Inhalt:

Einführung

Wem ging das nicht schon mal so: Eine (ähnliche, ja fast gleiche) Funktion oder Klasse musste mehrfach implementiert werden, weil wir sie für verschiedene Typen einsetzen wollten. Das Paradebeispiel sind Container-Klassen wie Listen: Einmal brauchen wir eine für int, dann eine für std::string und schließlich noch eine für unsere eigenen Klassen. Jedesmal eine Liste speziell für einen Typ zu schreiben, das wäre sehr zeitaufwändig und auch mühsam, abgesehen davon würden sich wahrscheinlich Fehler einschleichen, da viel mit Copy & Paste gearbeitet würde. Glücklicherweise bietet uns C++ aber ein Werkzeug an, mit dem wir "typunabhängig" programmieren können: Templates! Praktisch die gesamte C++-Standardbibliothek besteht aus Templates, angefangen bei std::string über std::vector bis zu den vielen Algorithmen wie std::copy oder std::find.

Die Compiler-Frage

Der immer noch sehr weitverbreitete und veraltete "VC++ 6"-Compiler ist leider nicht besonders gut für die (insbesondere fortgeschrittene) Template-Programmierung geeignet, für ein vernünftiges Arbeiten ist mindestens der VC 7.1 notwendig. Mit einem aktuellen g++ ist man auch auf der sicheren Seite.

Normale Templates wie Container sind für den VC++ 6 kein Problem, aber bei komplizierteren Deklarationen streikt er ziemlich schnell. Die Template-Klassen aus Abschnitt 5 z.B. wird der VC++ 6 mit vielen Fehlermeldungen quittieren, obwohl der Code korrekt ist (im Extremfall kann der Compiler abstürzen).

Im Übrigen ist selbst der sehr gute g++ nicht unverwundbar. Es gibt einige Sachen, die der Compiler einfach (noch) nicht verträgt.

Wir empfehlen die aktuelle Visual C++ Express Edition zu benutzen.

Gerade als Anfänger hat man keinen Grund, noch mit dem veralteten VC++ 6 zu beginnen, sondern sollte gleich mit einer neueren, besseren Version in die C++-Programmierung einsteigen. Die neuen Compiler werden sich auch abseits des Template-Schlachtfelds positiv mit schnellerem und besserem Code bemerkbar machen. Ich habe die Beispiele alle mit dem g++ 3.3.6 problemlos kompilieren können. Man sollte sich im Übrigen von den "umfangreichen" Fehlermeldungen des Compilers bei Templates nicht einschüchtern lassen, auch wenn sie zu Beginn kaum lesbar erscheinen, mit der Zeit gewöhnt man sich daran.

Definition von Templates

Und los geht's: Um dem Compiler mitzuteilen, dass man ein Template definieren möchte, bedient man sich folgendem Präfix, welches einer Funktion oder Klasse vorangestellt wird:

[code xml:lang="cpp"]template // oder: template [/code]

T stellt einen Parameter mit einem beliebigen Typ dar, und obwohl hier das Schlüsselwort class steht, kann man auch char oder double einsetzen. Das Schlüsselwort typename ist gleichwertig mit class, allerdings kann man die Verwendung von beiden wie folgt einteilen: typename wird verwendet, wenn ein built-in oder eine Klasse als Parameter kommen kann, class wird benutzt, wenn ausschließlich Klassen erwartet werden. Diese Einteilung dient nur der Übersichtlichkeit und hat sonst keine Auswirkungen.

Selbstverständlich kann man auch mehrere Template-Parameter angeben:

[code xml:lang="cpp"]// Zwei Parameter, einer vom Typ T und einer vom Typ U template ... template // Ein Parameter vom Typ T und einer vom Typ int ...[/code]

Für "nicht-Typ-Parameter", also built-ins, gelten folgende Einschränkungen:

1. Sie dürfen nicht verändert werden
2. Sie dürfen nur ganzzahlig sein

Im Übrigen werden built-ins als Konstanten behandelt, auf diesem Wege könnten wir z.B. bei einem Array Klassen-Template die Größe festlegen.

Es ist jedoch möglich, Referenzen oder Zeiger auf Gleitpunkt-Typen als Parameter anzugeben:

[code xml:lang="cpp"]template ...[/code]

Außerdem kann man den Parametern, wie gewohnt, Default-Werte geben:

[code xml:lang="cpp"]// FastCopy ist irgendeine Klasse template ...[/code]

Hierbei gelten die gleichen Regeln wie bei normalen Default-Parametern:

1. Wenn ein Parameter einen Default-Wert bekommt, so müssen alle nachfolgenden Parameter einen bekommen
2. Wird bei der Instanzierung ein Argument weggelassen, so müssen alle nachfolgenden Argumente weggelassen werden

Funktions-Templates

Früher, als die Gummistiefel noch auch Holz waren ;) , war min ein äußerst beliebtes und bekanntes Makro, um den kleineren von zwei Werten herauszufinden:

[code xml:lang="cpp"]#include #define MIN(a,b) ((a < b) ? a : b) using namespace std; int main (int argc, char **argv) { int x = 5,y = 6; int z = MIN(x, y); cout