Vorwort

"The world is built on C++ (and its C subset)."

– Herb Sutter

Die Infrastrukturen bei Google, Amazon und Facebook bestehen aus Komponenten und Diensten, die in C++ entworfen und implementiert wurden. Auch ein erheblicher Teil der Technologie von Betriebssystemen, Netzwerkgeräten und Speichersystemen ist mit C++ verwirklicht. In Telekommunikationssystemen werden fast alle Festnetz- und Mobilfunkverbindungen mit C++-Software aufgebaut. Und Schlüsselkomponenten in Industrie- und Transportsystemen – wie automatisierte Mautsysteme und autonome Fahrzeuge – basieren auf C++.

In Wissenschaft und Technik werden die meisten hochwertigen Softwarepakete heute in C++ realisiert. Die Stärke der Sprache beweist sich, wenn Projekte eine bestimmte Größe überschreiten und Datenstrukturen und Algorithmen nicht mehr trivial sind. Es ist kein Wunder, dass viele – wenn nicht die meisten – Simulationssoftwareprogramme in der Informatik heute in C++ realisiert werden: FLUENT, Abaqus, deal.II, FEniCS, OpenFOAM, G+SMO. Auch Embedded-Systeme werden dank leistungsfähigerer Embedded-Prozessoren und verbesserter Compiler zunehmend in C++ programmiert. Und die neuen Anwendungsbereiche Internet of Things (IoT) und Embedded Edge Intelligence werden alle von C++-Plattformen wie TensorFlow, Caffe2 und CNTK beherrscht.

Wesentliche, täglich von Ihnen genutzte Dienste basieren auf C++: Von Ihrem Mobiltelefon bis zu Ihrem Auto, in der Kommunikations- und Industrieinfrastruktur sowie wichtige Elemente in Medien- und Unterhaltungsdiensten enthalten alle C++-Komponenten. C++-Dienste und -Anwendungen sind in der modernen Gesellschaft allgegenwärtig. Der Grund ist einfach. Die Sprache C++ hat sich mit ihren Anforderungen weiterentwickelt und ist in vielerlei Hinsicht führend bei der Produktivität der Programmierung und der Ausführungseffizienz. Beide Charakteristika machen es zur bevorzugten Sprache für Anwendungen, die skalierbar sein müssen.

Gründe, C++ zu lernen

Wie keine andere Sprache meistert C++ das gesamte Spektrum von der hardwarenahen Programmierung auf der einen bis hin zur abstrakten High-Level-Programmierung auf der anderen Seite. Die Low-Level-Programmierung – wie nutzerdefinierbare Speicherverwaltung – ermöglicht es Ihnen als Programmierer zu verstehen, was wirklich während der Ausführung passiert. Dies hilft Ihnen wiederum, das Verhalten von Programmen in anderen Sprachen zu verstehen. In C++ können Sie extrem effiziente Programme schreiben, deren Performance der von in Maschinensprache geschriebenem Code nur geringfügig nachsteht, wobei letzteres ein Vielfaches an Entwicklungsaufwand erfordert.

Allerdings sollten Sie mit dem Hardcore-Performance-Tuning erst einmal warten und sich zunächst auf klare und aussagekräftige Software konzentrieren. Hier kommen die High-Level-Features von C++ ins Spiel. Die Sprache unterstützt eine Vielzahl von Programmierparadigmen direkt: objekt-orientierte Programmierung (Kapitel 6), generische (Kapitel 3), Meta-Programmierung (Kapitel 5), parallele Programmierung (Abschnitt 4.6), prozedurale (Abschnitt 1.5) und weitere.

Verschiedene Programmiertechniken – wie RAII (Abschnitt 2.4.2.1) und Expressions-Templates (Abschnitt 5.3) – wurden in und für C++ erfunden. Da die Sprache so ausdrucksstark ist, war es oft möglich, diese neuen Techniken zu etablieren, ohne die Sprache zu ändern. Und wer weiß, vielleicht erfinden Sie eines Tages auch eine neue Technik.

