TDP004 Objektorienterad Programmering
Fö 5 Minneshantering

Introduktion

- Vi har tidigare nämnt dynamisk minneshantering i Fö 2.
- Pekare och referenser.
- Allokera minne och skapa objekt mha new, lämna tillbaka minne och ta bort objekt med delete.
- struct som alternativ till class
- enum för uppräkningstyper.

Pekare *

- ”A pointer points to an object”
- STL-iteratorer som vi såg i Fö 3 är en specialisering av pekarbegreppet.
- Pekare används bla för dynamisk minneshantering.
- Ex:

```
int* p1;
float* p2;
vector<int*> * vp;        /* vp är en pekare till en vector
                             som innehåller element av
                             typen pekare till int. */
ExampleClass* myClass = new ExampleClass();
```

Pekare *, forts.

- En pekare måste inte, men bör initieras.
- Man kan kontrollera om en pekare är giltig genom jämförelse med NULL.
- NULL är en preprocessorvariabel med värdet 0.

```
if(p1 != NULL)          //Vi kan även skriva if(p1 != 0)
{
    //Förhoppningvis giltig pekare.
}
```

Pekare och const

```
const float pi = 3.14159f;
float e = 2.71828f;
const float * piPointer= &pi;
```
piPointer pekar på konstanten pi.

```
float * const ePointer = &e;
```
ePointer är const och pekar på variabeln e.

```
const float * const newPiPointer = &pi;
```
newPiPointer är const och pekar på konstanten pi

Läs från höger till vänster!

Använda pekare

- Om man har en pekare till ett objekt och vill anropa dess funktioner används notationen: variabelnamn->funktionsnamn(parametrar)

Ex:

```
ExampleClass* myClass = new ExampleClass();
myClass->func(1.1, 3.5, 5.7);
```

Ex:

- Läser medlemsvariabel med liknande syntax.

```
int id = myClass->m_ID;
```

Referens &

- Alternativ till pekare.
- Vi har använt referenser i Fö 1:

  ”Referensöverföring – call by reference

  int max(int& a,int& b)

  a och b refereras direkt, inga kopior skapas”
- Med referenser kan vi undvika att data kopieras i onödan.

Referens &, forts.

- ”A reference is an alias”
- En referens måste initieras.
- int i = 1;
- int& intRef = i;   //intRef refererar nu till i.
- All operationer på en referens är egentligen operationer på den underliggande objektet.
- ”A reference is just another name for an object”

Referens &, forts.

- *En referens får inte vara ogilitig.*
- *Det går inte att kontrollera om en referens är giltig.*

Använda referenser

- Om man har en referens till ett objekt kan man anropa dess funktioner med variabelnamn.funktionsnamn(parametrar)

Ex:

```
ExampleClass myClass2 = &myClass;
myClass2.func(1.1, 3.5, 5.7);
```

Ex:

- Läser medlemsvariabel med liknande syntax.

```
int id = myClass.m_ID;
```

Minneshantering

- Hur man skapar nya objekt och tar bort gamla objekt.
- Minneshantering står för en stor del av felen i programmen, genom att begära för mycket/lite minne och att inte lämna tillbaka det korrekt.
- Vi löser en stor del av dessa problem genom användning av STLs containerklasser.

Allokera med new

- En instans av ett objekt kan skapas med kommandot new:

```
ExampleClass* myClass = new ExampleClass();
```

- En instans kan skapas utan att använda new/pekare.

```
ExampleClass m_Class = ExampleClass();
```

- Man bör (men nästan ingen gör det) kontrollera resultatet av new. Det kan hända att new misslyckas med att allokera minne till objektet. Om new misslyckas att allokera minne så kommer std::bad_alloc exception att kastas.

Ta bort med delete

- En instans av ett objekt tas bort med kommandot delete:
  ```
  delete myClass;
  myClass = NULL;
  ```
- myClass borttagen och allt minne har lämnats tillbaka (om destruktorn i ExampleClass är korrekt).
- Dessutom kan man kontrollera om myClass är borttagen genom:

```
if(myClass == NULL)
{
   …
}
delete myClass;   /* Garanterat att det inte
                     kraschar om myClass = NULL. */
```

Minneshantering

- C++ har tre olika sorters minneshantering:
- Statiskt minne
  - Globala och namespace-variabler, static variabler.
- Automatiskt minne/stack
  - Funktionsargument och lokala variabler.
- Free store/heap
  - Allokeras med new.

