JAZYK C

Céčko na přelomu tisíciletí

V závěrečné části povídání o novinkách, které přinesla druhá verze standardu ISO jazyka C, bude řeč především o ukazatelích, polích, funkcích a makrech. Povšimneme si však i standardní knihovny a implementací novinek v běžných překladačích.

UKAZATELE

V deklaraci ukazatele na objekt a v typech odvozených lze nyní použít modifikátor `restrict`. (Termín *objekt* zde znamená jenom oblast paměti sloužící k ukládání dat, nikoli pojem z objektově orientovaného programování.) Po stránce syntaxe se tento modifikátor podobá modifikátorům `const` a `volatile`; standard jazyka ho řadí spolu s nimi do skupiny tzv. **typových kvalifikátorů**. Například deklarací

```
int * restrict rpa;
```

zavádíme „restringovaný“ ukazatel na typ `int`.

VÝZNAM

Mezi objektem, s nímž pracujeme pomocí restringovaného ukazatele, a tímto ukazatelem je vztah, který vyžaduje, aby se při každém – přímém nebo nepřímém – přístupu k onomu objektu využívala hodnota daného ukazatele. Jestliže např. použijeme ukazatel `rpa` deklarovaný v předchozím odstavci a přiřadíme mu adresu paměti alokované pomocí funkce `malloc()`,

```
rpa = malloc(N*sizeof(int));
```

vznikne takovýto vztah mezi `rpa` a alokovanou pamětí. Při jakékoli manipulaci s alokovanou pamětí bychom měli používat pouze ukazatel `rpa` a hodnotu tohoto ukazatele bychom neměli přiřazovat jiným ukazatelům. Také změna hodnoty restringovaných ukazatelů je omezena; restringovaný ukazatel by měl v jednom bloku ukazovat stále na stejný objekt.

Modifikátor `restrict` tedy znamená, že k objektu, který takovýto ukazatel zpřístupňuje, nepřistupujeme jiným způsobem, pomocí jiných ukazatelů. Pokud toto pravidlo porušíme, může se program chovat nedefinovaným způsobem.

Smyslem vztahu vytvořeného modifikátorem `restrict` je umožnit překladači optimalizace, které by jinak nebyly možné s ohledem na aliasing, tj. modifikaci hodnot objektů prostřednictvím jiných ukazatelů.

PŘÍKLADY

Následující příklady jsou převzaty přímo ze standardu ISO/IEC 9899:1999. Deklarujeme-li na úrovni souboru ukazatele

```
int * restrict a;
int * restrict b;
extern int c[];
```

tvrdíme tím, že přistoupíme-li k nějakému objektu prostřednictvím jednoho z těchto ukazatelů a je-li tento objekt někde v programu měněn, nebudeme k němu přistupovat prostřednictvím zbývajících dvou.

Modifikátor `restrict` můžeme také použít v deklaraci parametrů funkce, například takto:

```
void f(int n, int * restrict p,
                int * restrict q)
{
        while(n-- > 0)
                *p++ = *q++;
}
```

Tím se zavazujeme, že při žádném volání této funkce nebude ukazatel `p` přistupovat ke stejnému objektu jako ukazatel `q`.

Tento příklad ukazuje obě protikladné stránky používání modifikátoru `restrict`. Jeho výhodou je, že umožňuje překladači provést efektivní analýzu závislostí funkce `f()`, aniž by musel zkoumat všechna její volání v programu, a podle toho optimalizovat její kód. Ovšem nic není zadarmo. Toto zkoumání se – alespoň zčásti – přenáší na bedra programátora, který si musí při každém volání uvědomit, zda použité parametry nepovedou k nedefinovanému chování.

Například volání

```
extern int A[100];
f(50, A, A+50);        // OK
```

je v pořádku, zatímco volání

```
f(50, A, A+1); // NEDEFINOVANÉ
CHOVÁNÍ
```

může způsobit nedefinované chování programu. Problém je v tom, že s prvky `A[1]` až `A[49]` pole `A` pracujeme v rámci jediného volání funkce `f()` jak pomocí restringovaného ukazatele `p`, tak i pomocí restringovaného ukazatele `q`.

