C99 jaunās iespējas

Andrejs Vihrovs, av07006

2011. gada 4. janvārī

Anotācija

Programmēšanas valoda C ir viena no visvairāk izplatītām un lietotām datoru pasaulē. Tā apvieno sevī gan minimālismu un efektivitāti, gan iespēju veidot pārnesamās programmas. Vairāku desmitu gadu attīstība ir radījusi dažādas standartizētas C valodas versijas, no kurām visvairāk izplatīts ir t. s. C89 variants. Šajā referātā tiek aprakstītas jaunāka C99 varianta ieviestās iespējas, kas nav tik plaši pazīstamas.

Saturs

1. C vēsture

Programmēšanas valodas C rašanās ir cieši saistīta ar Unix operētājsistēmas izstrādi 20. gadsimta 70. gados, ko vadīja Kens Tompsons, Deniss Ričijs un Braiens Kernigans no Bell Laboratories. Unix sākumā bija uzrakstīta asemblera valodā priekš PDP-7 un PDP-11/20 platformām. Tomēr tās autori vēlējās asemblera vietā izmantot augstāka līmeņa valodu, kas padarītu programmas viegli rakstāmas un saprotamas, kā arī nepiesaistītas pie konkrētas platformas [1].

C pirmsākumi ir parādījušies B programmēšanas valodā, ko pārsvarā izveidoja Kens Tompsons, izslēdzot daudzas viņaprāt nevajadzīgas lietas no agrākās BCPL valodas, ar kuru bija saskarušies Unix izstrādātāji. BCPL valodas galvenie principi bija dot programmētājam brīvību līdzekļu un realizācijas izvēles ziņā, kā arī ļaut veidot mazus un efektīvus kompilatorus priekš šīs valodas [2]. BCPL valodā parādījās dažas jaunas lietas, kas šodien ir plaši izplatītas: figūras iekavas bloku atdalīšanai, «`//`» tipa komentāri, starpkoda ģenerēšana.

Pēc neilga laika B kļuva par pamatu Denisa Ričija izstrādātajai C programmēšanas valodai. Arī nosaukums «C» tika izvēlēts daļēji tāpēc, ka tas ir nākamais latīņu alfabēta burts pēc B. C tālāk attīstīja ideju par programmas koda neatkarību no platformas, bet saglabāja iespēju strādāt arī zemā līmenī; tika pievienota standarta (no fizisko iekārtu detaļām neatkarīga) ievada/izvada funkcionalitāte, ko pazīst kā «stdio» bibliotēku. Unix operētājsistēma tika pārrakstīta C valodā.

Vairākus gadus vēlāk C nostiprinājās kā sistēmu programmēšanas valoda. C pārnesamība ļāva bez īpašām pūlēm pārnest Unix uz jaunām platformām, kas sekmēja šīs operētājsistēmas izplatīšanos. 1978. gadā Braiens Kernigans un Deniss Ričijs uzrakstīja grāmatu «The C Programming Language» ar valodas aprakstu un programmu piemēriem, kas uz ilgu laiku kļuva par neformālu valodas standartu. Šajā grāmatā aprakstīto valodu sauc par K&R C, lai atšķirtu to no velākajām valodas versijām.

80. gados C popularitāte pieauga; šai valodai pastāvēja jau daudzi kompilatori priekš dažādām platformām. Kļuva nepieciešams izveidot vienotu valodas standartu, kas nodrošinātu dažādu C realizāciju savietojamību. To 1989. gadā izdeva American National Standards Institute (ANSI) un gadu vēlāk — International Organization for Standardization (ISO). Šis C variants ir pazīstams kā ANSI C, C89 vai C90. Šajā variantā tika ieviestas jauna tipa funkciju deklarācijas, norādes uz **`void`** un internacionalizācijas atbalsts, bet kopumā valoda saglabāja lielu līdzību K&R C variantam. C89 ir mūsdienās visizplatītākais C variants. To atbalsta vairākums izplatīto C kompilatoru, kā arī šī valodas versija ir pamats C++ valodas 1998. un 2003. gadu standartiem.

Pēc C89 standarta pieņemšanas C attīstība palēninājās uz gandrīz desmit gadiem. 1995. gadā tika pieņemts standarta papildinājums priekš labāka internacionalizācijas atbalsta, bet tas nesaturēja būtiskas izmaiņas valodā. Tikai 1999. gadā izdeva jaunu C standartu ar lielu skaitu jauninājumu [3]. Šis standarts, C99, atbrīvojas no novecojušām lietām, padara valodas sintaksi un lietojumu vairāk konsistentu, kā arī ievieš daudzas modernākas iespējas. Dažas no tām arī tiek aprakstītas šajā referātā.

