Advanced Software Development

Templates und Template-Metaprogrammierung

Übersicht

- Varianten
- Template-Spezialisierung
- Partielle Spezialisierung
- Templates und Vererbung
- Variadic Templates
- Template-Metaprogrammierung

Varianten

- Arten von Templates
  - Template-Funktionen/Methoden
  - Template-Klassen
- Arten von Template-Parametern
  - Typparameter (*typename*/*class*)
  - Parameter mit einfachen Typen (int, bool, enum, Pointer, …)
  - Template Template Parameter

Template Template Parameter

```
template <template <typename, typename> class C>
class IntContainer {
public:
  …
private:
  C<int, std::allocator<int>> cont;
};


IntContainer<vector> ic1;
IntContainer<list> ic2;
```

Templates und Typnamen

```
template <typename TCont>
void f(TCont &c) {
  TCont::iterator iter = c.begin();      // Fehler
  typename TCont::iterator iter = c.begin();
  …
}
```

- *typename* immer erforderlich, wenn Typ von Template-Parameter abhängt

```
template <typename T>
void f2(vector<T> &v) {
  typename vector<T>::iterator iter = v.begin();
  …
}
```

Template-Spezialisierung (1)

- Spezielle Varianten für bestimmte Parametertypen

```
template <typename T>
class MyTemplate {  // Generische Implementierung
  …
};

template <>
class MyTemplate<int> {  // Spezialisierung für int
  …
};

MyTemplate<double> x;  // generische Variante
MyTemplate<int> y;     // spezialisierte Variante
```

Template-Spezialisierung (2)

- Mit allen Arten von Parametern möglich

```
template <int size>
struct Array {
  int arr[size];
};

template <>
struct Array<0> {
};


template <template <typename, typename> class C>
class IntContainer …

template <>
class IntContainer<vector> …
```

Spezialisierung – Beispiel (1)

```
template <typename T>
class Set {
public:
  bool Add(T const &x) { return mSet.insert(x).second; }
  bool Remove(T const &x) { return mSet.erase(x) > 0; }
  bool Contains(T const &x) { return mSet.count(x) > 0; }
private:
  std::set<T> mSet;
};

template <>
class Set<unsigned char> {
public:
  bool Add(unsigned char x);
  bool Remove(unsigned char x);
  bool Contains(unsigned char x) { return mSet.test(x); }
private:
  std::bitset<UCHAR_MAX+1> mSet;
};
```

Spezialisierung – Beispiel (2)

```
bool Set<unsigned char>::Add(unsigned char x) {
  if (mSet.test(x)) {
    return false;
  }
  mSet.set(x);
  return true;
}

bool Set<unsigned char>::Remove(unsigned char x) {
  if (!mSet.test(x)) {
    return false;
  }
  mSet.reset(x);
  return true;
}
```

Partielle Spezialisierung

- Nur für einen Teil der Parameter

```
template <typename T1, typename T2>
class MyTemplate2 {
  …
};

template <typename T1>
class MyTemplate2<T1, bool> { // Partielle Spezialisierung
  …
}
```

- Spezialisierung für alle Pointer-Typen

```
template <typename T>
class MyTemplate<T*> {
  // ...
};
```

Spezialisierungsmuster

- Pointer-Typen (T*)
- Referenztypen (T&)
- Templates

```
template <typename T>
class MyTemplate<std::vector<T>> {
  // ...
};
```

- Kombinationen möglich

```
template <typename T>
class MyTemplate<std::vector<T*>> { // alle Vektoren von Pointern
  // ...
};
```

Spezialisierung in der Standard-Library

- vector<bool>
  - Partielle Spezialisierung von vector
  - ein Bit pro Element
  - Vorteil: weniger Speicher
  - Nachteil: schlechtere Performance
  - Alternative: vector<char>
- numeric_limits
  - Spezialisierung für jeden numerischen Typ

Templates und Vererbung (1)

- Varianten
  - Template-Klasse erbt von normaler Klasse
  - Normale Klasse erbt von Template-Klasse

```
class IntFunctor: public std::binary_function<int, int, int> …
```

