IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz Universität Hannover

Programmierpraktikum Technische Informatik (C++)

30.6.2022

IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz
Universität
Hannover

Die Standardbibliothek
Container Adaptors

Spezielle Container

- C++ enthält einige spezielle Containertypen, die bestimmte häufig verwendete Konzepte modellieren
  - `std::stack`, `std::queue` und `std::priority_queue`
- Unterschied zu normalen Containern: Kein Zugriff auf beliebige Elemente möglich
- Nach außen sind je nach Container nur Anfang und/oder Ende des Containers sichtbar
  - Kein Zugriff auf Elemente in der Mitte des Containers
  - Für jeden Container genau eine Stelle, auf die zugegriffen und deren Element gelöscht werden kann, und eine Stelle, an der eingefügt werden kann
- Verhalten kann auch mit normalen Containern emuliert werden
- Warum also eingeschränktere Container verwenden?
- Verwendung spezialisierter Schnittstellen beugt Fehlern vor

std::stack

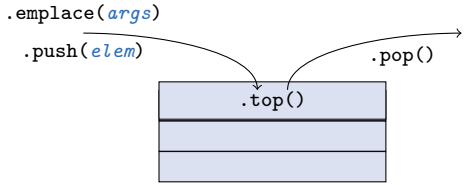
- `std::stack<T>` (Header: `stack`) wurde in der letzten Vorlesung und Übung bereits oberflächlich behandelt
- Modelliert einen Stapel (Stack)
  - Last-In-First-Out (LIFO)
  - Elemente werden oben auf den Stack gelegt und von oben wieder entfernt
- Welche Datenstruktur liegt einem Stack zugrunde?

Container Adaptors

- Stacks werden intern über dynamische Arrays, verlinkte Listen oder Deques implementiert
- `std::stack` ist ein Container Adaptor
  - Ein Adapter kapselt ein Objekt einer anderen Klasse hinter einem veränderten Interface
- `std::stack` hat zweiten, optionalen Templateparameter
  - `std::stack<double, std::list<double>>`
  - Gibt den intern zu verwendenden Containertyp an
  - Defaultwert für `std::stack<T>` ist `std::vector<T>`
- Laufzeitkomplexitäten entsprechen direkt der Komplexität auf dem zugrundeliegenden Container
  - Für `stack` und `queue` (später) normalerweise $O(1)$ für alle Operationen

Operationen von std::stack

- `std::stack` stellt lediglich sechs Methoden zur Verfügung:
  - `.pop()` entfernt das oberste Element vom Stapel ohne es zurückzugeben (Intern: `.pop_back()`)
  - `.push(`*e*`)` legt *e* oben auf den Stapel (Intern: `.push_back(`*e*`)`)
  - `.emplace(`*args*`)` ist analog zu `.push`, erstellt das Objekt aber Inplace (Intern: `.emplace_back(`*args*`)`)
  - `.top()` gibt das oberste Element des Stapels zurück ohne es zu entfernen (Intern: `.back()`)
  - `.empty()` und `.size()` funktionieren wie für andere Container



Beispiel für std::stack

```
std::string reverse(const std::string& str)
{
    std::stack<char, std::deque<char>> stack;
    for(char c: str)
        stack.push(c);
    std::string result;
    while(!stack.empty())
    {
        result.push_back(stack.top());
        stack.pop();
    }
    return result;
}
```

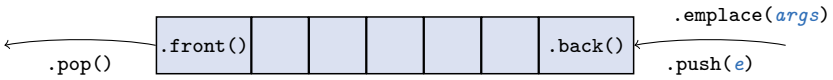
- In echtem Code natürlich besser `std::reverse` verwenden!

std::queue

- Die Standardbibliothek enthält weitere Adapter
- `std::queue<`*T*`, `*container<T>*`>` modelliert eine Warteschlange (Queue)
  - First-In-First-Out (FIFO)
  - Elemente werden ans Ende der Queue angehängt und vom Anfang entfernt
  - Angabe des zugrundeliegenden Containers wie bei `std::stack` optional, Default ist `std::deque<`*T*`>`

