Jürgen Lemke

Xpert.press

C++-Metaprogrammierung

Eine Einführung in die Präprozessor- und Template-Metaprogrammierung

Springer Vieweg

Xpert.press

Weitere Information zu dieser Reihe finden Sie auf
http://www.springer.com/series/4393

Die Reihe **Xpert.press** vermittelt Professionals in den Bereichen Softwareentwicklung, Internettechnologie und IT-Management aktuell und kompetent relevantes Fachwissen über Technologien und Produkte zur Entwicklung und Anwendung moderner Informationstechnologien.

Jürgen Lemke

C++-Metaprogrammierung

Eine Einführung in die Präprozessor- und Template-Metaprogrammierung

Springer Vieweg

Jürgen Lemke
Hilchenbach, Deutschland

ISSN 1439-5428
Xpert.press
ISBN 978-3-662-48549-1 ISBN 978-3-662-48550-7 (eBook)
DOI 10.1007/978-3-662-48550-7

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Springer Vieweg

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer-Verlag GmbH Berlin Heidelberg ist Teil der Fachverlagsgruppe Springer Science+Business Media (www.springer.com)

Dieses Buch widme ich meiner Frau Heike sowie unseren Kindern Annelie, Frieda und Torsten.

Vorwort

Kann man eine Programmiersprache wie C++ mit seinen Templates komplett beherrschen? Selbst Bjarne Stroustrup, der Erfinder von C++, musste sich vom Template-Mechanismus der Turing-Vollständigkeit eines Besseren belehren lassen. Auf dem C++-Standardisierungstreffen 1994 in SanDiego präsentierte Erwin Unruh zum ersten Mal ein rekursives C++-Metaprogramm zur Berechnung von Primzahlen.

Gibt es heute noch Grenzen von C++, wo liegen sie und was ist mit C++ möglich bzw. unmöglich? In meiner Arbeit mit C++-Schnittstellen (Interfaces) wurde ich mit der Behauptung konfrontiert, dass sich atomare (native) C++-Strukturen nicht automatisch innerhalb der Programmiersprache generieren lassen, im Gegensatz zu managed Strukturen von Java, C# und C++/CLI. Weil ich diese Aussage so nicht im Raum stehen lassen wollte, beschäftigte ich mich intensiv mit der Metaprogrammierung in C++. Das Ergebnis ist ein Konzept zur automatischen Generierung von atomaren Strukturen, einschließlich deren Serialisierung und Visualisierung, sowie eine große Sammlung von Template-Metafunktionen.

Doch wie kann nun der neue Stoff dem Leser vermittelt werden? Ein Nachschlagewerk für die C++-Metaprogrammierung zu schreiben ist kaum möglich, da die Metaprogrammierung stärker mit Programmiertechniken als mit C++-Sprachstandards verbunden ist. Eine Beschränkung auf die Darstellung von Programmiertechniken schränkt wiederum den Leserkreis zu stark ein, da bestimmte Vorkenntnisse der Präprozessor- und Templateprogrammierung vorhanden sein müssen. Aus diesem Grunde versuche ich einen Spagat zwischen beiden Ansätzen zu erreichen.

In den ersten Abschnitten des Buches werden die Grundlagen in Form eines Nachschlagewerkes sowohl für den Präprozessor als auch für den Templatemechanismus beschrieben. Diese Beschreibung zielt bereits auf die spätere Verwendung in der Metaprogrammierung ab, sodass diese Abschnitte in der Beschreibung der Grundlagen

bereits deutlich umfangreicher sind, als sie in Standard C++-Büchern sein können. In den weiteren Abschnitten werden dann Programmiertechniken und Anwendungsbeispiele vorgestellt, die auf die Grundlagenabschnitte aufbauen. Viele dieser Beispiele und Techniken ziehen sich dann wie ein roter Faden durch das gesamte Buch.

Bei der Untersuchung von Laufzeitfehlern fällt häufig auf, dass eigentlich schon der Compiler auf bestimmte Fehler hätte aufmerksam machen müssen, weil alle Informationen bereits zur Kompilationszeit zur Verfügung standen. Leider gehen zum Beispiel Längeninformationen von Feldern oder Typinformationen von Objekten bei der Parameterübergabe in Funktionen verloren. Mit Templates lassen sich solche Informationen in andere Programmcodes übertragen. Ein Schwerpunkt der Metaprogrammierung liegt daher auch in der Erzeugung von sicherem Programmcode.

