Standardbiblioteket STL

Nu när vi är experter på grunderna i C++ ska vi lära oss lite mer om standardbiblioteket i C++, vilka olika funktioner och strukturer det innehåller. Standardbiblioteket kallas ofta för *STL*, och står för *Standard Template Library*. Standardbiblioteket innehåller en stor mängd färdiga funktioner och datastrukturer som vi kan använda oss av i våra program genom att inkludera dessa.

Globala variabler

Innan vi ger oss in i standardbibliotekets underbara värd vill jag först nämna något om *globala variabler*. Vanligtvis har vi alltid deklarerat våra variabler innuti en funktion - vanligtvis main-funktionen, men även andra funktioner. Det visar sig dock att vi kan deklarera variabler *utanför* en funktion. I så fall kommer variabeln vara gemensam för samtliga funktioner i vårt program:

```
#include <iostream>

using namespace std;

// x är utanför alla funktioner - den är global
int x;

void addera(){
  x = x + 1;
}

int main(){
  x = 5;
  addera(); // Uppdaterar x till 6
  cout << x << endl;
  addera(); // Uppdaterar x till 7
  cout << x << endl;
}
```

Variabler som deklareras innuti en funktion kallas istället för *lokala variabler*. Du kommer behöva använda globala variabler för att lösa övningen *Telefonkatalog* senare.

`vector`

Under loop-avsnittet i föregående lektion stötte vi på problemet *N-summa*, där vi skulle läsa in N olika tal och summera dessa. En vanlig fråga är: vad händer om vi skulle vilja spara alla N talen i variabler *samtidigt*? Måste vi deklarera N olika variabler, ett för varje tal? Svaret är självfallet nej - och till vår hjälp har vi datastrukturen *vector*.

En *vector* är en av de mest grundläggande strukturerna som standardbiblioteket erbjuder. Det är helt enkelt en lista av saker, som man kan lägga till och ta element i. För att kunna använda strukturen måste man först inkludera den från standardbiblioteket genom att lägga till

```
#include <vector>
```

där man i sitt program har sina övriga `includes`.

På samma sätt som du när du deklarerar en variabel måste ange vilken datatyp variabeln ska innehålla, behöver din vektor också veta vilken datatyp dess element ska ha. Man kan alltså inte ha en vektor som innehåller både heltal och flyttal, utan alla element ska ha samma typ. Vi anger den när vi skapar en vector-variabel genom att lägga till typen inom vinkelparenteser (`<typnamnet>`) efter `vector`:

```
vector<string> ord;
```

Denna kodrad skapar alltså en vector vid namn `ord`, och deklarerar att den ska innehålla strängar - element av typen `string`.

När vi sedan skapat vectorn kan vi lägga till element i den med dess inbyggda funktion `push_back`. Den används på följande sätt:

```
ord.push_back("Simon");
ord.push_back("is");
ord.push_back("a");
ord.push_back("fish");
```

För att hämta ut värdena vi precis lade in i vektorn kan vi använda hakparentesoperatorn. Om vi skriver `ord[i]` får vi ut det i:te värdet i vektorn:

```
  cout << ord[0] << " " << ord[1] << " "; // Simon is
  cout << ord[2] << " " << ord[3] << endl; // a fish
```

Notera att det första elementet i vektorn har talet 0. Vi säger att vektorn är *noll-indexerad*.

Övning 1. Om 0 är det första elementet i en vektor, vilket index kommer det *sista* elementet ha? Antag att vektorn innehåller exakt N element. Skriv kod för att testa.

Notera här att det första elementet motsvaras av talet 0. Detta kalls för *noll-indexering*. Om du någonsin använder ett index som är utanför en vectors giltiga index, t.ex. `ord[5]` när vectorn bara har 5 element, så kan ditt program krascha, eller så får du ett slumpmässigt värde.