2. Jauno iespēju apraksts

2.1. Būla vērtību datu tips

Sākumā C valodā nebija atsevišķa Būla vērtību datu tipa. Tā funkcijas diezgan veiksmīgi veica **`int`** vai **`char`** datu tipi. C loģiskie operatori uzskata nulli (vai nulles norādi)

par aplamu vērtību, bet jebkuru citu skaitli/norādi — par patiesu. Šādai pieejai ir priekšrocība tajā, ka vienā mainīgajā var iekodēt gan Būla informāciju, gan cita veida datus. Piemēram, funkcija `fopen` veiksmīga izsaukuma gadījumā atgriež norādi uz `FILE` mainīgo, bet neveiksmīga izsaukuma gadījumā — nulles norādi, ko var izmantot **`if`** konstrukcijā:

```
/* fopen deklarācija */
FILE *fopen(const char * restrict filename, const char * restrict mode);

FILE *f = fopen("test", "r");
if (f)
{
  /* Normāla darbība */
}
else
{
  /* Kļūdas apstrāde */
}
```

Tomēr «tīrs» Būla tips arī ir noderīgs, piemēram, programmētājam tas ļauj veidot vairāk lasāmu kodu, bet kompilatoram — potenciāli optimizēt programmu. Šāds tips, **`bool`**, tika ieviests C99 standartā. Standarta izstrādātājiem nācās risināt savietojamības problēmu: daudzas C programmas jau definēja savu **`bool`** tipu uz **`int`** vai **`char`** pamata. Lai arī vecas programmas atbilstu jaunajam standartam, tika izvēlēts šāds risinājums: jaunu primitīvo Būla datu tipu nosaukt par `_Bool` (tas var saturēt vērtības 0 un 1), bet lietotāja līmeņa definīcijas ievietot atsevišķajā iekļaujamajā datnē `<stdbool.h>`, kuras saturs būtībā atbilstu šādam kodam:

```
#define bool _Bool
#define false 0
#define true 1
```

Šādā veidā vecas programmas darbojas kā līdz šim, bet jaunajām programmām pietiek ar rindu «**`#include <stdbool.h>`**», lai iegūtu Būla datu tipa atbalstu.

2.2. `long long int` datu tips

C89 apraksta vairākus veselu skaitļu datu tipus. To intervāli nav norādīti precīzi, bet gan ir norādītas tikai minimālas absolūtās vērtības, ko datu tipa mainīgajā var saglabāt. Tas ļauj veidot C kompilatorus priekš ļoti dažādām platformām, piemēram, arī tādām, kurās veselu skaitļu datu tipiem ir izlīdzinājuma (*padding*) biti vai baits nesatur 8 bitus (C standarts prasa tikai, lai baits jeb **`char`** datu tips būtu *vismaz* 8 bitu liels). C89 veselu skaitļu datu tipu robežas ir apkopotas 1. tabulā.

C99 izstrādes laikā pastāvēja daudzas realizācijas, kas kādreiz pieņēma **`long int`** datu tipa izmēru par 32 bitiem, bet pēc tam atklāja, ka ar šo izmēru vairs nepietiek, lai, piemēram, apstrādātu lielas datnes. **`long int`** izmēru vairs nevarēja mainīt, jo tas «salauztu» pārāk daudz programmu (piemēram, 64 bitu Windows platformām pat tagad **`long int`** ir 32 bitu). Šis bija viens no iemesliem jauna **`long long int`** tipa ieviešanai, kuram minimālas prasības skaitļu intervālam ir $[-(2^{63}-1); 2^{63}-1]$. Pastāv arī attiecīgs **`unsigned long long int`** tips ar minimālo intervālu $[0; 2^{64}-1]$.

1. tabula. C89 veselu skaitļu datu tipu minimālās robežas

Tips	Maksimums pēc moduļa	Komentārs
`signed char`	$2^7 - 1$	
`unsigned char`	$2^8 - 1$	
`char`	—	Vai nu kā **`unsigned char`**, vai kā **`signed char`**
`short int`	$2^{15} - 1$	
`unsigned short int`	$2^{16} - 1$	
`int`	$2^{15} - 1$	Parasti atbilst platformas datu vārdam
`unsigned int`	$2^{16} - 1$	
`long int`	$2^{31} - 1$	
`unsigned long int`	$2^{32} - 1$	

Šiem datu tipiem var veidot attiecīgos literāļus, piemēram, «`12345LL`» priekš **`long long int`** un «`45678ULL`» priekš **`unsigned long long int`**. Darbā ar `printf` un `scanf` var izmantot «`ll`» garuma modifikatoru, lai strādātu ar **`long long int`** tipa vērtībām.

