JAZYK C (1)

Céčko na přelomu tisíciletí

Skutečnost, že v poměrně nedávné době byla přijata nová verze standardu ISO jazyka C, je dostatečně známa, horší už to je se znalostí novinek, které přinesla. Není jich málo, ale snad vám orientaci usnadní jejich třídílný přehled, který právě zahajujeme.

Na malém povědomí programátorů o změnách v normě ISO má svůj podíl nepochybně i skutečnost, že nejrozšířenější komerční překladače jazyka C, dodávané jako součást vývojových nástrojů pro C++, je dosud neimplementují – což ovšem neznamená, že by o nich „céčkaři" nemohli předem vědět. Podívejme se tedy, o co jde.

Trocha historie

Jazyk C vytvořil, jak známo, D. Ritchie při práci na jedné z verzí operačního systému UNIX na počátku sedmdesátých let. Tento jazyk se poměrně brzy stal značně populárním a koncem sedmdesátých let o něm D. Ritchie napsal spolu s B. W. Kernighanem knihu *The C Programming Language* (Addison-Wesley 1978), která se stala jeho neoficiální normou. O této verzi jazyka C se zpravidla hovoří jako o „Céčku podle K&R". Poznamenejme, že první verze jazyka C prošly brzy poměrně výraznými změnami – můžeme se o nich dočíst v dodatku zmíněné knihy –, takže Céčko podle K&R nejenže není první verzí jazyka, ale není s ní ani kompatibilní.

Na počátku osmdesátých let již jazyk C patřil k nejpopulárnějším programovacím jazykům a byl k dispozici pro velkou většinu počítačů. To, spolu s narůstajícím počtem dialektů, vedlo pochopitelně ke snaze o standardizaci. První byly americké národní normy ANS X3.159, přijaté v letech 1988 a 1989. V přehledu amerických norem se dočteme, že v roce 1990 byla tato norma stažena a nahrazena mezinárodní normou ISO/IEC 9899:1990.

Brzy se ovšem ukázalo, že přijatá mezinárodní norma má některé nedostatky – především nevyhovovala požadavkům na používání jazyka C v mezinárodním (přesněji neanglickém) prostředí. Vedle toho se některé rysy jazyka ukázaly jako zastaralé a naopak se objevila řada rozšíření, mnohá zjevně inspirovaná jazykem C++.

Proto byly v polovině devadesátých let přijaty technické dodatky a opravy označené ISO/IEC 9899:1990/COR1:1994, ISO/IEC 9899:1990/AMD1:1995 a ISO/IEC 9899:1990/COR2:1996. V roce 1999 pak byla přijata druhá veze mezinárodního standardu jazyka C, označená ISO/IEC 9899:1999. Tato druhá verze v plném rozsahu ruší a nahrazuje předchozí verzi včetně všech oprav a dodatků k ní. A konečně v roce 2001 byly vydány opravy k nové normě pod označením **ISO/IEC 9899:1999/Cor. 1:2001(E)**. Tento dokument odstraňuje známé překlepy a nedopatření v původním znění normy.

Nejdůležitější změny

Než se pustíme do podrobností, shrneme nejdůležitější změny, které nová norma přinesla. Tento výběr bude poněkud subjektivní, úplný přehled by však byl příliš dlouhý.

- Součástí jazyka C se staly nástroje a knihovny pro práci s vícebajtovými znaky a znaky v kódování UNICODE.
- Z jazyka bylo odstraněno pravidlo známé pod zkratkou „implicitní `int`".
- Přibyly nové celočíselné typy a změnila se pravidla pro celočíselné konverze.
- Součástí jazyka se staly nástroje pro práci s komplexními čísly.
- Datové typy pro práci s reálnými čísly mají nyní přesně vymezenu vazbu na standard IEC 60559 (známý také jako IEC 559 nebo IEEE 754).
- Lokální pole mohou mít dynamické meze.
- V deklaraci funkce lze použít modifikátor `inline`.
- Modifikátor `restrict` usnadňuje optimalizaci při práci s ukazateli.
- Rozšířily se možnosti inicializace polí, struktur a unií.
- Důsledkem a součástí změn normy jazyka C je i zavedení několika nových standardních hlavičkových souborů.

