Mallar – templates

Generaliserade kodmallar för

- funktioner
- klasser

Mallparametrisering görs ofta med avseende på *typ*

- bred användning – långt ifrån enbart typparametriserade datastrukturer
- viktig del i objektorienterad design i C++

Mallparametrar

Tre grundläggande slag:

- typparameter (template *type parameter*) – **typename** eller **class**

```
template<typename T> class C;

template<typename T> const T& max(const T&, const T&);
```

- icke-typparametrar (template *none-type parameter*)
  - anger ett värde – heltalstyper, som size_t, är i särklass vanligast

```
template<typename T, size_t N> class Array;

template<typename T, size_t N> void swap(T (&a)[N], T (&b)[N]);
```

- mall-parameter (template *template parameter*)

```
template< template <typename T> class X > class C;
```

Mallar hanteras av kompilatorn.

- säkert – instansiering och typkontroller görs under kompilering/länkning
- effektivt – ingen dynamisk bindning eller dynamiska kontroller
- i vissa avseenden kompilatorberoende

Funktionsmallar

Vanligtvis parametrisering m.a.p. *en* formell datatyp, `T`.

- ger möjlighet att skriva funktioner som kan användas för argument av godtycklig typ.
- deklaration och definition för övrigt är som för en vanlig funktion.
- ofta ett bättre alternativ än att överlagra funktioner
  - en kodinstans
  - flexiblare
- implicit instansiering görs om ett funktionsanrop kan lösas med en instans av en funktionsmall.
- instansieringsargument kan anges explicit
- funktionsmallar kan överlagras både av andra (vanliga) funktioner och av andra funktionsmallar.
  - kräver ett visst regelverk

Mall för max-funktion

Deklaration:

```
template <typename T>
const T& max(const T&, const T&);
```

Definition:

```
template <typename T>
const T max(const T& x, const T& y)
{
   return (y < x) ? x : y;
}
```

Användning av funktionsmall

Anrop skrivs vanligtvis som för en vanlig funktion.

```
{
    int  i, j;
    …
    int  m = max(i, j);
}
```

Regler för bestämning av funktionsanrop:

1. Det finns en vanlig funktion som *exakt* överensstämmer med anropet
- använd den

2. Det finns en funktionsmall som kan instansieras för *exakt* överensstämmelse med anropet
- generera en funktion ur mallen och använd den

3. Vanliga funktioner i kombination med automatisk typomvandling prövas
- om en vanlig funktion hittas och är en unik bästa match används den i kombination med implicit typomvandling av argument

4. Om ingen funktion hittats enligt 1, 2 eller 3 är anropet ett fel.
- det finns ingen matchande funktion, eller
- anropet är tvetydigt

Om en funktionsmall inte kan instansieras för en viss datatyp är det inget kompileringsfel – SFINAE (substitution failure is not an error)

Exempel

Antag att det inte finns någon vanlig funktion som heter max.

```
int main()
{
   int    i, j, k;
   double a, b, c;

  …

  i = max(j, k);              // int-instans (regel 2)

  …

  a = max(b, c);              // double-instans (regel 2)

  …

  a = max(b, i);              // Fel! (regel 3)

  …

  a = max<double>(b, i);      // Tillåtet (explicit instans + omvandling)
  …
}
```

Instans av funktionsmall – genererad funktion

I föregående exempel skapades två instanser av `max`.

```
const int& max(const int& x, const int& y)
{
   return (y < x) ? x : y;
}

const double& max(const double& x, const double& y)
{
   return (y < x) ? x : y;
}
```

- dessa hanteras på något sätt av kompilatorn

Enda kravet på instansieringstypen:

- **`operator`**< måste vara definierad

Överlagring för specialfall

En vanlig, överlagrad funktion kan definieras om exempelvis

- en funktionsmall inte fungerar för en viss datatyp
- en effektivare implementering kan hittas för en viss datatyp

Pekare kan typiskt vara problem.

- till exempel C-strängar (**char***):

```
#include <cstring>

const char* max(const char* x, const char* y)
{
   if (std::strcmp(y, x) < 0)
      return x;
   return y;
}
```

- exakt överensstämmelse går före instansiering – *regel 1*

```
const char* s = max("foo", "bar");
```

Klassmall – Array

```
template<typename T, std::size_t N>
class Array {
public:
    // Kompilatorgenererade speciella medlemsfunktioner är bra.

    void fill(const T& value);

    size_t size() const;
    size_t max_size() const;
    bool   empty() const;

    T&       operator[](std::size_t n);
    const T& operator[](std::size_t n) const;
    …

    T&       front();
    const T& front() const;
    …

    void swap(Array<T, N>& other);

private:
    T elems_[N ? N : 1];
};
```

