Artūrs Verza

Veselu skaitļu datu tipi

Ievads

Šis dokuments satur īsu aprakstu un ieteikumus par veselu skaitļu datu tipu lietošanu programmēšanas olimpiādēs. Galvenokārt tiek apskatīta 64 bitu mainīgo lietošana valodās Pascal un C++ (kompilatori FreePascal, GNU C++ un Microsoft Visual C++).

Pārskats par veselu skaitļu datu tipiem

Aplūkosim pieejamos veselu skaitļu datu tipus.

Baiti	Intervāls	Pascal	C++
1	-128..127	shortint	char
1	0..255	byte	unsigned char
2	-32768..32767	int	short int
2	0..65535	word	unsigned short
4	-2147483648..2147483647	longint	int
4	0.. 4294967295	longword	unsigned int
8	$-2^{63}..2^{63}-1$ (aptuveni $-9*10^{18}..9*10^{18}$)	int64	long long vai __int64
8	$0..2^{64}-1$ (aptuveni $1,8*10^{18}$)	qword	unsigned long long

Jāņem vērā, ka C++ dažādiem kompilatoriem intervāli var atšķirties. Norādītie intervāli ir Microsoft Visual C++ 2005/2008 un GNU C++ 3.4.2 kompilatoriem.

Kurš datu tips ir vislabākais?

Risinot olimpiādes uzdevumus, parasti vislabāk ir izmantot 32-bitu mainīgos ar zīmi (longint valodā Pascal un int valodā C++). Tam ir vairāki iemesli:

1. Ļoti bieži uzdevumu ierobežojumi ir veidoti tā, lai uzdevumu varētu atrisināt, izmantojot tieši 32 bitu mainīgos.
2. Lietojot mazākus tipus, pastāv ļoti liels risks, ka mainīgo vērtības var pārsniegt to maksimālās pieļaujamās vērtības un var rasties kļūdas.
3. Operācijas ar 32 bitu mainīgiem parasti strādā visātrāk (skat. punktu Ātrdarbība).

Kad ir vērts lietot mazus mainīgos?

Izmantot mazākus mainīgos var būt vērts tad, ja citādi pieejamais operatīvās atmiņas apjoms ir nepietiekošs. Nav ieteicams speciāli taupīt atmiņu šim nolūkam. Iegūto punktu skaitu praktiski vienmēr nosaka vienīgi risinājuma efektivitāte. Sīkas optimizācijas ļoti retos gadījumos var nedaudz palielināt punktu skaitu, bet visbiežāk notiek pretējs efekts, jo optimizāciju rezultātā programmā ieviešas kļūdas.

