

Architektur und Programmierung von Grafik- und Koprozessoren

C++ Wiederholung

Stefan Zellmann

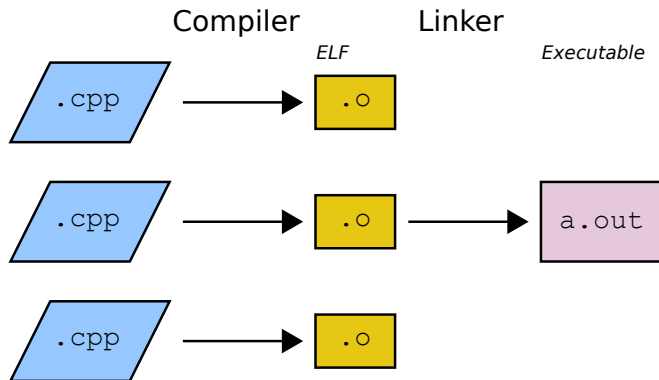
Lehrstuhl für Informatik, Universität zu Köln

SS2019

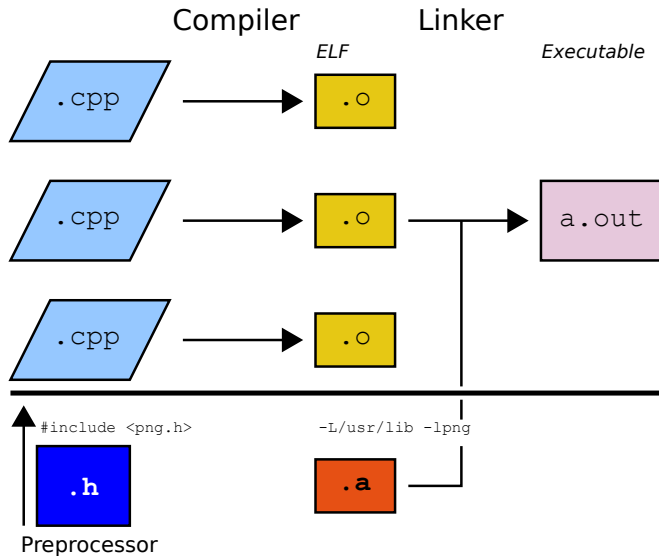
C++

- ▶ **Kompilierte** Programmiersprache.
- ▶ Ausgelegt auf hohe **Laufzeit-Performance**.
 - ▶ “As opposed to”: hohe Performance beim Kompilieren, hohe Entwicklerproduktivität, etc.
- ▶ Superset von **ANSI-C**. Unix: “Betriebssystem mit C-Compiler”.
- ▶ Standardisiert (aktuell: ISO/IEC 14882:2017(E)) (“C++17”).
 - ▶ C++11 Standard hat nach vielen Jahren grundlegende Neuerungen mit sich gebracht. C++14 und C++17 eher Inkremente.
 - ▶ Geplant: jeder C++11/17/23/.. Standard “große” Änderungen, jeder C++14/20/26 inkrementelle Änderungen.
- ▶ Streng typisierte Sprache.
- ▶ Wir verwenden Features von C++11. NVIDIA CUDA Compiler unterstützt neuere Standards nicht oder nicht vollständig.

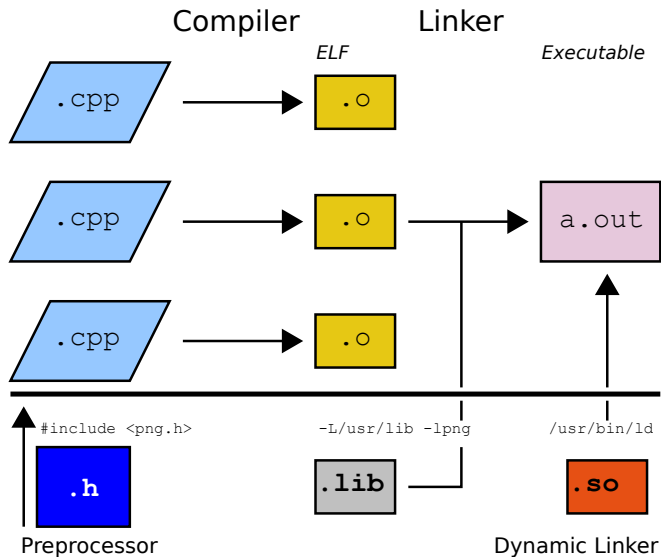
Kompilieren und Linken eines Ausführbares Programms



