazem. Jeho typ je v podstatě "pole z char", ale platí pro něj ta samá pravidla, která již byla popsána pro identifikátory. Tj. typ je modifikován na "pointer na char" a výsledkem je ukazovátka na první znak řetězce. Existuje výjimka z těchto pravidel, která se uplatňuje při inicializaci řetězce, viz. odstavec 8.6.

Ozávorkovaný výraz je základním výrazem, jehož typ je identický s typem neozávorkovaného výrazu. Přítomnost závorek nemá vliv na to, je-li výraz hodnotou nebo není.

Základní výraz následovaný výrazem v hranatých závorkách je základním výrazem. Intuitivní pojetí takového výrazu je,

že se jedná o indexový výraz. Základní výraz má typ "pointer na ...", indexový výraz má typ int a typ výsledku je "...". Výraz $V1 [V2]$ je identický (svoji definicí) s výrazem $*((V1) + (V2))$. Další informace potřebné pro pochopení výše uvedené notace jsou uvedeny v odstavcích 7.1, 7.2 a 7.4. Celkové shrnutí je dále obsaženo v odstavci 14.3.

Funkční volání následované seznamem výrazů, které jsou uzavřeny v závorkách, je základním výrazem. Seznam výrazů může být prázdný. Základní výraz je typu "funkce vracějící..." a výsledek vrácený funkčním voláním je typu "...". Identifikátor, který je bezprostředně následován levou závorkou je kontextově chápán jako funkce, která vrací hodnotu typu int a nemusí být proto deklarován.

Všechny skutečné argumenty typu float jsou konvertovány před vyvoláním funkce na typ double. Argumenty typu char a short jsou obdobně konvertovány na typ int a názvy polí jsou konvertovány na typ pointer. Žádné další konverze se automaticky neuskutečňují, překladač nekontroluje, zda-li typy skutečných argumentů odpovídají typům argumentů formálních. Jestliže je taková konverze potřebná, je nutné použít explicitní konverzi, viz. odstavce 7.2 a 8.7.

Při přípravě funkčního volání se vytvoří kopie každého skutečného argumentu, jedná se tedy o přenos argumentu jejich hodnotou. Funkce smí změnit hodnoty všech svých formálních argumentů, aniž dojde ke změně hodnot skutečných argumentů. Na druhé straně je možné přenášet ukazovátka na dané objekty, které pak mohou být funkcí měněny. Název pole je chápán jako ukazovátko. Pořadí vyhodnocení argumentů není jazykem definováno, závisí od použitého překladače. Jsou dovolená rekurzivní volání libovolné funkce.

Základní výraz bezprostředně následovaný tečkou a identifikátorem je základním výrazem. První výraz musí být l-hodnotou určující strukturu nebo union. Identifikátor musí být název položky struktury nebo unionu.

Základní výraz následovaný šipkou (je vytvořen ze znaku "<-" a ">") a identifikátorem je výrazem. První výraz musí být 1-hodnotou určující strukturu nebo union. Identifikátor musí být název položky struktury nebo unionu. Výsledkem je 1-hodnota určující danou položku struktury nebo unionu, na kterou ukazuje výraz typu ukazovátko.

Výše uvedená pravidla znamenají, že výrazy L1->MOS a (*L1).MOS jsou totožné. Struktury a uniony jsou objasněny v odstavci 8.5.

7.2 Unární operátory

Výrazy s unárními operátory jsou vyhodnocovány zprava doleva.

unární výraz:

- *výraz
- &l-hodnota
- výraz
- !výraz
- ~výraz
- ++l-hodnota
- l-hodnota
- l-hodnota++
- l-hodnota--
- (název_typu) výraz
- sizeof výraz
- sizeof (název_typu)

Unární operátor "*" znamená tzv. nepřímý přístup. Výraz musí být ukazovátko a výsledkem je 1-hodnota určující objekt, na který výraz ukazuje. Jestliže typ výrazu je "pointer na...", potom typ výsledku je "...".

Výsledkem unárního operátoru "&" je ukazovátko na objekt určený 1-hodnotou. Jestliže typ 1-hodnoty je "...", potom typ výsledku je "pointer na...".

Výsledek unárního operátoru "-" je negace jeho operandu. Jsou uskutečněny obvykle aritmetické konverze. Negace hodnoty typu unsigned je vypočtena tak, že hodnota výraz je odečtena od hodnoty 2^{*n} , kde n je počet bitů čísla typu int. Neexistuje unární operátor +.

Výsledkem operátoru logické negace "!" je hodnota 1, jestliže hodnota operandu je 0, a hodnota 0, jestliže hodnota operandu je různá od hodnoty 0. Typ výsledku je int. Může být použit ve výrazech s libovolným aritmetickým typem nebo ve výrazech s ukazovátky.

Operátor "-" vytváří jednotkový komplement svého operandu. Při jeho použití se uskuteční obvyklé aritmetické konverze. Operand musí být typu int.

Operátor "++", který předchází svůj operand, způsobí jeho inkrementaci. Tato hodnota se stane novou hodnotou objektu, který je určen l-hodnotou, ale samotná tato hodnota není l-hodnotou. Výraz $++x$ je rovnocenný výrazu $x += 1$. Další informace jsou v odstavcích 7.4 (sčítání) a 7.14 (operátory přiřazení). V těchto odstavcích jsou také informace o konverzích, které se při použití tohoto operátoru uskuteční.

Operátor "--", který předchází svůj operand, způsobí jeho dekrementaci. Ostatní vlastnosti jsou analogické vlastnostem předcházejícího operátoru.

Při použití operátoru "++", který následuje svůj operand, se stane výsledkem hodnota objektu, která je určena l-hodnotou a teprve potom dojde k inkrementaci objektu. Typ výsledku je tentýž jako je typ l-hodnoty.

Operátor "--", který následuje svůj operand, má vlastnosti analogické vlastnostem předcházejícího operátoru.

Jméno typu, které je uzavřeno mezi závorky a je následováno výrazem, způsobí, že hodnota výrazu bude konvertována na odpovídající hodnotu daného typu. Konstrukce tohoto typu se nazývá "cast". Jména typu jsou vysvětlena v odstavci 8.7.

Operátor sizeof poskytuje hodnotu, která udává velikost operandu (ve slabikách). Termín slabika (byte) není jazykem definován výjma vztahu k hodnotě, která je poskytována operátorem sizeof. Nicméně, ve všech existujících implementacích jazyka C je byte chápán jako rozměr objektu char. Pokud je tento operátor použit na objekt typu pole, je vrácena hodnota udávající celkový počet slabik pole. Velikost operandu je určena prostřednictvím deklarací jednotlivých objektů ve výrazu. Tento výraz je semanticky celočíselnou konstantou a může být použit všude tam, kde je dovoleno použít konstanty. Hlavní použití tohoto operátoru je v komunikacích s podprogramy, jako jsou alokační podprogramy a podprogramy systémového vstupu a výstupu.

Operátor sizeof může být také použit se jménem typu, které je uzavřeno v závorkách. V tomto případě poskytne velikost daného typu ve slabikách. Konstrukce sizeof (type) je chápána jako celek. Výraz sizeof (type)-2 je totožný s výrazem (sizeof (type))-2.

7.3 Multiplikativní operátory

Multiplikativní operátory jsou *, / a %. Jsou vyhodnocovány zleva doprava a taky se při jejich zpracování uskuteční obvyklé aritmetické konverze.

Multiplikativní výraz:

výraz * výraz

výraz / výraz

výraz % výraz

Binární operátor "*" označuje součin. Tento operátor je asociativní a výraz, který obsahuje několik součinů na té samé úrovni, může být překladačem přeskupen.

Binární operátor "/" označuje dělení. Při dělení kladných čísel je výsledek zaokrouhlen směrem k nule, při dělení záporných čísel je způsob zaokrouhlení závislý na technickém vybavení

počítače. Platí následující vztah: $(a/b)*b + a\%b = a$ (pokud je b různé od 0).

Binární operátor "%" poskytuje zbytek po dělení prvního operandu druhým. Uskuteční se obvykle aritmetické konverze. Operandy nesmějí být typu float.

7.4 Aditivní operátory

Aditivní operátory jsou + a -. Jsou vyhodnocovány zleva doprava a také při jejich zpracování se uskuteční obvykle aritmetické konverze. V některých případech mohou být operandy různých typů.