Operationen von std::queue

- `std::queue` bietet neben `.empty()` und `.size()` die folgenden Operationen:
  - `.pop()` entfernt das erste Element der Queue ohne es zurückzugeben (Intern: `.pop_front()`)
  - `.push(`*e*`)` hängt *e* an das Ende der Queue an (Intern: `.push_back(`*e*`)`)
  - `.emplace(`*args*`)` ist analog zu `.push`, erstellt das Objekt aber Inplace (Intern: `.emplace_back(`*args*`)`)
  - `.front()` gibt das erste Element der Schlange zurück ohne es zu entfernen (Intern: `.front()`)
  - `.back()` gibt das letzte Element der Schlange (Intern: `.back()`)



std::priority_queue

- Im Gegensatz zu den anderen Containeradaptern ist `std::priority_queue` kein sequentieller Container
  - Reihenfolge beim Entfernen hängt nicht direkt von der Einfügereihenfolge ab
- Einträge sind in einer `std::priority_queue` anhand einer Priorität angeordnet
- Das Element, auf das zugegriffen und das gelöscht werden kann, ist immer das mit der höchsten Priorität
- Einfügeposition ist nicht genauer spezifiziert, Element wird anhand seiner Priorität einsortiert
- Verwendet die Heapalgorithmen der Standardbibliothek (`std::push_heap`, `std::pop_heap` und `std::make_heap`)
  - Benötigt Random-Access-Iteratoren, daher als Container nur `std::vector<`*T*`>` und `std::deque<`*T*`>`

Operationen von std::priority_queue

- `std::priority_queue` bietet neben `.empty()` und `.size()` die folgenden Operationen:
  - `.pop()` entfernt das beste Element ohne es zurückzugeben
    - Intern: `std::pop_heap(...); c.pop_front()`
  - `.push(`*e*`)` fügt *e* hinzu
    - Intern: `c.push_back(`*e*`);std::push_heap(...);`
  - `.emplace(`*args*`)` ist analog zu `.push`, erstellt das Objekt aber Inplace
    - Intern: `c.emplace_back(`*args*`);std::push_heap(...);)`
  - `.top()` gibt das beste Element der Prioritätssschlange zurück ohne es zu entfernen (Intern: `.front()`)
- `.top()` ist wie `.empty()` und `.size()` auch $O(1)$, andere Operationen sind $O(\log(n))$

Priorisierung von Elementen

- In einer `std::priority_queue` wird auf das beste Element zugegriffen
  - Wie wird die Priorisierung bestimmt?
- `std::priority_queue` hat wie sortierte Container ein zusätzliches Templateargument, das die Ordnung der Elemente bestimmt
  - `std::priority_queue<`*T*`, `*Container*`, `*Comparison*`>`
  - *Comparison* ist eine Vergleichsfunktion wie für sortierte Container (Verhalten entspricht dem Operator $<$)
  - Standard für *Comparison*: Der $<$-Operator

IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz
Universität
Hannover

Die Standardbibliothek
Verschiedenes

Filesystem

- Vor C++17 existierten nur begrenzte Möglichkeiten, mit dem Dateisystem zu interagieren
  - Lediglich einlesen und schreiben von Dateien
  - Keine Möglichkeiten, Ordner zu durchsuchen oder zu erzeugen, ...
- Die Filesystem Library ist plattformunabhängige Kapselung des Dateisystems
- Ist seit C++17 in den C++-Standard aufgenommen

Beispiel Filesystem

```
int main(int argc, char* argv[]) {
    fs::path p(argv[1]);
    if(fs::exists(p)) {
        if(fs::is_regular_file(p))
        std::cout << p << " size is " << fs::file_size(p) << '\n';
        else if(fs::is_directory(p)) {
            std::cout << p << "is a directory\n";
            for(auto& f: fs::directory_iterator(p))
            std::cout<< f <<"\n";
        }
        else
        std::cout << p << "exists, but is neither a regular file nor a directory\n";
    }
    else {
        std::cout << p << "does not exist, creating\n";
        fs::create_directory(p);
    }
}
```

- Ein `path` beschreibt einen Dateisystempfad und das dazugehörige Dateisystemobjekt
- Kann in einen Stream ausgegeben werden, gibt dann den Pfad aus

Beispiel Filesystem

```
int main(int argc, char* argv[]) {
    fs::path p(argv[1]);
    if(fs::exists(p)) {
        if(fs::is_regular_file(p))
        std::cout << p << " size is " << fs::file_size(p) << '\n';
        else if(fs::is_directory(p)) {
            std::cout << p << "is a directory\n";
            for(auto& f: fs::directory_iterator(p))
            std::cout<< f <<"\n";
        }
        else
        std::cout << p << "exists, but is neither a regular file nor a directory\n";
    }
    else {
        std::cout << p << "does not exist, creating\n";
        fs::create_directory(p);
    }
}
```