Statisches Linken



Dynamisches Linken



Basiskonstrukte

Hauptfunktion / Einsprungspunkt für Betriebssystem

```
int main(int argc, char** argv){  
    return 0;  
}
```

Basiskonstrukte

Statements: `<command>* ;`

Funktionen: `<return_type> name (<param_list>) {<body>}`

Seit C++11 auch:

`auto name (<param_list>) - > <return_type> {<body>}`

Seit C++14 auch ("automatic return type deduction"):

`auto name (<param_list>) {<body>}`

Basiskonstrukte

Kontrollstrukturen

```
// Comments
```

```
/* Comments */
```

```
if (condition) { } else if { } else {}
```

```
while (condition) {}
```

```
do { } while (condition);
```

```
for (long l = 0; l < 1000; ++l) {}
```

```
label:
```

```
goto label;
```

```
switch (i) {
```

```
case 0: break;
```

```
case 1: break;
```

```
default: break;
```

```
}
```


Basiskonstrukte

Programmstruktur: namespaces

```
// ANSI-C
struct myType {};

// C++
namespace my {
    struct Type {};
}

void useLikeC()    { myType mt; }

void useLikeCpp() {
    my::Type mt;
    { using my::Type;
      Type mt; }
    { using namespace my;
      Type mt; }
}
```

Niemals: `using namespace X;` in Header! *Namespace pollution.*

Basiskonstrukte

Programmstruktur: Typen

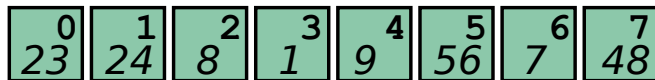
```
struct type1 {  
    int a, b, c;  
private:  
    int d_, e_, f_;  
};
```


```
class type2 {  
public:  
    type2() {}  
    ~type2() {}  
private:  
    int a, b, c;  
};
```

```
union u {  
    float f;  
    unsigned u;  
};
```

Basiskonstrukte

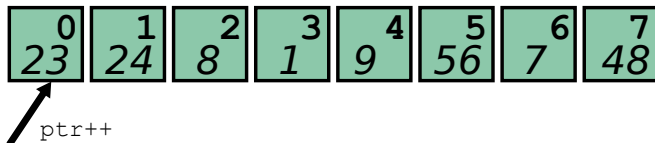
Zeiger: Abstraktionstyp für linearen Speicher.



 `ptr == 0, *ptr == 23`

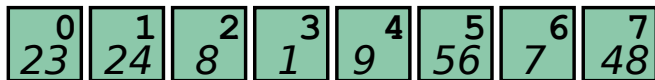
Basiskonstrukte

Zeiger: Abstraktionstyp für linearen Speicher.



Basiskonstrukte

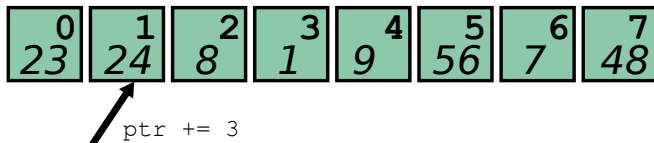
Zeiger: Abstraktionstyp für linearen Speicher.



`ptr == 1, *ptr == 24`

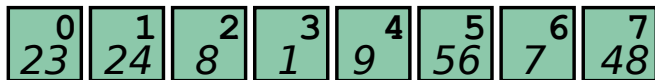
Basiskonstrukte

Zeiger: Abstraktionstyp für linearen Speicher.