64 bitu mainīgie

Kopš ir kļuvuši pieejami 64 bitu mainīgie, kas spēj glabāt krietni lielākas vērtības, arvien biežāk uzdevumu ierobežojumi tiek veidoti tā, lai uzdevumu varētu atrisināt, izmantojot 64 bitu mainīgos. Par laimi, to lietošana praktiski neatšķiras no citu tipu lietošanas. Aplūkosim nelielu piemēru valodā C++ (*piemers1.cpp* un *piemers2.cpp*). Analoģisku piemēru valodā Pascal skat. failos *piemers1.pas* un *piemers2.pas*.

```
// piemers1.cpp
#include <cmath>
#include <iostream>
int main() {
	int a,b,c,min,max;

	printf("Ievadiet skaitli a: "); scanf("%d", &a);
	printf("Ievadiet skaitli b: "); scanf("%d", &b);
	printf("a + b = %d\n", a + b);
	printf("a * b = %d\n", a * b);
	printf("a %% b = %d\n", a % b);

	c = 87658765;
	printf("c = %d\n", c);
	printf("|c| = %d\n", abs(c));
	printf("sqrt(c) = %d\n", (int) sqrt((double) c));

	min = INT_MIN;
	max = INT_MAX;
	printf("mazakais = %d\n", min);
	printf("lielakais = %d\n", max);

	for (int i = c; i < c + 3; i++) {
		printf("%d %% 10^6 = %d\n", i, i % 1000000);
	}

	return 0;
}

// piemers2.cpp
#include <cmath>
#include <iostream>
int main() {
	long long a,b,c,min,max;

	printf("Ievadiet skaitli a: "); scanf("%lld", &a);
	printf("Ievadiet skaitli b: "); scanf("%lld", &b);
	printf("a + b = %lld\n", a + b);
	printf("a * b = %lld\n", a * b);
	printf("a %% b = %lld\n", a % b);

	c = 8765876587658765876;
	printf("c = %lld\n", c);
	printf("|c| = %lld\n", (long long) abs((double)c));
	printf("|-c| = %lld\n", (long long) abs((double)(-c)));
	printf("sqrt(c) = %lld\n", (long long) sqrt((double) c));

	min = LLONG_MIN;
	max = LLONG_MAX;
	printf("mazakais = %lld\n", min);
	printf("lielakais = %lld\n", max);

	for (long long i = c; i < c + 3; i++) {
		printf("%lld %% 10^13 = %lld\n", i, i % 10000000000000);
	}

	return 0;
}
```

Pirmās programmas izpildes variants:

```
Ievadiet skaitli a: 123456
Ievadiet skaitli b: 987
a + b = 124443
a * b = 121851072
a % b = 81
c = 87658765
|c| = 87658765
sqrt(c) = 9362
mazakais = -2147483648
lielakais = 2147483647
87658765 % 10^6 = 658765
87658766 % 10^6 = 658766
87658767 % 10^6 = 658767
```

Otrās programmas izpildes variants:

```
Ievadiet skaitli a: 1234567890123
Ievadiet skaitli b: 4567
a + b = 1234567894690
a * b = 5638271554191741
a % b = 4356
c = 876587658765876
|c| = 876587658765876336
|-c| = 876587658765876336
sqrt(c) = 2960722308
mazakais = -9223372036854775808
lielakais = 9223372036854775807
876587658765876 % 10^13 = 6587658765876
876587658765877 % 10^13 = 6587658765877
876587658765878 % 10^13 = 6587658765878
```

Atšķirības, lietojot long long datu tipu:

- ielasot ar scanf vai izvadot ar printf, jālieto %lld (GNU C++ kompilatora gadījumā – %I64d).
- vismazākais pieļaujamais skaitlis – LLONG_MIN
- vislielākais pieļaujamais skaitlis – LLONG_MAX
- funkcijas abs un sqrt nav paredzētas lietošanai 64 bitu mainīgajiem. Pārveidojot vērtības uz tipu double un atpakaļ, var rasties informācijas zudumi. Redzam, ka abs funkcija lieliem skaitļiem strādā nepareizi. Iespējams, ka arī funkcija sqrt ne vienmēr strādā pareizi.

Jāņem vērā, ka nedrīkst rakstīt šādi:

```
int i = 2000000000;
long long x = i * i;
```

vai pat vienkārši:

```
long long x = 1000000 * 2000000;
```

i vērtība tiek reizināta kā 32 bitu, tāpēc notiek pārpildīšanās. Vismaz viena no vērtībām jāpārveido uz 64 bitu. Visi šie varianti ir pareizi:

```
long long x = (long long) i * i;
long long x = i *(long long) i;
long long x = (long long) i * (long long) i;
long long x = 1000000LL * 2000000;
long long x = 1000000 * 2000000LL;
long long x = 1000000LL * 2000000LL;
```

Aplūkosim vēl vienu piemēru:

```
int i = 1000000;
int j = 2000000;
// nepareizi! long long x = i * i + 2 * i * j + j * j;
// nepareizi! long long x = (long long) i * i + 2 * i * j + j * j;
// pareizi, bet gari! long long x = (long long) i * i + (long long) 2
* i * j + (long long) j * j;
// pareizi, nedaudz īsāk! long long x = (long long) i * i + 2LL * i *
j + (long long) j * j;

// šādi ir vismazāk iespēju nokļūdīties
long long li = i;
long long lj = j;
long long x = li * li + 2 * li * lj + lj * lj;
```

Atšķirības, lietojot int64 datu tipu (Pascal)

Funkcijām abs un sqrt nesakritības netika novēroras, tāpēc iespējams, ka tās strādā pareizi.

