Programovacie rozhranie

Obsah

1 Základné triedy - Okruhy

- `ZZ`: veľké celé čísla
- `ZZ_p`: veľké celé čísla modulo p
- `zz_p`: celé čísla mod p s jednoduchou presnosťou (single precision)
- `GF2`: celé čísla mod 2
- `ZZX`: polynómy s jednou premennou ZZ
- `ZZ_pX`: polynómy s jednou premennou ZZ_p
- `zz_pX`: polynómy s jednou premennou zz_p
- `GF2X`: polynómy GF2
- `ZZ_pE`: rozšírenie poľa/okruhu ZZ_p
- `zz_pE`: rozšírenie poľa/okruhu zz_p
- `GF2E`: rozšírenie poľa/okruhu GF2
- `ZZ_pEX`: polynómy s jednou premennou ZZ_pE
- `zz_pEX`: polynómy s jednou premennou zz_pE
- `GF2EX`: polynómy s jednou premennou GF2E

Všetky triedy podporujú **základné aritmetické operátory**:

$$+, -, -(\text{unárny}), +=, -=, ++, --,$$
$$*, *=, /, /=, \%, \%= .$$

Avšak operácie

$$\%, \%=$$

sa dajú použiť len pre triedy celých čísel a polynómov a neexistujú pre triedy

$$ZZ\_p, zz\_p, GF2, ZZ\_pE, zz\_pE, GF2E.$$

Štandardné **operátory rovnosti** (== a !=) sa dajú použiť pre každú triedu. Trieda `ZZ` podporuje aj obvyklé operátory nerovnosti. Triedy celých čísel a polynómov tiež podporujú "shift" operátory pre posun doľava a doprava. Pre triedy polynómov to znamená násobenie alebo delenie mocninou x.

2 Triedy s pohyblivou desatinnou čiarkou

NTL poskytuje tri rôzne triedy:

- `xdouble`: čísla s dvojitou presnosťou (double precision) s pohyblivou desatinnou čiarkou s rozšíreným rozsahom exponentu (pre veľmi veľké čísla)
- `quad_float`: čísla s štvornásobnou presnosťou (quadruple precision) s pohyblivou desatinnou čiarkou
- `RR`: čísla s voliteľnou presnosťou s pohyblivou desatinnou čiarkou

3 Vektory a matice

Vektory a matice nad

$$ZZ, ZZ\_p, zz\_p, GF2, ZZ\_pE, zz\_pE, GF2E, RR$$

podporujú bežné aritmetické operácie.

4 Funkcionálna a procedurálna forma

Všeobecne pre každú funkciu definovanú v NTL existuje jej funkcionálna a procedurálne forma. Napríklad: `ZZ x, a, n;`

```
x = InvMod(a, n); // functional form
InvMod(x, a, n); // procedural form
```

Príklad ilustroval syntaktickú odlišnosť týchto dvoch foriem. Avšak existujú **výnimky**. Prvá, keď existuje **operátor** ktorý nahradí funkciu.

```
ZZ x, a, b;
x = a + b; // functional form
add(x, a, b); // procedural form
```

Druhá, keď meno funkcie vo funkcionálnej forme je nedostačujúce a je potrebné pridať **návratový typ** za toto meno funkcie.

```
ZZ_p x;
x = random_ZZ_p(); // functional form
random(x); // procedural form
```

Po tretie, je veľa **funkcií konverzie**, ktorých meno v procedurálnej forme je `conv`, ale vo funkcionálnej `to_T`, kde T predstavuje návratový typ.

```
ZZ x;
double a;
```

```
x = to_ZZ(a); // functional form
conv(x, a); // procedural form
```

Použitie procedurálnej formy môže byť účinnejšie, pretože nevytvorí dočasný objekt na uloženie výsledku. Väčšinou je ale preferovaná funkcionálna forma pre jej lepšie porozumenie v kóde. Vyššie uvedené pravidlá platia pre všetky aritmetické triedy podporované NTL, s výnimkou `xdouble` a `quad_float`. Tieto dve triedy podporujú iba funkcionálny zápis alebo zápis pomocou operátorov (ale podporujú obe formy pre konverziu).