Der interessierte Leser wird nach dem Studium des Buches einen erweiterten Blickwinkel auf die Möglichkeiten des C++-Compilers, der Sicherheit und der Effizienz von Programmcode erhalten. Er wird schon vor dem eigentlichen Schreiben von neuem Programmcode tiefergreifende Überlegungen anstellen, welche Programmteile generiert und welche Teile bereits zur Kompilationszeit erledigt werden können.

Auch dem .NET-Anwender steht die C++-Metaprogrammierung mit seinen Templates, über die erweiterte Programmiersprache C++/CLI, zur Verfügung. Damit können .NET-Anwendungen im Hinblick auf ihre Laufzeit beschleunigt werden. Mit Hilfe von Generatoren ist es leicht möglich, Strukturen nach .NET zu konvertieren und die Daten von unmanaged zu managed Objekten zu kopieren bzw. zu konvertieren.

In der vorliegenden Arbeit dreht sich aber nicht alles nur um die reine Metaprogrammierung. Das ist auch kaum möglich, denn die Metaprogrammierung ist immer auf Hilfsklassen angewiesen. Gerade dieses Zusammenspiel soll dargestellt werden, um den Leser für eigene Ansätze zu inspirieren.

Hilchenbach, 2015 Jürgen Lemke

Inhaltsverzeichnis

Teil II Techniken und Anwendungsfälle

Einleitung 1

Die Softwaretechnik ist einem ständigen Wandel unterzogen. Um heute effizient und wirtschaftlich Software entwickeln zu können ist es notwendig, einen großen Teil des Quellcodes aus allgemeingültigen Bibliotheken zu benutzen, um den kundenspezifischen Anteil möglichst gering zu halten. Software aus freien und tausendfach bewährten Bibliotheken, wie der Standard Template Library (STL) oder der Boost Bibliothek, sind dabei zu bevorzugen. Bei der Entwicklung von neuer Software sollte die erste Überlegung immer in Richtung einer allgemeingültigen Umsetzung gehen. So entstehen mit der Zeit Softwaresystemfamilien, die flexibel für unterschiedlichste Anwendungen eingesetzt werden können.

Auf die Entwicklung der Softwaresystemfamilien baut die generative Programmierung auf. Sie zielt dabei auf die nahezu automatische Erstellung von Softwaresystemen auf Grundlage der flexiblen Softwaresystemfamilien. Hierbei kommen vor allem Generatoren zum Einsatz, die auf Grundlage von Konfigurationswissen das Softwaresystem erzeugen.

Die Metaprogrammierung lässt sich wiederum als Programmierung des Generators auffassen, deren Ergebnis ein neuer Quellcode ist. Aber auch eine Erweiterung der Programmiersprache um neue Funktionalitäten oder optimierten Quellcode ist mit ihr möglich. Die griechische Vorsilbe *meta* steht dabei für *über* oder *hinter* und deutet auf eine übergeordnete Programmierung hin. Die hier vorgestellte C++-Metaprogrammierung beschreibt die Technik der Metaprogrammierung nur mit Mitteln der Programmiersprache C++ selbst. Hierfür stehen in C++ zwei unterschiedliche Mechanismen zur Verfügung, die Präprozessor- und die Template-Metaprogrammierung.

Die Präprozessor-Metaprogrammierung befasst sich mit dem Generieren und Einsetzen von Codesegmenten in den vorhandenen Quellcode vor dem Kompilieren. Es erfolgt dabei weder eine Typ- noch eine Syntaxprüfung, weshalb der Einsatz des Präprozessors auf diejenigen Einsatzfälle zu beschränken ist, die mit der Template-Metaprogrammierung nicht abgedeckt werden können.

J. Lemke, *C++-Metaprogrammierung*, DOI 10.1007/978-3-662-48550-7_1

Die Grundidee der Template-Metaprogrammierung besteht darin, dass Templates während der Kompilationszeit ausgewertet werden und daraus erst der eigentliche Quellcode generiert wird. Sie kann wie eine Programmiersprache zur Kompilationszeit mit Verzweigungen, Schleifen und Berechnungen angesehen werden. Die Template-Metaprogrammierung von C++ wurde bereits intensiv erforscht. Mit dem neuen C++11-Standard wurden zudem viele Neuerungen, wie zum Beispiel variadische Templates (Templates mit einer variablen Anzahl von Argumenten), neue Schlüsselwörter wie *constexpr* und *static_assert*, sowie neue Metafunktionen in der Standard Template Library eingeführt, die die Metaprogrammierung weiter unterstützen. Aus Gründen der Abwärtskompatibilität ist es leider manchmal notwendig, auf einige der neuen Features von C++11 zu verzichten.

