Linköpings Universitet
Institutionen för datavetenskap (IDA)
UPP-gruppen 2020-09-30

TDDC76 Seminarie 3

Bakgrund - Studentmaterial

Föreställ dig att du implementerat grunden för att bygga en stack. En pekare till en nod anger översta elementet på stacken. Varje nod kan sedan peka ut noden som ligger under.

```
class Node
{
public:
  Node(int i, Node* n = nullptr);

  int   get_value() { return value; }
  Node* get_below() { return below; }

private:
  int value{};
  Node* below{};
};
```

Hur gör du nu för att skapa en stack? Sätt in ett nytt värde? Titta överst? Ta bort ett värde? Kontrollera om stacken är tom? Kopiera en stack? Undvika minnesläckor?

```
Node* stack{}; // skapa tom stack

stack = new Node{7, stack}; // Lägg 7 överst

cout << stack->get_value() << endl; // Titta överst

Node* popped = stack; // Flera rader för att ta bort ett värde...
stack = stack->get_below();
delete popped;

if ( stack == nullptr )
{
  // Stacken är tom
}

Node* copy = nullptr; // Flera rader för att kopiera hela stacken...
while ( ... )
{
  // Kopiera en nod i taget till copy
}

while ( stack != nullptr ) // Flera rader för att undvika minnesläcka...
{
  // Ta bort ett värde
}
```

Fundera nu på ovan implementation. Är den bra? Skulle du kunna ge den till vilken C++-människa som helst och lita på att allt blir rätt när en stack används? What could possibly go wrong? Okej. Om det är glasklart för dig hur ovan stack fungerar - good for you! Men du som är lite mer mänsklig inser att ingenting med ovan kod är självklart och många saker är lätt att helt enkelt glömma av för det “verkar fungera ändå”.

Hur skulle vi vilja kunna använda stacken?

```
Stack stack{}; // Skapa tom stack

stack.push(7); // Lägg 7 överst

cout << stack.top() << endl; // Titta överst

stack.pop(); // Ta bort ett värde

if ( stack.is_empty() )
{
  // Stacken är tom
}

Stack copy = stack; // Kopiera hela stacken

// std::move upplyser kompilatorn om att argumentet inte behöver användas
// mer så att den effektiva flytt-varianten av kopiering kan användas
Stack steal = std::move(stack); // Flytta stacken

// Minnesläckor ska undvikas automatiskt!
```

Hur kan vi använda C++ för att gå från den jobbiga implementationen till den som är enkel och säker att använda? Vi vill göra det trivialt att skapa ett program som använder en stack. Här är ett exempel som parar ihop det först inmatade ord med det sist inmatatade order, det andra ordet med det näst sista ordet och så vidare:

```
#include <iostream>
#include "stack.h"
using namespace std;

int main()
{
    int x{};
    Stack {};

    while (cin >> x)
    {
        reverse.push(x);
    }

    Stack order;
    Stack rclone{reverse};

    while (!rclone.empty())
    {
        order.push(rclone.top());
        rclone.pop();
    }

    while (!reverse.empty())
    {
        cout << order.top() << ':' << reverse.top() << endl;
        reverse.pop();
        order.pop();
    }
}
```

För att få till detta måste vi i C++ förstå vad som behöver hända under stack-objekts livstid och vilket stöd C++ ger. Samtliga av nedan speciella medlemmar anropas av kompilatorn vid behov. Att kompilatorn tar hand om anrop gör att vi inte glömmer, men vi måste implementera allt som kan behöva anropas.

- Konstruktion: Vad behöver hända när objekt skapas? (C++ ger oss konstruktorn)
- Kopiering: Hur ska objekt kopieras? (C++ ger oss varianter för fyra fall)
  - Kopiera till befintligt objekt: tilldelningsoperatorn
  - Kopiera till nytt tomt objekt: kopieringskonstruktorn
  - Flytta döende objekt till befintligt objekt: flyttilldelningsoperatorn
  - Flytta döende objekt till nytt tomt objekt: flyttkonstruktorn
- Destruktion: Vad behöver hända när objekt inte behövs mer? (C++ ger oss destruktorn)

Ibland är det inte tydligt vilken kopieringsvariant kompilatorn kommer att använda. Det spelar normalt ingen roll eftersom kompilatorn alltid väljer bästa alternativet. Men om du vill testa programmet och felsöka behöver du förstås se till att varje kopieringsvariant anropas.

```
{
  Stack a{};  // vanlig konstruktor

  Stack b{a}; // kopieringskonstruktor
  Stack c = a; // fortfarande kopieringskonstruktor!

  Stack d{std::move(b)}; // flyttkonstruktor
  Stack e = std::move(c); // fortfarande flyttkonstruktor!

  a = e; // kopieringstilldelning: operator=(const&)
  a = std::move(e); // flyttilldelning: operator=(&&)

} // destruktor för a,b,c,d,e
```

På seminariet är det er uppgift att gemensamt komma fram till hur vi kan göra en klass som går att använda enligt ovanstående huvudprogram main. Klassen ska vara enkel och säker att använda. Det ska alltså vara svårt(omöjligt) att göra så minnesläckor uppstår eller något annat blir fel utan att det märks.