Aditivní_výraz:

výraz + výraz

výraz - výraz

Operátor "+" poskytuje sumu svých operandů. Může být vytvořen součet ukazovátka ukazujícího na daný objekt a hodnoty celočíselného typu. Druhý operand je ve všech případech konvertován na posunutí ve slabikách, které je získáno součinem druhého operandu s velikostí objektu, na který ukazuje ukazovátka. Výsledkem je ukazovátka na objekt téhož typu. Tedy jestliže P je ukazovátka na objekt v poli, potom $P+1$ je ukazovátka na následující objekt pole. Žádné další kombinace typu operandu nejsou dovoleny.

Operátor "+" je asociativní a výraz, který obsahuje několik součtů na téže samé úrovni, může být překladačem přeskupen.

Operátor "-" poskytuje rozdíl operandu. Při jeho vyhodnocení se uskuteční obvykle aritmetické konverze. Hodnota libovolného celočíselného typu může být odečtena od ukazovátka, potom se uskuteční analogické konverze, které se uskuteční při použití operátoru "+".

Jestliže se uskuteční rozdíl dvou ukazovátek, které ukazují na objekty stejného typu, potom je výsledek konvertován,

dělením velikosti objektu, na hodnotu typu `int`, která představuje počet objektů, které oddělují objekty, na které ukazuje ukazovátka. Tato konverze bude dávat nečekané výsledky pokud ukazovátka nebudou ukazovat do téhož pole, protože potom, i v situaci, kdy pole budou obsahovat objekty téhož typu, není zaručeno, že rozdíl ukazovátek bude celistvým násobkem délky objektu.

7.5 Operátory posuvu

Operátory posuvu `<<` a `>>` se vyhodnocují zleva doprava. Při jejich vyhodnocení se uskuteční obvykle aritmetické konverze. Oba operandy musí být celočíselných typů. Pravý operand je vždy konvertován na typ `int`. Typ výsledku je vždy roven typu levého operandu. Výsledek není definován, jestliže pravý operand je negativní nebo je větší anebo roven velikosti levého operandu v bitech.

výraz_posuvu:

výraz `<<` výraz

výraz `>>` výraz

Hodnota výrazu `R1<<R2` je rovna hodnotě `R1`, která je v tomto případě chápána jako řetězec bitu, posunutě o `R2` bitu vlevo. Hodnota výrazu `R1>>R2` je rovna hodnotě `R1` posunutě o `R2` bitu vpravo. Jestliže je operand na levé straně typu `unsigned` je posuv vpravo je logickým posuvem (uvolněné bitové pozice jsou doplňovány 0). Pokud není typu `unsigned` může se jednat o posuv aritmetický (uvolněné bity se zaplňují kopií znaménkového bitu).

7.6 Relační operátory

Relační operátory jsou vyhodnocovány zleva doprava, ale tato skutečnost není příliš užitečná. Výraz `a<b<c` nemá ten samý význam jako v algebře.

relační_výrazy:

výraz < výraz

výraz > výraz

výraz <= výraz

výraz >= výraz

Operátory < (menší než), > (větší než), <= (menší nebo rovno) a >= (větší nebo rovno) poskytují hodnotu 0, jestliže je daný výraz nepravdivý, a hodnotu 1, jestliže je daný výraz pravdivý. Typ výsledku je int. Při vyhodnocení se uskuteční obvykle aritmetické konverze. Mohou být porovnávány také dvě ukazovátka, výsledek je potom závislý na umístění daných objektů, na které obě ukazovátka ukazují, v adresním prostoru počítače. Porovnávání ukazovátek je přenositelné pouze v tom případě, kdy se jedná o ukazovátka ukazující do téhož pole.

7.7 Operátory porovnání

výraz_porovnání:

výraz == výraz

výraz != výraz

Operátory == (rovno) a != (nerovno) jsou analogické operátorům relačním, výjma jejich nižší priority. Tedy výraz $a < b == c > d$ je roven hodnotě 1, kdykoliv výrazy $a < b$ a $c > d$ poskytují tutéž pravdivostní hodnotu.

Objekt typu ukazovátka může být porovnáván s hodnotou typu int, ale výsledek je závislý na technických prostředcích výpočetního systému, pokud se nejedná o porovnání s konstantou typu int rovnou 0. Ukazovátka, které má hodnotu 0 neukazuje na žádný objekt. V běžném použití se předpokládá, že takové ukazovátka má hodnotu NULL.

7.8 Bitový operátor AND

AND_výraz:

výraz & výraz

Operátor "&" je asociativní a výraz obsahující více těchto operátorů může být přeskupen. Uskuteční se obvykle aritmetické konverze. Výsledek je provedení funkce AND na všech bitech operandu. Operátor může být použit pouze s operandy typu int.

7.9 Bitový operátor XOR

XOR_výraz:

výraz ^ výraz

Operátor "^" je asociativní a výraz obsahující více těchto operátorů může být přeskupen. Uskuteční se obvykle aritmetické konverze. Výsledek je provedení funkce XOR na všech bitech operandů. Operátor může být použit pouze s operandy typu int.

7.10 Bitový operátor OR

OR_výraz:

výraz | výraz

Operátor "|" je asociativní a výraz obsahující více těchto operátorů může být přeskupen. Uskuteční se obvykle aritmetické konverze. Výsledek je provedení funkce OR na všech bitech operandu. Operátor může být použit pouze s operandy typu int.

7.11 Logický operátor AND

logický_and_výraz:

výraz && výraz

Operátor "&&" je vyhodnocován zleva doprava. Vráti hodnotu 1, jestliže jsou oba operandy různé od 0, jinak vrátí hodnotu 0. Na rozdíl od operátoru "&" zaručuje operátor "&&" vyhodnocení výrazu zleva doprava. Navíc druhý operand není vůbec vyhodnocován, jestliže již byla vyhodnocena hodnota prvního operandu rovna 0.

Operandy nemusí být téhož typu, ale musí se jednat o typy základní nebo ukazovátka. Výsledek je vždy typu int.

7.12 Logický operátor OR

logický_or_výraz:

výraz || výraz

Operátor "||" je vyhodnocován zleva doprava. Vráť hodnotu 1, jestliže aspoň jeden z jeho operandů je různý od hodnoty 0, jinak vrátí hodnotu 0. Na rozdíl od operátoru "|" zaručuje operátor "||" vyhodnocení výrazu zleva doprava. Navíc druhý operand není vůbec vyhodnocován, jestliže již byla vyhodnocena hodnota prvního operandu různá od 0.

Operandy nemusí být téhož typu, ale musí se jednat o základní typy nebo ukazovátka. Výsledek je vždy typu int.

7.13 Operátor podmínky

výraz_podmínky:

výraz1 ? výraz2 : výraz3

Výrazy s podmínkou jsou vyhodnocovány zleva doprava. Nejprve je vyhodnocen výraz1, a jestliže je různý od 0, je výsledkem hodnota druhého výrazu, jinak je výsledkem hodnota třetího výrazu. Jestliže je to možné, uskuteční se obvyklé aritmetické konverze tak, aby druhý a třetí výraz měly tentýž typ. Jestliže se jedná o ukazovátka na objekty téhož typu, výsledek má tentýž typ. Pokud je jeden výraz ukazovátko a druhý konstanta rovna 0, má výsledek typ ukazovátka. Vyhodnocuje se vždy pouze jeden z výrazů za znakem "?".

7.14 Operátory přiřazení

Je velký počet operátorů přiřazení, které jsou vyhodnocovány zprava doleva. Všechny vyžadují na své levé straně 1-hodnotu, výsledný typ přiřazení je dán operandem na levé straně. Hodnota operátoru přiřazení je hodnota, která je uložena do operandu na levé straně. Obě dvě části složeného operátoru přiřazení jsou samostatně syntaktické jednotky.

výraz přiřazení:

l-hodnota = výraz
l-hodnota += výraz
l-hodnota -= výraz
l-hodnota *= výraz
l-hodnota /= výraz
l-hodnota %= výraz
l-hodnota >=>= výraz
l-hodnota <=<= výraz
l-hodnota &= výraz
l-hodnota ^= výraz
l-hodnota != výraz

Při jednoduchém přiřazení (=) je hodnota výrazu umístěná do objektu, který je určen l-hodnotou. Jestliže oba operandy mají aritmetický typ, je pravý operand před uskutečněním přiřazení konvertován na typ levého operandu.

Výraz přiřazení ve formě L1 op = L2 je analogický výrazu L1 = L1 op(L2), výjma toho faktu, že výraz L1 je vyhodnocován pouze jednou. U složených výrazů += a -= může být levý operand ukazovátkem. Potom je pravý (celočíselný) operand konvertován tak, jak je vysvětleno v odstavci 7.4. Všechny pravé operandy a všechny levé operandy, které nejsou ukazovátka musí mít aritmetické typy.

