

Übung zu Betriebssystembau

C++ Crashkurs

November 2022

Peter Ulbrich & Alexander Lochmann
(Mit Material vom Lehrstuhl 4 der FAU)

Arbeitsgruppe Systemsoftware
Technische Universität Dortmund

From: Linus Torvalds
Subject: Re: Compiling C++ kernel module + Makefile
Date: Mon, 19 Jan 2004 22:46:23 -0800 (PST)

On Tue, 20 Jan 2004, Robin Rosenberg wrote:

>
> This is the "We've always used COBOL^H^H^H" argument.

In fact, in Linux we did try C++ once already, back in 1992.

It sucks. Trust me - writing kernel code in C++ is a BLOODY STUPID IDEA.

The fact is, C++ compilers are not trustworthy. They were even worse in 1992, but some fundamental facts haven't changed:

- the whole C++ exception handling thing is fundamentally broken. It's _especially_ broken for kernels.
- any compiler or language that likes to hide things like memory allocations behind your back just isn't a good choice for a kernel.
- you can write object-oriented code (useful for filesystems etc) in C, _without_ the crap that is C++.

In general, I'd say that anybody who designs his kernel modules for C++ is either

- (a) looking for problems
- (b) a C++ bigot that can't see what he is writing is really just C anyway
- (c) was given an assignment in CS class to do so.

Feel free to make up (d).

Linus

C++: As Close as Possible to C, but no Closer

Andrew Koenig and Bjarne Stroustrup, The C++ Report. July 1989

einfaches ANSI C ist (fast) valides C++

```
#include <stdio.h>
```

```
int main() {  
    const char * str = "Informatik";  
    int n = 4;  
    printf("%s %d\n", str, n);  
    return 0;  
}
```

```
#include <cstdio>

int main() {
    const char * str = "Informatik";
    int n = 4;
    printf("%s %d\n", str, n);
    return 0;
}
```

```
#include <iostream>

int main() {
    const char * str = "Informatik";
    int n = 4;
    std::cout << str << ' ' << n << std::endl;
    return 0;
}
```

```
#include <iostream>
```

```
int main() {  
    const std::string str = "Informatik";  
    int n = 4;  
    std::cout << str << ' ' << n << std::endl;  
    return 0;  
}
```



```
#include <iostream>
using namespace std;

int main() {
    const string str = "Informatik";
    int n = 4;
    cout << str << ' ' << n << endl;
    return 0;
}
```

Namensräume

```
namespace Foo {  
    int getNum() {  
        return 23;  
    }  
}
```

```
namespace Bar {  
    int getNum() {  
        return 42;  
    }  
}
```

```
std::cout << Foo::getNum() << std::endl  
          << Bar::getNum() << std::endl;
```

Referenzen

```
int foo = 23;  
int& bar = foo;  
std::cout << bar << std::endl; // Ausgabe: 23
```

```
bar = 42;  
std::cout << foo << std::endl; // Ausgabe: 42
```

Referenzen

```
int foo = 23;  
int& bar = foo;  
std::cout << bar << std::endl; // Ausgabe: 23
```

```
bar = 42;  
std::cout << foo << std::endl; // Ausgabe: 42
```

Unterschied zu Zeiger

- Initialisierung bei Definition
- kann nicht geändert werden
- kann nicht NULL sein

Referenzen als Funktionsparameter

Konstante Referenzparameter

```
void dump(std::ostream &os, const Complex &c) {  
    os << c.real << " + " << c.img << "i\n";  
}
```

Referenzen als Funktionsparameter

Konstante Referenzparameter

```
void dump(std::ostream &os, const Complex &c) {  
    os << c.real << " + " << c.img << "i\n";  
}
```

Nicht-konstante Referenzparameter

```
void inc(int& i) { i++; }
```

```
int foo = 23;  
inc(foo);
```

```
std::cout << foo << std::endl; // Ausgabe: 24
```

Standardparameter

```
void inc(int& i, int n = 1) {  
    i += n;  
}
```

```
int foo = 23;  
inc(foo);
```

```
std::cout << foo << std::endl; // Ausgabe: 24
```

```
inc(foo, 2);
```

```
std::cout << foo << std::endl; // Ausgabe: 26
```