1.1 Ziel und Zweck des Buches

Das vorliegende Buch soll dem C++-Programmierer helfen, die Metaprogrammierung zu verstehen und die Vorteile für den täglichen Gebrauch zu erkennen. Das Buch bietet einen strukturierten Einstieg, für den Grundkenntnisse in C++ ausreichend sind. Es werden neben den Grundlagen der Präprozessor- und Templateprogrammierung Programmiertechniken und Beispiele vorgestellt, die das Grundverständnis für die Metaprogrammierung wecken sollen. Alle Beispiele sind kompatibel zum Standard C++98 es sei denn, sie sind explizit für den Standard C++11 gekennzeichnet.

Ein Schwerpunkt des Buches liegt in der Realisierung von sicheren und performanten Schnittstellen. Es wird in Kap. 5 gezeigt, wie Schnittstellen geprüft oder mittels Generatoren sicherer bedient werden können. In Kap. 8 wird ein mit Templates realisiertes Einheitensystem vorgestellt, bei dem die Einheitenrechnung bereits zur Übersetzungszeit durch den Compiler durchgeführt wird. Auch die Umsetzung von Bitfeldern mit Templates in Kap. 9 ermöglicht die sichere Benutzung von Bitfeldern in Schnittstellen.

Ein zweiter Schwerpunkt des Buches liegt in der effektiven Realisierung von Modulschnittstellen mit Hilfe der Metaprogrammierung. In den Kap. 6 und 7 wird gezeigt, wie Schnittstellen generiert und serialisiert werden können. Gerade die Verbindung zu einer beliebigen Datenbank oder einer graphischen Oberfläche machen den Einsatz der Modulschnittstellen sehr komfortabel.

1.2 Vergleich zur Codegenerierung außerhalb von C++

Gegenüber der Codegenerierung mit Generatoren und deren domainspezifischer Sprache (DSL) außerhalb von C++ bietet der Einsatz der Metaprogrammierung folgende Vorteile [1]:

- Es ist kein zusätzlicher Aufwand für das Erlernen einer neuen domainspezifischen Sprache bei einem verhältnismäßig eingeschränkten Anwendungsbereich erforderlich.

- Es besteht keine interne Bindung an den Entwicklungsstand der Generatoren.
- Es ist keine externe Bindung an den Anbieter der eingesetzten Softwaretools (zum Beispiel von Office-Anwendungen) erforderlich. Weiterhin fallen keine zusätzlichen Lizenzkosten an.
- Es besteht kein Zwang, den gesamten Sourcecode zu generieren. Es können jederzeit effizientere Alternativen oder andere Abstraktionsmöglichkeiten herangezogen werden.
- Der Anwender muss nicht ständig zwischen unterschiedlichen Sprachen und Entwicklungstools wechseln.
- Eine weitere Typspezialisierung oder Schnittstellenerweiterung ist schnell umsetzbar.

Gegenüber dem Einsatz von .NET-basierten Generatoren bietet die Metaprogrammierung zusätzlich die folgenden Vorteile:

- Es ist keine Laufzeitumgebung auf den Zielsystemen erforderlich.
- Es gibt keine Performanceverluste durch das Umwandeln von Datentypen zur API von Betriebssystem oder Datenbanken.
- Die Softwaresysteme bleiben unabhängig vom Betriebssystem.

1.3 Syntax der verwendeten Beispiele

In C++ gibt es eine Reihe unterschiedlicher Schreibweisen und Programmierrichtlinien. Zum besseren Verständnis der Beispiele in diesem Buch sollen die wichtigsten verwendeten Schreibweisen vorgestellt werden.

Variablen werden groß geschrieben
Variablen sind Substantive bzw. eigenständige Namen und werden daher groß geschrieben. Der Name sollte so gewählt werden, dass er lesbar den Inhalt beschreibt, der auch ohne zusätzlichen Kommentar zu verstehen ist.

```
long  MaterialId;  // material id
char  Unit[16];    // unit
float Price;       // price in EURO
```

Variablen von Klassen beginnen mit dem Präfix m_ (engl. member variable). Damit kann die Zugehörigkeit einer Variablen zu einer Klasse leicht erkannt werden.

```
struct SMaterial
{
  long  m_MaterialId;  // material id
  char  m_Unit[16];    // unit
  float m_Price;       // price in EURO
};
```