2.3. «`//`» komentāri

C89 aprakstīja tikai vienu komentāru veidu — «`/* … */`» tipa komentārus. C99 atļāva lietot arī t. s. vienas rindas komentārus, kas sākas ar «`//`» un turpinās līdz rindas beigām. Šādus komentārus tajā laikā jau lietoja C++ un citas valodas.

2.4. Deklarāciju un koda kopējā lietošana

K&R C un C89 valodas variantos vienā blokā bija sākumā jādefinē visi mainīgie, bet pēc tam jāraksta pirmkods. Tas saglabājās no vēl senāku valodu sintakses un neļāva lietot pieeju «mainīgo definē tieši pirms lietošanas», kas var uzlabot pirmkoda lasāmību. C99 atcela šo ierobežojumu un atļāva lietot deklarācijas konstrukcijas kopā ar pirmkodu.

2.5. Netiešā `int` tipa un vecu funkciju deklarāciju aizliegums

K&R C un C89 varēja definēt mainīgos bez tiešas datu tipa norādīšanas. Šajā gadījumā tika pieņemts, ka mainīgā datu tips ir **`int`**. Tas pats attiecās arī uz funkciju atgriežamajām vērtībām. Piemēram, visās sekojošajās vietās tiek lietots netiešs **`int`** datu tips:

```
/* int */ f();

/* int */ g()
{
  auto /* int */ i = 42;
  return i;
}
```

C99 aizliedz lietot netiešo **int**, jo tā lietošana pasliktina pirmkoda lasāmību un ievieš «speciālgadījumu» deklarāciju sintaksē.

Funkciju deklarācijas K&R C variantā arī atšķirās no mūsdienu pieraksta. Piemēram, funkcijas ar diviem parametriem deklarācija un definīcija izskatījās šādi:

```
add();

add(a, b)
int a, b;
{
  return a + b;
}
```

Ievērosim, ka funkcijas deklarācija nesatur nekādu informāciju par parametru skaitu vai to tipiem. Tas viegli varēja novest pie programmētāja kļūdām. Tāpēc C89 un pēc tam C99 pieņem jaunu deklarāciju sintaksi, kas jau ļauj veikt datu tipu kontroli:

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

int add(int a, int b)
{
  return a + b;
}
```

2.6. Mainīgā garuma masīvi

Vēsturiski masīva garumam C valodā bija jābūt konstantai izteiksmei. Situācijās, kad bija nepieciešams no, piemēram, funkcijas parametra atkarīgs atmiņas apjoms, bija jālieto `malloc` un `free` funkcijas. Tomēr `malloc` izsaukums var būt lēnāks, nekā parastā (ar *automatic storage duration*) mainīgā izveide, tāpēc šāda pieeja var nebūt optimāla masīviem ar mazu izmēru.

Šo problēmu risina mainīgā garuma masīvi — C99 funkcionalitāte, kas ļauj kā masīva garumu norādīt jebkuru izteiksmi, kas rezultātā dod veselu skaitli. Tas ir lielā mērā līdzīgs nestandarta `alloca` funkcijai. Piemēram, sekojošā funkcija izveido tik lielu **char** masīvu, cik liels ir tās parametrs `n`. Funkcijai beidzoties, atmiņa tiek automātiski atbrīvota, kā jebkuram citam mainīgajam ar *automatic storage duration*.