Gründe, dieses Buch zu lesen

Das Material dieses Buches wurde an echten Menschen getestet. Der Autor hielt drei Jahre lang die Vorlesung "C++ für Wissenschaftler". Die Studenten, meist aus dem Fachbereich Mathematik, sowie einige aus der Physik und den Ingenieurwissenschaften, kannten C++ teilweise vorher gar nicht und waren am Ende des Kurses in der Lage, fortgeschrittene Techniken wie Expressions-Templates (Abschnitt 5.3) zu implementieren.

Sie können dieses Buch in Ihrem eigenen Tempo lesen: Direkt zur Sache, indem Sie dem Hauptpfad folgen, oder ausführlicher, indem Sie zusätzliche Beispiele und Hintergrundinformationen in Anhang A lesen (wobei es auch dann noch recht intensiv ist).

Die Schöne und das Biest

C++-Programme können auf vielfältige Weise geschrieben werden. In diesem Buch führen wir Sie sanft zu den anspruchsvolleren Stilen. Dies erfordert die Verwendung von fortgeschrittenen Features, die zunächst einschüchternd wirken könnten, was sich aber geben wird, sobald Sie sich daran gewöhnt haben. Dabei ist die High-Level-Programmierung nicht nur in einem breiteren Spektrum einsetzbar, sondern in der Regel auch genauso effizient, gelegentlich sogar effizienter und natürlich deutlich besser lesbar als die Low-Level-Programmierung.

Wir geben Ihnen einen ersten Eindruck mit einem einfachen Beispiel: Gradientenabstieg mit konstanter Schrittweite. Das Prinzip ist extrem einfach: Wir berechnen den steilsten Abstieg von $f(x)$ mit seinem Gradienten $g(x)$ und folgen dieser Richtung mit Schritten fester Größe zum nächsten lokalen Minimum. Selbst der algorithmische Pseudocode ist so einfach wie diese Beschreibung:

Input : Startwert x, Schrittweite s, Abbruchkriterium ε, Funktion f, Gradient g
Output : Lokales Minimum x
do
  $x = x - s \cdot g(x)$
while $|\Delta f(x)| \geq \varepsilon$;

Algorithmus 1 : Gradientenabstiegsverfahren

Für diesen einfachen Algorithmus haben wir zwei recht unterschiedliche Implementierungen geschrieben. Schauen Sie doch einfach mal rein, ohne zu versuchen, die technischen Details zu verstehen.

```
void gradient_descent(double* x,
    double* y, double s, double eps,
    double(*f)(double, double),
    double(*gx)(double, double),
    double(*gy)(double, double))
{
    double val= f(*x, *y), delta;
    do {
        *x-= s * gx(*x, *y);
        *y-= s * gy(*x, *y);
        double new_val= f(*x, *y);
        delta= abs(new_val - val);
        val= new_val;
    } while (delta > eps);
}
```

```
template <typename Value, typename P1,
          typename P2, typename F,
          typename G>
Value gradient_descent(Value x, P1 s,
                       P2 eps, F f, G g)
{
    auto val= f(x), delta= val;
    do {
        x-= s * g(x);
        auto new_val= f(x);
        delta= abs(new_val - val);
        val= new_val;
    } while (delta > eps);
    return x;
}
```

Auf den ersten Blick sehen sich beide Version recht ähnlich, und wir werden Ihnen gleich sagen, welche wir bevorzugen. Die Erste ist im Prinzip reines C, d.h. auch mit einem C-Compiler kompilierbar. Ihr Vorteil ist, dass die berechnete Optimierung direkt sichtbar ist: eine 2D-Funktion mit `double`-Werten (dargestellt durch die hervorgehobenen Funktionsparameter). Wir bevorzugen die zweite Version, da sie allgemeiner anwendbar ist: Funktionen beliebiger Dimension mit beliebigen Werttypen, (erkennbar an den markierten Typen und Funktionsparametern). Überraschenderweise ist die allgemeinere Umsetzung nicht weniger effizient. Im Gegenteil, die als `F` und `G` übergebenen Funktionen können `inline` verwendet werden (siehe Abschnitt 1.5.3), so dass der Overhead des Funktionsaufrufs eingespart wird, während die explizite Verwendung von (unschönen) Funktionszeigern in der linken Version diese Optimierung erschwert oder komplett unmöglich macht.