Zeigervariablen beginnen mit dem Präfix p (engl. pointer).

```
SMaterial *pMaterial;
```

Wenn sich eine Zeigervariable gleichzeitig innerhalb einer Klasse befindet, dann wird das Präfix m_ zusätzlich vorangestellt.

```
SMaterial *m_pMaterial;
```

Globale Variablen beginnen mit dem Präfix g (engl. global).

```
const char gVersion[] = "version 1.00-02";
```

Enumeratoren beginnt mit dem Präfix e (engl. enumerator).

```
enum {eValue = 1};
```

In Metafunktionen wird der Enumerator häufig klein und ohne dem Präfix e geschrieben, um die stärkere Zugehörigkeit zu Funktionsnamen zu verdeutlichen (siehe Funktionsnamen).

```
// meta function for type information
template <typename T>
struct TypeInfo
{
  enum {isInt   = 0,
        isFloat = 0};
};
```

Funktionsnamen beginnen mit einem Kleinbuchstaben
Funktionsnamen sollten mit einem Verb beginnen welches beschreibt, was die Funktion macht. Sie werden daher klein geschrieben. Zudem erhöht die Kleinschreibung des Verbs die Lesbarkeit des Sourcecodes, weil sich somit Funktionen und Variablen deutlich voneinander unterscheiden lassen.

```
static const char* getNamespace();
void sort(int &Value1, int &Value2);
```

Klassennamen beginnen mit einem Präfix
Klassennamen beginnen immer mit einem Großbuchstaben als Präfix, welches den Typ der Klasse beschreibt. In der Tab. 1.1 sind für ausgewählte Klassentypen die Präfixe dargestellt.

Tab. 1.1 Überblick über den Präfix von Klassen

Präfix	Klassentyp	Deklaration
I	Interfaceklasse mit abstrakt definierten Funktionen	class
C	Klasse mit Konstruktor und Funktionen	class
S	Struktur nur mit Variablen, ggf. mit statischen Funktionen	struct
U	Union	union
T	Klassentemplate	template <typename T> class
E	Enumerator	enum

```
// structure
struct SMaterial
{
  long  m_MaterialId;  // material id
  char  m_Unit[16];    // unit
  float m_Price;       // price in EURO
  ...
};

// class
class CMaterial
{
public:
  CMaterial(SMaterial *pMaterial) {}
  ...
};

// class template
template <typename T>
class TMatrix
{
  ...
};

// interface class
class IDBAccess
{
public:
  virtual ~IDBAccess() {};
  virtual bool open() = 0;
  virtual void close() = 0;
  virtual bool isOpen() const = 0;
  virtual bool execSql(const std::string &SqlStatement) = 0;
  ...
};

// enumerator
enum EDatabaseType
{
  eOracle,
  eMySQL,
  eSQLite
};
```

Das Präfix T als Kennzeichen für ein Klassentemplate wird manchmal als Gründen der Lesbarkeit weggelassen, wenn sich das Template im Namensbereich *tmp* für (template meta programming) befindet.

```
namespace tmp
{
  // static class for if condition
  template <bool Condition, typename Then, typename Else>
  class If
  {
    ...
  }
}
```

Makros bestehen nur aus Großbuchstaben
Makros werden durch den Präprozessor vor dem Kompilieren aufgelöst. Um sie von Funktions- und Klassennamen eindeutig unterscheiden zu können, bestehen Makronamen nur aus Großbuchstaben und zulässigen Sonderzeichen.

```
#define VERSION v3.02-13
#define PI 3.141592653589793
```

Englischer Sourcecode und Kommentar
Wie bereits in den oben aufgeführten Beispielen zu erkennen war, wird der Sourcecode und die Dokumentation in englischer Sprache geschrieben. Das erleichtert die Zusammenarbeit am Sourcecode mit ausländischen Mitarbeitern.

Verzicht auf Fehlermeldungen und Ausnahmebehandlungen
Der Sourcecode sollte zur Laufzeit sicher sein. Daher besteht ein realer Sourcecode aus einer Ausnahmebehandlung und zusätzlichen Ausgaben im Fehlerfall. Hierauf wird in den folgenden Beispielen der Einfachheit halber verzichtet. Es würde zudem die Lesbarkeit und das Verständnis der Beispiele stark einschränken.

