Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 2 –
Deep vs. shallow copy, smarta pekare och RAII

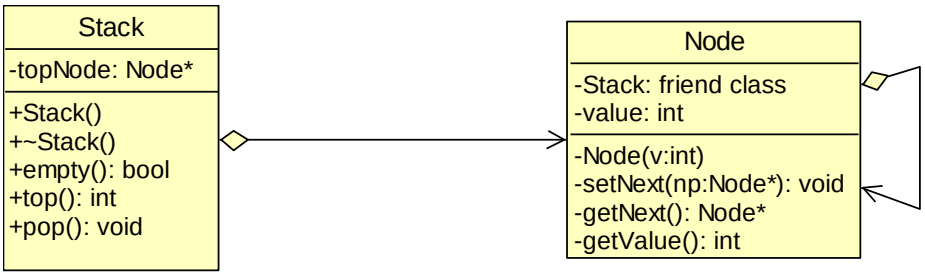
DT060G Objektorienterad programmering i C++

Behandlar
- dynamiskt minne med inbyggda pekar
- deep/shallow copy
- ”smart pointers”
- RAII

Lektion 2 - Shallow vs Deep copy, Smarta pekare och RAII

Fråga: När bör man definiera egna versioner av tilldelningsoperator och kopieringskonstruktor?
Det korta svaret är: när man har pekare till dataobjekt som attribut!

Detta gäller alltid när man har ”raw pointers” dvs. pekare av den inbyggda pekartypen, t.ex. int*. Från och med C++11 finns s.k. smarta pekare (smart pointers) som ändrar bilden men vi återkommer till detta.
Vi tittar på ett exempel med en stack (LIFO) implementerad som en dynamiskt länkad lista (exStack1.zip).




```
/*  File:     Stack.h
    Purpose:  Definition of a stack implemented as a dynamically
              linked list of Node objects. Default copy constructor
              and assignment operator give shallow copy semantics.
*/
#ifndef STACK_H
#define STACK_H

class Stack; // Forward declaration of Stack, more to come...

class Node {
// Give Stack access to private members
friend class Stack;

private:
    Node *next;
    int value;

    // Constructor
    Node(const int &v):value(v), next(nullptr) { }

    // Getters & Setters
    void setNext(Node *np) {
        next = np;
    }

    Node* getNext( ) const {
        return next;
    }

    int getValue( ) const {
        return value;
    }
}; // class Node
```

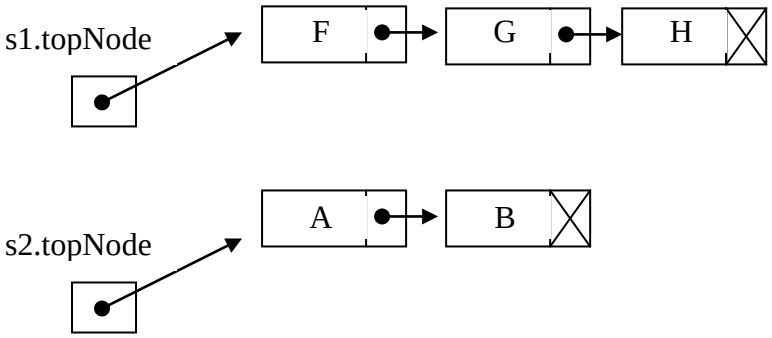
```
/*
    Operations on the stack:
    push    puts a new value on the top
    top     returns the top value
    pop     removes the top value
    empty   is the stack empty?

Implemented as a linked list of Node-objects. The nodes are
dynamically created and destroyed as the stack grows and
shrink. */


class Stack {
private:
    Node *topNode;  // Points to the top Node
public:

    Stack() :topNode(nullptr) { }
    ~Stack( );

    // Operations
    bool empty() const { return topNode == nullptr; }
    int top( ) const {return topNode->getValue( );}
    void pop( );
    void push(int v);
 };  // class Stack
```

```
/*
    File:       Stack.cpp
    Purpose:    implementation of class Stack
*/

#include "Stack.h"

    Stack::~Stack( ) {
        while(topNode !=nullptr)
            pop();
    }

    void Stack::pop( ) {
        Node *p = topNode;
        topNode = topNode->getNext( );
        delete p; // destroy the Node
    }


    void Stack::push(int v) {
        // create a new Node...
        Node *p = new Node(v);
        // ..and link it to the others
        if(empty())
                    p->setNext(nullptr);
        else
            p->setNext(topNode);

        topNode = p;
     }
```

