# Typparameterisering

# Typparameterisering (Templates)

## Templates TemplatesTemplatesTemplates

C++-programmeraren konstruerar ibland funktioner och klasser som liknar varandra. Detta är ofta fallet när man överlagrat en funktion och denna finns i versioner som skiljer sig åt endast i avseende på argumentens typ men är identiska i avseende på funktionalitet.

Med hjälp av **templates** behöver programmeraren endast skriva **en** sådan funktion. Denna får sedan fungera som mall (template). Kompilatorn genererar sedan en version för varje typ av anrop eller explicit deklaration som programmeraren gjort.

Templates (eller typparameterisering) är ett tillägg till C++ som finns i senare implementationer (2.1-) av C++. Det är därför inte säkert att din C++-kompilator kan använda templates.

# Template för funktioner.1  Template för funktioner.1 Template för funktioner.1  Template för funktioner

**template** är ett reserverat ord i C++. Med hjälp av template kan man använda en **typ som parameter**.

## Definition av templatefunktion

För att definiera en funktion som template gör man på följande sätt:

```
template <class Type>
void prompt(char *prompter, Type& value)
{
   cout.width(30);
   long old = cout.setf(ios::adjustfield,ios::left);
   cout << prompter<< ":";
   cout.setf(old);
   cin  >> value;
}

template <class Type>
void display(char *prompter, Type& value)
{
   cout.width(30);
   long old = cout.setf(ios::adjustfield,ios::left);
   cout << prompter<< ":" << value << endl;
   cout.setf(old);
}
```

Lägg märke till hur man använder typen som argument ("class Type" är "formell parameter") .

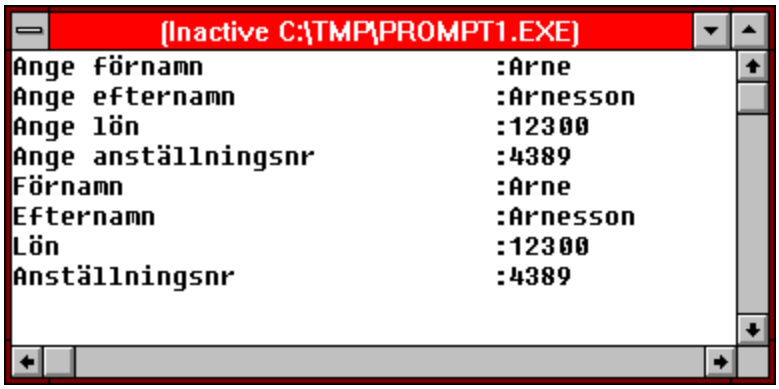
## Anrop av templatefunktion

För att sedan använda en template för en funktion räcker det att man anropar funktionen. Kompilatorn genererar funktioner som matchar anropen (enligt den mall (template) som angetts):

```
char   fnamn[50];
char   enamn[50];
double lon;
int    nr;

prompt("Ange förnamn",fnamn);
prompt("Ange efternamn",enamn);
prompt("Ange lön",lon);
prompt("Ange anställningsnr",nr);

display("Förnamn", fnamn);
display("Efternamn", enamn);
display("Lön", lon);
display("Anställningsnr", nr);
```

Kompilatorn genererar kod som matchar anropen. Som argument kan man även använda egendefinierade typer (klasser).

## Resultat



## Att använda egendefinierade typer som argument till templatefunktioner

Antag att vi har definierat klassen **Record** som nedan. Lägg märke till att vi har gett klassen en jämförelseoperator:

```
class ostream;
class Record {
    friend ostream& operator<<(ostream& os,const Record& rec);
public:
    Record(Streng);
    int operator >(const Record& rec) const;
private:
    Streng  namn;
};
```

Nedan visar vi hur klassens medlemmar är definierade:

```
Record::Record(Streng n)
:namn(n)
{
}
int Record::operator >(const Record& rec) const
{
    return strcmp(namn,rec.namn) > 0;
}
ostream& operator<<(ostream& os,const Record& rec)
{
    os << rec.namn;
}
```