Ein längeres Beispiel für den Vergleich von altem und neuem Stil finden Sie in Anhang A.1 (für den wirklich geduldigen Leser). Dort manifestiert sich der Nutzen der modernen Programmierung viel deutlicher als in dem hier gezeigten Einführungsbeispiel. Aber wir wollen Sie nicht zu lange mit dem Vorgeplänkel aufhalten.

Sprachen in Wissenschaft und Technik

> *"Es wäre schön, wenn jede Art von numerischer Software ohne Effizienzverlust in C++ geschrieben werden könnte; aber wenn dies nur durch Kompromittieren des C++-Typsystems möglich wäre, sollten wir uns lieber auf Fortran, Assembler oder architekturspezifische Erweiterungen verlassen."*
>
> *– Bjarne Stroustrup*

Wissenschaftliche und technische Software wird in verschiedenen Sprachen geschrieben, und welche sich am Besten geeignet, hängt von den Zielen und verfügbaren Ressourcen ab:

- Mathematische Werkzeuge wie MATLAB, Mathematica oder R sind ausgezeichnet, wenn wir ihre vorhandenen Algorithmen verwenden können. Wenn wir unsere eigenen Algorithmen mit feingranularen (z.B. skalaren) Operationen implementieren, werden wir einen

deutlichen Leistungsabfall verzeichnen. Dies ist noch nicht kritisch, wenn die Probleme klein sind oder der Nutzer eine unendliche Geduld aufbringt; andernfalls sollten wir über alternative Sprachen nachdenken.

- Python eignet sich hervorragend für die schnelle Software-Entwicklung und enthält bereits wissenschaftliche Bibliotheken wie "SciPy" und "NumPy," Oft sind diese Bibliotheken in C oder C++ implementiert und daher auch einigermaßen effizient. Auch hier gilt: nutzerdefinierte Algorithmen mit vielen feingranularen Operationen führen zu einer deutlichen Leistungseinbuße. Python ist hervorragend, um kleine und mittlere Aufgaben effizient zu lösen. Wenn Projekte ausreichend groß werden, wird es immer wichtiger, dass der Compiler strenger wird (z.B. dass er Zuweisungen abgelehnt, wenn die Argumenttypen nicht passen).
- Fortran ist auch großartig, wenn wir bestehende, sorgfältig optimierte Operationen – wie dichte Matrixmultiplikation – nutzen können. Es ist auch bestens geeignet, um die Hausaufgaben von alten Professoren zu erledigen (wenn sie nur nach dem fragen, was in Fortran einfach ist). Die Einführung neuer Datenstrukturen ist nach den Erfahrungen des Autors recht umständlich, und das Schreiben eines großen Simulationsprogramms in Fortran ist eine ziemliche Herausforderung – heute nur noch freiwillig von einer schwindenden Minderheit in Angriff genommen.
- C ermöglicht eine gute Performance, und eine große Menge an Software ist in C geschrieben. Die Kernsprache ist relativ klein und leicht zu erlernen. Die Herausforderung besteht darin, große und fehlerfreie Software mit den einfachen und riskanten Sprachfeatures, insbesondere Zeiger (Abschnitt 1.8.2) und Makros (Abschnitt 1.9.2.1), zu erstellen. Der letzte Standard wurde 2011 veröffentlicht, daher der Name C11. Die meisten seiner Funktionen – aber nicht alle – sind seit C++14 auch in C++ enthalten.
- Sprachen wie Java, C# und PHP sind wahrscheinlich eine gute Wahl, wenn die Hauptkomponente der Anwendung eine Web- oder Grafikschnittstelle ist und nicht zu viele Berechnungen durchgeführt werden.
- C++ brilliert besonders, wenn wir große, hochwertige Software mit guter Performance entwickeln. Dennoch muss der Entwicklungsprozess nicht langsam und schmerzhaft sein. Mit den richtigen Abstraktionen können wir unsere C++-Programme sehr schnell entwickeln. Wir sind optimistisch, dass in Zukunft noch viele wissenschaftliche Bibliotheken entstehen werden.