Překladač skutečně umožňuje, aby hodnota ukazovátka mohla být přiřazena proměnné typu int, aby hodnota typu int mohla být přiřazena k ukazovátku a hodnota ukazovátka ukazovátku jiného typu. Operace přiřazení je pouhým překopírováním hodnot bez konverzí. Takovéto použití není přenositelné, mohou vzniknout ukazovátka, která nemají smysl. Nicméně při přiřazení konstanty 0 ukazovátku je zaručeno, že ukazovátko nebude ukazovat na žádný objekt.

7.15 Operátor čárky

výraz_s_čárkou:

výraz , výraz

Dvojice výrazů oddělených čárkou je vyhodnocena zleva doprava a hodnota levého výrazu je přenesena doprava. Typ a hodnota výsledku je stejná jako typ a hodnota pravého operandu. V takovém kontextu, kde čárka může mít nějaký speciální význam, například v seznamu skutečných argumentů funkce (odstavec 7.1) nebo v seznamu při inicializaci (odstavec 8.6), může se operátor "," ve významu, který je popisován v tomto odstavci vyskytovat pouze mezi závorkami. Například:

```
f(a, (t=3, t+2), c)
```

Tato funkce má tři argumenty, druhý z nich má hodnotu 5.

8. Deklarace

Deklarace jsou užívány pro specifikaci jednotlivých identifikátorů C jazyka. Nemusí vždy uskutečnit současně rezervaci místa paměti pro daný identifikátor. Deklarace mají následující formu:

deklarace:

```
specifikátor_deklarace seznam_deklarace???
```

Seznam_deklarace obsahuje názvy identifikátoru, které mají být deklarovány.

Specifikátor_deklarace se skládá ze specifikace typu a třídy uložení.

specifikátor_deklarace:

```
specifikace_typu specifikátor_deklarace???
```

```
specifikace_uložení specifikátor_deklarace???
```

Seznam deklarací musí být uspořádán daným způsobem, který bude dále vysvětlen.

8.1 Specifikace třídy uložení

specifikace_uložení:

auto

static

extern
register
typedef

Specifikace typedef nerezervuje žádnou paměť pro uložení objektu a je zařazena mezi specifikace uložení pouze z důvodu syntaktické konvence. Tato specifikace je vysvětlena v odstavci 8.8. Význam zbývajících specifikací byl objasněn ve 4. kapitole.

Specifikace auto, static a register jsou současně i definicemi a způsobují rezervování příslušného množství paměti. V případě specifikace extern se jedná o externí definici (viz. odstavec c.10.) identifikátoru definovaného mimo funkci, ve které je deklarován.

Deklarace register je nejlepší způsob deklarace objektu, pro který by mohla být použita deklarace auto. Tato deklarace naznačí překladači, že se jedná o proměnnou, která bude velmi často používána. Pouze několik prvních takových deklarací však bude efektivních. Navíc, pouze některé typy proměnných mohou být uloženy v registrech. Další výjimka, která platí pro registrové proměnné je ta, že na ně nelze použít adresový operátor "&". Při použití registrových proměnných je možné očekávat menší a rychlejší programy, ale budoucí zlepšení generace strojového kódu překladačem je mohou učinit zbytečnými.

Nejvýše jedna specifikace_uložení může být v jediné deklaraci. Jestliže specifikace_uložení chybí, předpokládá se specifikace auto uvnitř funkcí a extern mimo ně. Výjimkou jsou deklarace funkcí, které nikdy nejsou auto.

8.2 Specifikace typu

Specifikace_type:

char
short
int
long
unsigned

```
float
double
specifikator_structury
specifikator_unionu
typedef_jmeno
```

Názvy long, short a unsigned mohou být použita jako adjektiva. Jsou dovoleny následující kombinace:

```
short int
long int
unsigned int
long float
```

Význam poslední kombinace je tentýž jako specifikace double. Jinak smí být v deklaraci použita pouze jedna specifikace typu. Jestliže specifikace typu v deklaraci chybí, předpokládá se specifikace int.

Specifikace struktur a unionu jsou vysvětleny v odstavci 8.5, deklarace s konstrukcí typové_jméno jsou objasněny v odstavci 8.8.

8.3 Deklarátory

Seznam deklarátorů je seznam položek oddělených čárkami, z nichž každá může obsahovat inicializační část.

seznam_deklarátorů:

```
init_deklarátor
init_deklarátor seznam_deklarátorů
```

init_deklarátor:

deklarátor inicializátor???

Inicializátory jsou vysvětleny v odstavci 8.6. Specifikace v deklaraci určují typ a třídu uložení objektů, které jsou určeny deklarátory.

Syntaxe deklarátoru je následující:

deklarátor:

identifikátor
(deklarátor)
*deklarátor
deklarátor()
deklarátor [konstantní výraz???

8.4 Význam deklarátoru

Každý deklarátor je v podstatě tvrzením o vlastnostech nějakého objektu. Pokud se takový deklarátor té samé formy vyskytne ve výrazu, určuje objekt daného typu a třídy uložení. Každý deklarátor obsahuje pouze jediný identifikátor, a to ten, který je právě deklarován.

Pokud má deklarátor tvar pouhého identifikátoru, získá tento identifikátor vlastností určené specifikační částí deklarace.

Deklarátor uzavřený mezi závorkami je identický s deklarátorem, který mezi závorkami uzavřen není. Možnost závorek je zde proto, aby bylo možné vytvářet složené deklarátory, jejichž význam je určen umístěním závorek.

Předpokládejme následující deklaraci:

t d1

Kde t je specifikátor typu (například int atd.) a d1 je deklarátor. Předpokládejme, že tato deklarace vytvoří identifikátor mající typ "...t", kde "..." je prázdný řetězec, pokud se jedná o základní identifikátor. Tedy typ identifikátoru x je int, pokud byl deklarován deklarací: "int x". Pokud má deklarátor d1 tvar:

*d

typ identifikátoru je: "... pointer na t". Jestliže má deklarátor d1 tvar:

d()

typ identifikátoru je "... funkce vracející t". Jestliže má

deklarátor d1 tvar:

d [konstantní_výraz]

nebo

d[]

typ identifikátoru je "... pole z t". V prvním případě je konstantní výraz vyhodnocen v době překladu a jeho typ je int. Konstantní výrazy jsou definovány v kapitole 15. Pokud je několik specifikací "pole z" vedle sebe, je vytvořeno více-rozměrné pole. Konstantní výraz, který určuje velikost pole, je nepovinný pouze u prvního členu vícenásobné specifikace "pole z". Tato možnost je výhodná, pokud se jedná o externí pole a jeho skutečná definice je v jiné části programu. První konstantní výraz může chybět také v případě, kdy je deklarátor následován inicializátorem. V takovém případě je velikost pole určena z počtu položek inicializátoru.

Pole smí být utvořeno z jednoho ze základních typů, z ukazovátek, ze struktur nebo unionu, anebo z jiného pole (potom se jedná o pole vícerozměrné).

Ve skutečnosti však nejsou dovoleny všechny možnosti vytvoření deklarátoru, které povoluje výše uvedená syntaxe. Omezení jsou následující:

- funkce nesmějí vracet objekty typu pole, struktury, unionu nebo funkce, ačkoliv mohou vracet ukazovátka na výše uvedené objekty
- neexistují pole funkcí, ale mohou existovat pole ukazovátek na funkce
- struktury a uniony nesmějí obsahovat funkce, ale mohou obsahovat ukazovátka na funkce.

Například deklarace:

```
int i, *ip, f(), *fip(), (*pfi)();
```

deklarující celočíselnou proměnnou i, ukazovátka ip na proměnnou typu int, funkci f, která vrací proměnnou typu int, funkcí fip, která vrací ukazovátka na proměnnou typu int a ukazovátka

fip na funkci, která vrací proměnnou typu int. Je velmi užitečné srovnat poslední dvě deklarace. Deklaraci *fip() je možné také napsat ve tvaru *(fip()), ze kterého je vidět, že podobně jako při vyhodnocení výrazu se nejprve volá funkce fip a teprve poté se uskuteční přístup pomocí ukazovátka. Při deklaraci (*pfi()) jsou závorky navíc nutné, protože podobně jako při vyhodnocení výrazu se uskuteční nejprve přístup pomocí ukazovátka na funkci, která vrací hodnotu typu int.

Uvedeme další příklady:

```
float fa [17], *afp[17];
```

Výše uvedená deklarace deklaruje pole čísel typu float a pole ukazovátek na čísla typu float.