Dokumentation für Dokumentationsgeneratoren vorbereitet
Die Dokumentation des Sourcecodes beginnt häufig mit drei Schrägstrichen ///, damit Dokumentationsgeneratoren den Sourcecode parsen und Informationen über die jeweiligen Variablen, Klassen oder Funktionen herausfiltern können.

```
/// structure SMaterial
struct SMaterial
{
  /// material id
  long  m_MaterialId;
  /// unit
  char  m_Unit[16];
  /// price in EURO
  float m_Price;
  ...
};
```

Verzicht auf die Darstellung der #ifdef Makros
In vielen Beispielen wird aus Gründen der Übersichtlichkeit auf die Darstellung der notwendigen *#ifdef*-Makros oder *#pragma once*-Direktiven in den Headerdateien verzichtet.

Kursive Schreibweise
Ein kursiver Name oder eine kursive Wortgruppe im Text deutet auf einen Text im Sourcecode des Beispiels hin. Daher ist dieser Text in der Regel in englischer Sprache.

Geschweifte Klammern
Jeder öffnenden und schließenden geschweiften Klammer wird eine separate Zeile spendiert. Der Sourcecode innerhalb der geschweiften Klammern wird eingerückt. Diese Schreibweise ist übersichtlicher und lesbarer als eine geöffnete Klammer am Ende der vorherigen Zeile.

```
struct SMaterial
{
  ...
};
```

Wenn innerhalb der geschweiften Klammer nur eine Zeile Sourcecode vorhanden ist, dann wird innerhalb von Headerdateien der gesamte Sourcecode in einer Zeile geschrieben.

```
class CODBCAccess : public IDBAccess
{
private:
  /// data source name
  std::string m_DSN;
  /// user id
  std::string m_UserId;
  ...
public:
  ...
  /// set data source name
  void setDataSource(const std::string &DSN) {m_DSN = DSN;}
  /// get data source name
  const std::string& getDataSource()         {return m_DSN;}
  /// set uer identification
  void setUserId(const std::string &UserId)  {m_UserId = UserId;};
  /// get user identification
  const std::string& getUserId()             {return m_UserId;}
  ...
}
```

Literatur

1. Wikipedia. 2014. Domänenspezifische Sprache. *Domänenspezifische Sprache*. [Online]. http://de.wikipedia.org/wiki/Domänenspezifische_Sprache. Zugegriffen am 03.05.2014.

Teil I

Grundlagen

C-Präprozessor

2

Der C-Präprozessor ist Bestandteil der Programmiersprache C und damit auch C++. Mit ihm ist es möglich, den Quelltext in einem ersten Übersetzungsschritt zu verändern, bevor der Compiler den daraus entstandenen Quellcode übersetzt. Nach Bjarne Stroustrup [1] sollten Makros in C++ nur benutzt werden, wenn es unabdingbar ist, denn sie demonstrieren eine Schwäche in der Programmiersprache selbst, im Programm oder beim Programmierer.

Bei der Metaprogrammierung gehen wir oft bis an die Grenzen des Machbaren von C++ und seinen Templates. In solchen Situationen bietet uns der Präprozessor häufig die einzige Möglichkeit, die „Schwäche der Programmiersprache“ zu umgehen. So ist es beispielsweise mit Templates nicht möglich, den Namen von Variablen zu generieren. Aber auch zur übersichtlichen Darstellung von Sourcecode bietet sich der Einsatz des Präprozessors an. Der C-Präprozessor ist somit ein wichtiges Hilfsmittel der C++-Metaprogrammierung.

In diesem Kapitel wird gezeigt, dass der C-Präprozessor ein leistungsfähiges Werkzeug ist, um schnell Quellcode generieren zu können. Seine Einsatzmöglichkeiten werden dabei häufig unterschätzt. Mit dem Präprozessor lassen sich, wie bei Programmiersprachen auch, bedingte Verzweigungen und Schleifen realisieren. Leider können Makros nicht rekursiv aufgerufen werden, weshalb Rekursionen über Schleifen nachgebildet werden müssen. Weiterhin werden variadische Makros (Makros mit einer variablen Anzahl von Argumenten) vorgestellt, um den zu generierenden Quellcode übersichtlicher und allgemeingültiger definieren zu können. Schließlich wird gezeigt, dass auch einfache Grundrechenarten zur Übersetzungszeit realisiert werden können, um die Flexibilität der Makros weiter zu steigern. Zur Bestimmung von Primzahlen werden abschließend die vorgestellten Verzweigungen, Schleifen und Grundrechenarten in einem komplexen Beispiel zusammenfassend dargestellt.