Statiskt minne

- Variabler som är static, globala eller namespace-variabler allokeras automatiskt.
- ”Finns så länge du behöver dem”

Automatiskt minne / stack

- Ex: funktionsargument och lokala variabler i funktioner.
- Skapas och tas bort automatiskt.
- Automatiskt minne sägs ”vara på stacken”.
- När en funktion anropas skapas utrymme på stacken, sk ”stack frame”.
- När funktionen returnerar tas ”stack frame” för den returnerande funktionen bort.
- Vi bör inte returnera referenser till lokala variabler.

Free store / heap

- Alla program har en heap (free store area) för minne.
- När en variabel skapas med new allokeras minne på heapen.
- Det är på heapen minnesläckorna finns.
- Ex på minnesläcka:

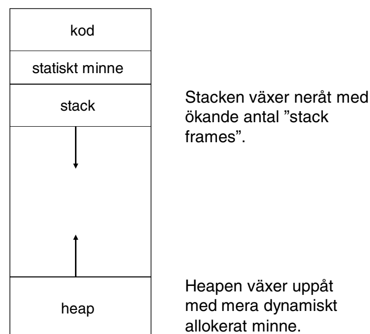
```
Test::TestExampleClass()
{
    ExampleClass* classPointer = new ExampleClass();
    classPointer->DoSomething(…)
}
```

  - delete classPointer fattas.
  - Minnet som allokerats med new kommer att läcka.

Minnesregel

- Vi behöver enbart ta bort minne med delete om vi har allokerat det med new.
- Vanliga variabler, vector, list etc tas bort automatiskt.
- Om vi istället hade använt gammaldags arrayer [] vilket allokeras med new måste denna tas bort i destruktorn/slutet av funktionen för lokala variabler.

Översiktlig bild över minnet



Stacken växer neråt med ökande antal ”stack frames”.

Heapen växer uppåt med mera dynamiskt allokerat minne.

struct

- Som alternativ till class finns även struct som alternativ att skapa objekt.
- Skillnaden är att alla funktioner/variabler i en struct har *public* tillgänglighet som default, medan den är *private* i class.
- Struct är lämpligt för mindre dataposter där allt ska vara offentligt.
  - Möjligt att specificera åtkomst på samma sätt som för class.
- Annars brukar class användas.
- I övrigt fungerar struct på samma sätt som class.

enum

- enum möjliggör uppräkningstyper.
- Exempel: Definiera färger på kort i en kortlek.

```
enum CardSuite
{
    Clubs,
    Diamonds,
    Spades,
    Hearts
};
```

enum, exempel

```
struct Card
{
    Card(CardSuite suite, CardValue value)
     : m_suit(suite), m_value(value) { }
    const CardSuite m_suite;
    const CardValue m_value;
};
```

- Direkta jämförelser mellan enum-variabler möjliga.

```
Card a = Card(Clubs, Two);
Card b = Card(Hearts, Three);
if (a.m_value > b.m_value)
{
…
}
```

enum, forts.

- Varje värde i en enum har ett numeriskt värde. Olika enum kan ha samma värde.
- Detta kan definieras av användaren.
- Om det numeriska värdet ej definieras börjar det med 0 för den första värdet o.s.v.

Kort om exceptions

- exception = exceptionell händelse, allvarligt fel
- Exempel

```
ExampleClass* myClass = NULL;
try
{
    myClass = new ExampleClass();
}
catch(std::bad_alloc e)     //Fånga endast bad_alloc
{
    //Hantera std::bad_alloc här.
}
```

- catch(…) fångar alla sorters exception
- Med throw() kan programmeraren kasta exceptions

Sammanfattning

- Pekare, *, -> notation.
- Referens, &, . notation.
- Skapa objekt med new.
- Ta bort objekt med delete.
- struct är ett alternativ till class, för mindre dataposter där all data ska vara offentlig.
- enum för uppräkningar.