- containrar kan vanligtvis ha storlek 0 – det bör även Array kunna ha
  - `N` ska kunna vara 0 men dimension 0 är inte tillåtet för fält – måste vara > 0

```
// Byta innehåll med annan Array

template<typename T, std::size_t N>
void swap(Array<T, N>& x, Array<T, N>& y);

// Jämföra Arrayer (==, !=, <, <=, > och >=)

template<typename T, std::size_t N>
bool operator==(const Array<T, N>& x, const Array<T, N>& y);

template<typename T, std::size_t N>
bool operator!=(const Array<T, N>& x, const Array<T, N>& y);

template<typename T, std::size_t N>
bool operator<(const Array<T, N>& x, const Array<T, N>& y);

template<typename T, std::size_t N>
bool operator>(const Array<T, N>& x, const Array<T, N>& y);

template<typename T, std::size_t N>
bool operator<=(const Array<T, N>& x, const Array<T, N>& y);

template<typename T, std::size_t N>
bool operator>=(const Array<T, N>& x, const Array<T, N>& y);

#include "Array.tcc"
```

- observera inkluderingen av tillhörande implementeringsfil `Array.tcc` sist i `Array.h`

Implementering av Array-operationer (Array.tcc)

```
template<typename T, size_t N>
void
Array<T, N>::
fill(const T& value)
{
   std::fill_n(elems_, elems_ + N, value);
}


template<typename T, size_t N>
size_t
Array<T, N>::
size() const
{
   return N;                                   // size() == max_size() == kapaciteten == N
}


template<typename T, size_t N>
bool
Array<T, N>::
empty() const
{
   return size() == 0;                         // en Array är tom endast om N == 0
}
```

```
template<typename T, size_t N>
T&
Array<T, N>::
back()
{
    if (0 < N)
        return elems_[N - 1];
    return elems_[N];                    // specialfall för N == 0, size_t är unsigned
}


template<typename T, size_t N>
T*
Array<T, N>::
data()
{
    return &elems_[0];
}
```

swap-funktionerna

```
template<typename T, size_t N>
void
Array<T, N>::
swap(Array<T, N>& other)
{
    std::swap_ranges(elems_, elems_ + N, other.elems_);
}


template<typename T, size_t N>
void
swap(Array<T, N>& x, Array<T, N>& y)
{
    x.swap(y);
}
```

Likhetsoperationerna

```
template<typename T, size_t N>
bool
operator==(const Array<T, N>& x, const Array<T, N>& y)
{
    return std::equal(x.data(), x.data() + x.size(), y.data());
}

template<typename T, size_t N>
bool
operator!=(const Array<T, N>& x, const Array<T, N>& y)
{
    return !(x == y);
}
```

Relationsoperatorerna

```
template<typename T, size_t N>
bool
operator<(const Array<T, N>& x, const Array<T, N>& y)
{
    return std::lexicographical_compare(x.data(), x.data() + x.size(),
                                        y.data(), y.data() + y.size());
}


template<typename T, size_t N>
bool
operator>(const Array<T, N>& x, const Array<T, N>& y) { return y < x; }


template<typename T, size_t N>
bool
operator<=(const Array<T, N>& x, const Array<T, N>& y) { return !(y < x); }


template<typename T, size_t N>
bool
operator>=(const Array<T, N>& x, const Array<T, N>& y) { return !(x < y); }
```

Användning av Array

```
{
   Array<double, 10> arr1;

   cout << "arr1 har storleken " << arr1.size() << '\n';

   for (size_t i = 0; i < arr1.size(); ++i)
   {
      arr1[i] = 3.1415 * (i + 1);
   }

   Array<int, 100> arr2;
   …

   Array<double, 10> arr3(arr1);
   …

   arr3 = arr1;

   if (arr1 == arr3)
   {
      cout << "arr1 är lika med arr3+n";
   }
   …
}
```

- vad krävs för att två Array-objekt ska vara typlika?

Beroende namn

Inuti en mall kan konstruktioners innebörd skilja mellan olika instanser.

- sådana konstruktioner *beror* på mallparametrarna
- speciellt *typer och uttryck* kan bero på typen och/eller värdet för mallparametrar, vilka bestäms av mallargumenten

Några exempel:

```
template <typename T>
struct C
{
   typename T::type x;                    // T::type beror på T

   int n = T::default_value;              // T::default_value beror på T

   ...
};
```

- *beroende namn* (”*dependent name*”) antas per definition *inte* vara namnet på en typ
  - `T::X` kan vara namnet på en datamedlem, en uppräknare (**enum**-värde) eller en funktion
  - för att ange typ kvalificerar man med **typename**

Kompileringsmodeller för mallar

Inkluderingsmodellen

- definitionen för mallen inkluderas i varje fil där mallen ska instansieras
- man kan fortfarande ha en klassmalldefinition på en inkluderingsfil (.h) och separata medlemsfunktionsdefinitioner på en tillhörande implementeringsfil (.tcc)
  - i sådant fall inkluderar h-filen sin implementeringsfil på slutet – `#include "???.tcc"`
- en variation tillåter att h-filen inte inkluderar tcc-filen
  - kompilatorn har regler för var tcc-filen ska eftersökas och hur den ska användas
- inkluderingsmodellen är den modell som de flesta C++-kompilatorer använder

Separatkompileringsmodellen

- i grunden den traditionella modellen med header-fil och motsvarande implementeringsfil
  - implementeringsfilen inkluderas inte av sin h-fil
  - kompileras sparat
  - svårt att implementera, få kompilatorer kan