- Für einen Pfad können mit ensprechenden Funktionen die verschiedenen Eigenschaften eines Dateisystemobjektes abgefragt werden
- Beispielsweise, ob der Pfad existiert und ob er ein Verzeichnis angibt

Beispiel Filesystem

```
int main(int argc, char* argv[]) {
    fs::path p(argv[1]);
    if(fs::exists(p)) {
        if(fs::is_regular_file(p))
        std::cout << p << " size is " << fs::file_size(p) << '\n';
        else if(fs::is_directory(p)) {
            std::cout << p << "is a directory\n";
            for(auto& f: fs::directory_iterator(p))
            std::cout<< f <<"\n";
        }
        else
        std::cout << p << "exists, but is neither a regular file nor a directory\n";
    }
    else {
        std::cout << p << "does not exist, creating\n";
        fs::create_directory(p);
    }
}
```

- Mit einem `directory_iterator` kann über die Dateien in einem Verzeichnis iteriert werden

Beispiel Filesystem

```
int main(int argc, char* argv[]) {
    fs::path p(argv[1]);
    if(fs::exists(p)) {
        if(fs::is_regular_file(p))
        std::cout << p << " size is " << fs::file_size(p) << '\n';
        else if(fs::is_directory(p)) {
            std::cout << p << "is a directory\n";
            for(auto& f: fs::directory_iterator(p))
            std::cout<< f <<"\n";
        }
        else
        std::cout << p << "exists, but is neither a regular file nor a directory\n";
    }
    else {
        std::cout << p << "does not exist, creating\n";
        fs::create_directory(p);
    }
}
```

- Es können auch Änderungen im Dateisystem vorgenommen werden
- Beispielsweise: Erstellen von Dateiordnern

std::variant (C++17)

- Type-safe union
- `std::variant<int, bool, std::string> x(5);`
- `bool y = std::get<bool>{x};` wirft `std::bad_variant_access`
- `bool* y = std::get_if<bool>(&x);` gibt `nullptr` zurück
- `x.emplace("foo");`
- Constructor/emplace so nur, wenn eindeutige Konstruktion vorliegt:
  - `std::variant<float, unsigned> x{1};` funktioniert nicht, da 1 gleichartig in `float` oder `unsigned` interpretierbar
  - In dem Fall Angabe mit `std::in_place_index<Int>`: `std::variant<float, unsigned> x(std::in_place_index<0>, 1);` für `float`-Komponente
  - `x.index()` gibt Index des aktiven Elements zurück

std::optional (C++17)

- Optionaler Wert: `std::optional<T>` hält entweder ein Element vom Typ `T` oder ist leer
- Verwendung: Funktionen mit optionalem Rückgabewert, Beispiel:

```
std::optional<int> try_get(const std::map<std::string, int>& lookup,
    const std::string& key)
{
auto iter = lookup.find(key);
return iter != lookup.end() ? *iter : std::nullopt;
}
```

- Erzeugung leerer Optionals mit Konstruktion aus `std::nullopt;`
- Überprüfung, ob Wert vorhanden, mit `.has_value()` oder bool-Konvertierung
- Zugriff auf Wert (wie bei Smartpointern) mit Dereferenzierungsop (Achtung: unchecked)
- `x.value()` wirft `bad_optional_access`, falls leer

Constraints (C++20)

- Templates funktionieren oft nicht für alle möglichen Typen
  - Fällt erst während der Instantiierung auf
  - Führt zu schwer verständlichen Fehlermeldungen
- Constraints ermöglichen eine Typprüfung vorweg
- Syntax: `requires <constraint-expression>`
- Beispiel:

```
#include <concepts>
template <typename T>
requires std::integral<T> || std::floating_point<T>
double average(const std::vector<T>& vec) {
    const double sum = std::accumulate(vec.begin(), vec.end(), 0.0);
    return sum / static_cast<double>(vec.size());
}
```

- Erwartet, dass T entweder ganzzahlig oder Fließkommazahl ist

Concepts (C++20)

- Constraints können zu Concepts zusammengefasst werden
- Beispiel:

```
template <typename T>
concept Cumulative = std::integral<T> || std::floating_point<T>

template <Cumulative T>
double average(const std::vector<T>& vec) {
    const double sum = std::accumulate(vec.begin(), vec.end(), 0.0);
    return sum / static_cast<double>(vec.size());
}
```

- Definiert ein eigenes Concept `Cumulative`
- Concept anstelle von `typename` verwendbar

Concepts (C++20)