Basiskonstrukte

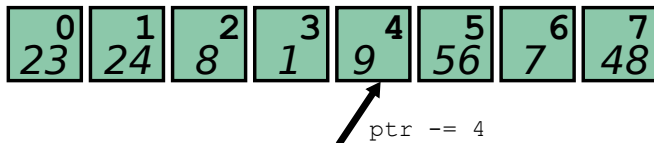
Zeiger: Abstraktionstyp für linearen Speicher.



`ptr == 4, *ptr == 9`

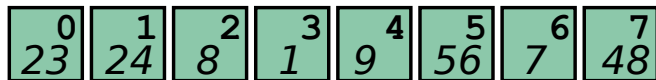
Basiskonstrukte


Zeiger: Abstraktionstyp für linearen Speicher.



Basiskonstrukte

Zeiger: Abstraktionstyp für linearen Speicher.



 `ptr == 0, *ptr == 23`

Basiskonstrukte

Zeiger

Initialisiere Zeiger. `Type* pointer = &data_array[0];`
(Achtung: Operator `&` hat drei verschiedene Bedeutungen in C++
(Adressoperator, Referenzoperator, bitweises Und)).

Springe:

```
pointer += 4;
```

Sprung vor den Anfang des Arrays:

```
pointer -= 5;
```

Aus Sicht der Sprache legal (Compiler). Zur Laufzeit (bestenfalls):
Memory Access Violation (Betriebssystem); womöglich: Zugriff auf
beliebigen Programmspeicher (Sicherheitslücke).

Dereferenzierung:

```
Type t = *pointer;
```

Achtung, auch Operator `*` hat mehrere Bedeutungen.

C++ und Ressourcen

C++ spezifiziert keine *Garbage Collection* (der Standard schließt sie nicht explizit aus, aber es gibt keine Implementierungen, die mir bekannt sind). Daher müssen alle Ressourcen, die reserviert werden, wieder freigegeben werden, z. B.

```
// Dynamisches Array
```

```
int* ptr1 = new int[5];
```

```
delete [] ptr1;
```

```
// Datei Handle
```

```
FILE* fp = fopen("/home/user/filename", "rw");
```

```
fclose(fp);
```

```
// Netzwerk Socket
```

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

```
close(sockfd);
```

C++ und Ressourcen

Ressourcenallokation \Rightarrow (i. d. R.) *System Calls*:

- ▶ Reserviere Speicherbereich mit `malloc` oder `new` \Rightarrow Betriebssystem gibt Zeiger auf Speicher zurück (oder `throw bad_alloc`).
- ▶ Öffne Datei: Dateisystemoperation erfordert Kommunikation mit Betriebssystem.
- ▶ ...

System Calls: API des Betriebssystemkernels. User Mode Library Funktionen wie `fopen` oder `new` leiten an Kernel Mode System Calls (z. B. `open`, `brk`) weiter (oder erweitern sie).

Unix: verschiedene Man-Page Sektionen für Library Funktionen (3) und System Calls (2):

`man 3 fopen`

`man 2 open`

C++ und Ressourcen

Ressource Acquisition is Initialization (RAII)

```
struct File {
    File(char const* fn)
        : fp(fopen(fn, "rw"))
    {
    }

    ~File()
    {
        fclose(fp);
    }

    FILE* fp;
};

void openFile() {
    // Stack variable, dtor is automatically
    // called when variable goes out of scope
    File f("/home/user/myfile");
}
```

C++ und Ressourcen

- ▶ Nutzen Sie RAII!
- ▶ Eingebaute, “deterministische” Garbage Collection. Destruktor wird garantiert aufgerufen, selbst wenn der Funktionsstack wegen Exceptions abgebaut wird.
- ▶ Prüfen Sie immer erst, ob es für Ihre Anwendung eine STL (oder Boost etc.) Template Klasse gibt (im Beispiel: `std::fstream`).