## Funktionsdefinitioner

Antag sedan att vi har definierat de tre templatefunktionerna **swap,sortera** och **skrivut** som nedan:

```
template <class Type>
void swap(Type& i,Type& j)  {
   Type temp = i;
   i = j;
   j = temp;
}
template <class Type>
void sortera( Type faelt[], unsigned antal)
{
     for(register unsigned  i=0;i<antal-1;i++)
     {
       for( register int j=i+1;j<antal;j++)
       {
           if ( faelt[i] > faelt[j])
           {
               swap(faelt[i],faelt[j]);
           }
       }
     }
}
template <class Type>
void skrivut(Type faelt[], unsigned antal)
{
   cout << "*********** Tabell ***********\n";
   for(register unsigned i=0;i<antal;i++)
   {
       cout << "(" << (i+1) << ") " << faelt[i] << endl;
   }
}
```

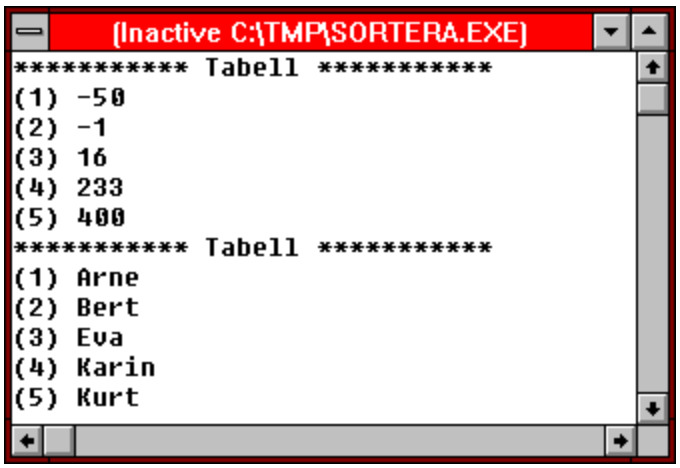
## Användning

Sedan kan vi använda vår template på nedanstående sätt:

```
main()
{
    int    ifield[]  = { -1 , -50 , 400 , 233 , 16 };
    Record sfield[] = { "Kurt","Arne" ,"Eva","Bert","Karin"};

    sortera(ifield,sizeof(ifield)/sizeof(ifield[0]));
    sortera(sfield,sizeof(sfield)/sizeof(sfield[0]));
```

```
    skrivut(ifield,sizeof(ifield)/sizeof(ifield[0]));
    skrivut(sfield,sizeof(sfield)/sizeof(sfield[0]));

   return 0;
}
```

## Resultat




En förutsättning för att vi skall kunna använda en viss typ som argument till vår template-funktion är att den går att använda på det sätt som definieras i funktionen.

I ovanstående exempel krävs att det går att använda jämförelseoperatorn  > för objekt av den typ som används i sorteringsfunktionen. I utskriftsfunktionen krävs att vi har definierat <<-operatorn för typen i fråga.

# Klass som template.3 Klass som template.3 Klass som template.3 Klass som template

Det är också möjligt att deklarera en **templateklass**. Denna kan sedan vara mall för en viss typ av klasser. Så här deklareras en **templateklass**:

## Deklaration av templateklass

```
template <class Type>
class Value {
    friend ostream& operator<< (ostream& os,const Value<Type>&);
    friend istream& operator>> (istream& is,Value<Type>&);
public:
    Value();
    Value(Type init);
    void print();
    operator Type();
private:
    Type value;
};
```

## Definition av klassmedlemmar i templateklass

Nedan visar vi hur medlemmarna skall definieras om vi gör det utanför klassdefinitionen:

```
template <class Type>
Value<Type>::Value()
:value(0)
{
}
template <class Type>
Value<Type>::Value(Type init)
:value(init)
{
}
template <class Type>
Value<Type>::operator Type()
{
    return value;
}
template <class Type>
ostream& operator<<(ostream& os,const Value<Type>& v)
{
    return os << v.value;
}
template <class Type>
istream& operator>>(istream& is,Value<Type>& v)
{
    return is >> v.value;
}
```

## Definiera specialfall av template

Man kan definiera specialfall av en template. Antag att vi skulle vilja använda en **char*** som parameter till template och i så fall hantera detta fall separat.

```
class Value <char*> {
    friend ostream& operator<<(ostream& os,const Value<char*>&);
    friend istream& operator>>(istream& is,Value<char*>&);
public:
    Value<char*>();
    Value<char*>(char* init);
    ~Value();
    void print();
    operator char*();
private:
    char* value;
};

Value<char*>::Value()
:value(0)
{
}
Value<char*>::Value(char* init)
:value(strcpy(new char[strlen(init)+1],init))
{
}

Value<char*>::~Value()
{
    delete [] value;
}

Value<char*>::operator char*()
{
    return value;
}

ostream& operator<<(ostream& os,const Value<char*>& v)
{
    return os << v.value;
}

istream& operator>>(istream& is,Value<char*>& v)
{
    return is >> v.value;
}
```