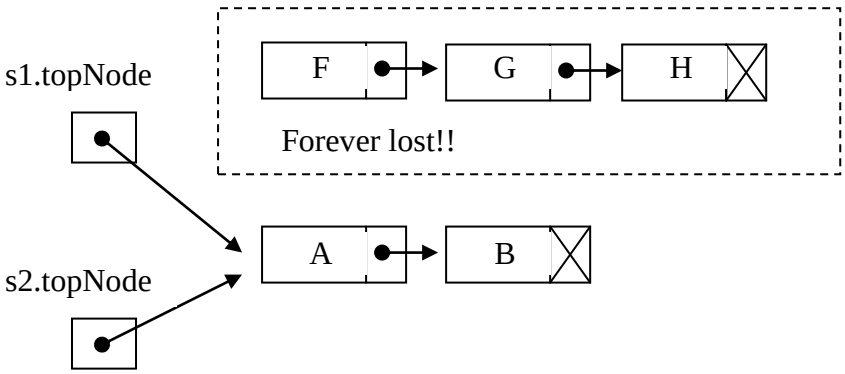
Körningsexempel:

```
Stack s1, s2
```

Aktuellt läge efter diverse push / pop …

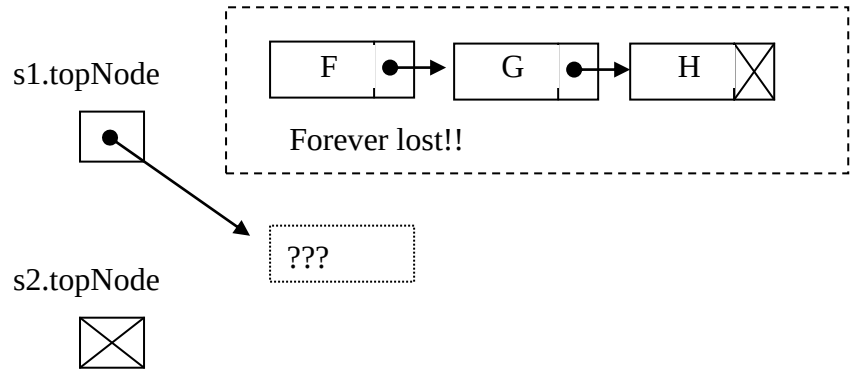



```
s1 = s2;
```



`s1` och `s2` är nu i praktiken samma stack!

```
s2.pop( );
s2.pop( );
```



`s1.pop( )` → KRASCH!!

Anledningen till misslyckandet är att default-versionen av operator = har fel *semantik* (innebörd) i detta fall, den fungerar men utför ’fel’ sak. Den gör en direkt kopiering av datamedlemmarna (memberwise copy) vilket ger en ”*shallow copy*”. Istället för att kopiera

objektet som pekaren pekar på så kopieras enbart pekaren. Pekaren `topNode` får samma värde i både `s1` och `s2`, den pekar alltså på samma Node vilket till slut leder till en krasch.

Tre problem måste lösas:

1. Tilldelningsoperatorn = måste definieras så att den utför en korrekt *'deep copy'*. Alla noder i den befintliga listan (target) ska deallokeras, därefter ska nya noder skapas och värdena kopieras från den andra listan (source).