C++ und Ressourcen

STL RAII Klassen:

```
// Dynamic arrays
#include <vector>
std::vector<int> v1(3); v1.push_back(4);
v1[2] = 23;
```

```
// Smart pointers
#include <memory>
std::shared_ptr<int> sp = nullptr;
std::unique_ptr<int> up = nullptr;
```

```
// File handling
#include <fstream>
std::ofstream file("/home/user/myfile");
file << "text";
```

- ▶ Viele andere Beispiele!
- ▶ Vorsicht mit `std::shared_ptr`, sehr schwergewichtig.

Fundamentaltypen

Booleans, Characters, sonst.

```
void; std::nullptr_t;  
bool; char;
```

Integer Typen

```
short; int; long; long long; // + unsigned etc.
```

Datenmodelle, z. B. LP64 (64-bit Unix, Linux, Mac): long & pointer 64-bit, LLP64 (Win64): long 32-bit, pointer 64-bit.

Floating-Point Typen

```
float; double; long double;
```

Typeeigenschaft

```
static_assert(std::is_fundamental<T>::value == true);
```


Zusammengesetzte Typen

Typ ist eines von: Array, Funktion, Zeiger auf Objekt, Zeiger auf Funktion, Zeiger auf Member, Referenz, Klasse, Union oder Enumeration.

Typeeigenschaft

```
static_assert(std::is_compound<T>::value == true);
```

Typkategorien

Wichtige:

- ▶ **trivial** - `std::is_trivial`
- ▶ **POD** - `std::is_pod`

Triviale Typen

Trivialer Default Konstruktor

```
// Generated
struct type1 { int i; };
// Defaulted:
struct type2 {
    type2() = default;
    int i;
}
// Not trivial:
struct type2 {
    type2() {}
    int i;
}
```

Trivial kopierbar

```
type1 a, b, c;
b = a; memcpy(&c, &b, sizeof(b));
```

Typeeigenschaft

```
static_assert(std::is_trivial<T>::value == true);
```

POD Typen

“Plain Old Data Types”

Standard Layout Ist bitweise kompatibel mit einfachem ANSI-C Typen, insb. bzgl. Reihenfolge der Member Felder.

```
struct type1 { int i; };           // standard-layout
struct type2 : type1 { };         // standard-layout
struct type3 : type2 { };         // standard-layout
struct type4 : type2, type3 { };  // kein standard-layout
```

Typeeigenschaft

```
static_assert(std::is_pod<T>::value == true);
```

Einfache Typen (Trivial, POD)

- ▶ Zeichnen sich insb. dadurch aus, dass sie mit einfachem `memcpy` im Speicher bewegt werden können.
- ▶ Keine virtuelle Vererbung \Rightarrow keine erst zur Laufzeit bekannten Konstrukte.
- ▶ Einfache Typen besser für Optimizer. In inneren Schleifen sollten einfache Typen oder Varianten verwendet werden.

Smart Pointer und Referenzzählen

Prinzip sehr einfach (konkrete Implementierung nicht..). Copy Konstruktor zählt ref-count hoch, Destruktor zählt ref-count runter, löscht wenn ref-count 0.

Smart Pointer und Referenzzählen

```
struct smart_ptr {
    smart_ptr(Object* o) : obj_(o), cnt_(new int) {
        *cnt_ = 1;
    }
    smart_ptr(smart_ptr& rhs) : obj_(rhs.obj_)
                               , cnt_(rhs.cnt_) {
        *cnt_++;
    }
    ~smart_ptr() {
        if (--(*cnt_) == 0) {
            delete obj_;
            delete cnt_;
        }
    }
    // Interface
    Object& operator*() { return *obj_; }
    Object* operator->() { return obj_; }
    // Data
    Object* obj_;
    int* cnt_;
};
```

Smart Pointer und Referenzzählen

Ownership

Problem: wem gehört der Speicher. Bei dem Smart Pointer von oben geht man davon aus, dass Ownership geteilt wird (daher Referenzzählung). Tatsächliche Implementierungen (z. B. `std::shared_ptr`, `boost::shared_ptr`) recht schwergewichtig (insb. wegen Thread Safety).