Nedrīkst rakstīt šādi (ja i – longint mainīgais, bet x – int64 mainīgais):

```
i := 2000000000;
x := i * i;
```

Arī te vismaz viena no vērtībām jāpārveido uz 64 bitu. Visi šie varianti ir pareizi:

```
x := int64(i) * i;
x := i * int64(i);
x := int64(i) * int64(i);
x := 1000000 * 2000000;
x := 2000000000000;
```

Cikla skaitītājs nevar būt int64 tipa. Piemēram, ja x – int64 mainīgais, nedrīkst rakstīt:

```
for x := c to c + 2 ...
```

Šajā gadījumā var rakstīt tā:

```
for i := 0 to 2 do begin
  x := c + i;
  ...
end;
```

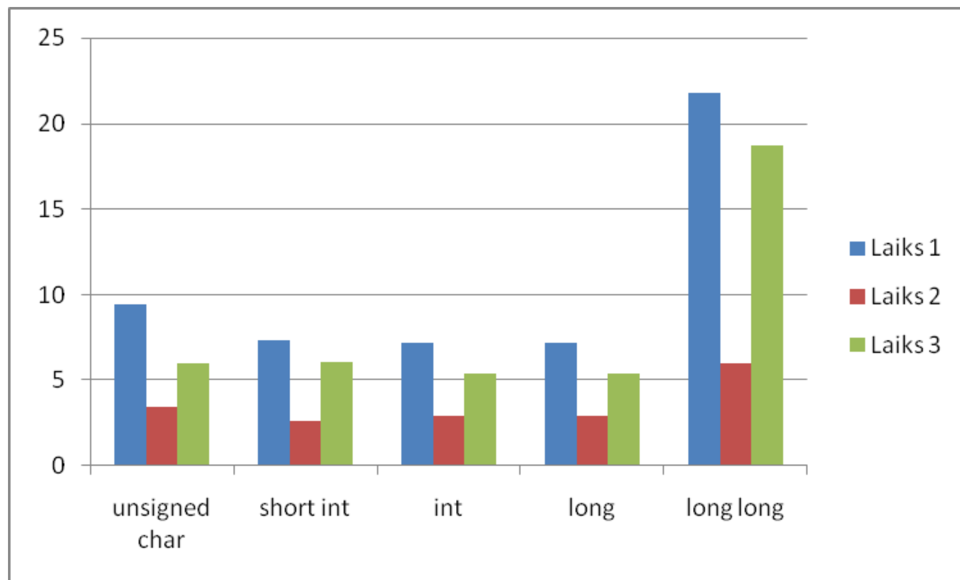
Ātrdarbība (C++)

Ātrdarbības pārbaudei tika izmantota programma valodā C++ (*atrd.cpp*), kas veic šādus aprēķinus:

1) aizpilda masīvu ar izmēriem 255×255×255 ar rēķinātām vērtībām;
2) sakārto visu masīvu;
3) veic aprēķinus ar masīva vērtībām.

Visi skaitļi pēc vērtības nepārsniedz 255. Katrai no 3 daļām tiek mērīts izpildes laiks.

Programma tika izpildīta ar dažādiem datu tipiem (visus int failā aizvietojot ar short int, long utt.). Lūk, rezultāti:



Redzam, ka long long strādā ievērojami lēnāk, bet pārējie datu tipi – aptuveni vienādi.