```
void foo(size_t n)
{
  char arr[n];

  /* Strādā ar arr */
}
```

Mainīgā garuma masīviem ir arī trūkumi: programmas uzvedība nav definēta, ja atmiņas nepietiek. `malloc` kļūdas gadījumā atgriež `NULL` un programma šo situāciju var apstrādāt; mainīgā garuma masīva gadījumā tādas iespējas nav. Tāpēc mainīgā garuma masīvi būtu jālieto tikai nelieliem datu apjomiem.

2.7. Makrosi ar mainīgo parametru skaitu

Tradicionāli C makrosi varēja saņemt noteiktu skaitu parametru (vai arī nevienu). Piemēram, «**#define** `FOO bar`» definē makrosu bez parametriem, bet «**#define** `MERGE(a, b) a ## b`» definē makrosu ar diviem parametriem. C99 piedāvā jaunu iespēju — makrosus ar mainīgo parametru skaitu. Tie ir ērti gadījumos, kad makross izsauc funkciju ar mainīgo parametru skaitu.

Makrosu ar mainīgo parametru skaitu definē, pievienojot «`...`» pēc makrosa parametru identifikatoriem (ja tādi ir). Tad makrosa tekstā var lietot īpašu identifikatoru `__VA_ARGS__`, kas atbilst visiem ar «`...`» saistītiem parametriem. Piemēram definēsim `DEBUG` makrosu, kas palīdz izvadīt atkļūdošanas informāciju:

```
#define DEBUG(...) fprintf(stderr, __VA_ARGS__)

/* Lietošanas piemēri */
DEBUG("test␣message\n");
DEBUG("another␣message␣with␣a␣number:␣%d\n", 42);
```

2.8. «inline» funkcijas

Ilgu laiku C makrosi kalpoja kā veids vairākas reizes izmantot to pašu pirmkodu, neveidojot funkciju izsaukumus, kas parasti ir salīdzinoši lēni. Tomēr makrosiem ir vairākas problēmas: (a) nav datu tipu kontroles to parametriem; (b) ja parametra vērtība ir izteiksme, kurai ir blakus efekts, tad neuzmanīgi uzrakstīts makross var atkārtot šo blakus efektu vairākas reizes; (c) makrosos ir grūti lietot sarežģītas programmas vadības konstrukcijas. Klasisks piemērs b punktam ir `MAX` makross un tā izsaukums:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int i = 42, j;
j = MAX(i++, 0); /* Tagad i ir 44, nevis 43 */
```

C99 ievieš «**inline**» funkcijas, kas vairākumā gadījumu var aizvietot makrosus un parasti ir tikpat efektīvas. **inline** ir specifikators, ko lieto funkcijas deklarācijā. C standarts norāda, ka «**inline**» funkciju izsaukumiem jābūt tik ātriem, cik tas ir iespējams. Praksē tas parasti nozīmē, ka funkcijas izsaukums nenotiek un funkcijas kods tiek ievietots tās izsaukuma vietā. Ja funkcija turklāt deklarēta ar **static** specifikatoru, kas nozīmē iekšējo saistīšanu (*internal linkage*), tad rezultējošajā programmā šī funkcija vispār var neparādīties kā atsevišķs simbols ar vārdu un to var lietot pat iekļaujamajās datnēs. Kā piemēru pārrakstīsim `MAX` makrosu ar «**inline**» funkcijas palīdzību:

```
static inline int max(int a, int b)
{
  return a > b ? a : b;
}
```

2.9. `__func__` identifikators

Ir plaši pazīstami `__FILE__` un `__LINE__` makrosi, kas ir vienmēr definēti un satur, attiecīgi, pirmkoda datnes nosaukumu un rindas numuru, kurā šie makrosi tiek izvērsti. Tos var izmantot atkļūdošanas vai diagnostikas paziņojumiem, kā, piemēram, šajā `assert` definīcijā:

```
#define assert(e) ((e) ? (void)0 : (fprintf(stderr, __FILE__ ":%d:␣
    assertion␣`" #e "'␣failed.\n", __LINE__), abort()))
```

C99 standarts piedāvā arī iespēju iegūt tagadējās funkcijas nosaukumu. Tas tiek panākts ar `__func__` identifikatoru (nevis makrosu!), kas ir netieši definēts katrā funkcijā tā, it kā funkcijas sākumā būtu rakstīts

```
static const char __func__[] = "funkcijas nosaukums";
```

Tātad, ar `__func__` ir jāstrādā tāpat, kā ar parastu C simbolu virkni. Iepriekš minēto `assert` realizāciju varētu izmainīt sekojošā veidā:

```
#define assert(e) ((e) ? (void)0 : (fprintf(stderr, __FILE__ ":%s:%d:␣
    assertion␣`" #e "'␣failed.\n", __func__, __LINE__), abort()))
```