Na druhé straně k objektu, který se v daném bloku nemění, lze přistupovat i prostřednictvím několika restringovaných ukazatelů. Definujeme-li například funkci

```
void h(int n, int * restrict p, int
    * restrict q, int * restrict r)
{
        int i;
        for(i = 0; i < n; i++)
                p[i] = q[i]+r[i];
}
```

je volání

```
h(100, a, b, b);       // OK
```

zcela v pořádku, pokud `a` a `b` ukazují na různá pole. Pole `b` se totiž ve funkci `h()` nemění, a proto nevadí, že k němu přistupujeme prostřednictvím dvou různých restringovaných ukazatelů.

Hodnota restringovaného ukazatele by se neměla „vynášet“ ven z bloku, v němž byl ukazatel definován. Ovšem toto pravidlo má své výjimky. Následující příklad je v pořádku:

```
typedef struct {
int n;
double * restrict v
} vektor;
```

```
vektor Novej(int n)
{
        vektor V;
        V.n = n;
        V.v = malloc(sizeof
                        (double)*n);
        return V;
}
```

Příkazy

Drobné, nikoli však bezvýznamné změny se dotkly i některých příkazů. Podívejme se na jejich přehled.

- Příkaz `if` se chová jako blok. Příkazy, které jsou jeho součástí (tj. příkaz za `if`, a případně i příkaz za `else`), představují také samostatné bloky, a to bez ohledu na to, zda jsou zapsány ve složených závorkách.
- Příkaz `switch` se chová jako samostatný blok.
- Skok pomocí příkazu `goto` nesmí vést dovnitř bloku s dynamicky modifikovaným typem (např. s polem o proměnné délce).
- Příkazy cyklu se chovají jako samostatné bloky. Také tělo cyklu představuje blok bez ohledu na to, zda je zapsáno ve složených závorkách.
- V inicializační části příkazu `for` můžeme deklarovat proměnné s paměťovou třídou `auto` nebo `register` (podobně jako v C++).

Poslední bod znamená, že jsme cyklus `for` v deklaraci funkce `h()` o několik odstavců výše mohli také zapsat takto:

```
for(int i = 0; i < n; i++)
        p[i] = q[i]+r[i];
```

Proměnná `i` je lokální v těle příkazu `for`; můžeme ji vedle inicializační části použít v podmínce, v reinicializaci a v těle tohoto příkazu. Po opuštění těla cyklu zanikne, neboť příkaz `for` představuje blok.

Pole

Možnosti deklarace pole v jazyce C nová norma podstatně rozšířila:

- Lokální pole může mít „proměnnou" délku.
- Ve specifikaci indexů pole předávaného jako parametr se může objevit modifikátor `static` nebo kvalifikátor typu.
- Specifikaci indexu pole předávaného jako parametr lze nahradit znakem `*`.

Proměnná délka pole

Pole s proměnnou délkou jsou pole, u nichž není počet prvků v deklaraci vyjádřen konstantním výrazem. Takové pole lze deklarovat pouze jako lokální nestatickou proměnnou. Řečeno slovy standardu, je-li identifikátor pole deklarován jako objekt se statickou dobou života, nesmí mít proměnné meze. Identifikátor pole s proměnnými mezemi nesmí mít definován způsob sestavování *(linkage)*. Nemůže to také být složka struktury nebo unie. (Omezení je tedy spousta; i tak je to ale užitečná novinka.)

Podívejme se na příklad. Následující konstrukce je **správná**, neboť `A` deklarujeme jako lokální automatickou proměnou:

```
void f(int n)
{
        int A[n+7];
        // ...
        int k = sizeof(A);
        // ...
}
```

V různých voláních funkce `f()` bude mít pole `A` různou délku; operátor `sizeof` v této situaci vrátí skutečnou velikost pole. (To znamená, že se bude v této situaci vyhodnocovat dynamicky, za běhu programu, nikoli v době překladu.)