Konečně deklarace:

```
static int x3d[3][5][7];
```

deklaruje statické trojrozměrné pole proměnných typu int, s celkovým rozměrem 3x5x7. Pole x3d se skládá ze tří položek, každá tato položka je polem, které se skládá z pěti položek, které jsou také poli, která mají 7 položek typu int. Ve výrazech se může vyskytovat libovolný z následujících výrazů, které odkazují na dané pole: x3d, x3d[i], x3d[i][j], x3d[i][j][k]. První tři mají typ "pole" a poslední má typ int.

8.5 Deklarace struktur a unionu

Struktura je objekt skládající se z řady pojmenovaných položek. Každá položka může mít libovolný typ. Union je objekt, který může mít v daném okamžiku jeden z více libovolných typů. Struktury a uniony se specifikují tou samou formou.

specifikátor struktury nebo unionu:

```
struktura/union {seznam_struktury}  
struktura/union identifikátor {seznam_struktury}  
struktura/union identifikátor
```

struktura/union:

struct
union

Seznam_struktury je řada deklarací pro jednotlivé členy struktury nebo unionu.

seznam_struktury:

deklarace_struktury
deklarace_struktury seznam_struktury

deklarace_struktury:

specifikátor_typu seznam_deklarátorů;

seznam_deklarátorů:

deklarátor_struktury
deklarátor_struktury, seznam_deklarátorů

V obvyklém případě je deklarátor_struktury obyčejným deklarátorem jednoduché struktury nebo unionu. Struktura se může ale také skládat ze specifikovaného počtu bitu. Taková položka se nazývá "pole bitu". Jeho velikost je dána konstantním výrazem, který je od názvu pole bitu oddělen znakem ":".

deklarátor_struktury:

deklarátor
deklarátor: konstantní_výraz
: konstantní výraz

Objekty uvnitř struktury jsou umístěny na adresy podle toho, jaké je jejich pořadí v deklaraci struktury, tedy objekty, které jsou v deklaraci více vpravo, mají vyšší adresy. Každý člen struktury, který není polem bitu, je umístěn na adresu podle svého typu. Proto mohou vzniknout uvnitř struktury místa, která nejsou přiřazena žádné proměnné. Pole bitu jsou zarovnána do objektu typu int. Jedno pole bitu nesmí být nikdy současně umístěno ve dvou takových objektech. Pole bitu, které se nevejde do prostoru, který již v použitém objektu typu int zbývá, je umístěno celé do dalšího objektu. Žádné pole bitu nemůže být větší, než je objekt typu int, který je dán technickými prostředky výpočetního systému.

Deklarátor_struktury, který se skládá pouze ze znaku ":" a definice velikosti pole bitu, ustanovuje nepojmenované pole, které může být použito pro uspokojení požadavků, které jsou dány vnějším vybavením systému. Speciálním případem je pole bitu délky 0. Taková definice vynutí umístění dalšího pole bitu na počátku nového objektu typu int. Normální členy struktury, tj. nikoliv pole bitu, se vždy umísťují od počátku objektu typu int.

Jazyk C nedefinuje, jakého typu mají být pole bitu, ale na implementacích jazyka C není požadováno, aby dovolovaly jiná pole bitu, než celočíselného typu. Tato pole mohou být však považována za pole typu unsigned.

Adresový operátor "&" nemůže být na pole bitu aplikován, ani nemohou existovat ukazovátka na pole bitu.

Union představuje ve své podstatě strukturu, jejíž všechny členy počínají na té samé adrese (relativní adresa od počátku unionu je vždy rovna 0) a velikost unionu je vždy taková, aby byla dostatečná i pro nejrozměrnější člen unionu. V jediném okamžiku může být v unionu uložen pouze jediný člen jeho struktury.

Specifikátor struktury nebo unionu, který má následující formu:

```
struct   identifikátor   seznam_struktury
union    identifikátor   seznam_struktury
```

deklaruje identifikátor, jako tzv. značku struktury (značku unionu), která je specifikována daným seznamem struktury. Další deklarace již mohou používat třetí formu specifikace struktury nebo unionu:

```
struct   identifikátor
union    identifikátor
```

Značky struktury umožňují definovat struktury, které se odkazují na sebe samé. Jejich výhoda spočívá také v tom, že dovolují, aby dlouhé definice struktury byly zapsány pouze jedinkrát

a posléze použity vícekrát. Je zakázáno deklarovat struktury nebo uniony, které mohou obsahovat samy sebe, ale je možné deklarovat struktury nebo uniony, které obsahují ukazovátka na sebe sama.

Názvy členů struktury nebo název značky struktury je vytvořen podle stejných pravidel, jaká platí pro názvy proměnných. Nicméně název značky struktury musí být rozdílný od názvů členů struktury.

Dvě struktury mohou mít společné počáteční položky, tj. ty samé členy se mohou vyskytovat ve dvou rozdílných strukturách, jestliže jsou téhož typu a všechny předcházející položky jsou totožné. Ve skutečnosti překladač kontroluje, zda totožná jména ve dvou rozdílných strukturách jsou téhož typu a zda mají stejné relativní posunutí od počátku struktury. Pokud jsou předcházející členy struktury rozdílné, není taková konstrukce přenositelná.

Následuje jednoduchý příklad deklarace struktury:

```
struct tnode{
    char tword[20];
    int count;
    struct tnode *left;
    struct tnode *right;
};
```

Tato struktura obsahuje znakové pole o 20 znacích, celočíselnou proměnnou typu int a dvě ukazovátka na tutéž strukturu. Jakkmile je jednou taková struktura definována, je možné ji použít v dalších deklaracích:

```
struct tnode s,*sp;
```

Výše uvedená deklarace deklaruje strukturu s s již výše uvedenou strukturou a ukazovátko sp, které ukazuje na strukturu výše již uvedené struktury.

S využitím výše uvedených deklarací následující výraz:

```
sp -> count;
```

určuje položku count struktury, která je určena ukazovátkem sp.

```
s.left;
```

Tento výraz určuje ukazovátko, které určuje levou větev struktury s.

```
s.right -> tword[0];
```

V tomto výrazu ukazovátko pravé větve struktury s ukazuje na první člen položky tword dané struktury.

8.6 Inicializace

Při deklaraci je možné specifikovat počáteční hodnoty jednotlivých proměnných, které jsou deklarovány. Inicializátor počíná znakem "=" a sestává se z výrazu nebo seznamu hodnot uzavřených mezi složené závorky.

Inicializátor:

```
= výraz  
= {seznam_inicializací}  
= {seznam_inicializací,}
```

Seznam_inicializací:

```
výraz  
seznam_inicializací, seznam_inicializací  
{seznam_inicializací}
```

Všechny inicializace statických nebo externích proměnných musí být složeny z konstantních výrazů (viz. kapitola 15) nebo výrazů, které určují adresu již dříve deklarované proměnné, případně její relativní umístění.

Automatické nebo registrové proměnné mohou být inicializovány výrazy obsahující konstanty a již dříve deklarované proměnné a funkce.

Statistické a externí proměnné, které nebyly inicializovány, mají počáteční hodnotu rovnu 0. Automatické a registrové proměnné, které nebyly inicializovány, mají počáteční hodnotu nedefinovanou (náhodnou).

Pokud je inicializován skalár (ukazovátka nebo objekt aritmetického typu), skládá se inicializace z jednoduchého výrazu, který může být uzavřen do složených závorek. Počáteční hodnota daného objektu je určena tímto výrazem, při jeho vyčíslení se uskuteční stejné konverze jako při uskutečnění přiřazení.

Pokud je deklarovaná hodnota agregát (tj. struktura nebo pole), je inicializátor vytvořen ze seznamu inicializací jednotlivých položek agregátu oddělených čárkami a uzavřených mezi složené závorky, které jsou psány dle vzrůstajícího indexu nebo pořadí položek. Pokud agregát obsahuje včleněné agregáty, mohou se tato pravidla aplikovat rekurzivně. Jestliže je inicializováno méně položek než agregát obsahuje, jsou zbývající položky vyplněny 0. Není dovoleno inicializovat uniony a automatické agregáty.

Složené závorky smí být vynechány za následujících podmínek. Jestliže inicializátor začíná levou složenou závorkou, následující seznam inicializací oddělených čárkami inicializuje agregát. Je chybou pokud tento seznam obsahuje více členů, než je položek agregátu. Naopak, jestliže inicializátor nezačíná levou složenou závorkou, jsou inicializovány pouze nezbytné položky daného agregátu. Zbývající inicializace jsou ponechány pro inicializaci položek agregátu.

