Fakulteten för ekonomi, kommunikation och IT

Avancerad programmering i C++

Laboration 1

Datum: 2007-05-10
Namn: Henrik Bäck
Mathias Andersson
Kurs: DAV C05

Innehållsförteckning

Bakgrund och syfte

Livstiden för ett dataobjekt i C++ beror främst på två saker. Detta är vilket Scope ett dataobjekt har skapats i men även i vilket minnesområde objektet har allokerats.

I C++ finns det fyra olika Scope där dataobjekt kan deklareras. Dessa är i ett NameSpace, en klass, ett lokalt eller det globala scopet.

Vad det gäller programspråket C++ finns det tre olika områden där dataobjekt kan allokeras. Dessa objekt är allmänt kända som Stacken, Heapen och det Statiska minnesutrymmet. Beroende på vilken metod i C++ som används för att skapa dataobjektet så kommer det att befinnas sig på olika plats i minnet.

Utförande

För att kunna avgöra ett dataobjekts livstid i C++ fastställdes först samtliga testfall som kan uppstå. Efter detta så skapades en testmiljö bestående av ett C++-program. Mjukvaran var simpel i sitt utförande och dess utseende var lite annorlunda från testfall till testfall.

Testmjukvaran

För att kunna avgöra när ett dataobjekt skapas och dör är det viktigt att kunna få ut denna information från mjukvaran. Om en standardiserad datayp, exempelvis `char` eller `int`, skulle användas för att representera detta skulle information om när objektet skapas och dör inte kunna erhållas.

För att komma förbi detta har en standardiserad datatyp kapslas in i en klass. Denna klass kan man sedan använda som en ny datatyp. Till denna klass adderar man då en konstruktör och en destruktör vilket ger stora fördelar. Eftersom en konstruktör alltid körs när objektet skapas kan man låta denna skriva ut information om detta, samma sak gäller destruktören.

Genom att lägga upp testmjukvaran på detta sätt kan man använda klassen för att se när ett objekt skapas och raderas från minnet i förhållande till övrig programkod. Genom at också för varje test lägga in utskrift om var i exekveringen man befinner sig kan man tydligt se när ett objekt skapas och sedan inte längre finns med.

Testfall

Det finns totalt 12 stycken fall vilka följer av de fyra Scope som finns samt de tre olika minnesområdena som C++ kan allokera minne i. Vilka fall som finns åskådliggörs i tabellen på nästa sida, denna tabell innehåller kortfattad information om vad varje fall innebär.

En mer utförlig beskrivning av varje testfall och dess utfall återfinns längre ned i denna underrubrik.

		Minnesområde		
		Run-Time Stack	**Static memory**	**Heap**
Scope	**NameSpace**	Ej definierat.	Statiska dataobjekt deklarereade direkt i namnormårdet med `static`.	Dataobjekt som skapats med hjälp av en pekare och nyckelordet `new`.
	Class	Alla dataobjekt i klassen som deklareras i privat eller public utan `static` eller `new`.	Statiska dataobjekt i klassen med hjälp av nyckelordet `static` i privat eller public.	Dataobjekt som skapats med en pekare och nyckelordet `new` i privat eller public.
	Local	Alla dataobjekt som deklareras inuti ex. en funktion utan `static` eller `new`.	Statiska dataobjekt deklarerade inuti ex. en funktion med `static`.	Dataobjekt som skapats med en pekare och nyckelordet `new`.
	Global	Ej definierat.	Statiska dataobjekt deklarerade inuti ett nytt scope med `static`.	Dataobjekt som skapats i ett nytt scope med hjälp av en pekare och nyckelordet `new`.

Tabell 1 – Alla testfall

Fall – NameSpace och Run-Time Stack

Det här fallet existerar inte, det går inte att allokera dataobjekt i Run-Time Stacken i ett NameSpace.