Na druhé straně, není-li `m` konstanta, bude deklarace

```
extern int B[m];        // CHYBA
```

nesprávná, ať se objeví kdekoli, neboť předepisuje poli `B` externí sestavování, a to v případě pole s proměnnou délkou nelze.

Jestliže v deklaraci pole vynecháme specifikaci počtu prvků, dostaneme deklaraci neúplného typu *(incomplete type)*.

Hvězdička v indexu

Uvedeme-li místo specifikace počtu prvků znak `*`, dostaneme specifikaci pole proměnné délky s neznámým počtem prvků. Takovou deklaraci smíme použít jen v prototypu funkce. (Poznamenejme, že takovýto typ se považuje za kompletní, tedy plně definovaný.) Následující prototypy jsou podle standardu kompatibilní:

```
double Min(int m, int n, double a[m][n]);
double Min(int m, int n, double a[*][*]);
double Min(int m, int n, double a[ ][*]);
double Min(int m, int n, double a[ ][n]);
```

Modifikátor static v indexu

V deklaraci pole ve specifikaci parametrů funkce můžeme v indexových závorkách použít klíčové slovo `static`. Tím slibujeme, že při každém volání této funkce bude skutečným parametrem nenulový ukazatel na první prvek pole, jež bude mít alespoň tolik prvků, kolik je jich uvedeno v deklaraci.

- Například

```
void f(double C[static 3]);
```

je prototyp funkce, které je třeba při každém volání předat jako skutečný parametr ukazatel na první prvek pole s alespoň třemi prvky typu `double`.

Kvalifikátory typu v indexu

Parametry funkcí deklarované jako pole se i podle nové verze standardu předávají jako ukazatele na první prvek. To znamená, že deklarujeme-li jako parametr pole typu `T`, přebere si to překladač jako kvalifikovaný ukazatel na typ `T`. Přitom kvalifikátory si vezme ze specifikace indexů. To zní nejspíš nesrozumitelně, nicméně příklad nám ukáže, že nejde o nic složitého.
Následující prototypy funkcí jsou podle standardu kompatibilní:

```
void f(double (* restrict a)[5]);
void f(double a[restrict][5]);
void f(double a[restrict 3][5]);
```

První prototyp má jako parametr restringovaný ukazatel na pole o pěti prvcích typu `double`, takže skutečným parametrem budou dvourozměrná pole. Další dva prototypy mají jako parametr pole, překladač si ho ovšem přebere jako ukazatel uvedený v prvním prototypu. Podobný význam bude mít i prototyp

```
void f(double a[restrict static 3][5]);
```

který ovšem navíc slibuje, že při každém volání bude skutečným parametrem nenulový ukazatel na první z nejméně tří polí o pěti prvcích typu `double`.

Funkce

O některých změnách, které se týkají funkcí, jsme již hovořili v předchozím oddílu věnovaném polím, v první části článku jsme se také zmínili o předdefinovaném identifikátoru `__func__`.

Je zajímavé, že i nová norma stále připouští definici funkce podle Kernighana a Ritchieho, i když ji označuje za zastaralou.

Modifikátor inline

Jednou z výrazných novinek je možnost použít v deklaraci funkce modifikátor `inline`. Jeho význam je podobný jako v C++. Deklarací

```
inline void f() { /* ... */ }
```

říkáme, že volání této funkce má být co nejrychlejší. To může znamenat, že se tělo funkce dosadí na místo volání (podobně jako při rozvoji maker), standard nicméně nepředepisuje, jak má být rychlosti volání dosaženo.

Modifikátor `inline` lze použít pro jakoukoli funkci s vnitřním sestavováním *(internal linkage)*.

Pro funkce s modifikátorem `inline` platí mj. následující omezení:

- Funkce nesmí obsahovat definici statické lokální proměnné a nesmí takové proměnné používat.
- Definice funkce s modifikátorem `inline` nepředstavuje externí definici, takže v jiné samostatně překládané části programu ji lze definovat jinak.

