NPRG041 – 2024/25 ZS – Cvičení MS – **Malý úkol C01**

Podmnožiny

Cílem prvního domácího úkolu je naprogramovat jednoduchou aplikaci, která pro zadanou množinu prvků najde a vypíše všechny její podmnožiny. Abychom si co nejvíce zjednodušili naši situaci, budeme předpokládat, že prvky této množiny jsou jen písmena anglické abecedy.

Samotnou vstupní množinu pak zadáme v podobě globální konstanty umístěné v hlavičkovém souboru `Input.h`, a to formou klasického céčkového pole `constexpr char items[] = { 'A', 'B', 'C', 'D' }`. Můžeme předpokládat, že toto vstupní pole bude validní, tedy bez duplicitních prvků. Jeho velikost však není implicitně omezena, může být i zcela prázdné.

Je zřejmé, že budeme muset vhodně pracovat i s velikostí tohoto vstupního pole. A jelikož chceme vytvořit univerzálně použitelný a dobře strukturovaný kód, určitě ji nikde nechceme napevno zadrátovat. Pokud bychom totiž počet prvků zadali jako fixní číslo, museli bychom pak takový údaj manuálně měnit při každé změně vstupní množiny (a to konzistentně na všech místech). Použijeme proto konstrukci `constexpr size_t count = sizeof(items) / sizeof(items[0])`, přičemž definici této konstanty opět umístíme do uvedeného hlavičkového souboru.

Celý program dekomponujeme do vhodně navržených funkcí a zcela se vyvarujeme použití globálních proměnných. Všechna potřebná data tedy budeme při volání funkcí předávat pomocí parametrů očekávaných jejich rozhraními. Hlavní funkce odpovídající za celý proces hledání a vypisování podmnožin musí být navržena univerzálně. Budeme ji tedy moci potenciálně volat opakovaně, a to i pro jakékoli jiné nebo jinak velké vstupní množiny.

Abychom rozumně pracovali s pamětí a současně zajistili, že budeme schopni všechny možné podmnožiny najít ve stejném pořadí, budeme je generovat pomocí prohledávání do hloubky. Konkrétně tak, že u každého prvku v pořadí zleva doprava rozhodneme, jestli jej v aktuální podmnožině chceme, nebo naopak nikoli. Pořadí prvků nesmíme měnit, přítomnost daného prvku pak má přednost před jeho vynecháním.

Příznaky u jednotlivých prvků si budeme pamatovat pomocí pole logických hodnot. Hodnota `true` na dané pozici bude znamenat, že odpovídající prvek do podmnožiny patřit má, `false` naopak ne. Nejde tedy o nic jiného než o jakousi masku, matematicky pak charakteristickou funkci dané podmnožiny.

Velikost tohoto pole ale nemůžeme znát předem (myšleno v okamžiku kompilace), takže jej budeme muset vytvořit pomocí mechanismu dynamické alokace. Konkrétně stačí použít konstrukci `bool* signature = new bool[count]`, čímž získáme ukazatel na první pozici takového pole. Na samotném konci ale nesmíme zapomenout toto pole zase dealokovat, a to pomocí `delete[] signature`. Pokud bychom to neudělali, danou paměť bychom nevratně ztratili.

Nalezené podmnožiny budeme postupně vypisovat na standardní výstup, a to v následujícím formátu: `{ A, C, D }`. Celou množinu tedy ohraničíme pomocí páru složených závorek, jednotlivé prvky budeme oddělovat čárkou. Pozor musíme dát na mezery, za posledním prvkem už žádnou čárku nepíšeme a v případě prázdné množiny vypíšeme jen mezeru jednu `{ }`. Na každý řádek umístíme právě jednu podmnožinu, každý řádek ukončíme pomocí `std::endl`.

Pokud bychom měli vstupní množinu např. `{ '1', '2', '3' }`, očekávaný výstup celého programu by byl následující:

```
{ 1, 2, 3 }
{ 1, 2 }
{ 1, 3 }
{ 1 }
{ 2, 3 }
{ 2 }
{ 3 }
{ }
```

Řešení vypracujte do jednoho jediného `*.cpp` souboru, ten také jako jediný soubor odevzdáte. Jinými slovy soubor `Input.h` odevzdávat nebudete, ten totiž už součástí jednotlivých připravených testů je. Řešení musí být správné, testy musí skončit úspěšně bez jakékoli chyby. Také je potřeba zařídit, že během kompilace nevzniknou žádná varování, natož chyby. Cílem úkolu je ověřit především schopnost práce s uživatelsky

definovanými funkcemi, předáváním argumentů hodnotou nebo ukazatelem, klasickými poli a standardním výstupem. Je potřeba se zaměřit nejenom na věcnou správnost, ale také na kvalitu kódu, zejména vhodnou dekompozici do funkcí nebo jejich rozhraní.