J. Lemke, *C++-Metaprogrammierung*, DOI 10.1007/978-3-662-48550-7_2

2.1 Einbinden von Dateiinhalten

Die sicherlich bekannteste Anweisung des C-Präprozessors ist die *#include*-Direktive.

```
#include <iostream>
```

Der Präprozessor durchsucht die Standardverzeichnisse des Compilers nach der in spitzen Klammern angegebenen Datei, hier im Beispiel *iostream*, und ersetzt die Zeile *#include <iostream>* mit dem Inhalt der Datei.

```
#include "Typelist.h"
```

Wird der Dateiname mit doppelten Anführungszeichen angegeben, durchsucht der Präprozessor zusätzlich das aktuelle Verzeichnis. Alle Header-Dateien eines Projektes sollten mit doppelten Anführungszeichen angegeben werden, um die Zugehörigkeit der Dateien zum Projekt zu verdeutlichen. Header-Dateien anderer Projekte oder Bibliotheken sollten in spitzen Klammern angegeben werden.

2.2 Einfache Makros

Makros werden mit der *#define*-Direktive deklariert. Die einfachsten Makros sind symbolische Konstanten und bestehen aus einem Makronamen (Makrosymbol) und einer Zeichenkette, die sich bis zum Zeilenende erstreckt. Ein Semikolon wird dabei wie ein Zeichen der Zeichenkette behandelt. Der Präprozessor ersetzt vor dem Kompilieren den Makronomen durch die Zeichenkette. Mit einfachen Makros kann der Sourcecode übersichtlicher und lesbarer gestalten werden.

```
#define VERSION v3.02-13
#define PI 3.141592653589793
```

Die Verwendung von Makros hat aber den Nachteil, dass beim Debuggen der Inhalt der Makrosymbole nicht sichtbar ist, da dieser im Quellcode bereits ersetzt wurde [2]. Für die Deklaration von PI und anderen konstanten Zahlen wäre es daher besser, diese als konstante Variable zu deklarieren.

```
const double Pi = 3.141592653589793;
```

Der Name von Makros sollte neben Zahlen und zulässigen Sonderzeichen nur aus Großbuchstaben bestehen, um sie von Funktionen deutlich unterscheiden zu können.

Es gibt bereits einige ANSI-C-kompatible vordefinierten Makros [3], die in der Tab. 2.1 dargestellt sind.

Zusätzlich stehen viele compilerabhängige vordefinierte Makros zur Verfügung. In der Tab. 2.2 werden einige wichtige Microsoft-spezifische vordefinierte Makros vorgestellt [4].

Tab. 2.1 Auswahl von ANSI-kompatiblen vordefinierten Makros

__DATE__	Kompilierungsdatum der aktuellen Datei
__FILE__	Name der aktuellen Datei
__LINE__	Zeilennummer der aktuellen Datei
__TIME__	Kompilierungszeit der aktuellen Datei
__STDC__	Liefert 1, wenn der Compiler sich nach dem ANSI C-Standard richtet

Tab. 2.2 Auswahl von Microsoft-spezifisch vordefinierten Makros

__cplusplus_cli	Definiert, wenn es sich um ein C++/CLI-Programm handelt, d. h. der Compilerschalter auf / clr gesetzt ist
_MSC_VER	Versionsnummer des Microsoft Compilers
_WIN32	Definiert für Win32 und Win64 Anwendungen
_WIN64	Definiert für Win64 Anwendungen

2.3 Makros mit Parametern

Ähnlich wie bei einem Funktionsaufruf in C/C++ können Makros auch Parameter in runden Klammern übergeben werden. In der *#define*-Anweisung ist zwischen dem Makronamen und der öffnenden Klammer kein Leerzeichen zulässig, da sonst das Makro als Makrosymbol mit dem anschließenden Text als Inhalt definiert wird (siehe Abschn. 2.2).

```
#define MM_TO_INCH(mm) (mm) / 25.4

#define CONVERT_C_TO_F(Celsius) (Celsius) * 1.8 + 32
#define CONVERT_F_TO_C(Fahrenheit) ((Fahrenheit) - 32) / 1.8
```

Die Parameter werden ohne jegliche Typüberprüfungen in das Makro substituiert. Dieses wird im folgenden Beispiel deutlich.

```
double Inch = MM_TO_INCH(This is not a number);
```

Durch die Parametersubstitution entsteht folgender Text:

```
double Inch = (This is not a number) / 25.4;// from Preprocessor
```