- Concept in <constraint-expression> verwendbar
- Beispiel:

```
template <typename T>
concept Cumulative = std::integral<T> || std::floating_point<T>

template <typename T>
concept SignedCumulative = Cumulative<T> && std::is_signed<T>::value;

template <SignedCumulative T>
double signedAverage(const std::vector<T>& vec) {
    ...
}
```

- Liste vordefinierter Concepts: https://en.cppreference.com/w/cpp/concepts
- Weitere Details: https://en.cppreference.com/w/cpp/language/constraints

Weitere Elemente der Standardbibliothek

- Komplette Behandlung der Standardbibliothek würde den Rahmen dieser Veranstaltung sprengen
- Einige Bereiche der Standardbibliothek wurden (und werden) daher nicht näher behandelt:
  - Zeitmessungen
  - Threading
  - Atomare Ausdrücke
  - Regular Expressions
  - Lokalisierung
  - IOStreams
- Bei Intresse unter `http://en.cppreference.com/w/` anschauen

IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz
Universität
Hannover

Tools und Bibliotheken

Doxygen

- Doxygen ist ein Codedokumentationswerkzeug
  - Ähnlich zu Javadoc
  - Unterstützt verschiedene Programmiersprachen
- Dokumentation wird in Form spezieller Kommentare im Quellcode geschrieben
- Doxygencompiler erstellt daraus eine Dokumentation
  - Dokumentation enthält u.A. Klassen- und Funktionslisten sowie Vererbungshierarchien
  - Unterstützt verschiedene Ausgabeformate, z.B. Html

Doxygenkommentare

```
/**
 * Splits the content of a stream by a given delimeter.
 *
 * @author Bjoern Bredthauer
 * @param is Stream to get input from
 * @param delim Character to split the input with
 * @return Vector containing the parts of the input
 */
std::vector<std::string> split(std::istream& is, char delim);
```

- Doxygenkommentare beginnen mit `/**`
  - Alternativ ist auch `/*!` möglich
  - Normale Kommentare werden von Doxygen nicht berücksichtigt
  - Sterne (∗) am Anfang jeder Kommentarzeile sind gängige Konvention, aber nicht zwingend erforderlich

Doxygenkommentare

- Kommentierung unmittelbar vor der kommentierten Entität
  - Kann mit entsprechenden Sonderbefehlen auch anders platziert werden, ist aber eher unüblich
  - Nähe von Kommentar zum Sourcecode macht es einfacher, beide synchron zu halten
- Doxygenanweisungen beginnen wie bei Javadoc mit `@`
  - Alternativ auch \ möglich, z.B. `\param` statt `@param`
  - Für Funktionen sind vor allem Beschreibung, Parameter (`@param` *`name description`*), Rückgabewert (`@return`) und möglicherweise eine Exceptionspezifikation (`@throws` *`name description`*) wichtig.

Weitere Anweisungen

- Doxygen unterstützt neben den genannten Anweisungen auch viele weitere
  - Weitere Annotationen
  - Bedingte Verwendung von Beschreibung
  - Auslassen von Codeblöcken in der Dokumentation
  - Liste: `http://www.stack.nl/~dimitri/doxygen/manual/commands.html`
- Auch ohne Doxygenkommentar wird eine Dokumentation für den entsprechenden Code erzeugt
  - Enthält dann lediglich aus der Deklaration erkenntliche Informationen wie die Signatur einer Funktion

Verwendung von Doxygen

- Doxygen benötigt zunächst eine Konfigurationsdatei
  - Dateiname meist `Doxyfile` in Anlehnung an `make`
- Erzeugen einer Konfigurationsdatei mit `doxygen -g` *filename*
  - z.B. `doxygen -g Doxyfile`
- Konfigurationsdatei besteht aus Einträgen der Form *Name*=*Value*
  - Besonders wichtig: `PROJECT_NAME` und `PROJECT_BRIEF` zur Beschreibung des Projekts und `FILE_PATTERN` und `RECURSIVE` zum Einstellen der Eingabedateien
- Dann Doxygen mit `doxygen` *configfile* aufrufen
  - `doxygen` ohne weitere Angaben entspricht `doxygen Doxyfile`

Boost

- Hauptschwäche von C++ ist der geringe Umfang der Standardbibliothek
  - Standardbiliothek von Java ist um ein Vielfaches größer
- Führte zur Entstehung einer Sammlung von C++-Bibliotheken: Boost
- Ursprüngliche Projektintention: Proving Ground für Bibliotheken, die für die Aufnahme in den Standard vorgeschlagen wurden
  - Daher von Mitgliedern des Standardkommitees ins Leben gerufen
  - Inzwischen Quelle für einen großen Teil der Erweiterungen der Standardbibliothek
- Enthält inzwischen auch Bibliotheken, die nicht für eine Standardisierung vorgesehen sind
  - Unterliegen relativ freien und verständlichen Lizenzbedingungen
  - Müssen portabel sein

Woher bekommt man die Boost-Bibliotheken?