Wenn klar ist, dass Datum, auf das gezeigt wird, nie geteilt wird, verwende exklusiven Smart Pointer (z. B. `std::unique_ptr`). Dieser implementiert RAII, aber keine Verweiszählung \Rightarrow kaum Verwaltungs-Overhead.

Generische Programmierung

```
// returns any linear container (e.g. std::vector)
auto container = generate_some_numbers();
// sort any linear container
std::sort(container.begin(), container.end());
```

`std::sort` ist ein *Algorithmus*, d. h. (im Sinne von C++) eine Funktion, die für verschiedene Eingabetypen *templatisiert* ist (insb.: die beiden Parameter `first` und `last` erfüllen das `RandomAccessIterator` *Konzept*).

Generische Programmierung

```
template <typename T>
struct generic_type {
    T data;
};

template <typename T>
void generic_func(T param) {
}

int main() {
    generic_type<int> gti;
    using concrete_type = generic_type<double>;
    concrete_type ctd;

    generic_func(gti.data); // Instantiate with int
    generic_func(ctd.data); // Instantiate with double
}
```

Generische Programmierung

STL und Boost stellen eine Vielzahl von Template Klassen und Funktionen bereit:

```
std::vector<int>;  
std::list<double>;  
std::map<int, char>;  
boost::bimap<int, std::string>;  
boost::circular_buffer<double>;  
//...
```

Generische Programmierung

STL und Boost stellen eine Vielzahl von Template Klassen und Funktionen bereit:

```
template <typename InputIt, typename T>
std::find(InputIt first, InputIt last, T value);
template <typename InputIt, typename T, typename Pred>
std::find_if(InputIt first, InputIt last, Pred pred);

template <typename RandomIt>
std::sort(RandomIt first, RandomIt last);
template <typename RandomIt, typename Compare>
std::sort(RandomIt first, RandomIt last, Compare comp);

//...
std::accumulate(..);
std::rotate(..);
std::partition(..);
```

Generische Programmierung

Concepts

Konzepte (engl./gebräuchlicher: Concepts) legen fest, welche Typen mit welchem Algorithmus etc. kompatibel sind.

Beispielsweise **EqualityComparable**: für den Typen ist die Funktion `operator==()` überladen, das Ergebnis von `operator==()` ist *implizit* nach `bool` konvertierbar.

Generische Programmierung erlaubt *Compile Zeit Polymorphismus* - also das Erweitern von Template Library Algorithmen durch überschreiben für eigene Typen - ohne Laufzeit Overhead (aber ggf. höhere Compile Zeiten und Code Komplexität).

Generische Programmierung – Patterns

Iterator Pattern

```
template <typename Container>
void iterate(Container cont) {
    for (auto it = cont.begin(); it != cont.end(); ++it) {
        std::cout << *it << '\n';
    }
}

int main() {
    std::vector<int> v({1,2,3,4,5});
    iterate(v);

    std::list<double> l({1.0,2.0,3.0});
    iterate(l);
}
```

Generische Programmierung – Patterns

Iterator Pattern Häufig wird generisch über eine *range* iteriert. Dann gilt (STL, Boost, andere):

```
range := [container.begin..container.end)
```

Der end (manchmal last Iterator zeigt also auf das (ungültige) Element direkt nach dem letzten Element.

Beim iterieren prüft man auf *Ungleichheit* (`it != end` anstatt `it < end`), um Datenstrukturen zu unterstützen, über die nicht sequentiell iteriert wird.

Zeiger erfüllen das RandomAccessIterator Konzept.