Die Parameter von Makros sollten grundsätzlich in Klammern gesetzt werden. Durch die Parametersubstitution entsteht beispielsweise mit dem gleichen Makro:

```
double Length = 50.8;                      // [mm]
double Inch   = MM_TO_INCH(Length + 25.4);  // [inch]
```

der folgende Quellcode:

```
double Inch   = (Length + 25.4) / 25.4;     // from Preprocessor
```

Ohne Klammersetzung sieht die Gleichung anders aus und führt demzufolge zu einem falschen Ergebnis.

```
#define MM_TO_INCH(mm) mm / 25.4

double Length = 50.8;                       // [mm]
double Inch   = MM_TO_INCH(Length + 25.4); // [inch]

double Inch   = Length + 25.4 / 25.4;       // from Preprocessor
```

In einigen Fällen ist es notwendig, einem Makro einen Parameter zu übergeben, der in der Definition ein Komma enthält. Ein Beispiel hierfür wäre die Containerklasse *std::map*, dem zwei Templateargumente übergeben werden müssen.

```
#define SETTYPE(Type, String) Type String

SETTYPE(std::map<std::string, double>, m_MapValue);
```

Der Präprozessor gibt eine Warnung aus, dass zu viele Parameter dem Makro *SETTYPE* übergeben wurden. Weil das Komma in der Typdefinition fehlt, ist die Definition falsch und würde sich nicht kompilieren lassen.

```
std::map<std::string double>;  // from Preprocessor
```

Wird die Definition der Klasse *std::map* in runde Klammern gesetzt, erkennt der Präprozessor diese als ein Argument an und würde folgenden Code generieren, der sich auch kompilieren lässt.

```
SETTYPE((std::map<std::string, double>), m_MapValue);

(std::map<std::string, double>) m_MapValue; // from Preprocessor
```

Eine weitere Möglichkeit besteht darin, die Definition der Klasse *std::map* in ein separates Makro zu verlagern. Allerdings funktioniert das nicht bei den GNU Compilern.[1]

```
#define MAPTYPE(T1, T2) std::map<T1, T2>

SETTYPE(MAPTYPE(std::string, double), m_MapValue);
```

Der Präprozessor würde das Makro *MAPTYPE* als ein Argument dem Makro *SETTYPE* übergeben. Das Ergebnis ist ein korrekt generierter Code.

[1] gcc Debian 4.6.3.

```
std::map<std::string, double> m_MapValue;  // from Preprocessor
```

Bei geschachtelten variadischen Makros (siehe Abschn. 2.8) würde diese Lösung aber fehlschlagen, da das Makro *MAPTYPE* bereits substituiert wurde, obwohl noch weiteren Makros die Parameter übergeben werden müssen. Eine Lösung hierfür wäre eine Typdefinition mit *typedef*.

```
typedef std::map<std::string, double> MapType;

SETTYPE(MapType, m_MapValue);
```

Mit einer Typdefinition lassen sich die Kommaprobleme sicher beheben. Der Präprozessor erzeugt folgenden kompilierfähigen Code.

```
MapType m_MapValue;  // from Preprocessor
```

Die Makros mit Parametern bilden die Grundlage für die weitere Verwendung in der Metaprogrammierung, weil sie sowohl Typen als auch Bezeichner verarbeiten können. In den gezeigten Beispielen wurde durch das Makro *SETTYPE* sowohl der Typ *std::map <std::string, double>* als auch der Bezeichner *m_MapValue* verarbeitet. Das ist mit C- bzw. C++-Funktionen so nicht möglich. In den weiteren Abschnitten dieses Kapitels wird diese Fähigkeit der Makros noch weiter detailliert beschrieben.

2.4 Mehrzeilige Makros

Der C-Präprozessor erlaubt es, Makros über mehrere Zeilen zu definieren. Damit können lange und komplexe Makros übersichtlicher dargestellt werden. Mit dem \-Zeichen am Ende einer Zeile wird festgelegt, dass das Makro in der nächsten Zeile fortgesetzt wird. Für die Lesbarkeit des Makros ist es förderlich, die \ -Zeichen untereinander auszurichten.

```
#define OUTPUT(X, Y, Z)                              \
std::cout << std::setiosflags(std::ios_base::fixed) \
          << std::setprecision(4)                    \
          << "x = "   << X                          \
          << ", y = " << Y                           \
          << ", z = " << Z                           \
          << std::resetiosflags(std::ios::fixed)     \
          << std::endl;
```

Mehrzeilige Makros werden durch den Präprozessor als nur eine Zeile interpretiert. Dieses kann das Debuggen erschweren, da der Compiler im Fehlerfall nur eine Zeilennummer kennt.