- Boost gehört nicht zum Standard und ist daher üblicherweise nicht mit dem Compiler mitgeliefert
- Kann auf `www.boost.org` heruntergeladen werden
- Für Linux-Systeme in der Regel im Softwarerepository der Distribution vorhanden
  - Auf Institutsrechnern und in virtueller Maschine bereits installiert
- Header liegen im Ordner `boost`, enden mit `.hpp`
  - Für Includes also beispielsweise `#include <boost/crc.hpp>`
- Die meisten Bibliotheken sind Header-only, einige allerdings nicht
  - Nicht Header-only Bibliotheken müssen explizit mitgelinkt werden
  - Bei gcc mit `-l` angeben, z.B. `-lboost_filesystem`

Überblick Boost

- Boost enthält inzwischen über 100 Bibliotheken
- Diverse Boost-Bibliotheken sind für kommende Standarderweiterungen vorgesehen
- Bietet Bibliothen für verschiedene Bereiche, u.a.:
  - String-Algorithmen
  - Graphen-Algorithmen
  - Parsen von Eingabedaten
  - Kommunikation zwischen verschiedenen Prozessen
  - Esoterischere Containertypen
  - Diverse Hilfskonstrukte für z.B. Operatoren und Iteratoren
- Im Folgenden werden einige der Bibliotheken näher betrachtet

Boost.Program_options

- Wesentliches Problem für viele Programme: Verarbeiten von Kommandozeilenparametern
- Für einfache Probleme simpel, für Kommandozeilentools mit vielen (optionalen) Parametern schnell problematisch
- Durch Verwendung von Boost.Program_options lösbar
- Boost.Program_options werden die Parameter, ihr Verhalten und ihre Beschreibung angegeben
- Parsen der Parameter übernimmt Boost
- Elemente liegen im Namespace `boost::progam_options`, im Folgenden mit `po` abgekürzt
- Nicht Header-Only, muss mit `-lboost_program_options` explizit gelinkt werden

Boost.Program_options Beispiel

```
int main(int argc, char** argv) {
    po::options_description desc("Allowed options");
    desc.add_options() ("help", "produce help message")
                       ("compression", po::value<int>(), "set compression level");
    po::variables_map vm;
    po::store(po::parse_command_line(argc, argv, desc), vm);
    po::notify(vm);
    if(vm.count("help")) {
        std::cout << desc << "\n";  return 1;
    }
    if(vm.count("compression"))
        std::cout << "Compression level was set to "
                  << vm["compression"].as<int>() << ".\n";
    else
        std::cout << "Compression level was not set.\n";
}
```

- `po::options_description` definiert die unterstützten Parameter
- `.add_options()` nutzt Operatorüberladungen, um eine generische Syntax zum Hinzufügen von Options zu erlauben
  - Pre-C++11-Bibliothek, emuliert Initializer-Lists
  - Elemente bestehen jeweils aus Argumentname und Beschreibungstext

Boost.Program_options Beispiel

```
int main(int argc, char** argv) {
    po::options_description desc("Allowed options");
    desc.add_options() ("help", "produce help message")
                       ("compression", po::value<int>(), "set compression level");
    po::variables_map vm;
    po::store(po::parse_command_line(argc, argv, desc), vm);
    po::notify(vm);
    if(vm.count("help")) {
        std::cout << desc << "\n";  return 1;
    }
    if(vm.count("compression"))
        std::cout << "Compression level was set to "
                  << vm["compression"].as<int>() << ".\n";
    else
        std::cout << "Compression level was not set.\n";
}
```

- Parameter in einer `po::variables_map` mit `po::store` speichern
  - Funktioniert ähnlich einer normalen Map
- Program_options kann auch Konfigurationsdateien einlesen
  - Daher müssen Parameter explizit als Kommandozeilenparameter geparst werden
- `vm.notify()` schließt das Einlesen von Parametern ab

Boost.Program_options Beispiel

```
int main(int argc, char** argv) {
    po::options_description desc("Allowed options");
    desc.add_options() ("help", "produce help message")
                       ("compression", po::value<int>(), "set compression level");
    po::variables_map vm;
    po::store(po::parse_command_line(argc, argv, desc), vm);
    po::notify(vm);
    if(vm.count("help")) {
        std::cout << desc << "\n"; return 1;
    }
    if(vm.count("compression"))
        std::cout << "Compression level was set to "
                  << vm["compression"].as<int>() << ".\n";
    else
        std::cout << "Compression level was not set.\n";
}
```