Je také možné inicializovat pole znaku řetězcem. V takovém případě jednotlivé znaky řetězce inicializují jednotlivé položky znakového pole.

Následují příklady inicializace:

```
int x[] = {1, 3, 5};
```

Uvedený příkaz deklaruje a inicializuje x jako jednorozměrné pole, které má tři členy určené počtem inicializací.

```
float y[4][3] = {  
    {1, 3, 5},  
    {2, 4, 6},  
    {3, 5, 7},  
};
```

Uvedená inicializace plně využívá závorek. Hodnoty 1, 3 a 5 inicializují první řádek pole y (jmenovitě položky y[0][0], y[0][1], y[0][2]). Podobně jsou inicializovány i dva následující řádky pole y[1] a y[2]. Protože inicializace je zakončena předčasně, je poslední řádek pole inicializován hodnotami 0. Totéž je možné uskutečnit následovně:

```
float y[4][3] = {  
    1, 3, 5, 2, 4, 6, 3, 5, 7  
};
```

Inicializace pro pole y začíná levou závorkou, ale inicializace pro řádek y[0] nikoliv, proto jsou použity při inicializaci první tři hodnoty ze seznamu. Podobně další tři jsou použity pro inicializaci y[1] a další tři pro inicializaci y[2].

```
float y[4][3] = {  
    {1}, {2}, {3}, {4}  
};
```

Tato inicializace inicializuje první sloupec pole y (jedná se o dvourozměrné pole) a zbývající sloupce pole vyplní hodnotami 0.

```
char msg[] = "Syntaktická chyba na řádce %s\n";
```

Zde se jedná o inicializaci pole řetězcem znaků.

8.7 Názvy typu

Názvy typu dat je nutné uvádět ve dvou kontextech.

Při specifikaci typu konverze v konstrukci "cast" a jako argument operátoru sizeof. Je to možné použitím názvu typu, což je v podstatě část deklarace typu objektu, ve kterém chybí jméno vlastního objektu.

Jméno typu:

specifikace__typu abstraktní__deklarátor

Abstraktní__deklarátor:

prázdný
(abstraktní__deklarátor)
*abstraktní__deklarátor
abstraktní__deklarátor()
abstraktní__deklarátor [konstantní_výraz??]

Aby nedošlo ke dvojznačnosti v konstrukci typu:

(abstraktní__deklarátor)

je požadováno, aby v této konstrukci nebyl abstraktní__deklarátor prázdný. S tímto omezením je možné vytvořit stejné konstrukce s abstraktním deklarátorem, jako kdyby se jednalo o skutečný deklarátor při deklaraci. Typ této konstrukce je stejný jako typ hypotetického identifikátoru.

V následujících příkladech

int
int *
int *[3]
int (*)[3]
int *()
int (*)()

se jedná postupně o typy: "integer", "ukazovátka na integer", "pole tří ukazovátek na integer", "ukazovátka na pole tří integer", "funkce vracející ukazovátka na integer", "ukazovátka na funkci vracející integer".

8.8 Typedef

Deklarace, jejíž třída uložení je typedef, ve skutečnosti žádnou paměť nepřiděluje, namísto toho definuje identifikátor, který může být později použit jako definice základního či odvozeného typu.

typové_jméno:

identifikátor

Uvnitř typedef deklarace se každý identifikátor, který je součástí některého deklarátoru, stává později syntaktickým ekvivalentem klíčového slova pojmenovávajícího typ, který je identifikátoru přiřazen.

Příklad:

```
typedef int MILES, *KLIKSP;  
typedef struct {double re, im;} COMPLEX;
```

užité v konstrukcích:

```
MILES distance;  
extern KLIKSP metricp;  
COMPLEX z, *zp;
```

jsou správné deklarace. Typ distance je int, typ metricp je "pointer na int", typ z je výše uvedená struktura a typ zp je ukazovátka na danou strukturu.

Konstrukce typedef nevytváří nové typy, pouze synonyma typu, které by mohly být utvořeny normálním postupem. Například distance má úplně stejný typ jako objekt, který byl deklarován s typem int.

9. Příkazy

Příkazy jsou normálně vykonávány v tom pořadí, v jakém jsou zapsány.

9.1 Výrazové příkazy

Základními příkazy jsou výrazové příkazy, které mají následující formu:

výraz;

Nejčastějším typem těchto příkazů jsou přiřazení nebo funkční volání.

9.2 Složené příkazy nebo bloky

Je možné použít několik příkazů tam, kde by mohl být použit pouze jediný. Je to možné pomocí složených příkazů, které jsou také nazývány bloky.

```
složený_příkaz:
    {seznam_deklarací??? seznam_příkazů???}

seznam_deklarací:
    deklarace
    deklarace      seznam_deklarací

seznam_příkazů:
    příkaz
    příkaz      seznam_příkazů
```

Jestliže je některý z identifikátorů, které jsou deklarovány, totožný s identifikátorem již dříve deklarovaným, je vnější deklarace během vykonávání bloku potlačena a po ukončení bloku opět obnovena.

Všechny inicializace proměnných typu auto nebo register jsou provedeny vždy při opětovném přístupu do bloku, pokud je přístup uskutečněn na začátek bloku. Je sice možné přenést řízení přímo dprostřed bloku, což ale není dobrá praxe, potom k inicializacím nedojde. Inicializace proměnných typu static se uskuteční pouze při spuštění programu. Uvnitř bloku externí deklarace nerezervují žádné paměťové místo, inicializace není proto dovolena.

9.3 Podmínkové příkazy

Podmínkové příkazy existují ve dvou tvarech:

```
if(výraz) příkaz1
if(výraz) příkaz1      else příkaz2
```

V obou případech je nejprve vyhodnocen výraz, jestliže je různý od nuly, je vykonán příkaz1. V případě druhého tvaru podmínkového příkazu se vykoná příkaz2, jestliže má výraz hodnotu 0.

Obvyklá dvojznačnost konstrukce s else při vnořených podmínkových příkazech je řešená tak, že část else je spojena vždy s posledním příkazem if, ke kterému dosud žádná část else nenáleží.

9.4 Příkaz while

Příkaz while má tvar:

while(výraz) příkaz

příkaz je opakovaně vykonáván, dokud je hodnota výrazu nenulová. Test této hodnoty se uskuteční vždy před započítím vykonání příkazu.

9.5 Příkaz do

Příkaz do má tvar:

do příkaz while(výraz);

Příkaz je opakovaně vykonáván dokud hodnota výrazu není nulová. Test se provádí vždy po uskutečnění příkazu.

9.6 Příkaz for

Příkaz for má tvar:

for(výraz-1??? ; výraz-2??? ; výraz-3???) příkaz

Příkaz for je ekvivalentní následujícím příkazům:

```
výraz-1;
while(výraz-2)
{
    příkaz
    výraz-3;
}
```

To znamená, první příkaz inicializuje smyčku. Druhý výraz specifikuje test, který se vykoná před každým opakováním a způsobí ukončení smyčky, pokud je hodnota výrazu rovna 0.

Třetí výraz nejčastěji specifikuje inkrementaci, která se uskuteční po každém opakování smyčky.

Libovolný nebo všechny z výrazu mohou chybět. Chybějící výraz-2 způsobí, že je příkaz for ekvivalentní while(1).

9.7 Příkaz switch

Příkaz switch způsobí, že řízení programu bude přeneseno na jeden z několika příkazů v závislosti na hodnotě výrazu. Příkaz switch má následující tvar:

switch(výraz) příkaz

Při zpracování výrazu se uskuteční obvykle aritmetické konverze, ale výsledná hodnota musí být typu int. Příkaz je většinou složeným příkazem. Libovolný příkaz uvnitř tohoto příkazu může být označen následující formou:

case konstantní_výraz:

Konstantní výraz musí být typu int. V jediném příkazu switch nesmějí mít konstantní výrazy u různých příkazů case tutéž hodnotu. Konstantní výrazy jsou definovány v odstavci c. 15.

Nejvíce jeden příkaz uvnitř příkazu switch může být označen následující formou:

default

Při vykonávání příkazu switch je nejprve vyhodnocen výraz a jeho hodnota je porovnávána s každým konstantním výrazem, který přísluší příkazu case. Jestliže je hodnota výrazu rovna některému z konstantních výrazů, je řízení programu přeneseno na příkazy, které následují za příslušným příkazem case. Jestliže hodnota výrazu není rovna žádnému konstantnímu výrazu a příkaz switch obsahuje příkaz default, je řízení přeneseno na příkazy, které následují za příkazem default. Jestliže hodnota výrazu není rovna žádnému konstantnímu výrazu ani žádný příkaz není označen default, je řízení přeneseno za příkaz switch.

