

dt047g Programmeringsmetodik

# Laboration: Dynamisk minneshantering, RAII och merge

Martin Kjellqvist\*

merge.tex 190 2016-12-05 22:15:47Z martin

## 1 Introduktion

RAII - [http://en.wikipedia.org/wiki/Resource\\_Acquisition\\_Is\\_Initialization](http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization) är ett idiom i C++ som gör vår programkod fri från minnesläckage. Iden är enkel, använd enkla behållare som då de konstrueras också skapar resursen. Vid destruktion frigörs resursen. Använd sedan stacken för att skapa och förstöra dessa enkla behållare. Stacken kommer att garantera att resursen frigörs oavsett vad som händer i programmet.

Vi strävar efter att få den dynamiska minneshanteringen transparent och enkel.

Uppgiften tillåter oss att sedan implementera en kraftfull sorteringsmetod på ett enkelt sätt.

## 2 Syfte

Syftet med denna laborationen är

- Fräscha upp kunskaperna om pekare och referenser.
- Bli välbekant med RAII idiomet.

---

\*E-post: [martin.kjellqvist@miun.se](mailto:martin.kjellqvist@miun.se).

- Tillämpa flera idiomatiska lösningar, bla RAII, copy-and-swap, container för range-for.

### 3 Läsanvisningar

Detta är en inledande laboration. Du använder kurslitteraturen som referens tillsammans med dina tidigare erhållna c++ kunskaper.

### 4 Genomförande

För varje klass ska du skapa en cpp-fil och en h-fil med include-guards. [http://en.wikipedia.org/wiki/Include\\_guard](http://en.wikipedia.org/wiki/Include_guard). Detta för att garantera unika definitioner, [http://en.wikipedia.org/wiki/One\\_Definition\\_Rule](http://en.wikipedia.org/wiki/One_Definition_Rule).

Genomför uppgifterna i den ordning de ges.

1. Skapa en klass `int_buffer` som sköter en minnesresurs av typen `int`.

Klassen ska ha följande gränssnitt.

```
class int_buffer{
public:
    explicit int_buffer(size_t size); // size_t
        is defined in cstdlib
    int_buffer(const int* source, size_t size);
    int_buffer(const int_buffer& rhs); // copy
        construct
    int_buffer(int_buffer&& rhs); // move
        construct
    int_buffer & operator=(const int_buffer&
        rhs); // copy assign
    int_buffer & operator=(int_buffer&& rhs);
        // move assign
    size_t size() const;
    int* begin();
    int* end();
    const int* begin() const;
    const int* end() const;
    ~int_buffer();
};
```

Att tänka på: storleksförändringar sköts enkelt av konstruerare nummer två.

- Skriv klassdefinitionen med nödvändiga tillägg i `int_buffer.h` och implementation i `int_buffer.cpp`.
- Skriv följande test i `main`:
  - Anropa en funktion `f` med argumentet `int_buffer(10)`

```
void f(int_buffer buf);
```

och kontrollera med hjälp av breakpoints vilka konstruere och destruerare som körs. Anteckna detta i labben.

- I funktionen `f`, skriv två forsatser med följande form

```
for(int* i = buf.begin(); i != buf.end();  
    i++)
```

och

```
for(const int* i = buf.begin(); i !=  
    buf.end(); i++)
```

där den första formen tilldelar buffern värdet 1 och uppåt. Buffern kommer efter tilldelningen att innehålla värdena 1 till 10. Den andra formen skriver ut innehållet i buffern. Kontrollera med debuggern att rätt version av `begin` och `end` anropas i de båda fallen. Anteckna detta i labben.

Detta avslutar testning för `int_buffer`.

2. Skapa en klass med följande gränssnitt.

```
class int_sorted{  
public:  
    int_sorted(const int* source, size_t size);  
    size_t size() const;  
    int* insert(int value); // returns the  
        insertion point.  
    const int* begin() const;  
    const int* end() const;  
    int_sorted merge(const int_sorted&  
        merge_with) const;  
};
```

En formell beskrivning av merge finner du i algoritm 2.

Notera att denna klass ska inte hantera något dynamiska minne alls. Det sköter buffern.

Klassen har minst ett privat attribut av typen `int_buffer`.

Testa klassen genom att göra insert på något hundratal slump-tal och därefter kontrollera att `begin()` till `end()` är i stigande ordning. Alternativt gör du en medlemsfunktion som returnerar `true` om den är sorterad. Nackdelen med det tillvägagångssättet är att den medlemsfunktionen inte är meningsfull annat än i test-hänseende.

Formell beskrivning av en gångbar metod finner du i algoritm 1.

3. Skriv följande sortering. Argumenten är en array av heltal.

```
int_sorted sort(const int* begin, const
    int* end){
    if ( begin == end ) return int_sorted();
    if ( begin == end -1 ) return
        int_sorted(*begin, 1);

    ptrdiff_t half = ( end-begin )/2; //
        pointer diff type
    const int* mid = begin + half;
    return sort( begin, mid ).merge( sort(
        mid, end ) );
}
```

Välj en av två följande uppgifter:

- Implementera en selection sort. Gör en tidsmätning på hur lång tid de båda metoderna tar att sortera 400000 `rand()` element. Kör några gånger så din tidtagning blir tillförlitlig. Se upp så att du inte sorterar redan sorterade element.

Det finns även en metod `sort` i standardbiblioteket, header `<algorithm>`. Kontrollera hur lång tid den tar på sig. Standard `sort` använder troligtvis en metod `intro-sort` som är en variant på `quick sort`, och bör vara något snabbare.

- Beskriv utförligt hur metoden `sort` fungerar.

## 5 Examination

Du ska lämna in header och implementation för varje uppgift, tillsammans med ett dokument som besvarar frågorna i texten. Filerna zippas i ett arkiv och lämnas in på kurswebbplatsen. Använd inte icke-standardformat som zipx och liknande.

## 6 Algoritmer

---

**Algorithm 1** Avgör om A är sorterad

---

**Input:** A innehåller numeriska värden

**Output:** Filen A är sorterad

```
a ← A.next()
while A.hasNext() do
  b ← A.next()
  if a > b then
    return false
  end if
  a ← b
end while
return true
```

---

---

**Algorithm 2** Merge med två filer

---

**Input:** A och B är sorterade enligt algoritm 1

**Output:** C innehåller samtliga värden från A och B i sorterad ordning

**while** *A.hasNext()* And *B.hasNext()* **do** ▷ Avgör vilket värde som ska skrivas till C

*a* ← *A.next()*

*b* ← *B.next()*

**if** *a* < *b* **then**

        Skriv *a* till C

*a* ← *A.next()*

**else**

        Skriv *b* till C

*b* ← *B.next()*

**end if**

**end while**

▷ A eller B är slut, skriv klart båda

**while** *A.hasNext()* **do**

    Skriv *A.next()* till C

**end while**

**while** *B.hasNext()* **do**

    Skriv *B.next()* till C

**end while**

---