- Parameter sind in `vm` vorhanden, wenn sie angegeben wurden
- `vm.count("hel`
- Streamausgabe

```
$ ./main --help
Allowed options:
  --help                : produce help message
  --compression arg     : set compression level
```

Boost.Program_options Beispiel

```
int main(int argc, char** argv) {
    po::options_description desc("Allowed options");
    desc.add_options() ("help", "produce help message")
                       ("compression", po::value<int>(), "set compression level");
    po::variables_map vm;
    po::store(po::parse_command_line(argc, argv, desc), vm);
    po::notify(vm);
    if(vm.count("help")) {
        std::cout << desc << "\n";  return 1;
    }
    if(vm.count("compression"))
        std::cout << "Compression level was set to "
                  << vm["compression"].as<int>() << ".\n";
    else
        std::cout << "Compression level was not set.\n";
}
```

- Argumente werden in nichttypisierter Form gespeichert, daher expliziter Cast über `.as<T>()` notwendig

```
$ ./main
Compression level was not set.
$ ./main --compression 10
Compression level was set to 10.
```

Boost.Operators

- Einige Operatoren werden in der Regel auf Basis anderer Operatoren definiert
  - + auf Basis von +=, > auf Basis von <, ...
- Code für diese Operatoren ist immer gleich
- Besser: Automatisierung der Erstellung dieser Operatoren
- Boost.Operators bietet diese Funktionalität

Beispiel Boost.Operators

```
class point
    : boost::addable<point, boost::multipliable2<point, double >> {
public:
    point(double, double);
    double x() const;
    double y() const;

    point& operator+=(const point&);
    point& operator*=(double);
private:
    double xVal;
    double yVal;
};
```

- Klassen können von Boost.Operators-Klassen erben
  - `private`-Vererbung reicht aus
- Die Basisklassen definieren jeweils die entsprechenden Operatoren
  - `boost::addable` definiert den + Operator unter Verwendung von +=
  - Je nach Basisklasse muss der *op*=-Operator, der <-Operator oder der ==-Operator existieren, damit die Operatoren erzeugt werden

Beispiel Boost.Operators

```
class point
    : boost::addable<point, boost::multipliable2<point, double >> {
public:
    point(double, double);
    double x() const;
    double y() const;

    point& operator+=(const point&);
    point& operator*=(double);
private:
    double xVal;
    double yVal;
};
```

- Boost.Operators enthält auch Oberklassen, die mehrere Operatoren erstellen
  - `boost::arithmetic<T>` erzeugt beispielsweise die Operatoren + und *

Andere Boost-Bibliotheken

- Gezeigt wurde nur eine kleine Auswahl der Boost-Bibliotheken
- Deckt nicht ansatzweise die Menge der nützlichen Bibliotheken in Boost ab
- Vollständige Liste: `http://www.boost.org/doc/libs/`
- Bei Problemen, die so aussehen, als müssten sie häufiger vorkommen, in Boost nachschauen, ob es dort vielleicht schon eine Lösung gibt

GUI-Bibliotheken

- Standard-C++ definiert keine Möglichkeiten zur Gestaltung von graphischen Oberflächen
  - Sinnvoll, da C++ beispielsweise auch auf embedded Systemen läuft
- Für Betriebssysteme existieren in der Regel systemspezifische GUI-Bibliotheken
  - Aus Portabilitätsgründen hier nicht interessant
- Cross-Plattform Bibliotheken existieren ebenfalls
  - Qt
  - Gtkmm
  - wxWidgets

Qt

- Ursprünglich von Trolltech entwickelt
  - Inzwischen Open Source, wurde von Nokia unterstützt
  - Basistoolkit für KDE-Anwendungen
- GPL-lizensiert
  - Kommerzielle Lizenzen auch verfügbar
- Enthält auch andere Komponenten abseits der GUI-Entwicklung
  - Netzwerk- und Multimediabibliotheken
  - Webbibliotheken inklusive Javascript, Sql und Xml Verarbeitung
  - Eigene Container- und Stringtypen, Algorithmen, ...
- Kann die Standardbibliothek zu großen Teilen ersetzen
  - Ist auch Nachteil von Qt: Qt-Klassen entsprechen nicht dem gewohnten Verhalten in C++
  - Verwendung von Qt und Klassen der Standardbibliothek kann problematisch sein
- Verwendet u.a. in Autodesk Maya, Virtualbox und VLC Player