2. Kopieringskonstrukorn måste allokera nya noder och kopiera innehållet från källobjektet.

3. Eventuellt bör jämförelseoperatorn == defineras så att två stackobjekt betraktas som lika endast om de innehåller samma värden i samma ordning.

Pseudokod för operator= vid anropet target = source;

- avbryt om source och target är samma objekt

- deallokera alla befintliga noder i target

- följ den länkade listan som börjar i source.topNode och som slutar med NULL i sista noden. För varje nod i source:
  - skapa ny nod i target och kopiera motsvarande värde från source
  - länka noden sist i target

- returnera en referens till target för att möjliggöra en kedja med tilldelningar som t.ex. s1 = s2 = s3;

Implementation av tilldelningsoperatorn:

```
Stack & Stack::operator=(const Stack &source) {
   // Avoid assignment 'source = source'
   if(this == &source)   // Same object?
       return *this;     // We are finished!

   // Delete all nodes in this stack (target)
   while (topNode != nullptr)
       pop();
   topNode = nullptr;

   if(source.empty())   // Empty source?
       return *this;    // We are finished !

   /* OK there is at least one value to copy,
      let's create a new Node, set topNode
      to point at it and copy the value from
      source stack */

   topNode = new Node(source.topNode->value);

   // fortsättning på nästa sida!
```

```
    /* For each of the remaining Nodes in source, create
       a new Node, copy the value and link it as the last */

    // Get a pointer to the next Node in source
    Node *nextSourceNode = source.topNode->next;
    Node *lastNode = topNode; // Last in target

    while (nextSourceNode != nullptr) {
      // Create a new node and copy value from source
      lastNode->next = new Node(nextSourceNode->value);
      lastNode = lastNode->next; // Advance lastNode in target
      nextSourceNode = nextSourceNode->next; // Advance in source
    }
    return *this;
}
```

Hela implementationen finns i exStack2.zip.

”Smart pointers” i C++11

Smarta pekare är klassobjekt som inte bara beter sig som inbyggda pekare utan också hanterar deallokering av dynamiskt allokerade objekt. Du behöver själv inte explicit anropa delete - de smarta pekarna gör det åt dig vid lämplig tidpunkt. Den smarta pekaren är definierad på ett sådant sätt att den kan användas syntaktiskt nästan exakt som en inbyggd (eller "raw") pekare. Du kan oftast använda dem bara genom att använda en smart pekare där du annars skulle använt en inbyggd pekare. En smart pekare innehåller en inbyggd pekare, och definieras som en klassmall vars typparameter är typen för det utpekade objektet.
När det gäller dynamiskt allokerade objekt, talar vi ofta om vem som "äger" objektet. "Ägande" innbär inte bara möjlighet att använda objektet utan också ansvar för att deallokera det. Utan smarta pekare bestäms deallokeringen av var i koden vi placerar anropet till delete. Om vi glömmer det leder det till minnesläckage och kanske också till odefinierat och felaktigt beteende.
Smarta pekare hjälper oss att implementera ägarskapet genom att delete på den lagrade ”raw”-pekaren anropas i destruktorn. Eftersom C++ garanterar att destruktorn för ett objekt anropas när det förstörs kommer alltså delete att anropas för det utpekade objektet när smart-pointer-objektet själv förstörs.
Observera att smarta pekare kan bara användas med inbyggda pekare som returnerats från `new`. Använding med pekare till lokala stackobjekt kommer att ge programkörningsfel.