  - Template-Klasse erbt von Template-Klasse

```
template <typename T>
class GenericFunctor: public std::binary_function<T, T, T> …
```

  - Template-Klasse erbt von Template-Parameter

```
template <typename T>
class ExtensionClass: public T …

ExtensionClass<MyClass> obj1;
ExtensionClass<MyTemplateClass<int>> obj2;
```

Templates und Vererbung (2)

- Zugriff auf Members in Basisklasse, die von Template-Parametern abhängt

```
template <typename T>
struct Base {
  void m1() {}
};

template <typename T>
class Extended: public Base<T> {
public:
  void m2() {
    this->m1();  // nicht: m1(); => Fehler
    …
  }
};
```

- Zugriff via *this* oder Klassenname erforderlich

Template-Vererbung – Beispiel

```
template <typename T, typename TPred = std::less<T>>
class GenericSet {
public:
  bool Add(T const &x) { return mSet.insert(x).second; }
  bool Remove(T const &x) { return mSet.erase(x) > 0; }
  bool Contains(T const &x) { return mSet.count(x) > 0; }
protected:
  std::set<T, TPred> mSet;
};

template <typename T>
class Set: public GenericSet<T> {
};
```

CRTP

- „Curiously Recurring Template Pattern“
- Klasse leitet von einem mit sich selbst instanziiertem Template ab

```
template <typename T>
class Base {
  …
};

class X: public Base<X> {
  …
};

template <typename T>
class Y: public Base<Y<T>> …
```

CRTP – Beispiel (1)

Generische Klasse mit Vergleichsoperatoren:

```
// define all comparison operators based on operator <
template <typename C>
class ComparisonOperators {
public:
  bool operator>(C const &x) const {
    return x < static_cast<C const&>(*this);
  }

  bool operator<=(C const &x) const {
    return !(x < static_cast<C const&>(*this));
  }

  bool operator>=(C const &x) const {
    return !(static_cast<C const&>(*this) < x);
  }
```

CRTP – Beispiel (2)

```
  bool operator==(C const &x) const {
    return !(x < static_cast<C const&>(*this)) &&
           !(static_cast<C const&>(*this) < x);
  }

  bool operator!=(C const &x) const {
    return x < static_cast<C const&>(*this) ||
           static_cast<C const&>(*this) < x;
  }
};

class MyNumber: public ComparisonOperators<MyNumber> {
public:
  bool operator<(MyNumber const &x) const;
  // operators >, <=, >=, ==, != are inherited
  …
};
```

Variadic Templates (1)

- Template-Funktion oder Klasse mit beliebig vielen Parametern

```
class MyClass {
public:

  template <typename... T>
  static unique_ptr<MyClass> *create(T &&...args) {
    return new MyClass(std::forward<T>(args)...);
  }

private:
  MyClass();
  MyClass(int x, int y);
  MyClass(string const &s);
  // ...
};
```

Variadic Templates (2)

```
template <typename T>
T sum(T value) {
  return value;
}

template <typename T1, typename... T>
T1 sum(T1 value1, T ...values) {
  return value1 + sum(values...);
}

int x = sum(1, 2, 3, 4, 5);

template <typename T1, typename... T>
T1 avg(T1 value1, T ...values) {
  return sum(value1, values...) / (sizeof...(values) + 1);
}
```

Variadic Template Klassen (1)

```
Tuple<int, string, double> t(3, "def", 3.5);
int x;
string s;
double d;
t.get(x, s, d);


template <typename...> class Tuple;

template <> class Tuple<> {
public:
  void get() {}
};
```

Variadic Template Klassen (2)

```
template <typename T1, typename... T>
class Tuple<T1, T...> : private Tuple<T...> {
  using base = Tuple<T...>;
public:
  Tuple() : mElem();
  Tuple(T1 const &elem1, T const &...elems)
    : mElem(elem1), base(elems...) {}

  void get(T1 &elem1, T &...elems) {
    elem1 = mElem;
    base::get(elems...);
  }

private:
  T1 mElem;
};
```