Qt-Beispiel

```
#include <QtGui>
int main(int argc, char **argv) {
    QApplication app(argc, argv);

    QWidget window;
    window.setWindowTitle("Qt4-Example");

    std::unique_ptr<QLabel> label = std::make_unique<QLabel>("Hello World!");
    label->setAlignment(Qt::AlignCenter);
    std::unique_ptr<QPushButton> button = std::make_unique<QPushButton>("&Exit");
    QObject::connect(button, SIGNAL(clicked()), &app, SLOT(quit()));

    std::unique_ptr<QVBoxLayout> layout = std::make_unique<QVBoxLayout>();
    layout->addWidget(label.release());
    layout->addWidget(button.release());
    window.setLayout(layout.release());

    window.show();
    return app.exec();
}
```

- Eine `QApplication` steuert die Ausführung einer Qt-GUI-Anwendung

Qt-Beispiel

```
#include <QtGui>
int main(int argc, char **argv) {
    QApplication app(argc, argv);

    QWidget window;
    window.setWindowTitle("Qt4-Example");

    std::unique_ptr<QLabel> label = std::make_unique<QLabel>("Hello World!");
    label->setAlignment(Qt::AlignCenter);
    std::unique_ptr<QPushButton> button = std::make_unique<QPushButton>("&Exit");
    QObject::connect(button, SIGNAL(clicked()), &app, SLOT(quit()));

    std::unique_ptr<QVBoxLayout> layout = std::make_unique<QVBoxLayout>();
    layout->addWidget(label.release());
    layout->addWidget(button.release());
    window.setLayout(layout.release());

    window.show();
    return app.exec();
}
```

- Ein Widget ist allgemein eine beliebige Komponente, dient hier als Fenster

Qt-Beispiel

```
#include <QtGui>
int main(int argc, char **argv) {
    QApplication app(argc, argv);

    QWidget window;
    window.setWindowTitle("Qt4-Example");

    std::unique_ptr<QLabel> label = std::make_unique<QLabel>("Hello World!");
    label->setAlignment(Qt::AlignCenter);
    std::unique_ptr<QPushButton> button = std::make_unique<QPushButton>("&Exit");
    QObject::connect(button, SIGNAL(clicked()), &app, SLOT(quit()));

    std::unique_ptr<QVBoxLayout> layout = std::make_unique<QVBoxLayout>();
    layout->addWidget(label.release());
    layout->addWidget(button.release());
    window.setLayout(layout.release());

    window.show();
    return app.exec();
}
```

- GUI-Elemente müssen meistens Heap-alloziert werden, da die Deallokation für Widgets, die bei anderen Widgets registriert wurden, automatisch vorgenommen wird

Qt-Beispiel

```
#include <QtGui>
int main(int argc, char **argv) {
    QApplication app(argc, argv);

    QWidget window;
    window.setWindowTitle("Qt4-Example");

    std::unique_ptr<QLabel> label = std::make_unique<QLabel>("Hello World!");
    label->setAlignment(Qt::AlignCenter);
    std::unique_ptr<QPushButton> button = std::make_unique<QPushButton>("&Exit");
    QObject::connect(button, SIGNAL(clicked()), &app, SLOT(quit()));

    std::unique_ptr<QVBoxLayout> layout = std::make_unique<QVBoxLayout>();
    layout->addWidget(label.release());
    layout->addWidget(button.release());
    window.setLayout(layout.release());

    window.show();
    return app.exec();
}
```

- Qt bietet einen Signal-Slot-Mechanismus, mit dem Aktionen (hier: Klicken des Buttons) bestimmte Reaktionen (hier: Beenden der Anwendung) zugeordnet werden können

Qt-Beispiel

```
#include <QtGui>
int main(int argc, char **argv) {
    QApplication app(argc, argv);

    QWidget window;
    window.setWindowTitle("Qt4-Example");

    std::unique_ptr<QLabel> label = std::make_unique<QLabel>("Hello World!");
    label->setAlignment(Qt::AlignCenter);
    std::unique_ptr<QPushButton> button = std::make_unique<QPushButton>("&Exit");
    QObject::connect(button, SIGNAL(clicked()), &app, SLOT(quit()));

    std::unique_ptr<QVBoxLayout> layout = std::make_unique<QVBoxLayout>();
    layout->addWidget(label.release());
    layout->addWidget(button.release());
    window.setLayout(layout.release());

    window.show();
    return app.exec();
}
```