C++11 definierar tre olika smarta pekare,
`shared_ptr`, `weak_ptr` och `unique_ptr`. Alla tre är klassmallar och kräver inkludering av headerfilen `<memory>` och i denna lektion introducerar vi nu `unique_ptr`.

unique_ptr

Vi tittar på ett exempel. Funktionen `foo()` här nedan utnyttjar `unique_ptr` för att hantera

dynamiska objekt av klassen Magic:

```
class Magic {
private:
	int value;
public:
	Magic(int v = 42)
		:value(v){ }

	~Magic() { cout << "Magic::~Magic" << endl; }

	void mystify() {
		value *= 3;
	}
	int reveal() {
		return value;
	}
};

void foo() {
	unique_ptr<Magic> mp(new Magic(10)); // p owns the Magic
	mp->mystify(); // mp works as a raw pointer
	cout << mp->reveal() << endl;
} // The Magic is destroyed when mp goes out of scope
```

Anrop till `foo()` ger utskriften

```
30
Magic::~Magic
```

- Operatoröverlagringar gör att mp kan användas som om den vore en inbyggd pekare.
- Gör alltid anropet till `new` direkt i konstruktoranropet till din smarta pekare. Den inbyggda pekaren kan då aldrig ”smita iväg” och av misstag användas någon annanstans. Den finns bara i smart-pekare-objektet.
- Deallokeringen av Magic-objektet sker alltså automatiskt när anropet till `foo` är klart.

Som klassmedlem kan en unique_ptr inte initieras direkt eller i initieringslistan. Initieringen måste göras i konstruktorkroppen genom anrop till medlemsfunktionen `reset()` som deallokerar det objekt som den interna pekaren hittills pekat på (delete på `nullptr` är ofarligt) och ersätter den med den nya pekaren:

```
class MagicUser {
private:
	unique_ptr<Magic> p; // p holds a nullptr
public:
	MagicUser();
	void print() { cout << p->reveal() << endl; }
};

MagicUser::MagicUser() {
	p.reset(new Magic(20)); // Initialization
	p->mystify();
}
```

En körning av programmet

```
int main()
{
	MagicUser mu;
	mu.print();
}
```

ger som förväntat utskriften

```
60
Magic::~Magic
```

Medlemsfunktionen `reset` är överlagrad. Versionen som används i exemplet här ovan, `reset(pekare)`, deallokerar det objekt som den interna pekaren hittills (eventuellt) har pekat på (delete på `nullptr` är ofarligt) och ersätter den sedan med den pekare som skickas med i anropet.
Utan argument, `reset()`, så deallokeras det hittills hållna objektet men den interna pekare sätts därefter till `nullptr`.

För att manuellt deallokera det hållna objektet anropar man alltså `reset` utan argument.

Man kan enkelt testa om en `unique_ptr` äger ett objekt eller inte. Mallen innehåller en operator för konvertering till `bool` som returnerar `false` om den interna pekaren är `nullptr`, annars returneras `true`:

```
if (myUniquePtr)
		myUniquePtr->doSomething;
```

Namnet på klassmallen `unique_ptr` kommer från det faktum att den implementerar ett unikt ägarskap av ett dynamiskt allokerat objekt. Detta inplementeras genom att tilldelningsoperatorn och kopieringskonstruktorn är ”deletade”. Koden för detta är i princip

```
unique_ptr(const unique_ptr &) = delete;
unique_ptr &operator=(const unique_ptr &) = delete;
```

[Genom tillägget `= delete;` till en prototyp för någon av de inbyggda operatorerna eller konstruktorerna tas motsvarande funktion bort. Inga defaultversioner genereras av kompilatorn och försök till anrop ger kompileringsfel.]

Några små exempel:

```
unique_ptr<Magic> p1(new Magic); // OK, p1 owns the Macic
unique_ptr<Magic> p2(p1); // ERROR - copy construction not allowed.
unique_ptr<Magic> p3; // OK, an empty unique_ptr;
p3 = p1; // ERROR, copy assignment not allowed.
```

Eftersom kopiering inte är tillåten måste man använda referensanrop om en `unique_ptr` ska skickas som argument till en funktion.