Je mehr Sprachen wir kennen, desto mehr Auswahl haben wir natürlich. Je besser wir diese Sprachen beherrschen, desto fundierter wird unsere Auswahl sein. Zudem enthalten große Projekte oft Komponenten in verschiedenen Sprachen, wobei in den meisten Fällen zumindest die leistungskritischen Kerne in C oder C++ realisiert sind. Alles in allem ist das Lernen von C++ eine faszinierende Reise, und ein tiefes Verständnis davon wird Sie auf jeden Fall zu einem großartigen Programmierer machen.

Typographische Konventionen

Neue Termini werden *"kursiv in Anführungszeichen"* dargestellt. Wichtige Begriffe werden *kursiv* gesetzt.

Wichtige Hinweise werden in einer Box mit Pfeil gegeben. Wenn es sich um Tipps zum Programmieren handelt, sollten Sie nur aus sehr gewichtigen Gründen dagegen verstoßen.

Boxen mit Ausrufungszeichen enthalten Regeln, die unbedingt befolgt werden sollten.

C++-Quellen sind `blau und nichtproportional` gedruckt. Wichtige Programmdetails werden durch **`fette Schrift`** hervorgehoben. Innerhalb von Programmen sind Klassen, Funktionen, Variablen und Konstanten kleingeschrieben und können optional Unterstriche enthalten (snake_case). Eine Ausnahme bilden Matrizen, die in der Regel mit einem Großbuchstaben benannt werden. Template-Parameter beginnen mit einem Großbuchstaben und können weitere enthalten (CamelCase). Programmausgaben und Kommandozeilenbefehle sind in der gleichen nichtproportionalen Schrift gesetzt, jedoch in schwarz.

Programme, die C++11, C++14 oder C++17-Funktionen erfordern, sind mit entsprechenden Randboxen markiert. Einige Programme, die nur wenige C++11-Features verwenden, welche einfach durch C++03-Ausdrücke ersetzt werden können, sind nicht immer explizit gekennzeichnet.

⇒ `verzeichnis/quell_code.cpp`

Bis auf sehr kurze Code-Illustrationen wurden alle Programmierbeispiele in diesem Buch auf drei Compilern getestet: `g++`, `clang++` und Visual Studio. Die Dateinamen mit Pfad innerhalb des Repos sind für die getesteten Programmbeispiele, welche für das betreffende Thema relevant sind, am Anfang des entsprechenden Absatzes oder Abschnitts gekennzeichnet.

Alle Programme sind in einem öffentlichen Repository auf GitHub – *https://github.com/petergottschling/dmc2* – verfügbar und können mit dem folgenden Befehl geklont werden:

```
git clone https://github.com/petergottschling/dmc2.git
```

Unter Windows ist es praktischer, TortoiseGit zu verwenden; siehe *tortoisegit.org*.

Da wir aus eigener Erfahrung wissen, dass jede Redundanz die Gefahr von Inkonsistenz in sich birgt, haben wir die Programmbeispiele für die 2. englische Auflage und die 1. deutsche Ausgabe in einem gemeinsamen Repository bereitgestellt. In der deutschen Version sind die meisten Kommentare und Ausgaben nach dem Kopieren ins Buch übersetzt worden, aber sonst unterscheiden sich die Programmschnipsel im Buch nicht von den Quellen auf GitHub.