Označení case a default samy o sobě nemění řízení programu, který tato označení ignoruje. Pro ukončení jedné větve příkazu switch je pak nutné použít příkaz break (viz odstavec 9.8).

Ve většině případů je příkaz switch příkazem složeným. Na začátku takového příkazu se mohou vyskytovat deklarace, ale inicializace automatických i registrových proměnných je neúčinná.

9.8 Příkaz break

Příkaz break má tvar:

```
break;
```

Tento příkaz způsobí ukončení nejvnitřnějšího příkazu while, do, for nebo switch. Řízení programu se přenesese bezprostředně za ukončený příkaz.

9.9 Příkaz continue

Příkaz continue má tvar:

```
continue;
```

Tento příkaz způsobí přenesení řízení na konec nejvnitřnější smyčky while, do nebo for. Podrobněji vysvětleno, v každém z následujících příkazů

while(...)	do	for(...)
{	{	{
...
contin;;	contin;;	contin;;
	while(...);	
}	}	}

je příkaz continue ekvivalentní příkazu goto contin. Příkaz, který následuje návěští contin je tzv. prázdný příkaz (viz odstavec 9.13).

9.10 Příkaz return

Příkaz return vrátí řízení programu z vnitřku funkce do místa, odkud byla tato funkce vyvolána. Má následující formy:

```
return;  
return výraz;
```

V prvním tvaru je vrácená hodnota nedefinovaná. Ve druhém je hodnota výrazu vrácená té části programu, která danou funkci vyvolala. Jestliže je to nutné, je hodnota výrazu konvertována, jako při příkazu přiřazení, na typ shodný s typem funkce. Přenesení řízení za konec funkce je ekvivaletní návratu bez žádné vrácené hodnoty.

9.11 Příkaz goto

Řízení programu může být nepodmíněně přeneseno pomocí příkazu goto, který má následující formu:

```
goto identifikátor;
```

Identifikátor musí být návěštím (viz odstavec 9.12), které je umístěno v právě vykonávané funkci.

9.12 Návěští příkazu

Libovolný příkaz smí být označen návěštím, které má následující formu:

```
identifikátor:
```

Tento příkaz způsobí, že daný identifikátor je deklarován jako návěští. Hlavní použití příkazu návěští je při použití příkazu goto. Platnost daného návěští je omezena na právě vykonávanou funkci, výjma bloky do ní vnořené, kde je tentýž identifikátor opět deklarován (viz kapitola 11).

9.13 Prázdný příkaz

Prázdný příkaz má následující formu:

Prázdný příkaz se používá s návěštím bezprostředně před koncem složeného příkazu nebo jako prázdné tělo příkazu smyčky, jako je například příkaz while.

10. Externí definice

Program v jazyku C obsahuje řadu externích definic. Externí definice deklarují identifikátory, které mají třídu uložení extern nebo static a daný typ. Specifikace typu může být prázdná (viz odstavec 8.2), v takovém případě se předpokládá typ int. Externí definice platí v celém souboru, ve kterém se taková definice vyskytne. Syntaxe externích definicí je stejná, jako je syntaxe normálních deklarací.

10.1 Definice externích funkcí

Definice funkce má následující tvar:

definice_funkce:

specifikátor_deklarace??? deklarátor_funkce tělo_funkce

Jediné specifikace uložení, které jsou povoleny v specifikátoru_deklarace, jsou extern nebo static, (podrobněji viz odstavec 11.2). Deklarátor_funkce je podobný deklarátoru typu "funkce vracející..." až na definici seznamu formálních parametrů.

deklarátor_funkce:

deklarátor (seznam_parametrů???)

seznam_parametrů:

identifikátor

identifikátor, seznam_parametrů

Tělo funkce má tvar:

tělo_funkce:

seznam_deklarací složený_výraz

Pouze identifikátory ze seznam_parametrů mohou být deklarovány v seznamu_deklarací. Každý identifikátor, kterého typ není určen, má typ int. Jediná třída uložení, která je dovolená ve specifikaci, je třída register. Jestliže je tato třída specifikována, je odpovídající skutečný parametr zkopírován, pokud je to možné, přímo do registru výpočetního systému vně funkce.

Uvedeme jednoduchý příklad kompletní definice funkce:

```
int max (a, b, c,)
int a, b, c;
{
    int m;
    m=(a>b) ? a:b;
    return((m>c) ? m:c);
}
```

Ve výše uvedeném příkladu je int specifikátor typu funkce, max(a,b,c) je deklarátor_funkce, int a,b,c je seznam_parametrů a {...} je blok, který tvoří tělo funkce.

Protože jazyk C konvertuje všechny skutečné parametry, které mají typ float, na typ double, jsou formální parametry, které mají typ float, schopné číst parametry typu double. Protože všechny odkazy na pole jsou chápány jako odkazy na první prvek takového pole, je při deklaraci formálního parametru jako "pole z..." takováto deklarace chápána jako deklarace "pointer na...". Protože struktury, uniony a funkce nemohou být funkcemi jako parametry akceptovány, nemohou být formální parametry deklarovány s typem struktury, unionu nebo funkce. Samozřejmě je dovoleno deklarovat ukazovátka na takové typy.

10.2 Externí definice dat

Definice externích dat má následující formu:

definice_dat:

deklarace

Třída uložení externích dat může být extern nebo static, ale nesmí být auto nebo register. Při neuvedení třídy uložení se předpokládá třída extern.

11. Pravidla rozsahu platnosti

Program v C jazyku nemusí být kompilován najednou v celku. Zdrojový text programu může být uložen v několika souborech a předem přeložené programy mohou být připojeny z knihoven. Komunikace mezi funkcemi programu se může uskutečnit pomocí funkčních volání nebo pomocí externích dat.

Existují dva pohledy na pojem rozsahu platnosti. První z nich určuje lexikální rozsah platnosti identifikátoru. Tento rozsah by mohl být intuitivně chápán jako úsek programu, ve kterém může být tento identifikátor použit, aniž dojde k chybě "nedefinovaný identifikátor". Druhý pohled je spojen s použitím externích identifikátorů, které jsou charakterizovány pravidlem, které tvrdí, že totožné externí identifikátory určují tytéž objekty.

11.1 Lexikální rozsah platnosti

Lexikální rozsah platnosti externích identifikátorů je od místa jejich definice do konce souboru, ve kterém jsou definovány. Lexikální rozsah formálních parametrů je funkce, ve které jsou definovány. Lexikální rozsah identifikátorů definovaných na počátku bloku je od místa jejich definování do konce bloku. Lexikální rozsah návěští je celá funkce, ve které jsou definována.

Protože všechny odkazy na tentýž externí identifikátor musí určovat tentýž objekt (viz odstavec 11.2), kontroluje překladač všechny deklarace téhož externího identifikátoru na kompatibilitu. Výsledkem je pak rozsah platnosti takového identifikátoru po celém souboru, ve kterém se jeho deklarace vyskytují.

Dále platí následující pravidlo. Jestliže je nějaký identifikátor deklarován na počátku bloku, včetně bloku tvořícího definici funkce, jakákoliv deklarace téhož identifikátoru vně tohoto bloku je potlačena do konce daného bloku.

Je nutné připomenout (viz odstavec 8.5), že identifikátor přiřazený skutečné proměnné a identifikátor přiřazený položce struktury nebo unionu či značce struktury nebo unionu patří do dvou různých tříd, které jsou bezkonfliktní. Pro položky a značky struktur a unionu platí stejná pravidla rozsahu jako pro ostatní identifikátory. Mohou být znovu deklarovány ve vnitřním bloku, ale musí být při nové deklaraci explicitně uveden typ.

```
typedef float distance;  
...  
{  
    auto int distance;  
    ...
```

Ve výše uvedeném příkladu musí být ve druhé definici explicitně uveden typ `int`, jinak by tato definice byla pochopena jako definice bez deklarátoru a s typem `distance`. Je nutné přiznat, že tato vlastnost jazyka je poněkud problematická.

11.2 Rozsah platnosti externích proměnných

Jestliže funkce používá identifikátor, který je deklarován jako extern, musí být tento identifikátor jako externí deklarován v některém souboru nebo knihovně, ze kterých je vytvořen komplexní program. Všechny funkce, které pracují se stejným externím identifikátorem, pracují se stejným objektem, musí být tedy zajištěno, že typ a velikost daná definicí identifikátoru je kompatibilní s typem a velikostí specifikované ve všech funkcích, které tento identifikátor používají.