Variadic Templates in der STL

- emplace... Methoden
- std::make_shared/std::make_unique
- std::tuple

Benutzerdefinierte Literale

- Verarbeitung von benutzerdefinierten numerischen Literalen
- Implementierung als Variadic Template
- Anwendungsbeispiele: große ganze Zahlen, Festkommazahlen

Bsp.: Benutzerdefinierte Literale

```
class BigInteger { ... }

template <typename T>
T IntLiteral(T val) {
  return val;
}

template <typename T, char firstDigit, char... digits>
T IntLiteral(T val) {
  val = val * 10 + (firstDigit - '0');
  return IntLiteral<T, digits...>(val);
}

template <char... digits>
BigInteger operator "" _bi() {
  return IntLiteral<BigInteger, digits...>(0);
}

BigInteger x = 100000000000000000000000_bi;
```

Template-Metaprogrammierung

- Ermöglicht die Definition von Funktionen, die vom Compiler berechnet werden
- Definition als Template-Klassen
  - Funktionswert als rekursiv definierte Konstante
  - Rekursionsanker durch Template-Spezialisierung
- Nicht nur für numerische Berechnungen, sondern auch zur Manipulation von Typen

Beispiel: Fakultätsberechnung

```
template <unsigned int x>
struct Faculty {
  static unsigned int const value = Faculty<x - 1>::value * x;
};

template <>
struct Faculty<0> {
  static unsigned int const value = 1;
};


cout << Faculty<5>::value << endl; // 120
int values[Faculty<4>::value];  // Compile-Time-Konstante!
```

Konstante Funktionen

- Ergebnis werden zur Compilezeit berechnet, wenn Parameter konstant

```
constexpr unsigned int Faculty(unsigned int x) {
  return x <= 1 ? 1 : Faculty(x-1) * x;
}
```

- C++ 11: Funktion darf nur aus return-Anweisung bestehen
- C++ 14: beliebige Anweisungen (mit Einschränkungen)

Typen als Parameter

Beispiel: Prüfen, ob Typ ein Pointer-Typ ist

```
template <typename T>
struct is_pointer {
  static bool const value = false;
};

template <typename T>
struct is_pointer<T*> {
  static bool const value = true;
};
```

Manipulation von Typen

Beispiel: Pointer-Deklaration entfernen

```
template <typename T>
struct remove_pointer {
  typedef T type;
};

template <typename T>
struct remove_pointer<T*> {
  typedef T type;
};
```

Type Traits

- Bestandteil von C++ 11 (<type_traits>)
- Metafunktionen zur Abfrage von Typeigenschaften
  - Abgeleitet von true_type bzw. false_type
  - Ermöglicht Überladen und Spezialisierung anhand von Typeigenschaften
- Metafunktionen zur Typtransformation

Typeigenschaften

- Typkategorie: is_integral, is_float, is_pointer, is_reference, …
- Allgemeine Eigenschaften: is_const, is_signed, is_unsigned, is_polymorphic, is_abstract, …
- Beziehungen zwischen Typen: is_same, is_convertible, is_base_of

Beispiel: Überladen

```
// Implementierung für alle integralen Typen
template <typename T> T MyFunction(T x, true_type) {
  …
}

// Implementierung für alle anderen Typen
template <typename T> T MyFunction(T x, false_type) {
  …
}

template <typename T> T MyFunction(T x) {
  return MyFunction(x, is_integral<T>());
}
```

Beispiel: Template-Spezialisierung

```
template <typename T, bool isIntegral>
class MyBaseClass;

template <typename T>
class MyBaseClass<T, true> {  // Variante für integrale Typen
  …
};

template <typename T>
class MyBaseClass<T, false> { // Variante für andere Typen
  …
};

template <typename T>
class MyClass: public MyBaseClass<T, is_integral<T>::value> {};
```

Typtransformation

- Entfernen von Typbestandteilen: remove_const, remove_pointer, remove_reference, …
- Hinzufügen von Typbestandteilen: add_const, add_pointer, add_reference, …
- Umwandlung zwischen signed/unsigned: make_signed, make_unsigned