Fall – NameSpace och Static Memory

Detta fall uppträder när ett godtyckligt dataobjekt statiskt deklareras direkt i namnområrdet på följande sätt

```
namespace Namn
{
            static dataTyp objekt;
}
```

När det gäller livstiden för ett sådant här objekt är det ingen skillnad alls från att deklarera dataobjektet utan static eftersom objektet kommer att vara ett icke-lokalt objekt. Dock är det så att detta objekt kommer att allokeras i det statiska minnet istället för Run-Time Stacken. Objektet kommer att leva ända till exekveringen avslutas. Detta på grund av att objektet är deklarerat statiskt.

Slutsats för detta fall: Objektet allokeras i det statiska minnet när det körs första gången men försvinner inte ur minnet förrän exekveringen upphör.

Fall – NameSpace och Heap

Detta fall uppträder när ett objekt skapas under körning. En pekare till objektet läggs upp och därefter skapas objetet med nyckelordet `new` enligt följande

```
namespace Namn
{
            dataTyp* objPek;
            objPek = new dataTyp;
}
```

I det här fallet kommer objektet att allokeras i det fria minnet när de först påträffas i programkoden. Det finns dock inget ställe där objektet upphör att leva, eftersom nyckelordet `delete` inte används på objektet. Objektet kommer dock in direkt att försvinna när mjukvaran har slutat att exekvera och operativsystemet rensar upp minnet efter mjukvaran.

Slutsats för detta fall: Objektet allokeras i det fria minnet när nyckelordet `new` körs men det finns inget ställe där objetet upphör att finnas i det fria minnet.

Fall – Class och Run-Time Stack

Det här fallet uppstår när ett privat eller publikt dataobjekt skapas i en klass. Sedan skall även denna klass skapas med hjälp av att från någon stans i mjukvaran deklarera utan att använda `static` eller `new`. Detta skulle kunna göras som nedan.

```
class testKlass
{
            public:
            dataTyp nyttObjekt;

}

int main()
{
            testKlass nyttTest;

            return 0;
}
```

Detta kommer att resultera i att dataobjektet deklarerat i `public` allokeras när instansen av klassen skapas. Objektet kommer att raderas från minnet samtidigt som `nyttTest`-objektet inte längre lever. När det slutar leva beror på när och hur det är deklarerat. I detta fall när `main` har körts klart.

Slutsats för detta fall: Dataobjektet inuti klassen allokeras i Run-Time Stacken och kommer att ligga kvar där tills instansen av klassen inte längre körs.

Fall – Class och Static Memory

I det fallet skapas ett dataobjekt i en klass med hjälp av nyckelordet `static`. Dessutom definieras dataobjektet utanför klassen enligt exemplet nedan.

```
class testKlass
{
            private:
            static dataTyp objekt;
}

dataTyp testKlass::objekt;
```

Vid programkörning kommer minne allokeras för `objekt` i det statiska minnesområdet. Dataobjektet `objekt` kommer att ligga kvar i det statiska minnet ända till exekveringen av programmet upphör. Detta trots att eventuella instanser av klassen `testKlass` upphör existera tidigare.

Slutsats för detta fall: Dataobjekt allokeras i det statiska minnesområdet och upphör inte att existera förrän programmera har exekverat klart.

Fall – Class och Heap

Det här fallet är likt fallet för Run-Time Stack till skillnad från att dataobjektet i klassen nu deklareras med en pekare. I konstruktören för klassen lägges en allokering av objektet.

```
class testKlass
{
            private:
            dataTyp* nyObj;


            public:
            //Konstruktör
            testKlass(){nyObj = new dataTyp;}
}

int main()
{
            testKlass nyttTest;

            return 0;
}
```

Trots sättet att skapa en instans av klassen kommer objektet som av klassen allokeras i det dynamiska minnet att finnas kvar ända till programmet har körts klart. Objektet allokeras på Heap när raden som skapar en instans av klassen först uppträder. Oavsett om instansen av klassen slutar att existera tidigare eller inte. För att åtgärda detta problem måste man i destruktören till klassen implementera ett `delete nyObj;`.

Slutsats för detta fall: Objektet allokeras i minnet av klassen men kommer aldrig att avallokeras eftersom inget `delete` körs i destruktören.