Příjemné drobnosti

Začneme tím, že se podíváme na několik novinek, které nejsou nijak zásadní, ale zpříjemní nám život, až se stanou běžnou součástí implementací jazyka C.

Komentáře

Jedním z nejčastějších rozšíření, s nimiž jsme se mohli v překladačích jazyka C v posledních letech setkat, byl jednořádkový komentář ve stylu jazyka C++. Takovýto komentář začíná dvojicí znaků `//` bezprostředně za sebou a končí na konci řádku. S příklady se setkáme dále.

Kód a deklarace

V klasickém C bylo třeba v bloku nejprve deklarovat proměnné a teprve za deklaracemi mohly následovat příkazy. Nyní smíme deklarace a kód libovolně míchat. (Také zde se nejspíš uplatnil vliv C++.) To znamená, že nyní můžeme napsat

```
int A[N], i;
for(i=1; i < N; i++) fread(A+i, 1, sizeof(int),
    vstup);
int x = g(A);          // To lze
```

V tomto fragmentu zdrojového kódu předpokládáme, že `vstup` je binární soubor otevřený pro čtení a `g()` je funkce očekávající jako skutečný parametr ukazatel na první prvek celočíselného pole a vracející celočíselnou hodnotu.

Deklarace se tedy chová v podstatě jako příkaz. Přesto syntaktická pravidla jazyka C rozlišují příkazy a deklarace jako samostatné kategorie.

Pojmenované inicializátory složek

Při inicializaci rozsáhlých polí a struktur jsme občas naráželi na problém, jak elegantně zadat jen některé hodnoty – např. hodnoty prvků polí s vysokým indexem. Jazyk C také neumožňoval inicializovat jinou složku unie než tu, která byla v deklaraci zapsána jako první.

Tyto problémy odstraňují pojmenované inicializátory. **Pojmenovaný inicializátor** (*designator*) prvku pole se skládá z indexu uzavřeného v indexových závorkách, za nímž následuje znak `=` a výraz představující počáteční hodnotu; pojmenovaný inicializátor složky struktury

nebo unie začíná tečkou, za níž následuje jméno složky, znak = a výraz určující hodnotu. U složitějších datových struktur se pojmenované inicializátory vytvářejí skládáním uvedených konstrukcí. Pojmenované inicializátory lze samozřejmě kombinovat s nepojmenovanými. To nám umožňuje např. napsat

```
int A[N] = {1, [300]=16};
```

Všechny prvky tohoto pole budou inicializovány hodnotou 0, s výjimkou `A[0]`, jenž bude mít hodnotu 1, a `A[300]`s hodnotou 16. Index v pojmenovaném inicializátoru musí pochopitelně ležet v rozmezí daném deklarací typu pole.
Podobně můžeme napsat

```
struct alfa {int x; double y;};
struct alfa a = {.y = 5.23};
```

Složka `a.x` bude inicializována hodnotou 0.

Složené literály

Za tímto divným označením (anglicky *composite literals*) se skrývá velice užitečný prostředek, který vyplňuje zjevnou a stěží pochopitelnou mezeru ve vyjadřovacích možnostech jazyka C. (Podobnou ovšem najdeme i v C++ a v mnoha dalších jazycích.)
Předchozí verze jazyka dovolovala napsat deklarace s inicializací

```
int A[] = {1,2,3,4,5};
struct alfa a = {9, 3.21};
```

kde typ `struct alfa` je stejný jako v předchozím odstavci. Nebylo však možné napsat analogické přiřazení, např.

```
a = {0, 2.13}; // Nešlo (a nejde)
```

Přesně řečeno, v jazyce C neexistovala možnost, jak – s výjimkou znakových řetězců – vytvořit literál představující pole nebo strukturu. Inicializátor pole nebo struktury se totiž za literál nepovažuje.
Nová norma tuto možnost poskytuje; složený literál má tvar

```
( jméno_typu ) { seznam_inicializátorů }
```