Hakparentesoperatorn kan även användas när vi vill ändra ett visst element i vectorn:

```
ord[0] = "Aron";
ord[3] = "vegetarian";

cout << ord[0] << " " << ord[1] << " "; // Aron is
cout << ord[2] << " " << ord[3] << endl; // a vegetarian
```

Du kan även få reda på längden av en vector med den inbyggda funktionen `size()`. Detta är en inbyggd funktion som många av datastrukturerna i STL har.

```
int storlek = heltal.size();

cout << storlek << endl;
```

Övning 2. Lös problemet Omvändning (id: reverse) på Kattis.

Skriv tre olika sorters lösningar:

- Läs in orden i en vector, skriv ut dem i omvänd ordning direkt.
- Läs in orden i en vector, kopiera in dem i omvänd ordning i en annan vector.
- Läs in orden i en vector, vänd ordningen genom att byta plats på två element i taget (först 0 och `N - 1`, sedan 1 och `N - 2`...). Se till att du stannar i mitten!

Övning 3. Lös problemet Vectorfunktioner (id: vectorfunktioner) på Kattis.

Övning 4. Man kan skapa en vector som redan från början har ett visst antal element på följande vis:

```
vector<int> vec(5); // skapar int-vector med 5 element
```

Frågan är: vad kommer dessa element faktiskt vara? Prova att skapa en vector som redan har ett antal element, och kolla vad för element den fylls med.

Iteratorer

Ett centralt koncept i STL är *iteratorn*. En iterator är ett objekt som pekar på ett element i någon datastruktur, t.ex. en vector. Iteratorns typ är samma som din datastrukturs typ, följt av `::iterator`. Iteratorn som pekar på det första elementet kallas för `begin`, och kan användas genom:

```
vector<string>::iterator first = ord.begin();
```

Iteratorn `first` motsvarar här alltså det första elementet. För att få ut värdet på det element som iteratorn motsvarar skriver vi en asterisk innan iteratorns namn:

```
string val = *first; // = "Aron"

cout << val << endl;
```

Iteratorer kan också gå fram och tillbaka i datastrukturen med hjälp av ökningsoperatorn och minskningsoperatorn:

```
//Först pekar first på det första elementet

first++;
//Nu pekar first på det andra elementet

cout << (*first) << endl; // = "is"

first--;
//Nu pekar first på det första elementet igen

cout << (*first) << endl; // = "Aron"
```

Det finns också en iterator som pekar på elementet *efter* det sista elementet. Den heter `end`, och med hjälp av den kan man bygga en loop som går igenom alla element i en datastruktur.

```
// Börja på start-iteratorn
// Så länge iteratorn inte är slut-iteratorn
// Flytta fram iteratorn ett steg
for(vector<string>::iterator it = ord.begin(); it != ord.end(); it++){
  string value = *it;
  cout << value << endl;
}
```

Det första vi gör i loopen är att skapa en iterator som motsvarar det första elementet i vectorn. Vårt villkor är sedan att denna iterator aldrig ska motsvarar `ord.end()`. Den iteratorn är ju elementet *efter* det sista elementet, så det är ju rimligt att vi aldrig vill gå utanför. Till slut flyttar vi iteratorn ett steg fram.

Denna process går alltså ut på att iteratorn först pekar på det första elementet, sedan ökar vi iteratorn så den pekar på det andra elementet, och så vidare. Så fort iteratorn går utanför vår vector blir den lika med `end`-iteratorn, och då ska vi ju avsluta loopen.

Övning 5. Vad händer om du använder `*` på en iterator som befinner sig före `begin()`? Efter?