2.5 Umwandlung und Bearbeitung von Zeichenketten

Makroparameter können mit dem #-Makrooperator im Ersatztext des Makros in eine Zeichenkette umgewandelt werden. Hierzu wird der Parameter in doppelte Hochkommata eingeschlossen. Im folgenden Beispiel definiert das Makros *SETNAME* eine statische globale Variable.

```
#define SETNAME(Name) extern const char Name[] = #Name
```

Mit diesem Makro können nun sehr einfach globale Variablen festgelegt werden.

```
SETNAME(MaterialId);
```

Der Präprozessor schließt die Zeichenkette *MaterialId* in doppelte Hochkommata ein.

```
extern const char MaterialId[] = "MaterialId";  // from Preproc.
```

Zwei Zeichenketten können mittels des ##-Makrooperators zu einer neuen Zeichenkette verbunden werden. Das Makro *SETNAME* wird hier erweitert, um das Präfix *g* der globalen Variablen voranzustellen.

```
#define SETNAME(Name) extern const char g ## Name[] = #Name
```

Der Präprozessor erweitert den Namen *MaterialId[]* zu *gMaterialId* und erzeugt somit die folgende Definition:

```
extern const char gMaterialId[] = "MaterialId"; // from Preproc.
```

Mit dem ##-Makrooperator können auch Verzweigungen realisiert werden. Im folgenden Beispiel wird in Abhängigkeit des Parameters *Flag* entschieden, welches der beiden Makros aufgerufen wird.

```
#define CONVERT_C_TO_F(Celsius) (Celsius) * 1.8 + 32
#define CONVERT_F_TO_C(Fahrenheit) ((Fahrenheit) - 32) / 1.8

#define CONVERT_TEMP(Celsius, Flag) CONVERT_ ## Flag(Celsius)
```

Das Makro verbindet die Zeichenkette *CONVERT_* mit dem Inhalt des Parameters *Flag*. Anschließend wird das entsprechende Makro herangezogen. Im folgenden Beispiel wird zunächst die Temperatur von 20 °C in Grad Fahrenheit umgerechnet und das Ergebnis dann wieder zurück in Grad Celsius.

```
double C   = 20.0;
double ToF = CONVERT_TEMP(C,   C_TO_F); // convert to Fahrenheit
double ToC = CONVERT_TEMP(ToF, F_TO_C); // convert to Celsius
```

Im ersten Schritt erzeugt der Präprozessor den Namen des Makros.

```
double C   = 20.0;
double ToF = CONVERT_C_TO_F(Celsius);   // 1. Step from Preproc.
double ToC = CONVERT_F_TO_C(ToF);       // 1. Step from Preproc.
```

Im zweiten Schritt wird dann das entsprechende Makro ersetzt.

```
double C   = 20.0;
double ToF = (20.0) * 1.8 + 32;         // 2. Step from Preproc.
double ToC = ((ToF) - 32) / 1.8         // 2. Step from Preproc.
```

2.6 Bedingte Ersetzung

Die Direktiven *#if*, *#ifdef*, *#elif*, *#else* und *#endif* ermöglichen es, Zeilen im Quelltext gezielt zu implementieren oder zu ignorieren. Im folgenden Makro wird eine statische globale Variable angelegt, wenn das Makrosymbol *DEF_GLOBAL_NAME* deklariert wurde. Andernfalls wird auf die globale Variable verwiesen.

```
#ifdef DEF_GLOBAL_NAMES
  #define SETNAME(Name) extern const char g ## Name[] = #Name
#else
  #define SETNAME(Name) extern const char g ## Name[]
#endif
```

Mit der *#if*- bzw. *#elif*-Direktive kann der Inhalt eines Makrosymbols ausgewertet und der entsprechende Quelltext implementiert werden. Im Beispiel wird die Fakultät der im Makrosymbol *NUM* zugewiesenen Zahl berechnet und ausgegeben.

```
#define NUM 4
#if NUM == 4
  std::cout << "The factorial of 3 is " << 1*2*3*4 << std::endl;
#elif NUM == 3
  std::cout << "The factorial of 3 is " << 1*2*3 << std::endl;
#elif NUM == 2
  std::cout << "The factorial of 2 is " << 1*2 << std::endl;
#else
  std::cout << "The factorial of 1 is " << 1 << std::endl;
#endif
```

Innerhalb von Makros ist es nicht zulässig die *#if*-Direktiven zu benutzen.