Inlämningsuppgift, EDAF30, 2023

1 Anvisningar för redovisning

Inlämningsuppgifterna ska redovisas med en kort rapport och de program som du har skrivit. Gör så här för att lämna in inlämningsuppgiften:

1. Arkivera lösningen (med zip eller tar, men *inte något annat arkivformat* som t ex rar), med rapport (som .pdf) och kompilerbar källkod, Makefile, och eventuella indatafiler. Var noga att inte ta med genererade filer (t ex `.o` eller `.exe`) i arkivet. Koden ska gå att bygga med `g++` eller `clang++` och `make` (samt eventuelt `cmake`).
2. Instruktioner för hur uppgiften ska skickas in publiceras senare.

2 Krav på uppgiften

2.1 Rapport

Uppgiften ska redovisas med en kort (ett par sidor) rapport som översiktligt presenterar din lösning. Följande punkter ska diskuteras:

1. Övergripande design: beskriv klasser och funktioner, och deras relationer till varandra.
2. En kort användarinstruktion: hur bygger och testar man programmet? Försök att paketera så mycket som möjligt med regler i makefilen. Det underlättar både under ert arbete och gör att denna instruktion blir väldigt enkel att skriva.
3. Brister och kommentarer: Finns det något i lösning som du i efterhand anser borde gjorts annorlunda? Andra kommentarer?

Rapporten ska lämnas in som pdf-fil.

2.2 Program

Er lösning ska naturligtvis fungera och lösa den angivna uppgiften. Testning ingår som ett krav i uppgiften, och både hur väl testerna täcker uppgiften, och hur väl programmen fungerar bedöms. Provkör allting ordentligt innan ni lämnar in er lösning.

Exempelprogram och testprogram

Det är ett krav att det ska finnas dels enhetstester för alla ingående delar (klasser och funktioner), och dels ett exempelprogram som visar hur er lösning fungerar genom ett exempel som producerar någon sorts utdata i terminalen, och eventuellt är interaktivt. Testprogrammen ska vara skrivna så att det inte krävs någon manuell kontroll av utdata för att avgöra om testet lyckades eller misslyckades: varje testfall ska – på något sätt – svara "ja" eller "nej". Det ska även finnas ett huvud-testprogram som kör alla enhetstester och tydligt visar vilka som inte uppfylldes.

Generella krav

Programkoden i lösningen ska uppfylla följande krav:

1. Programmet ska ha en vettig design.
2. Klasser och funktioner ska ha tydligt avgränsade uppgifter och ansvarsområden.
3. Minneshantering ska vara korrekt: programmet får inte läcka minne.
4. Programkoden ska vara lätt att följa och förstå.
5. Koden ska vara formatterad på ett sätt som underlättar läsning. Detta innebär en vettig indentering och att raderna inte är för långa.
6. Funktions- och variabelnamn ska vara väl valda och återspegla funktionens eller variabelns innebörd.
7. Programmet ska kompilera med `-Wall -Wextra -Werror -pedantic-errors`

   En tumregel, både för design och läsbarhet, är att en funktion inte får vara längre än 24 rader eller ha fler än 5–10 lokala variabler eller mer än tre indenterings-nivåer. Det finns ibland goda skäl att göra ett undantag från detta, men ofta är det bättre att dela upp en lång, komplex, funktion i några enkla hjälpfunktioner. Varje funktion ska bara göra *en* sak, gör den flera – dela upp den.

3 Rättning

Vi rättar uppgifterna så snart vi hinner, normalt inom en arbetsvecka räknat från inlämningsdag. När uppgiften är rättad får du besked om uppgiften är godkänd eller ej. Du får en kort sammanfattande kommentar om din lösning samt i de fall uppgiften inte är godkänd kommentarer om vad som behöver förbättras. Om uppgiften inte är godkänd ska du inom rimlig tid lämna in en förbättrad version.

Effektiv konkatenering av containers

1 Översiktlig beskrivning

I standardbiblioteket för C++ finns en mängd *containers* för att lagra samlingar av värden, och *algoritmer* för att på olika sätt bearbeta dessa datastrukturer. I denna uppgift ska vi studera fallet att vi har den data vi vill arbeta med lagrad i mer än en container. Vi vill kunna använda standard-algoritmerna på den totala mängden data utan att först kopiera alla värden till en ny container.