2.10. Inicializācija pēc vārdiem un saliktie literāļi

C masīvus un struktūras var inicializēt ar «`= { a, b, c }`» sintaksi. Kas ir jādara, ja programmētājs vēlas inicializēt tikai dažus saliktā tipa elementus vai arī lietot citu inicializācijas kārtību? C99 risina problēmu ar inicializāciju pēc vārdiem, funkcionalitāti, kas ļauj norādīt inicializējamā elementa vārdu. Struktūrām un apvienojuma tipiem inicializācijas sarakstā var norādīt «*`.elementa vārds = vērtība`*», bet masīviem — «`[`*`indekss`*`] = `*`vērtība`*». Elementi, kuru vārdi nav tikuši norādīti inicializācijas sarakstā, iegūst noklusēto vērtību. Tālāk parādīti daži piemēri inicializācijai pēc vārdiem:

```
struct bar
{
  int a, b, c;
} x = { .c = 42, .a = 0 };

int arr[3] = { [2] = 3, [1] = 2, [0] = 1 };
int arr2[2][2] = {
  [0][0] = 0, [0][1] = 1,
  [1][0] = 2, [1][1] = 3 };
```

Ar inicializāciju ir saistīti t. s. saliktie literāļi. Dažreiz ir nepieciešams norādīt struktūras, apvienojuma tipa vai masīva tipa konstanti, kas nekur citur netiks izmantota. Var definēt atsevišķo mainīgo un inicializēt to. Bet ievērosim, ka, lai padotu funkcijai `f` skaitli 42, mēs nerakstām `int tmp = 42; f(tmp);`, bet gan `f(42);`. Šajā gadījumā `42` ir skaitļu literāls. C99 ļauj izveidot šādus literāļus arī priekš saliktajiem datu tipiem. To sintaktiskais pieraksts ir «*`(tips){inicializācijas saraksts}`*». Piemēram var attēlot darbu ar `point` struktūru un masīvu literāļiem:

```
struct point
{
  int x, y;
};

void foo(struct point p);

void bar(void)
{
  foo((struct point){ 1, 2 });

  int *arr;
  arr = (int []){ 3, 2, 1 };
}
```

Ar saliktiem literāļiem var lietot arī inicializāciju pēc vārdiem.

2.11. Paplašināta veselu skaitļu datu tipu bibliotēka

Kā jau tika minēts 2.2. nodaļā, iebūvētie C veselu skaitļu datu tipi nav precīzi noteikti — ir norādīti tikai minimāli skaitļu intervāli, ko šie tipi spēj glabāt. Tāpat nav zināms šo tipu mainīgo izmērs atmiņā un ātrdarbība. Piemēram, datu tipi ar fiksētu, iepriekš zināmu izmēru ir noderīgi zema līmeņa programmēšanā, bināro datņu vai tīkla pakešu kodēšanā.

C99 piedāvā jaunu iekļaujamo datni `<stdint.h>`, kas definē papildus veselu skaitļu datu tipus piecās kategorijās:

1. Datu tipi ar fiksētu garumu. Tie ir formā `int`*N*`_t` un `uint`*N*`_t`, kur N ir bitu skaits. Neviens no šiem datu tipiem nav garantēti pieejams, tomēr, ja realizācijai ir pieejami datu tipi 8, 16, 32 vai 64 bitu garumā bez izlīdzinājuma bitiem *two's complement* reprezentācijā, tad tai ir jādefinē datu tipi `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`.

2. Vismazākie datu tipi ar vismaz noteiktu garumu, kuriem ir forma `int_least`*N*`_t` un `uint_least`*N*`_t`. Tie satur vismaz N bitus. Ir pieejami vismaz šādi tipi priekš $N = 8, 16, 32, 64$.

3. Visātrākie datu tipi ar vismaz noteiktu garumu. Tiem ir forma `int_fast`*N*`_t` un `uint_fast`*N*`_t`. Tiem būtu jāatbilst visātrākiem no primitīviem datu tipiem, kas satur vismaz N bitus. Tāpat kā iepriekšējā gadījumā, ir garantēti pieejami vismaz šādi tipi priekš $N = 8, 16, 32, 64$.

4. Datu tipi, kas ir pietiekami lieli, lai saglabātu objekta norādes vērtību: `intptr_t` un `uintptr_t`. Šie tipi var būt noderīgi atkļūdošanā vai *hash*-tabulās.

5. Vislielākie iespējamie datu tipi atbist primitīviem datu tipiem, kas spēj saglabāt jebkura veselu skaitļu datu tipa vērtību, attiecīgi, ar zīmi vai bez zīmes: `intmax_t` un `uintmax_t`.