To znamená, že proměnné `a` typu `struct alfa` můžeme přiřadit novou hodnotu příkazem

```
a = (struct alfa){2, 6.67};
```

Podobně lze vytvořit pole a použít ho:

```
int *p;
/* ... */
p = (int []){4, 5, 6};
```

Ve složených literálech můžeme použít i pojmenované inicializátory. Specifikace typu ve složeném literálu může obsahovat i modifikátor `const`:

```
(const float []){1.0, 2.0, 3.0}
```

Literály s modifikátorem `const` může překladač uložit do části operační paměti určené pouze pro čtení.

Složené literály představují nepojmenované objekty, a proto nemohou odkazovat samy na sebe.

Identifikátor _func_

V těle každé funkce je nyní automaticky deklarován identifikátor

```
static const char __func__[];
```

Jeho hodnotou je znakový řetězec obsahující identifikátor dané funkce. I když to možná vypadá podivně, jde o poměrně užitečnou věc, která se hodí zejména při ladění. Možné použití ukazuje následující fragment zdrojového kódu:

```
#ifdef LADIM
 #define JMENO printf("vstup do funkce %s \n", __func__)
#else
 #define JMENO (void)0
#endif
void Fun()
{
        JMENO;
        /* ... a další příkazy ... */
}
```

Bude-li #definován identifikátor `LADIM`, způsobí každé volání funkce `Fun()` výpis zprávy „*vstup do funkce Fun*“.

Ladění

Změny se dotkly i známého ladicího makra `assert()`. Pokud aserce neplatí, vypíše nová verze tohoto makra vedle textové podoby „prosazovaného“ výrazu, jména souboru a čísla řádky také jméno funkce, v níž byla chyba zjištěna. K tomu použije identifikátor `__func__`.

Co zmizelo

Podívejme se nyní na několik rysů jazyka C, které nová verze standardu odstranila.

Implicitní int

V předchozích verzích jazyka C jsme v jistých situacích mohli vynechat označení typu a překladač si doplnil `int`. Šlo např. o deklaraci funkce nebo o deklaraci proměnné s některým z modifikátorů (například `static` nebo `const`). To podle nové normy nelze. Znamená to také, že např. tradiční verze programu „Hello, world“, jíž začíná snad každá učebnice jazyka C, je podle nové normy nesprávná:

```
main() {                // Nyní NELZE
        printf("Helou");
}
```

Typ vracené hodnoty je třeba explicitně uvést,

```
int main() {            // OK
// ... atd.
```

Lze ale předpokládat, že ještě dlouhou dobu budou překladače při opomenutí v této situaci hlásit varování, nikoli chybu.

Poznamenejme, že toto pravidlo naštěstí neznamená, že bychom jména běžných celočíselných typů museli psát v plném znění. Například `unsigned`, resp. `short` jsou stále dovolené zkratky pro typy `unsigned int`, resp. `short int`.

Implicitní deklarace funkce

Jazyk C podle staré normy připouštěl použití nedeklarované funkce. Výše uvedený příklad funkce `main()` byl ve starších učebnicích tohoto jazyka zcela typický, teprve někdy v devadesátých letech začaly úvodní učebnicové příklady uvádět na počátku direktivu

```
#include <stdio.h>
```

Nová norma po vzoru jazyka C++ požaduje, aby prvnímu použití jakékoli funkce předcházela definiční deklarace nebo prototyp. (Slušné překladače jazyka C už dlouhou dobu hlásí varování, jestliže použijeme funkci, jejíž prototyp neznají.)

Definice funkce a typedef

Starší překladače dovolovaly deklarovat typ představující funkci jako `typedef` a takto deklarovaný identifikátor použít v definici funkce:

```
typedef int F(void);          // To je stále OK
F f, g;                       // I tohle je OK
F f { /* ... */ }             // Nyní CHYBA
F *g(void) { /* ... */ }      // OK
```

Poslední deklarace představuje funkci `g()` vracející ukazatel na funkci vracející `int` a bez parametrů.

