Labb 5 String och Deep Copy problemet

Denna laboration går ut på att tydliggöra problemen som uppstår när ett objekt:

- ”äger” minne på heapen
- ska kunna kopieras och tilldelas värden

samt ge ett enkelt exempel av hur container klasser fungerar för att ge bättre förståelse för STL:s container classes. Vi gör dock en container för char så vi slipper trassla med templates.

Uppgift:
Implementera en String klass (stort S för att skilja från STL klassen), den ska fungera ungefär som en förenklad std::vector<char> vilket även blir ganska likt std::string. Ni ska dock följa vector när metoden finns definierad där.
En String har värde semantik, dvs. när du skriva stringX = stringY sker en kopiering så att det nu finns två strängar som innehåller samma sak. En String hanterar själv sitt minne, dvs. själva tecknen i strängen finns allokerade i en C-array av char som ligger på heapen.
Det ska gå att lägga till tecken till en String obegränsat många gånger, när det allokerade minnet inte räcker till allokeras en ny större C-array av char på heapen och den gamla frigörs.

String klassen ska för G:

Funktion som fungera som i string	Kommentar
`~String();`	
`String();`	
`String(const String& rhs);`	
`String(const char* cstr);`	För teständamål
`String& operator=(const String& rhs);`	
`char& operator[](int i);`	indexerar utan range check
`const char& operator[](int i) const;`	indexerar utan range check
`int size() const;`	finns i container klasserna i STL, se vector
`int capacity() const;`	finns i container klasserna i STL, se vector
`void push_back(char c)`	lägger till ett tecken sist
`friend bool operator==(const String& lhs , const String& rhs)`	global function
`operator<<`	För testning, se nedan för kodexempel.

För VG ska även följande metoder implementeras

`char& at(int i) ;`	indexerar med range check *“Bounds checking is performed, exception of type std::out_of_range will be thrown on invalid access.”*
`const char& at(int i) const;`	
`char* ConvertToChars() const;`	Ger en heap allokerad kopia av strängen med ett ‘\0’ tecken tillagt.
`void reserve(int);`	finns i container klasserna i STL, se vector
`String& operator+=(const String& rhs)`	tolkas som konkatenering. Enklast att implementera += först och använda den för att göra +
`String operator+(const String& rhs)`	

Tips: Det kan vara klokt att göra hjälpfunktioner för att t.ex. allokera nytt minne.

Semantik för klassen.

```
String c(”huj”), d(”foo”);
c=d;
```

om vi nu ändrar på c eller d så ska det inte påverka värdet på den andra variabeln.

Observera att på c-string så finns det en avslutande '\0' på varje textsträng. Vi ska inte ha det utan en String "Olle" ska vara exakt 4 bytes lång.

Invarianter och Assert

Ni ska ha en privat metod som gör assert på allt som ni vet alltid är sant för ett String objekt. I exemplet så kollar vi att storleken inte kan bli större än kapaciteten (byt size och capacity mot det som stämmer med er klass). Det bör finnas andra saker som alltid är sanna.
Anropa Invariant i slutet av varje konstruktor och i början av destruktorn. Lägg gärna in anrop även i början och/eller slutet av andra metoder.
Assert finns i <cassert> headern.

```
void Invariant(){
    assert(size <= capacity);
}
```

Implementering av operator<<

```
friend void operator<< (std::ostream& out, const String& rhs) {
    for (int i=0; i<rhs.size();++i)
        out << rhs[i];
}
```

Testprogrammet i Main.cpp

Observera att det testprogram som finns i Main.cpp bara är en hjälp och varken fullständigt eller garanterat helt korrekt. Är program kan vara felaktigt fast testprogrammet fungera och tvärtom!

För G, kommentera bort raden ”`#define VG`”. För VG ha den med.

Krav för G

Implementera specifikationen ovan. Tänk på att ha med const där det är lämpligt. Det får inte finnas minnesläckor.

Krav för VG

Förutom att implementera all metoder ovan för G och VG så ska ni:

- Ha alla ”const” exakt rätt.
- Det hela ska vara effektivt – fast gå inte till överdrift.
    - All onödig allokering av dynamiskt minne kostar!
    - Om ni förlänger strängen t.ex. genom upprepade push_back tills den har längden N så skall det kosta O(N*log(N)), t.ex. så om ni gör 1000 push_back så ska det inte bli fler än ca 10000 (=1000*10) char som har kopierats.

Kommentar om ”Append problemet”

Om man i en ”container” lägger till saker sist så tar för eller senare kapaciten slut och nytt minne måste allokeras och data kopieras. Ni ska göra en lösning på detta som gör att tidskomplexiteten rimlig – O(Nlog(N)) - och inte kvadratisk – O(N^2) vid många tillägg.

Kommentar om ”Const problemet”

Man får inte ändra på en ”const variabel” och därför så behövs två versioner av de funktioner som lämnar ut en reference. T.ex.
char& operator[](int i) {….}” för anrop av normala String.
const char& operator[](int i) const {….}” för anrop av const String.m
Kompilatorn kommer att använda const varianten när man indexerar på en const String.

Kommentar om unsigned types:

En vanlig missuppfattning är att om man har en unsigned type så löser det problem som negativa värden kan föra med sig. Det är oftast inte fallet.
Exempel:

```
void Check_if_between_0_and_10(size_t n) { //size_t är en unsigned typ
    if (n < 0)
        std::cout << n << " är mindre än 0\n";
    else if (n > 10)
        std::cout << n << " är större än 10\n";
    else
        std::cout << n << " är nellan 0 och 10\n";
}

void TestUnsigned() {
    int i = 5;
    unsigned int j = -1;
    int x = i + j;
    int y = j;    //vad är x resp y?

    Check_if_between_0_and_10(5);
    Check_if_between_0_and_10(20);
    Check_if_between_0_and_10(-1);    //Vad skrivs ut?
}
```

Beroende på resultaten ovan så brukar jag (och Bjarne Stroustrup) nästan aldrig använda unsigned!