Výpustka

Jak známo, v jazyce C můžeme deklarovat funkce s proměnným počtem parametrů; tuto skutečnost specifikujeme pomocí výpustky (`...`). Nástroje pro práci s výpustkou najdeme v hlavičkovém souboru `stdarg.h`.

Připomeňme si, že k získání parametrů předaných na místě výpustky slouží typ `va_list` a makra `va_start`, `va_arg` a `va_end`. Novinkou je makro `va_copy(`*va_list dest*, *va_list src*`)`, které umožňuje kopírovat obsah proměnné typu `va_list` do jiné proměnné téhož typu.

Preprocesor

Preprocesoru se týkají dvě novinky: možnost definovat makra s proměnným počtem parametrů a zavedení standardních direktiv `#pragma`.

Makra

V definici maker s parametry můžeme nyní specifikovat proměnný počet parametrů. Podobně jako v případě funkcí k tomu poslouží výpustka (`...`), která musí být v seznamu formálních parametrů uvedena jako poslední.

S parametry makra předanými na místě výpustky zacházíme **jako s celkem** pomocí identifikátoru `__VA_ARGS__`.

V implementaci vyhovující novému standardu jsou #definována mj. makra `__STDC_VERSION__`, `__STDC_IEC_559__` a `__STDC_ISO_10646__`.

První z nich se rozvine v konstantu tvaru `yyyymmL` (např. `199901L`), vyjadřující rok a měsíc vydání verze standardu, jemuž tato implementace odpovídá. (Tato konstanta byla zavedena v dodatku ISO/IEC 9899:1990/AMD1 jako `199409L`.) Druhé makro se rozvine v konstantu `1`, pokud implementace vyhovuje standardu IEC 60559 pro počítání s reálnými čísly. Třetí makro se rozvine v konstantu tvaru `yyyymmL` (např. `199712L`), která vyjadřuje rok a měsíc vydání standardu ISO/IEC 10646 pro UNICODE (včetně dodatků a oprav), jímž se daná implementace řídí.

Standardní #pragma

Součástí standardu je nyní několik direktiv `#pragma`. Všechny mají tvar

`#pragma STDC` *jméno přepínač*

Přitom *jméno* je některý z identifikátorů `FP_CONTRACT`, `FENV_ACCESS` nebo `CX_LIMITED_RANGE` a *přepínač* je `ON`, `OFF` nebo `DEFAULT`. S prvními dvěma *jmény* jsme se v našem povídání již setkali, třetí se týká použití komplexních čísel. Všechny zapínají (`ON`) nebo vypínají (`OFF`) určitý způsob zpracování, popř. nastavují implicitní stav (`DEFAULT`).

Práce se znaky

Vzhledem k tomu, že standard jazyka C nyní počítá nejen s „obyčejnými“, tedy jednobajtovými znaky, ale i s vícebajtovými znaky a s kódováním UNICODE („širokými“ znaky – *wide characters*), přibyly v knihovně funkce pro práci se znakovými řetězci složenými z těchto znaků. Nové jsou také konverzní funkce pro vzájemné převody různých druhů řetězců. Nástroje pro práci s vícebajtovými znaky jsou obsaženy především v hlavičkových souborech `wchar.h` a `wctype.h`.

V souboru `wchar.h` najdeme mj. funkce pro převod řetězce se širokými znaky na číslo – `wcstod()`, `wcstof()`, `wcstold()` atd., dále funkce pro kopírování řetězců se širokými znaky (`wcscpy()` a některé další), pro spojování těchto řetězců (`wcscat()` a další) atd. Pro porovnávání řetězců podle lokálních zvyklostí (přesněji podle kategorie `LC_COLLATE` v nastavení lokálních zvyklostí) slouží funkce `wcscoll()`.

V této knihovně najdeme analogie prakticky všech funkcí pro práci s „obyčejnými“ řetězci. Většina jmen těchto funkcí vznikla ze jmen analogických funkcí pro obyčejné řetězce připojením předpony `wc`; často ale muselo být původní jméno zkráceno, neboť autoři této knihovny se snažili zachovat maximálně osmiznakové identifikátory.