5 Typové konverzie

NTL **neposkytuje automatické konverzie** napríklad z `int` na `ZZ`. Viacerí C++ experti posudzujú automatické konverzie za zlý návrh knižnice. Niektoré predchádzajúce knižnice NTL podporovali automatickú konverziu, ktorá ale spôsobovala veľa problémov. Ďalším problémom je kombinácia preťaženia funkcie a automatickej konverzie, pretože je ťažké rozhodnúť ktorá by mala byť skor volaná. C++ má komplexné pravidlá pre takéto prípady, ktoré sa postupne vyvíjali a žiadne dva prekladače neimplementujú rovnaké pravidlá. Pre tieto dôvody nie je implementovaná napr. `int` na `ZZ` konverzná funkcia ako `ZZ` konštruktor, ktorý vezme argument typu `int`, namiesto volania `to_ZZ`. Toto by spôsobilo automatickú konverziu, ktorej sa chceme vyhnúť. Konverzia taktiež nie je riešená explicitným konštruktorom z dôvodu, že táto vlastnosť nie je všeobecne dostupná.

Ako bolo spomenuté vyššie, je prístupných mnoho **nástrojov na explicitnú konverzi**u vo funkcionálnej aj procedurálnej forme. Zoznam je prístupný v súbore conversions.txt.
Napriek tomu, že neexistujú automatické konverzie, používatelia NTL neprídu o ich benefity, pretože všetky základné aritmetické operácie (vo funkcionálnej aj procedurálnej forme), operátory porovnania a priradenia sú **preťažené**.

```
ZZ x, a;
x = a + 1;
if (x < 0)
mul(x, 2, a);
else
x = -1;
```

Tieto "povýšenia" (promotions) sú dokumentované v .txt súboroch, väščinou s krátkou poznámkou. Napr.

```
ZZ operator+(const ZZ& a, const ZZ& b);
```

```
// PROMOTIONS: operator + promotes long to ZZ on (a, b).
```

To znamená, že popri deklarácii funkcie, existujú ďalšie dve funkcie, ktoré sú logicky ekvivalentné:

```
ZZ operator+(long a, const ZZ& b)  return to_ZZ(a) + b;
ZZ operator+(const ZZ& a, long b)  return a + to_ZZ(b);
```

Všimnite si, že NTL takto v skutočnosti neimplementuje funkcie. Všeobecne je efektívnejšie napísať

```
x = y + 2;
```

ako napísať

```
x = y + to_ZZ(2);
```

Prvý príklad nevyžaduje vytvorenie a zničenie dočasného `ZZ` objektu pre uloženie hodnoty 2.
Nebojte sa napísať testy ako

```
if (x == 0) ...
```

a priradenia

```
x = 1;
```

Všetky sú optimalizované. Nespúšťajte podstatne pomalšie

```
if (IsZero(x)) ...
```

a

```
set(x);
```

Pre niektoré typy existuje viac "povýšení". Napr. typ `ZZ_pX` má povýšenie aj z `long` a `ZZ_p`. Preto funkcia `add` pre `ZZ_pX` funguje pre nasledovné typy argumentov:

```
(ZZ_pX, ZZ_pX), (ZZ_pX, ZZ_p), (ZZ_pX, long), (ZZ_p, ZZ_pX),
(long, ZZ_pX)
```

Každá z týchto funkcií efektívne konvertuje argument a povyšuje ho na `ZZ_pX`. Všimnite si, že pri povýšení dvoch argumentov, aspoň jeden musí byť takého typu, ako je cieľ. V nasledujúcej tabuľke sa nachádza zoznam typov, ktoré sú **automaticky povýšené**.