- GUI-Elemente müssen eingehängt werden, Qt übernimmt von da an die Verwaltung des Elements

Qt kompilieren

- Qt fügt eigene Erweiterung zu C++ hinzu
  - zB. SIGNAL-SLOT-Mechanismus
- In Standard-C++ nicht in dieser Form umsetzbar
- Qt besitzt daher eigenen Precompiler, der Qt-Code in Standard-C++ umwandelt
  - Meta Object Compiler (MOC)
  - Zusätzlicher Kompilierungsschritt
- Code muss die entsprechenden Qt-Bibliotheken linken

Gtkmm und wxWidgets

- Gtkmm ist C++-Interface für die GTK+-Bibliothek
  - GTK+ (GIMP Toolkit) ist die primäre Grafikbibliothek für Gnome
  - Ursprünglich für das Bildbearbeitungsprogramm GIMP entwickelt
  - Weniger umfangreich als Qt, enthält vornehmlich GUI-Komponenten
  - LGPL-lizensiert
  - Verwendet u.a. in Firefox, GIMP und Inkscape
- wxWidgets ist weitere bedeutende GUI-Bibliothek
  - In C++ geschrieben
  - xwWidgets-Lizenz, ähnelt einer abgeschwächten LGPL
  - Verwendet u.a. für Audacity, Filezilla und Tom Tom

Probleme mit C++-GUI-Bibliotheken

- GUI-Bibliotheken integrieren nur begrenzt in modernen C++-Code
  - Hauptgrund: Ursprüngliche Implementation stammt aus Zeiten vor der Standardisierung von C++
  - Standardbibliothek war damals nicht in vollem Umfang und häufig nicht fehlerfrei verfügbar
  - Standard-C++-Programmierstil war noch nicht festgelegt
- Wesentliche Änderungen wegen Rückwärtskompatibilität unwahrscheinlich
- Keine moderneren Bibliotheken mit bedeutender Nutzerbasis
- Möglichkeit das Problem zu umgehen: GUI nicht in C++ schreiben
- Der Ansatz, die GUI als eigenes Programm zu schreiben, ist nicht unüblich
  - Kommunikation mit der eigentlichen Anwendung über festgelegte Schnittstellen wie Pipes oder Shared Memory

Verwendung in dieser Veranstaltung

- Inwieweit dürfen/müssen die vorgestellten Bibliotheken in Aufgaben und der Abschlussprüfung verwendet werden?
- Die GUI-Bibliotheken, insbesondere Qt, würden mehrere Veranstaltungen füllen, und sollen daher nicht in den folgenden Aufgaben verwendet werden
- Für Bearbeitung der Aufgaben sind die Bibliotheken nicht notwendig
- Dokumentation über Doxygen ist in den Aufgaben ebenfalls nicht notwendig

Hinweise zum Abschlusstest

Abschlusstest

- Termine: 21.7., 28.7., 25.8.
- Eintragung in Gruppen nötig
- In virtueller Maschine am Institutsrechner
  - cppreference.com mit Suchfunktion, sonst keine andere Internetverbindung
  - Visual Studio Code

Wichtige Themen

- Kernsprache: Rekursion, Forwarddeklaration, if, for, ...
- const
- vector/set/map
- Iteration/Iteratoren
- Stream-Ausgabe
- Call-By-Reference/Value
- Klassen schreiben: Konstruktoren, Member
- Polymorphie: virtuelle Methoden, Slicing
- RAII: unique_ptr
- Lambda-Funktionen
- Einfache Templates schreiben
- Standardalgorithmen verwenden (s. cppreference)

Randthemen

In der Vorlesung behandelt aber in der echten Programmierung selten notwendig:

- `const_cast` und `reinterpret_cast`
- `auto_ptr`
- C-style arrays
- Manuelles Speichermanagement mit `new` und `delete`

Für den Abschlusstest sind diese Sprachbereiche daher nicht relevant

Themen, die nicht ausführlich behandelt wurden

Nur oberflächlich behandelt und daher ebenfalls nicht im Abschlusstest:

- Mehrfachvererbung, virtuelle Vererbung, Nicht-public-Vererbung
- Funktionspointer, eigene Funktoren
- Namespace Lookup (Ausname: std)
- shared_ptr, weak_ptr, Pointer Aliasing
- enums
- Eigene Makros schreiben
- static_assert
- Exceptions, Exception Safety
- Eigene Destruktoren, Copy-Konstruktoren, ...
- inline, ODR

IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz
Universität
Hannover

Tutorial