Ansatsen är att skapa en datastruktur `concatentation(Collection a, Collection b)`, som uppför sig som en `Collection` som innehåller först alla element ur `a` följt av alla element ur `b`. Ett exempel på användning:

```
void test_concatenation()
{
    using IntVector = std::vector<int>;

    IntVector a{1,2,3,4,5,6,7,8};
    IntVector b{11,12,13,14,15};

    concatenation<IntVector> j(a, b);

    auto first7 = std::find(j.begin(), j.end(), 7);
    auto first12 = std::find(first7, j.end(), 12);

    std::deque<int> result;
    std::copy(first7, first12, std::back_inserter(result));
    // result ska nu innehålla {7,8,11};
}
```

Din uppgift är att implementera en sådan datastruktur. Eftersom detta är ett generellt problem, oberoende av typen av de underliggande datastrukturerna, ska den implementeras som en klassmall (template). I inlämningsuppgiften är det tillåtet att göra begränsningen att bara kontatenering av två *likadana containers*[1] stöds , även om man i det generella fallet vill kunna konkatenera två godtyckliga containers så länge *deras element-typer* är samma. Den förenklade versionen kan ha följande principiella utseende:

```
template<typename T>
class concatenation{
public:
    using value_type = typename T::value_type;
    using iter = // typdeklaration för iteratorer över en concatenation
    concatenation(T&, T&);

    iter begin();
    iter end();
};
```

[1] t ex två `std::vector<int>` eller två `std::deque<double>`, men inte nödvändigtvis en `std::vector<int>` och en `std::deque<int`

2 Iteratorer

Eftersom den övergripande abstraktionen som används för att genomlöpa en collection är en *iterator*, är det naturligt att utgå från hur en iterator över mer än en collection kan konstrueras. För uppgiften är det tillräckligt att bygga den enklaste sortens iterator, en `InputIterator` (och `OutputIterator` för skrivning), som kan användas till att genomlöpa en collection en gång. Den behöver ha följande operationer:

- `operator++`
- `operator++(int)`
- `operator==(...)`
- `operator!=(...)`
- `operator*()`

För att fungera med standard-algoritmerna måste en iterator även definiera ett antal typ-alias:

- `value_type` – typen som iteratorn "pekar på"
- `iterator_category` – vilken sorts iterator (`InputIterator`, `ForwardIterator`, ...)
- `difference_type` – en heltalstyp som kan användas för att mäta avståndet mellan två iteratorer (typiskt `size_t` eller `std::ptrdiff_t`)
- `pointer` – typen för en pekare till elementet, typiskt `value_type*`
- `reference` – typen för en referens till elementet, typiskt `value_type&`

För att uppfylla detta kan en iterator som på något sätt kapslar en annan iterator-typ definiera dessa typmedlemmar enligt följande

```
template <typename Iterator>
class high_level_iterator {
public:
    using value_type = typename Iterator::value_type;
    using iterator_category = std::input_iterator_tag;
    using difference_type = Iterator::difference_type;
    using pointer = value_type*;
    using reference = value_type&;

    //...
};
```

där uttrycket `using value_type = typename Iterator::value_type` betyder att iteratorn `high_level_iterator` ska ha samma `value_type` som den underliggande iteratorn (som ges av mall-parametern `Iterator`). Här måste man använda nyckelordet `typename` för att kompilatorn ska veta att namnet `Iterator::value_type` är en typ. Eftersom `Iterator` kan vara en mall och inte en klass så behöver kompilatorn denna hjälp att tolka koden.

Typen `iterator_category` anger vilken sorts iterator klassen är (d v s vilka operationer den stöder), och `std::input_iterator_tag` (som definieras i header-filen `<iterator>`) anger att en iterator är en InputIterator. Detta används för att veta vilka operationer man kan använda iteratorn (t ex måste en iterator vara `BidirectionalIterator` för att ha `operator--()`).

Ett annat sätt (som blir *deprecated* – ska inte användas – fr o m C++17) är att låta iteratorn ärva från, `std::iterator`, som tar iterator-typ och värde-typ som mall-parametrar. Ett exempel (en `ForwardIterator` över en mängd bitar (`bool`):

```
class MyIterator :public std::iterator<std::forward_iterator_tag, bool> {...};
```

3 Tips

Förslag till arbetsgång:

1. Försök arbeta test-drivet, och utnyttja testfallen (som är ett krav i uppgiften) för att effektivisera arbetet i stället för att skriva dem sist (och då inte få nån nytta av dem själv).

   Ha t ex en regel i er Makefile så att ni enkelt kan göra `make test` för att köra testerna, och gör det efter varje ändring för att tidigt upptäcka om något slutat fungera.

2. Konstruera en iterator som kapslar iteratorerna för två collections:

```
void test_join_iterator()
{
    std::vector<int> a{1,2,3,4,5};
    std::vector<int> b{6,7,8,9};

    using iter_type = decltype(a.begin());

    join_iterator<iter_type> it_begin( /* lämpliga parametrar */);
    join_iterator<iter_type> it_end( /* lämpliga parametrar */);

    for(auto i = it_begin; i != it_end; ++i) {
        std::cout << *i << " ";
    }
    std::cout << std::endl;
}
```

   Notera användningen av ett type-alias `iter_type` och hur uttrycket `decltype(a.begin())` ger typen för `a`:s iterator. Resultatet av ett `decltype`-uttryck är *typen* för argumentet, i detta fallet returtypen för `a.begin()`[2].