Många funktioner i standardbiblioteket använder sig av iteratorer, så de är oerhört viktiga att känna till

I en vector kan du också addera ett heltal till en vector, för att gå fram flera steg:

```
vector<string>::iterator first = ord.begin(); // Pekar på "Aron"
vector<string>::iterator third = first + 2;  // Vi går fram två steg, pekar på "a"
```

Övning 6. Det finns två andra iteratorer, `rbegin()` och `rend()`. Ta reda på vad dessa är, och använd dem för att skriva en loop som går baklänges.

sort

En av funktionerna i standardbiblioteket som gör just detta är `sort`. Den tar två iteratorer, en startiterator och en slutiterator, och sorterar sedan alla element som är mellan iteratorerna, exklusive den sista iteratorn men inklusive den första iteratorn. Funktionen ligger i filen `<algorithm>`, så glöm inte att inkludera den innan du använder `sort`.

```
vector<int> hej;
hej.push_back(1);
hej.push_back(4);
hej.push_back(2);
hej.push_back(3);

sort(hej.begin(), hej.end());

//Gå igenom alla element och skriv ut dem
for(int i = 0; i < hej.size(); i++){
  cout << hej[i] << endl;
}
```

För att använda `sort()` måste du inkludera `algorithm`:

```
#include <algorithm>
```

Övning 7. Hur skulle du göra för att sortera de N första elementen i en vector (och inte flytta de övriga alls)?.

Övning 8. Prova att göra en vector av strängar (`string`). Hur sorterar den strängarna då? Testa på strängarna `hej`, `hejhej`, `hehe`, `hejsan`, `heltal`.

Övning 9. Skriv om din lösning till problemet Tre-sortering (id: threesort) med `sort()`-funktionen.

Övning 10. Det finns en STL-funktion som vänder på en vector. Ta reda på vad den heter, och skriv om din lösning till Omvändning (id: reverse) med den.

Övning 11. Det finns STL-funktioner för att hitta det minsta elementet i en vector, samt att summera en vector. Ta reda på vad dessa heter och hur de funkar, och skriv om din lösning till Vectorfunktioner (id: vectorfunktioner) med dessa.

map

Ibland vill man kunna indexera över andra saker än bara 0, 1, ..., `vec.size() - 1` som för vektorn. För detta ändamålet erbjuder STL datastrukturen `map`. I en map kan du t.ex. indexera över strängar istället:

```
map<string, int> siffror;

siffror["zero"] = 0;
siffror["one"] = 1;
siffror["two"] = 2;
siffror["three"] = 3;
siffror["four"] = 4;
siffror["five"] = 5;
siffror["six"] = 6;
siffror["seven"] = 7;
siffror["eight"] = 8;
siffror["nine"] = 9;

cout << siffror["eight"] << endl;
```

Maps ligger i filen `map`.

```
#include <map>
```

Notera att vi här inte bara ger typen vi vill lagra (`int`) när vi deklarerar vår map, utan vi skriver också vilken typ vi ska indexera över (`string`).

Iteratorer finns även för maps, och fungerar på samma sätt som vector-iteratorerna, förutom att värdena här pekar på *par* av värden. Vi demonstrerar hur en loop över map-iteratorer kan se ut:

```
for(map<string, int>::iterator it = siffror.begin(); it != siffror.end(); ++it){
  pair<string, int> value = *it;
  cout << value.first << " = " << value.second << endl;
}
```

Övning 12. Märker du något speciellt med ordningen som indexen blev utskrivna? Är den alltid samma?

Övning 13. Vad händer om du använder ett index som inte är inlagt i en map? Prova för några olika datatyper. Exempelvis

```
cout << siffror["ten"] << endl;
```

i koden ovan.

Övning 14. Det finns två sätt för att ta reda på om ett index har deklarerats i en map. Ta reda på vilka de är genom att läsa dokumentationen för `map`.

Övning 15. Lös problemet Telefonkatalog (id: telefonkatalog) på Kattis. Du kommer behöva använda förgående övning, samt slå upp hur man tar bort värden ur en map.

set

Inom matematiken har vi *mängder* av element, där upprepningar inte spelar någon roll. Mängden $\{1, 1, 2, 3, 3\}$ är precis samma mängd som $\{1, 2, 3\}$. Det enda som spelar roll med en mängd är huruvuda den innehåller ett visst element eller inte.