Výskyt klíčového slova `extern` v externí definici znamená, že paměťové místo pro danou proměnnou, která je deklarovaná, bude rezervováno jinou částí programu, která je umístěna

v jiném souboru. Program, který je složen z více částí, které jsou umístěny v samostatných souborech, musí obsahovat pouze jedinou definici externích dat bez klíčového slova extern. V ostatních souborech, ve kterých mají být tato data použita, musí být při jejich definici uvedeno klíčové slovo extern.

Identifikátory, které jsou deklarovány na vnější úrovni programu s klíčovým slovem static, nejsou z ostatních souborů přístupné. Se specifikací static smějí být deklarovány i funkce.

12. Řídící řádky překladače

Překladač jazyka C obsahuje překladač schopný makrosubstituce, podmíněného překladu a začlenění "include" souboru. Řádky, které začínají znakem "#", řídí činnost tohoto předpřekladače. Tyto řádky mají syntaxi nezávislou na syntaxi jazyka C. Mohou se vyskytovat kdekoliv v programu a jejich účinek je od řádku, který následuje, až do konce programu.

12.1 Záměna syntaktické jednotky (znaková náhrada)

Řídící řádek předpřekladače má následující tvar:

```
#define identifikátor řetězec_syntaktických_jednotek
```

Řádek define nekončí středníkem, ale končí zadáním LF. Tato direktiva způsobí, že každý výskyt identifikátoru v programu bude nahrazen řetězcem_syntaktických_jednotek.

Řádek může mít alternativní formu:

```
#define identifikátor(identifikátor, ..., identifikátor)řetě-  
zec_syn._jedn.
```

Nesmí být mezera mezi prvním identifikátorem a znakem "(".
V tomto případě se jedná o definici makra s argumenty. Řada položek, která se skládá s prvního identifikátoru, levé závorky, seznamu syntaktických položek oddělených čárkami a pravé závorky, je nahrazena řetězcem syntaktických položek z řídícího

řádku. Každý výskyt identifikátoru v seznamu formálních parametrů v definici je nahrazen odpovídající syntaktickou položkou použitou při vyvolání makra. Skutečné argumenty při vyvolání makra jsou řetězce syntaktických jednotek oddělených čárkami. Čárky, které jsou v řetězci, který je uzavřen uvozovkami, neoddělují argumenty. Počet formálních a skutečných argumentů musí být stejný. Texty uvnitř řetězců nebo znakové konstanty nejsou nahrazovány.

Při obou formách je řetězec po záměně znovu zpracován předpřekladačem. Dlouhé definice mohou pokračovat na následujícím řádku, jestliže se předcházející řádek ukončí znakem "\".

Tato schopnost překladače je velmi výhodná při definici často používaných konstant, jako například:

```
#define TABSIZE 100
int table[TABSIZE];
```

Řídící řádek, který má následující formu:

```
#undef identifikátor
```

způsobí, že předešlá definice identifikátoru pomocí řídícího řádku `#define` bude překladačem zapomenuta.

12.2 Vkládání souboru

Řídící řádek, který má následující tvar:

```
#include "název_souboru"
```

způsobí nahrazení tohoto řádku obsahem souboru, jehož název je název souboru. Tento soubor je nejprve hledán v adresáři, ve kterém je umístěn zdrojový soubor, který je právě překládán. Pokud není v tomto adresáři nalezen, je postupně hledán v dalších standardních adresářích. Existuje alternativní forma tohoto řídícího řádku, který má následující tvar:

```
#include <název_souboru>
```

V tomto případě jsou prohledávány pouze standardní adresáře a

není prohledáván adresář, ve kterém je umístěn zdrojový soubor.

"Include" soubor může obsahovat další příkaz include.

12.3 Podmíněná kompilace

Řídící řádek překladače, který má následující tvar:

`#if konstantní_výraz`

Jestliže kontrolovaná podmínka je pravdivá, jsou řádky mezi řídícími řádky `#else` a `#endif` ignorovány. Jestliže kontrolovaná podmínka je nepravdivá, jsou ignorovány řádky mezi řádkem obsahujícím test a řádkem s řídícím příkazem `#else`, nebo jestliže není řádek `#else` uveden, řádkem obsahujícím řídící příkaz `#endif`.

Použití těchto konstrukcí může být vnořené.

12.4 Řízení řádkování

Pro použití s ostatními předpřekladači, které mohou generovat program v jazyku C, je zaveden řídící řádek ve tvaru:

`#line konstanta identifikátor???`

Tento řádek způsobí, že překladač bude předpokládat, že číslo následujícího řádku ve zdrojovém souboru je rovno konstantě uvedené v řídícím příkazu `line`, a započne od této hodnoty další řádky číslovat. Toto je užitečné například z důvodů diagnostiky chyb. Pokud je uveden i identifikátor, bude předpokládat, že název zdrojového souboru je totožný se jménem tohoto identifikátoru.

12.5 Připojení assembleru

Do zdrojového textu v C jazyku je možné připojit řádky zdrojového textu v assembleru uvedením příkazového řádku

`#asm`

Před dalšími řádky zdrojového textu C jazyka je nutné použít příkazový řádek

`#endasm`

Řádky mezi příkazy `#asm` a `#endasm` jsou kompilátorem C jazyka ignorovány.

13. Implicitní deklarace

Není vždy nutné specifikovat i třídu uložení i typ identifikátoru v deklaraci. Třída uložení je daná kontextem při externí definici a při deklaracích formálních parametrů a položek struktur. Při deklaracích uvnitř funkcí platí následující pravidla:

- je daná třída uložení, ale není dán typ, potom se předpokládá typ `int`
- je dán typ, ale není dána třída uložení, potom se předpokládá třída `auto`
- výjimkou z druhého pravidla je deklarace funkcí, protože funkce třídy `auto` nemohou existovat (jazyk C není schopen kompilace vykonatelného kódu do zásobníku)
- jestliže typ identifikátoru je "funkce vracející ..." je implicitně deklarován jako extern

Pokud se ve výrazech vyskytne identifikátor, který není ještě deklarován a je následován levou závorkou, je kontextově deklarován jako "funkce vracející `int`".

14. Další informace o typech

V této části budou shrnuty operace, které je možné vykonávat s objekty různých typů.

14.1 Struktury a uniony

Jsou dovoleny pouze dvě operace, které je možné uskutečnit s objekty, které mají typ struktury:

- přístup k jedné z jeho položek pomocí operátoru "." nebo ">"
- získání adresy dané struktury pomocí operátoru "&".

Jiné operace, jako je například přiřazení nebo použití struktury jako argumentu funkce, způsobí chybovou zprávu. Je možné očekávat, že při budoucích rozšířeních jazyka budou takové operace nebo některé z nich dovoleny.

V kapitole 7 je definováno, že při přímém nebo nepřímém přístupu do struktur (pomocí operátoru "." nebo ">") musí být člen na pravé straně položkou struktury a člen na straně levé název struktury nebo ukazovátka na takovou strukturu. Aby bylo možné změnit výše uvedené pravidlo, není překladačem prováděna důsledná kontrola tohoto pravidla. Ve skutečnosti může být na levé straně od operátoru "." libovolná l-hodnota, která má tvar struktury a na pravé straně název položky této struktury. Také na levé straně od operátoru ">" může být uveden libovolný výraz, který má tvar ukazovátka nebo celého čísla. Jestliže se jedná o ukazovátka, předpokládá se, že ukazuje na strukturu a že název jedné položky této struktury je uveden na pravé straně operátoru. Jestliže je na levé straně uveden výraz typu int, je chápán jako absolutní adresa v paměti počítače, kde je daná struktura umístěna.

Je nutné poznamenat, že takovéto konstrukce mohou být nepřenositelné.

14.2 Funkce

Jsou dovoleny pouze dvě operace, které je možné uskutečnit s objekty typu funkce:

- vyvolání funkce
- získání adresy vstupního bodu funkce

Jestliže se název funkce vyskytne ve výrazu, ale nikoliv ve tvaru vyvolání této funkce, je chápán jako ukazovátka na tuto funkci. Je tedy možné použít název funkce jako argument při vyvolání jiné funkce, například:

```
int f();  
...  
g(f);
```

Funkce g musí být definována následovně:

```
g(funcp)  
int(*funcp)();  
{  
    ...  
    (*funcp)();  
    ...  
}
```

Poznamenáváme, že funkce f musí být deklarována explicitně ve vyvolávající části programu, protože název funkce f není při vyvolání následován levou závorkou.