Popis všetkých povýšení sa nachádza v dokumentácii, priblížime si len niektoré základné pravidlá:

- Povýšenia platia všeobecne pre procedurálnu aj funkcionálnu formu, ako aj pre zodpovedajúci operátor.
  ```
  x = x + 2;
  add(x, x, 2);
  x += 2;
  ```

- Pri sčítaní, odčítaní, násobení, rovnosti a porovnaní budú vždy povýšené oba argumenty.
  ```
  x = 2 + y;
  ```

cieľ:	zdroj
xdouble:	double
quad_float:	double
RR:	double
ZZ:	long
ZZ_p:	long
ZZ_pX:	long, ZZ_p
zz_p:	long
ZZ_pX:	long, zz_p
ZZX:	long, ZZ
GF2:	long
GF2X:	long, GF2
GF2E:	long, GF2
GF2EX:	long, GF2, GF2E
ZZ_pE:	long, ZZ_p
ZZ_pEX:	long, ZZ_p, ZZ_pE
zz_pE:	long, zz_p
zz_pEX:	long, zz_p, zz_pE

```
add(x, 2, y);
if (3 > x || y == 5) ...
```

- Operátor priradenia vždy povýši pravú stranu.
  ```
  x = 2;
  ```

- Ak sa nejedná o celé čísla alebo polynómy, pri delení sú povýšené oba argumenty.
  ```
  RR x, y, z;
  ...
  x = 1.0/y;
  z = y/2.0;
  ```
  Pri celých číslach a polynómoch je pri delení povýšený len menovateľ.
  ```
  ZZ x, y;
  ...
  y = x/2;
  ```

- Pri násobení matice so skalárom alebo vektora so skalárom je povýšený len skalár.
  ```
  vec_ZZ v, w;
  ...
  ```

```
v = w*2;
v = 2*w;
v *= 2;
```

- Konštruktor jednočlena pre polynómy a príslušná funkcia `SetCoeff` povýši argument koeficienta.
  ```
  ZZX f;
  f = ZZX(3, 5); // f == 5 * X^3
  SetCoeff(f, 0, 2); // f == 5 * x^3 + 2
  ```

- Pri module `ZZ`, povýšenie argumentov nastáva pri funkciách modulárnej aritmetiky, bitovom and, or a xor. Existujú aj ďalšie funkcie modulu `ZZ`, ktoré sa dajú použiť aj s argumentmi typu `long` aj `ZZ`, napr. `NumBits, bit, weight`. Detaily sú popísané v dokumentácii ZZ.txt.

6 Technické podrobnosti typovej konverzie a povýšenia

Väčšinou sú konverzie a povýšenia **sémanticky ekvivalentné**. Existujú tri **výnimky**.
Prvou je konverzia typu `double` s pohyblivou desatinnou čiarkou na `ZZ`. Najbezpečnejšie je zvoliť explicitnú konverziu a nespoliehať sa na povýšenie. Napr.
```
ZZ a; double x;
a = a + x;
```
To je ekvivalentné s
```
a = a + long(x);
```
Použitá môže byť aj funkcia pre explicitnú konverziu.
```
a = a + to_ZZ(x);
```
Druhý príklad zabezpečuje, že nie je stratená presnosť, a tiež garantuje vypočítanie zaokrúhlenia smerom nadol z `x`. Pri prvom príklade môže byť stratená presnosť, keď je `x` konvertované na `long`, a tiež orezanie záporných čísel závisí od implementačných detailov (väčšinou sa číslo oreže smerom k nule namiesto jeho zaokrúhlenia).

Druhou výnimkou je konverzia `unsigned int` alebo `unsigned long` na `ZZ`. Opäť je najbezpečnejšie použiť operátor explicitnej konverzie. Pokiaľ by sme postupovali ako vo vyššie uvedenom príklade, `unsigned int` by bol najskôr skonvertovaný na `signed long`, čo pravdepodobne nebolo našim zámerom.

Tretia výnimka môže nastať na 64-bitovej architektúre pri konverzii `signed` alebo `unsigned long` na `RR` alebo `quad_float`. Tieto typy podporujú povýšenia len z typu `double` a konverzia `long` na `double` môže mať za následok stratu presnosti. Opäť pri použití NTL funkcie, žiadna strata presnosti nenastane.

Ďalšej pasci, ktorej sa treba vyhnuť je **inicializácia** typov `ZZ` príliš **veľkými celočíselnými konštantami**. Napr.

```
ZZ x;
x = 123456789012345678901 2;
```

Celočíselná konštanta je veľmi veľká, čo môže mať za následok chybu alebo varovanie kompilátora. Najjednoduchšia možnosť zadania takejto veľkej konštanty je nasledovná:

```
ZZ x;
x = to_ZZ(''123456789012345678901 2'');
```

Na konverziu reťazca znakov `C` na typy `ZZ`, `RR`, `quad_float` a `xdouble` sú k dispozícii konverzné funkcie.
Pozornosť treba venovať aj konverzii na typ `RR`. Všetky tieto konverzie zaokrúhľujú na súčasne nastavenú presnosť, ktorá nemusí byť vždy aj požadovaná.