Pro převod jednobajtových znaků na vícebajtové slouží funkce `wint_t btowc(int c)`; poznamenejme, že `wint_t` je celočíselný typ schopný pojmout kromě znaků také hodnotu `WEOF`, která představuje „širokou“ analogii konstanty `EOF`. Obrácený převod zajistí funkce `int wctob(wint_t c)`. Pro převody mezi vícebajtovými a širokými

- znaky jsou určeny funkce `wcrtomb()` a `mbrtowc()`.

Funkce a makra pro klasifikaci širokých znaků najdeme v souboru `wctype.h`. Jde např. o funkci `iswalnum()`, jež určí, zda jde o alfanumerický znak, `towupper()`, která převede malé písmeno na velké atd.

Datové proudy

Také v práci se vstupy a výstupy pomocí datových proudů se pochopitelně odrazilo zavedení vícebajtových a širokých znaků. Každý datový proud má svoji **orientaci** (na široké nebo úzké znaky). Bezprostředně po otevření je proud bez orientace; tu získá buď při první vstupní nebo výstupní operaci, nebo voláním funkce `fwide()`.

Nástroje pro vstupy a výstupy se širokými znaky najdeme v hlavičkovém souboru `wchar.h`. Jde především o funkci `fwprintf()`, jež obstarává „širokoznakový" formátovaný výstup, a o funkci `fwscanf()`pro formátované čtení ze souboru se širokými znaky. Práce s nimi je podobná jako s jejich „úzkoznakovými" analogiemi. Dále zde najdeme funkce `swprintf()` a `swscanf()` pro zápis do širokoznakových řetězců, pro čtení z nich apod. Z názvů následujících funkcí jistě poznáte, oč jde, neboť se od svých „úzkoznakových" analogií liší jen nějakým tím `w` navíc: `fgetwc()`, `fgetws()`, `fputwc()`, `getwc()`, `putwchar()`, `ungetwc()`. (Výčet není úplný.)

Další změny

Z dalších drobných změn stojí za zmínku přemístění deklarace funkcí `sprintf()` a `sscanf()` z hlavičkového souboru `string.h` do `stdio.h`. Novinkou je také konverze `%A` (příp. `%a`), jež umožňuje výstup reálných čísel pomocí funkce `fprintf()` a podobných v šestnáctkové soustavě (s dvojkovým exponentem vyznačeným pomocí `P`, resp. `p`). Nově jsou ve specifikacích konverzí také zavedeny specifikace délky `L` (pro typ `long double`), `ll` (pro typ `long long`), `hh` (pro typ `signed char` nebo `unsigned char`), `j` (pro typ `intmax_t` nebo `uintmax_t`), `z` (pro typ `size_t`) a `t` (pro typ `ptrdiff_t`).

Pár slov na závěr

Od vydání tohoto standardu uplynuly již tři roky. Přesto se nejrozšířenější překladače jazyka C pro pécéčka – překladače, které jsou součástí vývojových prostředí pro C a C++, např. *Visual Studio .NET* nebo *Borland C++Builder 6* – k němu zatím jen blíží. Implementují nástroje pro práci se širokými znaky a některé další drobnosti, jako je modifikátor `inline` u funkcí. Neobsahují však zatím např. pole proměnné délky, pojmenované inicializátory, komplexní čísla a další možnosti.

Důvod je, domnívám se, poměrně jednoduchý. Hlavní úsilí vývojových týmů těchto nástrojů se soustředilo na shodu s novým standardem jazyka C++, takže jazyk C zůstal jaksi v pozadí.

I když byl přehled novinek ve standardu jazyka C poměrně rozsáhlý, nebyl zcela vyčerpávající. Nicméně pokud jste si z něj odnesli alespoň povšechný přehled o možnostech, které nová norma nabízí a které se bezpochyby časem objeví v nových verzích překladačů, splnil svůj účel. ■ ■ ■

Miroslav Virius, *autor@chip.cz*