Om man följer de rekommenderade sätten att använda smarta pekare så innebär det att man inte kan ha två `unique_ptr`-objekt som innehåller samma ”raw”-pointer och därmed gör anspråk på samma objekt. Därmed elimineras också risken för dubbel deallokering. Ett sätt att sätta dessa egenskaper ur spel är att använda medlemsfunktionen `get()` som returnerar den ursprungliga inbyggda pekare som objektet initieras med. Använd den bara i undantagsfall!

Explicit överföring av ägarskap mellan två unique_ptr

Om man verkligen vill överföra ägarskap mellan två `unique_ptr`-objekt kan man utnyttja det som i C++11 kallas ”move semantics” med funktionsmallen `std:move()`. Exempel:

```
unique_ptr<Magic> mp(new Magic(10)); // mp owns the Magic
unique_ptr<Magic> mp2;  // mp2 owns nothing
mp2 = std::move(mp); // mp2 owns the Magic, mp owns nothing
unique_ptr<Magic> mp3 = std::move(mp2);// mp3 owns the Magic,
                                       // mp and mp2 owns nothing
```

Mallen `std::move()` konverterar sitt argument till en sk. ”rvalue reference” och sedan anropoas ”move”-versionen av en kopieringskonstruktor eller tilldelningsoperator. Så kallade move-versioner av konstruktorer och operatorer, liksom begreppet ”rvalue reference” är nya ingredienser i C++ fr.o.m. C++11. Vi ska inte fördjupa oss i dessa nu utan återkommer till det senare i kursen.

Vanliga användningsområden för `unique_ptr` är

- att erbjuda säkerhet vid exception genom att garantera att deallokering sker både vid normal avslutning och vid avslutning genom ett exception. Mer om detta i lektionen som behandlar exceptions (undantag).
- att överföra ägarskap av unikt ägda, dynamiskt allokerade objekt, till och från funktioner
- som elementyp i kontainrar som implementerar move-semantics, t.ex. `std::vector`.

En överlagring av `unique_ptr` finns för dynamiskt allokerade arrayer. Den definierar operator[] så att indexering kan utföras. Arrayen deallokeras när `unique_ptr`-objektet går ur scope.

Exempel

```
unique_ptr<int[]> arrPtr(new int[10]);
arrPtr[1] = 10;
```

`unique_ptr` och “Standard containers”

Man kan fylla en “standard container” (eller inbyggd array) med `unique_ptrs`:

- Om man tar bort (erase) en `unique_ptr` från kontainern så förstörs denna `unique_`ptr och det hållna objektet deallokeras.
- Om man tar förstör hela kontainern (eller anropar `clear`) så så förstörs alla dess `unique_`ptrs och de hållna objekten deallokeras.

- Om man flyttar ägandet av objekten ut från kontainerns `unique_`ptrs så blir det tomma `unique_`ptrs kvar i kontainern. Man måste då komma ihåg att testa om dessa pekar på något objekt innan man derefererar dem.

Ett exempel:

```
vector<unique_ptr<Magic>> vec; // vector för unique_ptr<Magic>
// Allokera en Magic till en unique_ptr i vectorn
vec.push_back(unique_ptr<Magic>(new Magic(100)));
vec.push_back(unique_ptr<Magic>(new Magic(200)));
cout << vec[0]->reveal() << endl;
cout << vec[1]->reveal() << endl;
vec[0].reset();// Deallokera Magic-objektet
vec[1].reset();
```

Utskrifter:

```
100
200
Magic::~Magic
Magic::~Magic
```

Ett andra exempel

```
vector<unique_ptr<Magic>> vec(2); // vector med två unique_ptr<Magic>
vec[0].reset(new Magic(30));  // Allokera Magic-objektet
vec[1].reset(new Magic(40));
cout << vec[0]->reveal() << endl;
cout << vec[1]->reveal() << endl;
vec[0].reset();  // Deallokera Magic-objektet
vec[1].reset();
```