3. Undersök om din iterator fungerar med standard-algoritmer och lägg till de operationer som eventuellt saknas för att `std::copy` ska fungera.

```
    std::vector<int> res;

    std::copy(it_begin, it_end, std::back_inserter(res));
```

4. Skapa en klassmall `concatenation`, och låt den kapsla iteratorn enligt ovan. Testa att *range-for* fungerar:

```
    concatenation</*template arguments*/> c(/*constructor arguments*/);

    for(auto& x : c) {
        std::cout << x << " ";
    }
    std::cout << std::endl;
```

5. Testa att `concatenation` fungerar med standard-algoritmer som `std::find` och `std::copy`. Åtgärda eventuella problem.

[2] I detta exemplet kan man även skriva ut typen explicit som `using iter_type = std::vector<int>::iterator`, men operatorn `decltype` gör det möjligt att skriva generell kod i templates. Notera att kompilatorn inte evaluerar eller genererar någon kod för uttrycket som ges till `decltype`, det används bara för att räkna ut en typ. Man kan t ex skriva `decltype(new T()->some_function())` utan att det betyder att någon allokering faktiskt görs. `decltype` används bara vid kompilering och uttrycket används bara för att beskriva en typ. Om man har en typ, `T` men inte nån varaibel av `T`, och behöver skriva ett `decltype`-uttryck för en medlem av `T`, kan man använda `std::declval<T>()`, som ger en referens-till-T (som bara får användas för att använda medlemmar i `T` i ett `decltype`-uttryck). `declval` finns i `<utility>`.

6. Så långt har vi bara använt iteratorerna för att läsa värden. Testa om skrivning till en `concatenation` fungerar och åtgärda eventuella problem. En algoritm att testa med är `std::copy`, till din `concatenation`.

   Andra enkla algoritmer som genererar data är `std::iota` (från `<numeric>`) och `std::generate` (dock kräver dessa en `ForwardIterator`). T ex så skriver
   ```
   std::iota(c.begin(), c.end(), 10);
   ```
   sekvensen $[10, 11, 12, ...]$ till intervallet $[$`c.begin`,`c.end()`$)$ (där `c` är en container av heltal).

7. För användaren är det praktiskt att kunna lita på typhärledning i stället för att behöva explicit ge mallparametrarna. I standardbiblioteket finns därför hjälpfunktioner, t ex `std::make_pair` som skapar ett std::pair med samma typer som argumentet. T ex skapar `make_pair(10,42.1)` ett `std::pair<int,double>`.

   Skapa en hjälpfunktion `concatenate`, som returnerar en `concatenation` med härledd typ. Exemplet från avsnitt 1 ska kunna skrivas:

```
void test_concatenation()
{
    using IntVector = std::vector<int>;

    IntVector a{1,2,3,4,5,6,7,8};
    IntVector b{11,12,13,14,15};

    auto j = concatenate(a, b);  // j har typen concatenation<IntVector>
    // Använd j...
}
```

4 Sammanfattning

Uppgiften är att skapa en datastruktur som uppför sig som en konkatenering av två containers men som inte gör någon kopiering av element när den skapas. En sådan konkatenering ska stödja

1. range-`for`
2. de standardalgoritmer som bara kräver en `InputIterator` respektive `OutputIterator`[3](t ex `std::copy`, `std::find`, `std::transform`).

Det är tillåtet att göra avgränsningar så länge ovanstående är uppfyllt, t ex behöver inte det som krävs för att algoritmer som `std::sort` ska fungera implementeras.

Avslutande kommentarer:

- En sak som kan vara lite besvärlig att få till korrekt är `const` och funktionerna `begin` och `end`. En vanlig variant är att `const`-versionen av `begin` delegerar till `cbegin` för att skapa en `const`-iterator (och motsvarande för `end`). Det är tillåtet att begränsa uppgiften till icke-`const` `begin` och `end`.
- När man arbetar med templates får man ofta väldigt långa och förvirrande kompileringsfel. Försök identifiera vad som är det egentliga felet, och vad som är följdfel. Arbeta i små steg och testa ofta.
- Dokumentera de designbeslut och avgränsningar ni gör under arbetet så att ni enkelt kan beskriva dem i rapporten. Notera även eventuella problem och frågor som dyker upp, och deras eventuella lösningar.

[3] Eller `ForwardIterator`. För uppgiften behöver ni inte hantera skillnaden mellan `ForwardIterator` och kombinationen av `InputIterator` och `OutputIterator`, så länge de angivna kraven uppfylls.