Současný standard dovoluje, zhruba řečeno, využít identifikátor zavedený pomocí uvedené deklarace `typedef` jen v situacích, kde neslouží k definici funkce – nenahrazuje vracený typ spolu se seznamem formálních parametrů. Jeho použití k definici funkce uvádí standard výslovně jako nedovolenou konstrukci. (Dovolím si poznamenat, že neznám nikoho, kdo by tuto možnost ve starších překladačích využíval.)

Celočíselné datové typy

Celočíselných typů se dotkla řada změn, a některé z nich jsou poměrně závažné. Připomeňme si, že pod souhrnným označením „celočíselné typy" se skrývají nejen typy, v jejichž jménech se může objevit klíčové slovo `int`, ale i znaky, výčtové typy a další.

Znaky

Jazyk C nyní rozlišuje tři jednobajtové znakové typy, a to `char`, `unisgned char` a `signed char`. Typ `char` je sice vždy implementován jako jeden ze zbývajících dvou (standard ponechává na implementaci, který), ale považuje se za zvláštní typ. (Není mi jasné, jaký to má význam; v C++ lze tyto tři znakové typy využít pro rozlišování přetížených funkcí, nic takového ovšem v jazyce C nemáme.)

Pro „široké" znaky je, podobně jako v předchozích verzích, k dispozici typ `wchar_t`, definovaný v souboru `stddef.h` jako `typedef` celočíselného typu schopného reprezentovat všechny znaky z rozšířené znakové sady ve všech podporovaných lokálních nastaveních (*locale*).

Pro vyjádření znakových konstant lze použít kromě možností známých z předchozí verze jazyka C také tzv. **univerzální jména** (identifikátory) znaků. Tato jména vyjadřují kód znaku v kódování UNICODE ve tvaru `\Uhhhhhhhh` nebo `\uhhhh`, kde `h` označuje šestnáctkovou číslici. Možnost `\uhhhh` znamená totéž co `\U0000hhhh`. Poznamenejme, že přesný význam univerzálních jmen znaků je popsán v mezinárodním standardu ISO/IEC 10646.

Univerzální jména znaků lze použít i ve znakových řetězcích, nesmějí však specifikovat hodnoty vyhrazené standardem ISO/IEC 10646 pro řídicí znaky, znak DELETE a znaky z tzv. S-zóny, používané kódováním UTF-16. Některá univerzální jména znaků lze používat také v identifikátorech.

K funkcím, které slouží k práci se „širokými" znaky, se vrátíme ve třetím dílu.

Celá čísla v počítači

Jak se v paměti počítače zobrazují celá čísla? V případě datových typů bez znaménka je to poměrně jednoduché, v případě typů se znaménkem se používá několik způsobů.

Poznamenejme předem, že pokud dojde k celočíselnému přetečení, tj. pokud výsledek celočíselné aritmetické operace neleží v rozsahu použitého datového typu, pro typy se znaménkem není výsledek v obecném případě definován.

Celá čísla bez znaménka

Pro zobrazení celých čísel bez znaménka se všeobecně používá tzv. **přímý kód**: Jednotlivé bity zobrazení čísla *x* v paměti představují jednotlivé číslice vyjadřující hodnotu *x* ve dvojkové soustavě (v případě potřeby zleva doplněné nulami).

Například hodnota 1, uložená ve dvou bajtech, je vyjádřena jako 0000000000000001, hodnota 65535 (maximální možná pro daný rozsah) je vyjádřena jako 1111111111111111.

Aritmetika čísel bez znaménka probíhá vždy modulo 2^N, kde *N* je počet bitů vyhrazených pro hodnotu daného typu. To mj. znamená, že překročení rozsahu se zpravidla (např. právě v jazyce C) nepovažuje za přetečení. Například pro dvoubajtový typ `unsigned int` (s nímž jsme se na PC setkávali v 16bitových implementacích jazyka C) bude platit `65535 + 1 == 0`.