Überladen von Funktionen

```
bool isZero(int i){  
    return i == 0;  
}
```

```
bool isZero(double i){  
    const double eps = 0.00001;  
    return i < eps && i > -eps;  
}
```


Überladen von Funktionen

```
bool isZero(int i){  
    return i == 0;  
}
```

```
bool isZero(double i){  
    const double eps = 0.00001;  
    return i < eps && i > -eps;  
}
```



Overload resolution ist nicht trivial!

```
struct Disp {  
    char val;  
    int x, y;  
};
```

disp.h:

```
struct Disp {  
    char val;  
    int x, y;  
};
```

disp.cc:

```
#include "disp.h"  
using namespace std;  
  
void func() {  
    struct Disp p;  
    p.val = 'X';  
    p.x = 2;  
    p.y = 0;  
    cout << p.val << endl;  
}  
  
// Geht in C++ nicht:  
// struct Disp p = {  
//     .val = 'X',  
//     .x = 2,  
//     .y = 0  
// }
```

disp.h:

```
struct Disp {
    char val;
    int x, y;

    Disp() {
        val = 'X';
        this->x = 2;
        this->y = 0;
    }
};
```

disp.cc:

```
#include "disp.h"
using namespace std;

void func() {
    Disp p;
    cout << p.val << endl;
}
```

disp.h:

```
struct Disp {
    char val;
    int x, y;

    Disp(char c, int x=0, int y=0) {
        val = c;
        this->x = x;
        this->y = y;
    }
};
```

disp.cc:

```
#include "disp.h"
using namespace std;

void func() {
    Disp p('X', 2);
    cout << p.val << endl;
}
```

disp.h:

```
struct Disp {
    char val;
    int x, y;

    Disp(char c, int x=0, int y=0) {
        val = c;
        this->x = x;
        this->y = y;
    }
};
```

disp.cc:

```
#include "disp.h"
using namespace std;

void func() {
    Disp p{'X', 2};
    cout << p.val << endl;
}
```

disp.h:

```
struct Disp {  
    char val;  
    int x, y;  
  
    Disp(char c, int x=0, int y=0)  
        : val(c), x(x), y(y) {}  
};
```

disp.cc:

```
#include "disp.h"  
using namespace std;  
  
void func() {  
    Disp p{'X', 2};  
    cout << p.val << endl;  
}
```

disp.h:

```
struct Disp {
    char val;
    int x, y;

    Disp(char c, int x=0, int y=0)
        : val(c), x(x), y(y) {}

    Disp(int p) : val('X'),
                x(p % 80), y(p / 80) {}
};
```

disp.cc:

```
#include "disp.h"
using namespace std;

void func() {
    Disp p{2};
    cout << p.val << endl;
}
```


disp.h:

```
struct Disp {
    char val;
    int x, y;

    Disp(char c, int x=0, int y=0)
        : val(c), x(x), y(y) {}

    Disp(int p)
        : Disp('X', p % 80, p / 80) {}
};
```

disp.cc:

```
#include "disp.h"
using namespace std;

void func() {
    Disp p{2};
    cout << p.val << endl;
}
```

disp.h:

```
int count = 0;
struct Disp {
    char val;
    int x, y;

    Disp(char c, int x=0, int y=0)
        : val(c), x(x), y(y) {
        count++;
    }

    Disp(int p)
        : Disp('X', p % 80, p / 80) {}

    ~Disp() { count--; }
};
```