Fall – Local och Run-Time Stack

Det här fallet uppträder när ett dataobjekt deklareras inuti ex. en funktion enligt exemplet nedan. Detta är ett mycket vanligt sätt att deklarera dataobjekt på.

```
void testfunktion()
{
            dataTyp objekt;

}
```

Det här dataobjektet kommer att leva lika länge som funktionen finns i minnet. Alltså så länge programkod exekveras inuti funktionen finns även objektet kvar. När

funktionen inte längre körs så kommer objektet att raderas. Det innebär att även om funktionen anropas många gånger så kommer objektet att skapas och tas bort varje gång.

Slutsats för detta fall: Objektet allokeras i Run-Time Stacken när programkoden för det körs och försvinner samtidigt som funktionen avslutas.

Fall – Local och Static Memory

Det här fallet är mycket likt det fallet som är beskrivet ovan men med en mycket markant skillnad. I det här fallet deklareras ett dataobjekt inuti en ex. funktion med nyckelordet `static` enligt exemplet nedan.

```
void testfunktion()
{
            static dataTyp objekt;

}
```

I likhet med det förra fallet kommer även här objektet att allokeras när det först påträffas. Dock kommer objektet inte att upphöra att existera när funktionen gör det. Istället kommer objektet att finnas kvar även när funktionen har körts klart och objektets datavärden kommer att finnas kvar och vara tillgängliga varje gång funktionen anropas.

Slutsats för detta fall: Dataobjektet allokeras i det statiska minnet och upphör inte att existera förrän programmet har exekverat klart.

Fall – Local och Heap

Det här fallet uppträder när ett objekt allokeras med hjälp av nyckelordet `new` enligt exemplet nedan.

```
void testfunktion()
{
            dataTyp* objPek;
            objPek = new dataTyp;

}
```

I likhet med de övriga fallen så kommer objektet att allokeras när det fört påträffas. Det här objektet kommer som fallet föreskriver att allokeras på heap. Objektets livstid är lika långt som programmets.

Slutsats för detta fall: Objektet allokeras när det körs första gången men upphör inte att existera förrän programmet har exekverat klart.

Fall – Global och Run-Time Stack

Det här fallet existerar inte, det går inte att allokera dataobjekt i Run-Time Stacken i det globala scopet.

Fall – Global och Static Memory

Det här fallet uppträder när ett dataobjekt deklareras i det globala scopet med hjälp av nyckelordet `static`.

```
static dataTyp objekt;
```

I det här fallet så kommer objektet att deklareras i det statiska minnet när det uppträder i programkoden. Dock kommer det därefter att finnas kvar i minnet ända till programmet avslutas.

Slutsats för detta fall: Objektet allokeras när dess programkod körs och upphör inte att existera förrän programmet har slutat att exekvera.

Fall – Global och Heap

Som för de övriga fallen med heap måste objektet allokeras med nyckelordet new.

```
dataTyp* objPek;
objPek = new dataTyp;
```

I likhet med de övriga fallen så kommer objektet att allokeras när det fört påträffas. Det här objektet kommer som fallet föreskriver att allokeras på heap. Objektets livstid är lika långt som programmets.

Slutsats för detta fall: Objektet allokeras när det körs första gången men upphör inte att existera förrän programmet har exekverat klart.

Sammanfattning

Från de fallen som testats ovan kan man se många likheter. Den största likheten är fallet för allokering av dataobjekt i heapen. I samtliga fall så avallokeras aldrig minnet om programmeraren inte själv använder sig av nyckelordet `delete`.

Vad det gäller att allokera dataobjekt i det statiska minnet är fallen också ganska lika. I samtliga fall kommer objektet att leva lika länge som programmet. All data lagrad i dataobjektet kommer att finnas kvar även om objektet i sig inte längre bör existera. Dock går det komma åt dess datavärde varje gång objektet påträffas.

Det är vid allokering i Run-Time Stacken som livslängden på objekten varierar mest. Här spelar det stor roll i vilket scope objektet har deklarerats men beteendet är det samma. Objektet upphör att existera efter att det scope som det deklarerats i upphör att existera. Vad det gäller det globala scopet så upphör det tillsammans med exekveringen.

Det vill säga att man fortfarande är och exekverar kod inom just det scopet.