Dvojkový doplněk

Máme-li pro daný celočíselný typ k dispozici *N* bitů, mohli bychom v něm zobrazit 2^N celých čísel bez znaménka (čísla 0 až 2^N - 1), jak jsme právě viděli. V případě datových typů se znaménkem je ale třeba vyhradit část rozsahu pro příznak znaménka.

Zobrazení označované jako **dvojkový doplněk** (*two's complement*) je definováno takto:

- kladné hodnoty *x* ležící v rozmezí 0 až 2^{N-1}-1 se zobrazí v přímém kódu,
- záporné hodnoty *x* v rozmezí -2^{N-1} až -1 se zobrazí jako $2^N + x$.

Nejvyšší bit kladných čísel má v tomto zobrazení vždy hodnotu 0, nejvyšší bit záporných čísel má hodnotu 1. Proto se o něm hovoří jako o „znaménkovém bitu". Např. hodnota 1 se zobrazí jako 0000000000000001, hodnota -1 jako 1111111111111111.

(Zobrazení ve dvojkovém doplňku charakterizuje standard formulací „znaménkový bit má hodnotu 2^N".)

Rozsah možných hodnot je při dvojkovém doplňku vždy nesymetrický, od -2^{N-1} do 2^{N-1}-1. To znamená, že dvoubajtový typ `int` může obsahovat celočíselné hodnoty v rozmezí od –32768 do 32767. To může působit u některých operací problémy; ve dvojkovém doplňku např. nelze vyjádřit absolutní hodnotu konstanty `INT_MIN`.

Neleží-li výsledek aritmetické operace v rozmezí zobrazitelných hodnot, dojde k přetečení (do znaménkového bitu) a dostaneme hodnotu „modulo 2^N posunutou o 2^{N-1}", tedy například `32767+1 == -32768`.

Procesor ovšem může nastavit příznak přetečení nebo vyvolat výjimku.

Dvojkový doplněk představuje patrně nejrozšířenější způsob zobrazování celých čísel se znaménkem; na osobních počítačích se s ním setkáme vždy.

Jednotkový doplněk

Zobrazení v **jednotkovém doplňku** (*one's complement*) se podobá zobrazení ve dvojkovém doplňku, avšak záporné hodnoty *x* jsou zobrazeny jako 2^{N-1}-1 + *x*. (Zobrazení v jednotkovém doplňku charakterizuje standard formulací „znaménkový bit má hodnotu 2^N-1".)

Na rozdíl od dvojkového doplňku je rozsah zobrazitelných hodnot symetrický, od -2^{N-1}+1 do 2^{N-1}-1. Na druhé straně zde ovšem existují dva obrazy nuly, a to 000...00 (kladná nula) a 111...11 (záporná nula).

Rozšířený přímý kód

Třetí ze zobrazení, která standard jazyka C připouští jako implementační možnost, je rozšíření přímého kódu anglicky zvané *sign and magnitude*, doslova „znaménko a velikost". Nejvyšší bit opět vyjadřuje znaménko, zbývajících *N*-1 bitů vyjadřuje absolutní hodnotu čísla v přímém kódu. Kladná čísla se v něm tedy opět zobrazují stejně jako v přímém kódu. Také toto zobrazení má symetrický rozsah, ovšem platí za to opět existencí kladné a záporné nuly (zobrazené jako 000...00 a 100...00).

Logické hodnoty

Novinkou je také datový typ `_Bool` pro vyjádření logických hodnot. Jde o celočíselný typ bez znaménka, který může obsahovat hodnoty 0 (nepravda, *false*) a 1 (pravda, *true*). V hlavičkovém souboru nazvaném `stdbool.h` jsou definována makra `true` a `false` se zřejmým významem, makro `bool`, které se rozvine v klíčové slovo `_Bool`, a také makro `__bool_true_false_are_defined` s hodnotou 1.

Pro typ `_Bool` jsou definovány implicitní konverze, které odpovídají tradičnímu používání číselných hodnot a ukazatelů v podmínkách: Hodnoty různé od nuly se konvertují na 1, nulové hodnoty všech typů se konvertují na 0 typu `_Bool`.