disp.cc:

```
#include "disp.h"
using namespace std;

void func() {
    Disp p{2};
    cout << p.val << "-"
         << count << endl;
}
```

```
disp.h:
struct Disp {
    static int count;
    char val;
    int x, y;

    Disp(char c, int x=0, int y=0)
        : val(c), x(x), y(y) {
        count++;
    }

    Disp(int p)
        : Disp('X', p % 80, p / 80) {}

    ~Disp() { count--; }
};

disp.cc:
#include "disp.h"
using namespace std;

void func() {
    Disp p{2};
    cout << p.val << "-"
         << Disp::count << endl;
}

int Disp::count = 0;
```

```
disp.h:
struct Disp {
    static int count;
    char val;
    int x, y;

    Disp(char c, int x=0, int y=0);

    Disp(int p)
        : Disp('X', p % 80, p / 80) {}

    ~Disp();
};

disp.cc:
#include "disp.h"
using namespace std;

void func() {
    Disp p{2};
    cout << p.val << "-"
         << Disp::count << endl;
}

int Disp::count = 0;

Disp::Disp(char c, int x, int y)
    : val(c), x(x), y(y) {
    count++;
}

Disp::~Disp() { count--; }
```

```
disp.h:
struct Disp {
    private:
        static int count;
        char val;
        int x, y;

    public:
        Disp(char c, int x=0, int y=0); }
        Disp(int p);
        ~Disp();
};

disp.cc:
#include "disp.h"
using namespace std;

void func() {
    Disp p{2};
    // Übersetzerfehler bei:
    cout << p.val << endl;

    int Disp::count = 0;

    Disp::Disp(char c, int x, int y)
        : val(c), x(x), y(y) {
        count++;
    }
    Disp::~Disp() { count--; }
```

Unterschied Standardsichtbarkeit

`public` bei struct (und union)

`private` bei class

```
struct Foo {  
    Foo(int i) { ... };  
    test() { ... };  
};
```

```
Foo foo1{23};  
foo1.test(); // OK
```

```
Foo foo2(23);  
foo2.test(); // OK
```

```
struct Bar {  
    Bar() { ... };  
    test() { ... };  
};
```

```
Bar bar0;  
bar0.test(); // OK
```

```
Bar bar1{};  
bar1.test(); // OK
```

```
Bar bar2();  
bar2.test(); // Fehler
```

```
struct Foo {  
    Foo(int i) { ... };  
    test() { ... };  
};
```

```
Foo foo1{23};  
foo1.test(); // OK
```

```
Foo foo2(23);  
foo2.test(); // OK
```

```
struct Bar {  
    Bar() { ... };  
    test() { ... };  
};
```

```
Bar bar0;  
bar0.test(); // OK
```

```
Bar bar1{};  
bar1.test(); // OK
```

```
Bar bar2();  
bar2.test(); // Fehler
```

error: request for member 'test' in 'bar2',
which is of non-class type 'Bar()'


```
struct Foo {  
    Foo(int i) { ... };  
    test() { ... };  
};
```

```
Foo foo1{23};  
foo1.test(); // OK
```

```
Foo foo2(23);  
foo2.test(); // OK
```

```
struct Bar {  
    Bar() { ... };  
    test() { ... };  
};
```

```
Bar bar0;  
bar0.test(); // OK
```

```
Bar bar1{};  
bar1.test(); // OK
```

```
Bar bar2();  
bar2.test(); // Fehler
```

**error: request for member 'test' in 'bar2',
which is of non-class type 'Bar()'**

→ Häufiger, „most vexing parse“ genannter Fehler!

Syntax: `class Abgeleitet: Vererbungsart Basis`

Vererbung

Syntax: `class Abgeleitet: Vererbungsart Basis`

```
class Foo {  
    int n;  
    protected:  
    int f1(int i){  
        return i * n;  
    }  
};
```

Vererbung

Syntax: `class Abgeleitet: Vererbungsart Basis`

```
class Foo {  
    int n;  
    protected:  
    int f1(int i){  
        return i * n;  
    }  
};
```

```
class Bar : Foo {  
    public:  
    int n; // eigene Var  
    int f2(int i){  
        return f1(i) * n;  
    }  
};
```

Vererbung

Syntax: `class Abgeleitet: Vererbungsart Basis`

```
class Foo {
    int n;
    protected:
    int f1(int i){
        return i * n;
    }
};

class Bar : Foo {
    public:
    int n; // eigene Var
    int f2(int i){
        return f1(i) * n;
    }
};
```

Vererbungsart	Elemente aus Basis
public	public und protected bleiben <i>Standard wenn Abgeleitet eine Struktur ist</i>
protected	public und protected werden zu protected
private	public und protected werden zu private <i>Standard wenn Abgeleitet eine Klasse ist</i>