`<stdint.h>` un `<inttypes.h>` iekļaujamās datnes definē arī palīglīdzekļus darbam ar aprakstītiem tipiem: minimālas un maksimālas vērtības, makrosus attiecīgā tipa literāļu veidošanai, dažas matemātiskas funkcijas, kā arī makrosus darbam ar `printf` un `scanf` funkciju saimēm.

2.12. «restrict» norādes

Daudzās potenciālās programmu optimizācijās, tādās, kā vērtības saglabāšana procesora reģistrā atkārtotai lietošanai vai ciklu paralēlā izpilde, ir svarīgas programmas mainīgo atkarības, it īpaši gadījumos, ja mainīgie ir norādes. Daudzkārt, ja kompilators zina, ka dati, uz kuriem rāda norāde, netiks izmantoti citās vietās, tas var ģenerēt labāku kodu. Tomēr kompilatoram pašam ir grūti vai pat neiespējami noteikt kompilācijas laikā, vai, piemēram, funkcijā `int foo(int *a, int *b)` parametri `a` un `b` rādīs uz dažādiem vai tiem pašiem apgabaliem atmiņā.

Kā piemēru var apskatīt sekojošu funkciju:

```
int foo(int *a, int *b)
{
  *a += *b;
  return *b;
}
```

Kompilators šai funkcijai varētu uzģenerēt šādu asemblera kodu (pieņemsim, ka procesoram ir reģistri R0–R3, parametri `a` un `b` tiek padoti reģistros R0 un R1, bet funkcijas rezultāts tiek saglabāts reģistrā R0):

```
load R2, R0
load R3, R1
add R2, R2, R3
store R2, R0
load R0, R1 /* *b ielādē otru reizi, jo, varbūt, a rādīja uz to pašu vietu */
ret
```

Ja kompilators zinātu, ka `*a` izmaiņa neietekmē `*b`, tad pēdējā `load` instrukcija kļūtu nevajadzīga. C99 patiešām ļauj to kompilatoram pateikt, izmantojot **`restrict`** kvalifikatoru. Ja norāde `p` ir deklarēta ar **`restrict`**, tas nozīmē, ka tikai uz `p` bāzētas izteiksmes un vērtības (tādas kā `p`, `p + 1`, utt.) tiks izmantotas piekļūvei datiem, uz kuriem rāda `p`, bet ne citas izteksmes (tādas kā `q`, kur `q` ir cita norāde). Ar **`restrict`** funkcijas `foo` deklarācija būtu

```
int foo(int * restrict a, int * restrict b);
```

Ir jāatceras, ka **`restrict`** ir tikai programmētāja padoms kompilatoram, kas nekādā veidā netiek pārbaudīts. Ja **`restrict`** tiek izmantots datu apgabaliem, kas pārklājas, tad programmas uzvedība nav definēta.

`restrict` tiek lietots daudzās standarta C un POSIX funkciju deklarācijās. Plaši pazīstama ir `memcpy` standarta funkcija:

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
```

Šī ir viena no funkcijām, kur **`restrict`** parasti ļauj kompilatoram veidot efektīvāku kodu. Bet, saskaņā ar **`restrict`** definīciju, `s1` un `s2` nedrīkst pārklāties. Šim gadījumam ir domāta (varbūt lēnāka) funkcija `memmove`.

3. Ieskats C nākotnē

Lai gan mūsdienās popularitāti iegūst augstāka līmeņa valodas, C joprojām paliek labs un efektīvs rīks noteiktu uzdevumu risināšanai, piemēram, iegulto iekārtu programmatūras izstrādei. Turpinās arī C attīstība. 2007. gadā tika uzsākts darbs pie nākamā C standarta, ko neoficiāli dēvē par C1X [4]. Starp citiem uzlabojumiem C1X piedāvās arī tādas lietas, kā lielākas optimizācijas iespējas, labāks Unicode atbalsts un paralēlā skaitļošana.

Literatūras saraksts

[1] **Ritchie, D.** The Development of the C Language. Pieejams http://cm.bell-labs.com/cm/cs/who/dmr/chist.html, pārbaudīts 03.01.2011.

[2] **Richards, M. and Whitby-Strevens, C.** BCPL—the language and its compiler. Cambridge, UK: Cambridge University Press, 1980, 173 p.

[3] ISO/IEC 9899:TC3. Programming languages—C. Pieejams http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf, pārbaudīts 03.01.2011.

[4] ISO/IEC 9899:201x. Programming languages—C. Pieejams http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1516.pdf, pārbaudīts 03.01.2011.