Celá čísla

Vedle typů `short`, `int` a `long` a odpovídajících celočíselných typů bez znaménka máme v jazyce C podle nové normy k dispozici také typ `long long int` a `unsigned long long int` (klíčové slovo `int` lze pochopitelně vynechat). Rozsah typu `long long int` nesmí být menší než rozsah typu `long int` a podobné pravidlo platí i pro variantu bez znaménka.

Literály typu `long long` mají příponu `LL` nebo `ll`.

Vedle těchto typů může konkrétní implementace obsahovat další celočíselné typy se znaménkem i bez něj. Kromě toho standard předepisuje, že implementace musí poskytovat datové typy s určitým rozsahem (např. 32bitová celá čísla se znaménkem a bez něj). O těchto typech budeme hovořit později.

Standard nyní také specifikuje způsob zobrazení celých čísel v počítači. Nabízí tři možnosti – dvojkový doplněk, jednotkový doplněk a přímý kód. (Podrobnosti viz rámeček.)

Další celočíselné typy

Předchozí celočíselné typy označujeme jako **základní** – každá implementace je musí poskytovat. Standard nestanoví, jaký přesně rozsah mají základní celočíselné typy mít (s výjimkou znakových typů). Pouze určuje omezení zdola – např. typ `int` musí mít rozsah alespoň od –32767 do 32767 – a požaduje, aby typ `int` byl „přirozený“ pro danou architekturu (tedy aby např. odpovídal šířce aritmetických registrů).

Vedle toho ovšem implementace mohou či musí poskytnout celočíselné typy s určitou přesně danou šířkou (počtem bitů), s určitou minimální šířkou, nejrychlejší typy s určitou minimální šířkou, typy, jež mohou obsahovat ukazatele na proměnné, a typy s maximální šířkou. Zpravidla jde jen o jiná jména pro základní celočíselné typy, zavedená pomocí deklarace `typedef` v hlavičkovém souboru `stdint.h`.

Celočíselné znaménkové typy s přesně určenou šířkou mají jména tvaru `intN_t`, kde *N* je číslo vyjadřující počet bitů; celočíselné typy bez znaménka s přesně určenou šířkou mají jména tvaru `uintN_t`. Pokud implementace obsahuje celočíselné typy s šířkou 8, 16, 32 nebo 64 bitů, musí nabízet odpovídající „typedefová“ jména, tedy `int8_t`, `int16_t`, `uint8_t` atd.

Jména tvaru `int_leastN_t`, resp. `uint_leastN_t` označují celočíselné typy se znaménkem, resp. bez něj, se šířkou alespoň *N* bitů. Každá implementace musí poskytnout implementace pro *N* rovné 8, 16, 32 a 64.

Jména tvaru `int_fastN_t`, resp. `uint_fastN_t` označují celočíselné typy se znaménkem, resp. bez něj, se šířkou alespoň *N* bitů, pro něž jsou v dané architektuře výpočty nejrychlejší. Každá implementace musí poskytnout implementace pro *N* rovné 8, 16, 32 a 64.

Typy `intptr_t`, resp. `uintptr_t` označují celočíselný typ se znaménkem, resp. bez něj, do něhož lze uložit hodnotu typu `void*` a pak ji přenést zpět, aniž se změní. Tyto typy nejsou povinné.

Typ `intmax_t` označuje celočíselný typ se znaménkem, schopný reprezentovat jakoukoli hodnotu jakéhokoli celočíselného znaménkového typu v dané implementaci (tedy celočíselný typ se znaménkem s největším rozsahem). Podobně typ `uintmax_t` označuje celočíselný typ bez znaménka s největším rozsahem. Tyto dva typy jsou povinné.

Příště

Povídání o celočíselných datových typech se do rozsahu tohoto článku celé nevejde, a dokončíme je proto až v příštím dílu. Tam se také podíváme na reálná čísla a na některé další novinky, které poslední verze standardu přinesla. ■ ■ ■ Miroslav Virius, *autor@chip.cz*