Motsvarande struktur heter i C++ `set`, och ligger i filen `set`:

```
#include <set>
```

Precis som med vektorn kan ett set endast innehålla element av en viss typ, som vi själva väljer när du skapar ett set.

```
set<int> heltal;
heltal.insert(1); // lägg till 1
heltal.insert(3); // lägg till 3
heltal.insert(3); // lägg till 3
heltal.insert(7); // lägg till 7

cout << heltal.size() << endl;
```

För att sätta in element i en set använder vi den inbyggda funktionen `insert()`.

Övning 16. Hur många element innehåller vårt set? Provkör koden.

Även set:s har iteratorer, och precis som med en map så itererar man genom ett set i *sorterad* ordning:

```
for(set<int>::iterator it = heltal.begin(); it != heltal.end(); ++it){
  cout << *it << endl;
}
```

Övning 17. Hur skriver man ut det minsta elementet i ett set?

Övning 18. Lös problemet Bijektion (id: bijektion) på Kattis.

string

Vissa av strängfunktionerna ligger i filen `string`, så ibland kan du behöva lägga till:

```
#include <strinsg>
```

En string kan också användas som om den vore en `vector<char>`. Man kan alltså sortera en sträng, använda iteratorer med den, lägga till saker i slutet med `push_back()`, och plocka ut tecken med index i genom hakparentesoperatorn:

```
string s = "Simon";

s.push_back('e'); // Vi gör push_back med en char - 'e'
cout << s << endl; // Skriver ut Simone

sort(s.begin(), s.end()); // Sorterar bokstäverna
cout << s << endl; // Skriver ut Seimno

s[4] = 'z'; // ändra n -> z
cout << s << endl; // Skriver ut Seimzo
```

Vi har nu blivit redo för att förstå hur en sträng lagras internet på datorn. Som vi precis konstaterade kan en sträng ses som en vector utav char, men vad precis är en char? Jo, en char är faktiskt som ett precis vanligt heltal. Prova följande kod:

```
  // Tilldelning int = char är OK, char är också heltal
  int A = 'A';
  int q = 'q';
  int plus = '+';
  cout << "A = " << A << " q = " << q << " + = " << plus << endl;
```

Det här exemplet illustrerar väldigt tydligt varför datatyper är oerhört viktigt. Vi har just sett att bokstaven `'A'` också kan betraktas som talet 65. Det som avgör när det är 65 och när det är `'A'` är helt enkelt vilken datatyp vi har.

Kodningen som används kallas för ASCII, och du kan hitta en tabell över vilka tal som motsvarar vilka tecken på `http://www.asciitable.com/`.

Övning 19. Lös problemet Caesar-chiffer (id: caesar) på Kattis.

Matematik

C++ har också ett stort antal matematiska funktioner: `http://www.cplusplus.com/reference/cmath/`. För att använda dessa måste du inkludera filen `cmath`:

```
#include <cmath>
```

Övning 20. Hitta funktioner för att:

- Beräkna de trigonometriska funktionerna sin, cos, tan.
- Beräkna inverser till de trigonometriska funktionerna sin, cos, tan.
- Ta logaritmen av ett tal i baserna 2, e och 10. Hur gör du för att ta logaritmen i en godtycklig bas?

- Beräkna e^x.
- Beräkna x^y.
- Beräkna $\sqrt{x}$
- Beräkna $\mid x \mid$, absolutbeloppet av x.

Hemtal

Kattisinlämning 1 (id: bookingaroom). Lätt

Kattisinlämning 2 (id: busnumbers). Lätt

Kattisinlämning 3 (id: color). Medel

Kattisinlämning 4 (id: parking). Medel

Kattisinlämning 5 (id: kolone). Svår

Kattisinlämning 6 (id: halfacookie). Svår