14.3 Pole, ukazovátka a indexy

Kdykoliv se ve výrazu objeví identifikátor pole, je konvertován na ukazovátka ukazující na první položku tohoto pole. Tato konverze je nutná, protože pole nejsou l-hodnotami. Podle definice je operátor indexu "[]" interpretován takovým způsobem, že výraz $L1[L2]$ je totožný s výrazem $((L1) + (L2))$. Jestliže $L1$ je pole a $L2$ celé číslo, potom tyto výrazy určují $L2$ -hou položku pole $L1$. Indexování je komutativní operace.

Podobná pravidla platí i pro vícerozměrná pole. Jestliže V je n -rozměrné pole o velikosti $i \times j \times \dots \times k$, potom je identifikátor V vyskytující se ve výrazu konvertován na ukazovátka, které ukazuje na $(n-1)$ -rozměrné pole o velikosti $j \times \dots \times k$. Jestliže je potom operátor "&", ať už vyjádřený explicitně nebo implicitně jako výsledek indexování, použit s takovým ukazovátkem, výsledek bude ukazovat na $(n-1)$ -rozměrné pole.

Tento výsledek bude okamžitě konvertován na ukazovátka. Uvedeme příklad:

```
int x[3][5];
```

V tomto příkladu se jedná o pole velikosti 3x5 s položkami typu int. Pokud se identifikátor x objeví ve výrazu, je konvertován na ukazovátka, které ukazují na první položku (první ze tří), kterou je pole obsahující 5 členů typu int. Ve výrazu x[i], který je ekvivalentní výrazu *(x+i), je identifikátor x nejprve konvertován na ukazovátka, jak bylo již výše popsáno a potom je index i konvertován na typ objektu x, což znamená, že i je násobeno délkou objektu, na který ukazuje výše zmíněné ukazovátka, tzn. na objekt obsahující 5 objektů typu int. Výsledek je přičten k ukazovátka. Tímto způsobem je nepřímě určeno pole (o 5-ti celočíselných položkách), které je opět chápáno jako ukazovátka na první jeho položku. Jestliže se tentýž argument použije na další index, výše zmíněná pravidla budou použita znovu. Výsledkem bude ale objekt typu int.

Důsledkem všech výše uvedených pravidel je to, že pole v jazyku C jsou ukládána po řádcích a poslední index se mění nejrychleji. První index se podílí na určení velikosti pole, ale nepodílí se na indexových výpočtech.

14.4 Explicitní konverze ukazovátek

Některé konverze ukazovátek jsou sice dovoleny, ale jsou závislé na implementaci jazyka. Takové konverze jsou specifikovány pomocí explicitního operátoru typu konverze (viz odstavce 7.2 a 8.7).

Ukazovátka může být konvertováno na libovolný celočíselný typ, který má vhodnou velikost. Je závislé na technických prostředcích systému, jestli bude vyžadován typ int nebo long. Mapovací funkce je také závislá na technickém vybavení, ale neměla by být neobvyklá pro uživatele, kteří jsou obeznámeni s adresní strukturou výpočetního systému.

Objekty celočíselných typů mohou být konvertovány na ukazovátka. Mapovací funkce by měla zajistit, aby celé číslo získané konverzí ukazovátka zpětnou konverzí poskytlo totožné ukazovátko. Jinak je ovšem způsob mapování zcela závislý na technických prostředcích systému.

Ukazovátko na objekt jednoho typu smí být konvertováno na ukazovátko na objekt jiného typu. Při použití výsledného ukazovátka mohou vzniknout chyby, pokud se nejedná o přiměřené objekty. Musí být zaručeno, že ukazovátko na objekt dané velikosti může být konvertováno na objekt s menší velikostí a zpět beze změny.

Například, funkce pro přidělování paměti přijímá jeden argument, který určuje velikost objektu, kterému má být přidělena paměť (ve slabikách), vrací ukazovátko na typ char. Tato funkce může být použita následujícím způsobem:

```
extern char *alloc();
double *dp;

dp = (double*)alloc(sizeof(double));
*dp = 22.0/7.0;
```

Funkce alloc musí zajistit technickými prostředky systému, že hodnota, kterou vrací, je vhodná pro konverzi na ukazovátko na objekt typu double, potom je použití takové funkce přenositelné.

15. Konstantní výrazy

V některých případech je požadováno, aby výsledkem nějakého výrazu byla konstanta. Například po příkazu case, jako velikost polí a při inicializaci. V prvních dvou případech takový výraz může obsahovat pouze celočíselnou konstantu, znakovou konstantu a operátor sizeof, které mohou být spojeny následujícími binárními operátory:

+ - * / % & ! << >> =x != <> <= >=

nebo unárními operátory: -

nebo ternárními operátory: ? :

Ve výrazu mohou být použity závorky, ale nikoliv funkční volání.

Větší možnosti jsou dovoleny při inicializaci. Mimo konstantních výrazů, tak jak je uvedeno výše, je možné použít unárního operátoru "&" na objekty typu extern nebo static nebo na pole typu extern nebo static s indexem, který je dán konstantním výrazem. Unární operátor "&" může být také použit implicitně tím, že je uveden název pole bez indexu či názvu funkce. Základní pravidlo je následující: vyhodnocený výraz musí být konstanta nebo adresa již dříve deklarovaného objektu typu extern či static, ke které může být přičtena nebo odečtena konstanta.

16. Úvahy o přenositelnosti

Některé aspekty jazyka C jsou závislé na technických prostředcích. Následující výklad některých potíží z toho vyplývajících si nečiní nárok na úplnost, ale naznačí nejdůležitější z nich.

Čisté technické problémy jako je velikost slova a zvláštností aritmetiky pohyblivé čárky a celočíselného dělení se ukázaly v praxi nevelkými. Ostatní vlastnosti technického vybavení se odrážejí v rozdílných implementacích. Některé z nich, zejména rozšíření znaménka (při konverzi záporného znaku do záporného čísla) a pořadí, ve kterém jsou slabiky ukládány do slova, jsou drobnosti, které musí být pečlivě studovány. Mnohé další problémy již nejsou závažné.

Počet registrových proměnných, které mohou být skutečně umístěny v registrech, je rozdílný na různých systémech, podobně jako soubor platných typů. Je věcí překladače, aby ošetřil

správně takové situace. Přebývajících nebo neplatné registrové deklarace jsou jednoduše ignorovány.

Některé problémy mohou vzniknout při použití pochybných programových praktik. Je velice nerozumné psát takové programy, které závisí na takových vlastnostech.

Pořadí vyhodnocení argumentů funkcí není v definici jazyka specifikováno. Pořadí, ve kterém může dojít k vedlejším efektům, není také specifikováno.

Protože znakové konstanty jsou ve skutečnosti objekty typu int, jsou dovoleny i víceznakové konstanty. Skutečná implementace je závislá na technických prostředcích, protože pořadí, ve kterém jsou znaky přiřazovány slovům, se na různých systémech liší.

Bitová pole jsou přiřazena slovům a znaky jsou přiřazeny celým číslům v závislosti na systému zprava doleva nebo zleva doprava. Tyto rozdíly jsou neviditelné samotným programům, pokud si nedopřávají rafinovanosti s typy dat (například, když skuteční konverzi ukazovátka na objekt typu int na ukazovátka na objekt typu char, které potom použije k prohlížení původního objektu), ale musí odpovídat vlastnostem technického vybavení.

Jazyk přijímaný různými překladači se liší nepatrnými detaily.

17. Anachronismy

Protože jazyk C je jazyk, který je ve vývoji, mohou být ve starších programech nalezeny některé neobvyklé konstrukce. Ačkoliv některé překladače dovoluují tyto anachronismy používat, budou nakonec odstraněny a problém přenositelnosti bude překonán.

Dřívější verze jazyka C používaly formu `=op` namísto formy `op=` pro operátory přiřazení. Tím byla způsobena dvojznačnost ve výrazu:

```
x=-1
```

Tento výraz opravdu bude dekrementovat proměnnou `x`, protože znaky `"="` a `"-"` jsou v bezprostředním sousedství, ale mohl by být velmi snadno zaměněn za příkaz přiřazení hodnota `-1` proměnné `x`.

Byla také změněna syntaxe inicializace, dříve nebyl používán při inicializaci znak přiřazení `"="`. Tedy dřívější zápis:

```
int x 1;
```

je nyní nahrazen zápisem:

```
int x = 1;
```

Název : CJAZYK - referenční příručka
Autor : RNDr. Viktor Fuhrmann
Zpracoval : OBZOR Praha
Určeno : Uživatelům TNS
Vydání : první
Výtisků : 1000

Tisk MTZ, provoz 34 Kyjov