7 Aliasing

Dôležitá vlastnosť knižnice NTL je, že **aliasing** vstupných a výstupných parametrov je **vždy povolený**. Napr., keď napíšete `mul(x, a, b)`, potom `a` alebo `b` môže mať rovnakú adresu ako `x` (alebo hocijaký objekt obsiahnutý v `x`, napr. násobenie skalár/vektor alebo skalár/polynóm).

8 Konštruktory, deštruktory a správa pamäte

Vo všeobecnosti NTL spravuje pamäť obsadenú veľkými, dynamicky vytvorenými objektmi, ako sú objekty triedy `ZZ` alebo hocijaké dynamické vektory v NTL. Avšak je dôležité pochopiť čo si pod tým máme predstaviť.

Najviac tried je implementovaných ako **smerník** a konštruktor nastaví jeho hodnotu na NULL. Potrebná pamäť je alokovaná a pri volaní deštruktora je uvoľnená. **Výnimky** tvoria modulárne triedy `ZZ_p`, `ZZ_pE`, `zz_pE` a `GFx2E`. Veľkosti týchto objektov sú nemenné, a preto konštruktor alokuje potrebnú veľkosť pamäte.

Pri **kopírovaní** sú skopírované celé údaje, nie iba smerníky. Pokiaľ cieľový objekt nemá alokovanú dostatočnú pamäť na skopírovanie zdrojových údajov, pamäť cieľového objektu je rozšírená pomocou príkazu jazyka C `realloc()`.

Pokiaľ je pôvodný objekt menší ako cieľový, veľkosť pamäte cieľového objektu je ponechaná. Táto stratégia má väčšinou uspokojivý výsledok, pokiaľ je ale potrebné, je možné použiť príkaz `kill()`, ktorý uvoľní všetku pamäť objektu a nastaví jeho stav na počiatočnú hodnotu (hodnotu 0 alebo nulový vektor).

Výnimku tvoria triedy `ZZ_pBak`, `ZZ_pContext` a analogické triedy pre `zz_p`, `ZZ_pE`, `zz_pE` a `GF2E`. Pri týchto objektoch sa kopírujú len smerníky.

Pri **inicializácii** stojí za zmienku, že je bezpečné deklarovať **globálne objekty** hocijakého NTL typu (okrem modulárnych typov), pokiaľ je použitý len implicitný konštruktor. Napr. globálna deklarácia

```
ZZ global_integer;
vec_ZZ_p global_vector;
```

by mala vždy fungovať, pretože inicializácia zahŕňa len nastavenie smerníka na 0. **Vyhnúť** sa ale treba inicializácii globálnych objektov pomocou **neimplicitných konštruktorov** a **netriviálnym výpočtom** s NTL pred začiatkom funkcie `main()`. Napr. chceme mať inicializovanú globálnu konštantu nasledovne:

```
const quad_float Pi =
to_quad_float(''3.14159265358979323846264338327950 29'');
```

Na väčšine platforiem to pravdepodobne bude fungovať, ale táto konštrukcia zahŕňa netriviálny výpočet pred začiatkom funkcie `main()`. Lepšia stratégia zahŕňa definovanie funkcie, ktorá vracia odkaz:

```
const quad_float& Pi()
{
static quad_float pi =
to_quad_float(''3.14159265358979323846264338327950 29'');
return pi;
}
```

a potom zavolať funkciu `Pi()`, ktorá vracia odkaz na hodnotu, ku ktorej je definovaný prístup len na čítanie.

```
area = Pi()*r*r;
```

Inicializácia sa vykoná prvý krát pri volaní funkcie `Pi()`, čo nastane po začatí funkcie `main()`. Popísali sme jednoduchú a všeobecnú stratégiu, odporúčanú väčšinou C++ expertov, na inicializáciu neglobálneho objektu pri ktorej sú potrebné netriviálne výpočty.