Mehrfachvererbung

```
class FooBaz: public Foo, protected Baz
{
    // ...
}
```

Mehrfachvererbung

```
class FooBaz: public Foo, protected Baz
{
    // ...
}
```

Falls Foo und Baz selbe Basisklasse haben → Diamond-Problem

Auswahl der **Methode**

nicht-virtuell zur Übersetzungszeit anhand des statischen Typs

virtuell zur Laufzeit anhand des „tatsächlichen“ Typs

Virtuelle Methoden

```
class Foo {
    public:
        void f1() { cout << "Foo::f1" << endl; };
        virtual void f2() { cout << "Foo::f2" << endl; };
        virtual void f3() = 0;
};

class Bar : public Foo {
    public:
        void f2() { cout << "Bar::f2" << endl; };
        void f3() { cout << "Bar::f3" << endl; };
};

class Baz : public Foo {
    public:
        void f1() { cout << "Baz::f1" << endl; };
        void f3() { cout << "Baz::f3" << endl; };
};
```

Virtuelle Methoden

```
class Foo {
    public:
        void f1() { cout << "Foo::f1" << endl; };
        virtual void f2() { cout << "Foo::f2" << endl; };
        virtual void f3() = 0;
};
class Bar : public Foo {
    public:
        void f2() override { cout << "Bar::f2" << endl; };
        void f3() override { cout << "Bar::f3" << endl; };
};
class Baz : public Foo {
    public:
        void f1() { cout << "Baz::f1" << endl; };
        void f3() override { cout << "Baz::f3" << endl; };
};
```

Virtuelle Methoden

```
class Foo {
    public:
        void f1() {...};
        virtual void f2() {...};
        virtual void f3() = 0;
};
class Bar : public Foo {
    public:
        void f2() override {...}
        void f3() override {...}
};
class Baz : public Foo {
    public:
        void f1() {...}
        void f3() override {...}
};
```

```
Foo foo;
// Nicht erlaubt
// Übersetzerfehler
```

Virtuelle Methoden

```
class Foo {
public:
    void f1() {...};
    virtual void f2() {...};
    virtual void f3() = 0;
};
class Bar : public Foo {
public:
    void f2() override {...}
    void f3() override {...}
};
class Baz : public Foo {
public:
    void f1() {...}
    void f3() override {...}
};
```

```
Bar bar;
bar.f1();
// "Foo::f1"
bar.f2();
// "Bar::f2"
bar.f3();
// "Bar::f3"
```

Virtuelle Methoden

```
class Foo {
public:
    void f1() {...};
    virtual void f2() {...};
    virtual void f3() = 0;
};
class Bar : public Foo {
public:
    void f2() override {...}
    void f3() override {...}
};
class Baz : public Foo {
public:
    void f1() {...}
    void f3() override {...}
};
```

```
Bar bar;
bar.f1();
// "Foo::f1"
bar.f2();
// "Bar::f2"
bar.f3();
// "Bar::f3"

Baz baz;
baz.f1();
// "Baz::f1"
baz.f2();
// "Foo::f2"
baz.f3();
// "Baz::f3"
```

Virtuelle Methoden

```
class Foo {
public:
    void f1() {...};
    virtual void f2() {...};
    virtual void f3() = 0;
};
class Bar : public Foo {
public:
    void f2() override {...}
    void f3() override {...}
};
class Baz : public Foo {
public:
    void f1() {...}
    void f3() override {...}
};
```

```
Foo * foo = &bar;
foo->f1();
// "Foo::f1"
foo->f2();
// "Bar::f2"
foo->f3();
// "Bar::f3"

foo = &baz;
foo->f1();
// "Foo::f1"
foo->f2();
// "Foo::f2"
foo->f3();
// "Baz::f3"
```

Operatorüberladung & Freundschaft

```
class Complex {
    int real, img;
public:
    Complex(int real, int img) : real(real), img(img) { }

    Complex operator+(const Complex &o) {
        return Complex{real + o.real, img + o.img};
    }

    friend Complex operator-(Complex l, Complex r);
};

Complex operator-(Complex l, Complex r) {
    return Complex{l.real - r.real, l.img - r.img};
}
```

```
std::cout << Complex{4,2} - Complex{2,1} << std::endl;
```

Operatorüberladung – wofür in STUBS?