## Använda templateklass

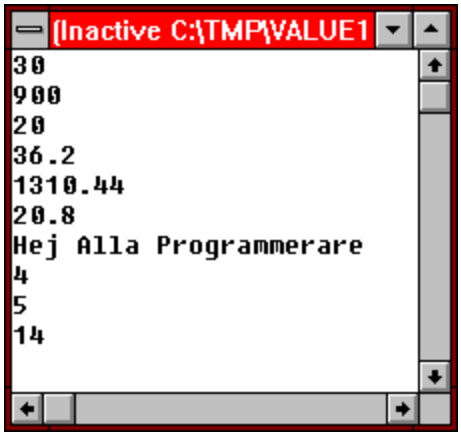
För att använda ovanstående templateklass gör man på nedanstående sätt, dvs "anropar" templateklassen med en typ som parameter (t.ex. int). Vi kan t.ex. använda vår mall när vi definerar typ på returvärden från en funktion, typ för formella parametrar samt lokala ellr globala variabler:

```
Value<int> funk(Value<int> a)
{
     Value <int> resultat;
     resultat = a * a;
     return resultat;
}
Value<double> funk(Value<double> d)
{
     Value <double> resultat;
     resultat = d * d;
     return resultat;
}
Value<int> funk(Value<char*> string)
{
     return strlen(string);
}
template <class Type>
Type max(Type a, Type b)
{
     return a > b ? a : b;
}
```

Nu kan vi anropa vår templateklass och våra templatefunktioner:

```
main()
{
     Value <int> a = 10;
     Value <int> b = 20;
     Value <int> c = a + b;

     cout << c <<  endl;
     cout << funk(c) << endl;
     cout << max(a,b) << endl;


     Value <double> d = 15.4;
     Value <double> e = 20.8;
     Value <double> f = d + e;

     cout << f <<  endl;
     cout << funk(f) << endl;
     cout << max(d,e) << endl;

     Value <char*> s1 = "Hej ";
     Value <char*> s2 = "Alla ";
     Value <char*> s3 = "Programmerare ";

     cout << s1 << s2 << s3 << endl;

     cout << funk(s1) << endl;
     cout << funk(s2) << endl;
     cout << funk(s3) << endl;

     return 0;
}
```

## Resultat


(Inactive C:\TMP\VALUE1
30
900
20
36.2
1310.44
20.8
Hej Alla Programmerare
4
5
14

Genom att vi gör anropet "Value <int> a" genereras nedanstående klass samt motsvarande medlemsfunktioner:

```
class Value {
      friend ostream&
            operator<<(ostream& os, const Value<int>&);
      friend istream& operator>> (istream& is,Value<int>&);
public:
      Value();
      Value(int init);
      void print();
      operator int ();
private:
      int value;
};
Value<int>::Value()
:value(0)
{
}

Value<int>::Value(int init)
:value(init)
{
}
Value<int>::operator int()
{
      return value;
}
```

Genom att vi gör anropet "**Value <double> d;**" genereras nedanstående klass samt motsvarande medlemsfunktioner:

```
class Value {
      friend ostream&
            operator<<(ostream& os, const Value<double>&);
      friend istream& operator>> (istream& is,Value<double>&);
public:
      Value();
      Value(double init);
      void print();
      operator double ();
private:
      double value;
};
Value<double>::Value()
:value(0)
{
}

Value<double>::Value(double init)
:value(init)
{
}
Value<double>::operator double()
{
      return value;
}
```

Genom att vi gör anropen "**cout** << **c** << **endl;**","**cout** << **c** << **endl;**" och "**cout** << **s1** << **endl;**" genereras nedanstående funktoner:

```
ostream& operator<<(ostream& os,const Value<double>& v)
{
	return os << v.value;
}

ostream& operator<<(ostream& os,const Value<int>& v)
{
	return os << v.value;
}
ostream& operator<<(ostream& os,const Value<char*>& v)
{
	return os << v.value;
}
```

Genom att vi gör nedanstående anrop:

```
	...
	cout << max(a,b) << endl;
	...
	cout << max(d,e) << endl;
	...
```

Genereras nedanstående funktioner:

```
Value<int> max(Value<int> a,Value<int> b)
{
	return a > b ? a : b;
}

Value<double> max(Value<double> a,Value<double> b)
{
	return a > b ? a : b;
}
```

# Templateklasser och aggregat

En templateklass kan anropas med flera typer som argument. Vissa av parametrarna kan som nedan användas för att definiera typen för ett attribut (datamedlem) som också är en template. I nedanstående exempel låter vi ett objekt av typen **C** bestå av ett objekt av typ **A** samt ett objekt av typ **B**.