Generische Programmierung – Patterns

Type Traits (Typeeigenschaften)

```
struct my_type { };

template <typename T>
struct is_my_type {
    static constexpr bool value = false; };

template <>
struct is_my_type<my_type> {
    static constexpr bool value = true; };

template <typename T>
void func(T) {
    std::cout << is_my_type<T>::value << '\n'; }

int main() {
    func(int{});
    func(my_type{});
}
```


Generische Programmierung – Patterns

SFINAE (*Substitution failure is not an Error*)

```
template <typename T>
void func(T x)
{
    std::cout << x.pretty_sure_this_member_doesnt_exist;
}

void func(int i)
{
    std::cout << i;
}

int main()
{
    func(23);
}
```

Generische Programmierung – Patterns

SFINAE (*Substitution failure is not an Error*)

```
template <typename I,  
         typename = typename std::enable_if<  
             std::is_integral<I>::value>::type  
         >  
void func(I)  
{  
}  
  
int main()  
{  
    func(23);  
    func(std::vector<int>()); // Error!  
}
```

Lambda Funktionen

```
void func() {  
    // Simple  
    auto mul = [](int i, int j) { return i * j; };  
    mul(23, 32);  
  
    // Algorithm  
    struct record { int key; std::string value; }  
    std::vector<record> records;  
    std::sort(records.begin(), records.end(),  
              [](record a, record b)  
              { return a.key < b.key; });  
  
    // Parallel algorithm  
    parallel_for(range.begin(), range.end(), [](index i)  
               {  
                   // Parallel logic  
               });  
}
```

Lambda Funktionen

```
void func() {  
    int a, b;  
    std::vector<int> c;  
  
    // Capture by value  
    auto do_it1 = [a,b,c]() { a = 23; // a is copy  
        c.push_back(0); // c is copy!  
};  
  
    // Capture c by reference  
    auto do_it2 = [a,b,&c]() { /* .. */ };  
  
    // Capture everything by value  
    auto do_it3 = [=]() { std::cout << a << '\n' };  
  
    // Capture everything by reference  
    auto do_it4 = [&]() { c.push_back(23); }  
}
```

Lambda Funktionen

Lambda Funktionen sind Funktionsobjekte. Ihr Typ ist Compiler-spezifisch, jedoch sind sie implizit nach `std::function<>` konvertierbar. Sie erleichtern Situationen, in denen man vor C++11 Helferklassen hätte schreiben müssen.

Move Semantik und Return Value Optimization

C++11 Konstrukt. Prinzip: anstatt copy-by-value oder referenzieren eines großen Objekts, *verschiebe* es einfach in einen anderen Scope (im ursprünglichen Scope wird es ungültig).

Wahrscheinlich größter Nutzen beim Kopieren von Funktionsrückgabewerten:

```
struct BIG {  
    char data[0xFFFF];  
};  
BIG tmp = make_BIG();
```

Alte C++ Compiler legen hier *zwei* Temporaries an! Es gibt aber keine Situation, in der man an die Temporaries gelangen kann, deshalb optimieren Compiler (auch < C++11) dies mit Move Semantics (sog. Return Value Optimization).

Move Semantik und Return Value Optimization

```
struct BIG {  
    char data[0xFFFF];  
};  
  
void func(BIG big) { // Copy 2^16 bytes  
}  
  
void func(BIG& big) { // Reference  
}  
  
BIG func() {  
    return BIG{}; // Since C++11: move semantics  
}  
  
BIG soon_no_longer_required;  
BIG big = std::move(soon_no_longer_required);  
// 1st variable is now also no longer *valid*
```

Tutorials

Bitte schauen Sie sich die “richtigen” Tutorials an! Viele Tutorials vermitteln C++ im Stil anderer Sprachen wie C oder Java.

“Richtige” Tutorials sind z. B.

Channel9 Lectures: Stephan T. Lavavej - Core C++

<https://channel9.msdn.com/Series/>

C9-Lectures-Stephan-T-Lavavej-Core-C-

Jason Turner: C++ Weekly (YouTube Channel)

Sehr unterhaltsames Tutorial:

CppCon 2016: Jason Turner, “Rich Code for Tiny Computers: A Simple Commodore 64 Game in C++17”.

(YouTube)