Streamoperatoren (z.B. cout, abgeleitet von ostream)

```
std::cout << "Die Antwort ist " << std::hex << 66 << std::endl;
```


Operatorüberladung – wofür in STUBS?

Streamoperatoren (z.B. `cout`, abgeleitet von `ostream`)

```
std::cout << "Die Antwort ist " << std::hex << 66 << std::endl;
```

Äquivalent in STUBS: `OutputStream`

OutputStream& OutputStream:: operator<< (bool b) {...}
Rückgabewert Namespace Überladung Parameter Rumpf

Operatorüberladung – wofür in STUBS?

Streamoperatoren (z.B. `cout`, abgeleitet von `ostream`)

```
std::cout << "Die Antwort ist " << std::hex << 66 << std::endl;
```

Äquivalent in STUBS: `OutputStream`

`OutputStream&` `OutputStream::operator<<(bool b) {...}`

Rückgabewert Namespace Überladung Parameter Rumpf

Manipulatoren:

```
OutputStream& hex(OutputStream&){...}
```

Casting & Typkonvertierungen

`const_cast` `static_cast` `dynamic_cast` `reinterpret_cast`

Hinzufügen oder Entfernen der Attribute `const` oder `volatile`

```
const int *x = get();  
int* y = const_cast<int*>(x);
```

Casting & Typkonvertierungen

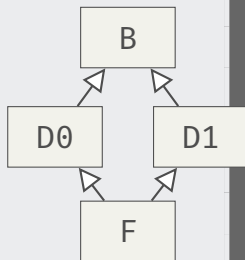
const_cast

static_cast

dynamic_cast

reinterpret_cast

```
D0 d{};
B *x = &d;
%D0 *a = static_cast<D0*>(x); ✓
%D1 *b = static_cast<D1*>(x); ⚡
// Laufzeit: Undef. Verhalten
D1 *c = static_cast<D1*>(a); ⚡
// Compiler: invalid static_cast from
// type 'D1*' to type 'D0*'
```



Casting & Typkonvertierungen

const_cast

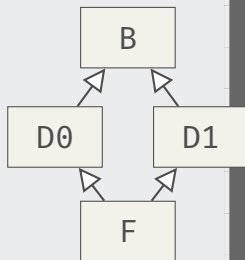
static_cast

dynamic_cast

reinterpret_cast

```
F f{};
B *x = &f;
D0 *a = static_cast<D0*>(x); ✓
D1 *b = static_cast<D1*>(x); ✓
// Okay: F erbt von D0 und D1

D1 *c = static_cast<D1*>(a); ⚡
// Compiler: invalid static_cast from
// type 'D1*' to type 'D0*'
```



Casting & Typkonvertierungen

const_cast

static_cast

dynamic_cast

reinterpret_cast

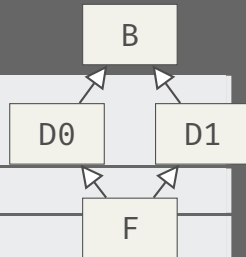
Typprüfung zur Laufzeit (nur polymorphe Klassen):

```
D0 d{};
```

```
B *x = &d;
```

```
D0 *a = dynamic_cast<D0*>(x); ✓
```

```
D1 *b = dynamic_cast<D1*>(x); ⚡ nullptr
```



Casting & Typkonvertierungen

`const_cast` `static_cast` `dynamic_cast` `reinterpret_cast`

„Reinterpretieren“ der Bitwerte

```
char *x = reinterpret_cast<char*>(0xb8000);
```

Casting & Typkonvertierungen

`const_cast` `static_cast` `dynamic_cast` `reinterpret_cast`

„Reinterpretieren“ der Bitwerte

```
char *x = reinterpret_cast<char*>(0xb8000);
```



C-style casts (`int x=(int) malloc(42)`) auch in C++ möglich,
trotzdem **nicht verwenden**

**Learning by Doing:
Aufgabe 0 (C++ Streams)**