## Klassdefinitioner

Vi visar först hur klasserna definieras. Vi har här valt att genomgående definiera medlemmarna som inlinefunktioner:

```
// tempaggr.cpp
#include <iostream.h>
// multipelt arv med templates
template <class Type>
class A {
public:
     A(Type init):value(init) {}
     Type get() { return value;}
     void print() { cout << value << endl;}
private:
     Type value;
};
template <class Type>
class B {
public:
     B(Type init):value(init) {}
     Type get() { return value;}
     void display() { cout << value << endl;}
private:
     Type value;
};
template <class AType,class BType>
class C {
public:
     C(AType ainit, BType binit)
     :avalue(ainit),bvalue(binit) {}
     void display() {
          avalue.print();
          bvalue.display();
        }
private:
     A<AType> avalue;
     B<BType> bvalue;
```

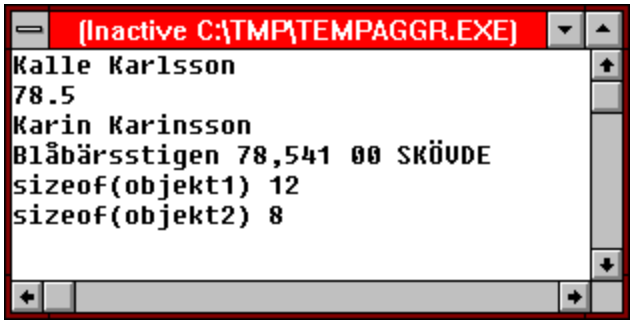
```
};
```

## Användning

Nedan visar vi hur man anropar templateklassen. Lägg märke till att **objekt1** och **objekt2** är av olika typ. För att visa detta prövar vi **sizeof**-operatorn på respektive objekt.

```
main()
{
      C <char*,double> objekt1("Kalle Karlsson",78.5);
      C <char*,char*> objekt2("Karin Karinsson",
                              "Blåbärsstigen 78,541 00 SKÖVDE");

      objekt1.display();
      objekt2.display();

      cout << "sizeof(objekt1) " << sizeof(objekt1) << endl;
      cout << "sizeof(objekt2) " << sizeof(objekt2) << endl;

        return 0;
}
```

## Resultat



# Templateklasser och arv

En templateklass kan anropas med flera typer som argument. Dessutom kan vi använda andra argument av godtycklig typ (se **Gamma** nedan). Vissa av parametrarna kan som nedan användas för att anropa en basklass som också är en template. I nedanstående exempel låter vi **Gamma** ärva av **Beta** som får ärva av **Alfa**.

## Klassdefinitioner

Vi visar först hur klasserna definieras.

```
// temparv.cpp
#include <iostream.h>
// arv med templates

template <class Type>
class Alfa {
public:
     Alfa(Type init);
     void show();
private:
     Type value;
};
template <class AType,class BType>
class Beta : public Alfa <AType> {
public:
     Beta(AType ainit,BType binit);
     void print();
private:
     BType value;
};

template <class AType,class BType, class GType , int
idnr>
class Gamma : public Beta <AType,BType> {
public:
     Gamma(AType ainit, BType binit, GType cinit);
     void display();
     static int ClassId();
private:
     GType value;
        static int id;
};
```

## Definition av medlemsfunktioner

Nedan visar vi hur man definierar medlemsfunktionerna i **Alfa,Beta** och **Gamma**. Lägg speciellt märke till hur vi anropar basklassernas konstruktorer och hur vi definierar en **static**-datamedlem i **Gamma**.