Utskrifter:

```
30
40
Magic::~Magic
Magic::~Magic
```

Ett tredje exempel:

```
unique_ptr<vector<unique_ptr<Magic>>> vecPtr(new vector<unique_ptr<Magic>>(2));
```

Skapar ett `unique_ptr`-objekt, `vecPtr`, som håller en pekare till en dynamiskt skapad `vector` för 2 `unique_ptr`-objekt för klassen Magic.

```
// Allokera en Magic och låt den första unique_ptr hålla pekaren
vecPtr->at(0).reset(new Magic(10));

// Allokera en Magic och låt den andra unique_ptr hålla pekaren
vecPtr->at(1).reset(new Magic(20));

// Skriv ut innehållet i det första Magic-objektet
cout << vecPtr->at(0)->reveal() << endl;
```

```
vecPtr->at(0).reset(); // Deallokera det första Magic-objektet
if (vecPtr->at(0)) // Testa -> false
	cout << vecPtr->at(0)->reveal() << endl;

// Skriv ut innehållet i det andra Magic-objektet
cout << vecPtr->at(1)->reveal() << endl;
```

Körning av koden ger utskrifterna

```
10
Magic::~Magic
20
Magic::~Magic
```

Den första destruktor-utskriften kommer från `vecPtr->at(0).reset();`
När `vecPtr` går ur scope deallokeras objektet vilket leder till att det hållna vector-objektet deallokeras vilket i sin tur leder till att destruktorn körs för de båda `unique_ptr`-objekten. Det första innehåller nu en `nullptr` men det andra håller en pekare till sitt Magic-objekt vilket deallokeras. Detta förklarar den andra destruktor-utskriften.

RAII

RAII är en förkortning av ”Resource Allocation Is Initialization” och står för ett viktigt koncept i C++-programmering. Idén är att utnyttja att C++ garanterar att en konstruktor anropas när ett objekt skapas och att destruktorn anropas när det förstörs. För dynamiskt skapade objekt sker detta som bekant när objektet skapas med `new` respektive deallokeras med `delete`. Lokala objekt definierade inom ett visst scope, dvs. ett block mellan { och }, skapas på stacken under exekveringen av blocket och poppas av stacken när exekveringen lämnar blocket. Även här körs konstruktorn när objektet skaps och destruktorn anropas när exekveringen lämnar blocket.
RAII innebär att man gör initiering och allokering i konstruktorn och motsvarande deallokering och uppstädning i destruktorn. Fördelen är att man senare inte behöver anropa kod för deallokering och uppstädning eftersom detruktorn kommer att göra jobbet när objektet förstörs. Rätt utyttjat leder RAII till säkrare kod med mindre risk för minnesläckor.
De smarta pekarna är exempel på tillämpning av RAII. Om du gör den dynamiska allokeringen direkt i konstruktoranropet så vet du att dellokeringen kommer att ske automatiskt så småningom, t.ex.

```
unique_ptr<vector<int>>  vec(new vector<int>);
```

Ett annat exempel där RAII används för spara den numeriska formatteringen för cout för att senare få den automatiskt återställd:

```
class CoutFormatSaver  {
public:
	CoutFormatSaver() :
		// save current IO flags and precision
		old_flags(cout.flags()), old_precision(cout.precision())
	{}

	~CoutFormatSaver() {
		// restore saved IO flags and precision
		cout.flags(old_flags);
		cout.precision(old_precision);
```

```
        }
private:
        ios::fmtflags old_flags;
        int old_precision;
};

// usage: foo needs to change the precision, etc, but caller needs them
// to be whatever they were upon return. Create an object before
// changing the settings; will automatically restore upon return.
void foo(double x1, double x2)
{
        CoutFormatSaver stateSaver;
        cout << fixed << setprecision(2) << x1 << endl;
        cout << setprecision(8) << x2 << endl;
}
```

RAII-konceptet är också viktigt i samband med undantagshantering. Vi återkommer till detta när exceptions tas upp.