Na závěr se ještě podíváme na několik konkrétních požadavků plynoucích z častých chyb nebo nevhodného návrhu. Konkrétně nepoužívejte pokročilejší konstrukty, se kterými jsme na cvičení ještě nepracovali. Jinými slovy si plně vystačíme s knihovnou `<iostream>`, operátorem `<<` pro zápis do streamu standardního výstupu `std::cout` a klasickým céčkovým polem. Jiné knihovny nepoužívejte, zejména tedy žádné kontejnery apod. Nepoužívejte ani třídy nebo šablony, cílem je vyřešit náš problém adekvátními prostředky.

Nepoužívejte více než jedno dynamicky alokované pole pro naší signaturu. Každou konkrétní podmnožinu vygenerujte právě jen jednou, není přípustné generovat případné duplicity s jejich následným dohledáváním a odstraňováním. Vypsání prvků konkrétní již nalezené podmnožiny je potřeba zvládnout jen na jeden jediný průchod (není tedy např. možné nejprve počítat skutečný počet prvků v dané podmnožině a teprve pak realizovat její vypsání).

Názvy všech funkcí, parametrů a proměnných volte jako dostatečně vysvětlující, vše včetně komentářů pište výhradně v angličtině. U každého parametru funkce zvažte, jestli by neměl být opatřen příznakem `const`. Ačkoli už víme, že tradiční céčkové pole je ve skutečnosti technicky chápáno jen jako ukazatel na jeho první prvek, přesto je lepší v rozhraních funkcí používat zápis `char items[]` namísto `char* items`, protože pak je na první pohled zřejmé, že opravdu má jít o pole, nikoli jen o jediný izolovaný prvek. Pro nejrůznější počty, velikosti, pozice nebo indexy vždy volte typ `size_t`.

Také se ujistěte, že každá funkce plní jen jeden dobře ohraničený a definovaný úkol, tedy že jste minimálně oddělili funkci na hledání podmnožin od jejich vypisování. Současně chcete koncovým uživatelům nabídnout funkci s přívětivým rozhraním, takže takovou veřejnou funkci musíme oddělit od té interní (rekurzivní). Mimo jiné i proto, že nechceme nikoho nutit alokovat a dealokovat naše čistě interní pole signatury, natož o něm vůbec vědět.

Před odezváním úkolu celý kód finalizujte a začistěte aneb očekává se odevzdání opravdu finální, ne jen nějaké pracovní verze. Odstraňte tedy všechny ladicí nebo jiné fragmenty kódu, které v něm nemají trvale zůstat. Pozornost věnujte i vizuálnímu stylu, tedy indentaci jednotlivých řádků, zápisu složených závorek kolem těl cyklů nebo větví podmínek, způsobu vytváření víceslovných názvů, ale i volným řádkům nebo jiným oddělovačům mezi jednotlivými částmi kódu. Konkrétní styl není zase až tak podstatný, nejdůležitější je v tomto smyslu konzistence.

Pokud jde o ladění odevzdaných řešení v systému ReCodEx, vyhodnocení automatických testů funguje na principu porovnávání skutečného a očekávaného výstupu programu, přičemž konkrétně očekávaný výstup se bere jako báze, vůči které se hledají rozdíly. Výpis všech takových detekovaných rozdílů pak najedeme v logu. Řádky označené jako `+` jsou navíc (ve výstupu by být neměly), řádky označené jako `-` naopak chybí. Pokud se očekávané a skutečné řádky liší jen velmi málo, můžeme se ještě setkat s kompaktním zápisem. Např. `-10/+10: [3]B != [3]C` znamená, že na řádcích 10 je ve sloupci 3 očekávána hodnota B, ale nalezena C. Čísla takto ztotožněných řádků se samozřejmě mohou lišit, stejně jako počáteční pozice nebo délky lišících se fragmentů.

Dodejme ještě, že pro hledání podmnožin není nezbytně nutné použít klasickou techniku prohledávání prostoru všech možných hodnot pomocí rekurzivní funkce. Ostatně, volání funkcí není zrovna nejrychlejší záležitostí. Použít tedy můžeme i metodu binárního počítání. V tom případě si vystačíme bez rekurze a jen s lokálními cykly. I tady však má smysl snažit se o snížení počtu průchodů signaturou na minimum, takže je vhodné její dekrementaci a test ukončovací podmínky (detekci nalezení úplně poslední podmnožiny, tedy té prázdné) spojit dohromady.