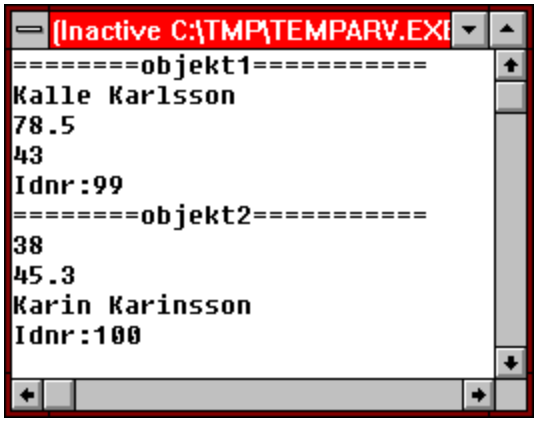
```
// Klassen Alfas funktionsdefinitioner
template <class Type>
Alfa<Type>::Alfa(Type init)
:value(init)
{
}
template <class Type>
void Alfa<Type>::show()
{
      cout << value << endl;
}
// Klassen Betas funktionsdefinitioner
template <class AType,class BType>
Beta<AType,BType>::Beta(AType ainit,BType binit)
: Alfa<AType>(ainit),value(binit)
{
}
template <class AType,class BType>
void Beta<AType,BType>::print()
{
      cout << value << endl;
}
// Klassen Gammas funktionsdefinitioner
// Definition av static medlem...
template <class AType,class BType, class GType , int idnr>
int Gamma<AType,BType,GType,idnr>::id = idnr;

template <class AType,class BType, class GType , int idnr>
Gamma<AType,BType,GType,idnr>::Gamma(AType ainit, BType binit, GType
cinit)
:Beta<AType,BType>(ainit,binit) ,value(cinit)
{
}
template <class AType,class BType, class GType ,int idnr>
void Gamma<AType,BType,GType,idnr>::display()
{
      cout << value << endl;
}
template <class AType,class BType, class GType ,int idnr>
int Gamma<AType,BType,GType,idnr>::ClassId()
{
      return id;
}
```

## Anrop

Nedan visar vi hur man anropar templateklassen **Gamma**. Observera att **objekt1** och **objekt2** nedan är av olika typ beroende på hur vi anropar **Gamma** för respektive objekt. Lägg speciellt märke till hur vi anropar en **static**-funktion i respektive klass

```
main()
{
      Gamma <char*,double,int,99>
                              objekt1("Kalle Karlsson",78.5,43);

      cout << "========objekt1===========\n";
      objekt1.show();
      objekt1.print();
      objekt1.display();

      // Anrop av static medlemsfunktion...
      cout << "Idnr:"
            << Gamma<char*,double,int,99>::ClassId() << endl;

      Gamma <int,double,char*,100>
                        objekt2(38,45.3,"Karin Karinsson");

      cout << "========objekt2===========\n";
      objekt2.show();
      objekt2.print();
      objekt2.display();

      // Anrop av static medlemsfunktion...
      cout << "Idnr:"
            << Gamma<int,double,char*,100>::ClassId() << endl;

      return 0;
}
```

## Resultat



# Templateklasser och multipelt arv

En templateklass kan anropas med flera typer som argument. Dessutom kan vi använda andra argument av godtycklig typ (se **Amfibie** nedan). Vissa av parametrarna kan som nedan användas för att anropa en basklass som också är en template. I nedanstående exempel låter vi **Amfibie** ärva av både **Car** och **Boat**. Vi använder alltså multipelt arv.

## Klassdefinitioner

Vi visar först hur klasserna definieras. Vi har här valt att genomgående definiera medlemmarna som inlinefunktioner. Lägg märke till hur klassen **Amfibie** definierar sina två basklasser samt hur basklassernas konstruktorer anropas.

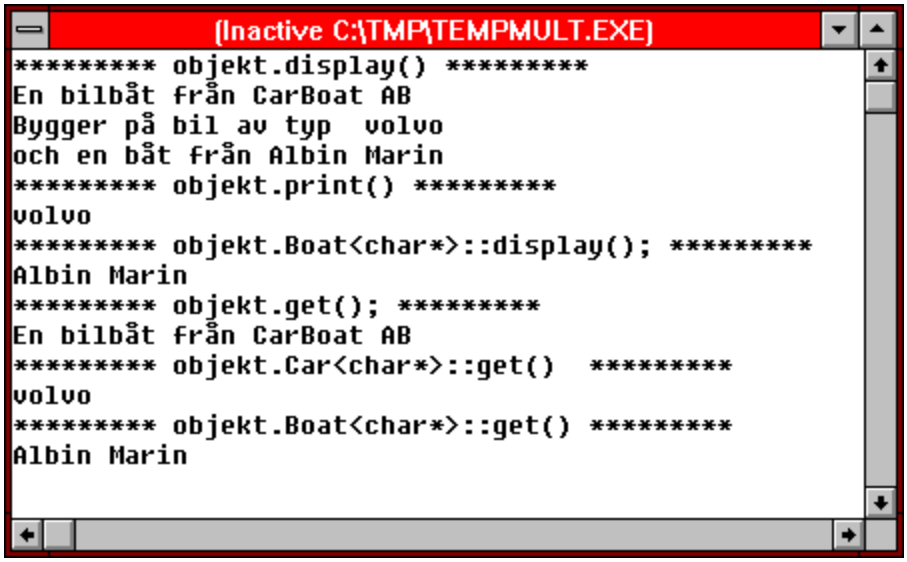
```
// tempmult.cpp
#include <iostream.h>
// template med multipelt arv
template <class Type>
class Car {
public:
      Car(Type init):value(init) {}
      Type get() { return value;}
      void print() { cout << value << endl;}
private:
      Type value;
};
template <class Type>
class Boat {
public:
      Boat(Type init):value(init) {}
      Type get() { return value;}
      void display() { cout << value << endl;}
private:
      Type value;
};
template <class AmfibieType, class CarType,class BoatType>
class Amfibie : public Car <CarType> , public Boat <BoatType>{
public:
      Amfibie(AmfibieType init, CarType ainit, BoatType binit)
      :Car<CarType>(ainit),Boat<BoatType>(binit),value(init) {}
      AmfibieType get() { return value;}
      void display() {
            cout << value << endl;
                cout << "Bygger på bil av typ  ";
            Car<CarType>::print();
                cout << "och en båt från ";
            Boat<BoatType>::display();
        }
private:
      AmfibieType value;
};
```

## Anrop

Nedan visar vi hur man anropar templateklassen **Amfibie**. Lägg märke till hur man anropar metoder i basklasserna som ärvs men döljs av samma funktionsnamn i klassen **Amfibie**.

```
main()
{
       Amfibie <char*,char*,char*>
                        objekt("En bilbåt från CarBoat AB",
                               "volvo",
                               "Albin Marin");

      cout << "********* objekt.display() *********\n";
      objekt.display();

      cout << "********* objekt.print() *********\n";
      objekt.print();

      cout <<
        "********* objekt.Boat<char*>::display(); *********\n";
      objekt.Boat<char*>::display();

      cout << "********* objekt.get(); *********\n";
      cout << objekt.get() << endl;

      cout << "********* objekt.Car<char*>::get()  *********\n";
      cout << objekt.Car<char*>::get() << endl;

      cout << "********* objekt.Boat<char*>::get() *********\n";
      cout << objekt.Boat<char*>::get() << endl;

      return 0;
}
```

## Resultat


(Inactive C:\TMP\TEMPMULT.EXE)
********** objekt.display() **********
En bilbåt från CarBoat AB
Bygger på bil av typ  volvo
och en båt från Albin Marin
********** objekt.print() **********
volvo
********** objekt.Boat<char*>::display(); **********
Albin Marin
********** objekt.get(); **********
En bilbåt från CarBoat AB
********** objekt.Car<char*>::get()  **********
volvo
********** objekt.Boat<char*>::get() **********
Albin Marin

# Övningar ÖvningarÖvningarÖvningar

## Övning 1

Gör en templateklass för en säker array med godtycklig dimension.. Dvs överlagra []-operatorn. Ge klassen defaultkonstruktor, copykonstruktor, tilldelningsoperator samt destruktor. Definiera medlemsfunktionerna utanför klassdefinitionen. Inför metoder eller klasser (iteratorklass) så att det är enkelt att iterera över objekten i arrayen.

## Övning 2

Konstruera templateklassen **SortedArray** som kan innehålla en **sorterad array** med objekt av godtycklig typ (som kan jämföras). Dimensionen skall vara dynamisk. Dvs arrayen expanderas när man lägger in fler objekt. Överlagra []-operatorn för access av objekt med hjälp av index. Ge klassen defaultkonstruktor, copykonstruktor, tilldelningsoperator samt destruktor. Definiera medlemsfunktionerna utanför klassdefinitionen. Inför metoder eller klasser (iteratorklass) så att det är enkelt att iterera över objekten i arrayen.