

Übung zu Betriebssystembau

C++ Crashkurs

Oktober 2023

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++

```
01 #include <stdio.h>
02
03
04 int main() {
05     const char * str = "Informatik";
06     int n = 4;
07     printf("%s□%d\n", str, n);
08     return 0;
09 }
```

```
01 #include <stdio>
02
03
04 int main() {
05     const char * str = "Informatik";
06     int n = 4;
07     printf("%s□%d\n", str, n);
08     return 0;
09 }
```

```
01 #include <iostream>
02
03
04 int main() {
05     const char * str = "Informatik";
06     int n = 4;
07     std::cout << str << ' ' << n << std::endl;
08     return 0;
09 }
```

```
01 #include <iostream>
02
03
04 int main() {
05     const std::string str = "Informatik";
06     int n = 4;
07     std::cout << str << ' ' << n << std::endl;
08     return 0;
09 }
```



```
01 #include <iostream>
02 using namespace std;
03
04 int main() {
05     const string str = "Informatik";
06     int n = 4;
07     cout << str << ' ' << n << endl;
08     return 0;
09 }
```

Namensräume

```
01 namespace Foo {
02     int getNum() {
03         return 23;
04     }
05 }
06
07 namespace Bar {
08     int getNum() {
09         return 42;
10     }
11 }
12
13 std::cout << Foo::getNum() << std::endl
14           << Bar::getNum() << std::endl;
```

Referenzen

```
01 int foo = 23;  
02 int& bar = foo;  
03 std::cout << bar << std::endl; // Ausgabe: 23  
04  
05 bar = 42;  
06 std::cout << foo << std::endl; // Ausgabe: 42
```

Referenzen

```
01 int foo = 23;
02 int& bar = foo;
03 std::cout << bar << std::endl; // Ausgabe: 23
04
05 bar = 42;
06 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

```
01 void dump(std::ostream &os, const Complex &c) {  
02     os << c.real << "_+_ " << c.img << "i\n";  
03 }
```

Referenzen als Funktionsparameter

Konstante Referenzparameter

```
01 void dump(std::ostream &os, const Complex &c) {  
02     os << c.real << " + " << c.img << "i\n";  
03 }
```

Nicht-konstante Referenzparameter

```
01 void inc(int& i) { i++; }  
02  
03 int foo = 23;  
04 inc(foo);  
05  
06 std::cout << foo << std::endl; // Ausgabe: 24
```

Standardparameter

```
01 void inc(int& i, int n = 1) {  
02     i += n;  
03 }  
04  
05 int foo = 23;  
06 inc(foo);  
07  
08 std::cout << foo << std::endl; // Ausgabe: 24  
09  
10 inc(foo, 2);  
11  
12 std::cout << foo << std::endl; // Ausgabe: 26
```

Überladen von Funktionen

```
01 bool isZero(int i){
02     return i == 0;
03 }
04
05 bool isZero(double i){
06     const double eps = 0.00001;
07     return i < eps && i > -eps;
08 }
```


Überladen von Funktionen

```
01 bool isZero(int i){  
02     return i == 0;  
03 }  
04  
05 bool isZero(double i){  
06     const double eps = 0.00001;  
07     return i < eps && i > -eps;  
08 }
```



Overload resolution ist nicht trivial!

```
01  
02 struct Disp {  
03     char val;  
04     int x, y;  
05 };
```

disp.h:

```
01
02 struct Disp {
03     char val;
04     int x, y;
05 };
```

disp.cc:

```
01 #include "disp.h"
02 using namespace std;
03
04 void func() {
05     struct Disp p;
06     p.val = 'X';
07     p.x = 2;
08     p.y = 0;
09     cout << p.val << endl;
10 }
11 // Geht in C++ nicht:
12 // struct Disp p = {
13 //     .val = 'X',
14 //     .x = 2,
15 //     .y = 0
16 // }
```

disp.h:

```
01
02 struct Disp {
03     char val;
04     int x, y;
05
06     Disp() {
07         val = 'X';
08         this->x = 2;
09         this->y = 0;
10     }
11 };
```

disp.cc:

```
01 #include "disp.h"
02 using namespace std;
03
04 void func() {
05     Disp p;
06     cout << p.val << endl;
07 }
```

disp.h:

```
01
02 struct Disp {
03     char val;
04     int x, y;
05
06     Disp(char c, int x=0, int y
07         =0) {
08         val = c;
09         this->x = x;
10         this->y = y;
11     };
12 }
```

disp.cc:

```
01 #include "disp.h"
02 using namespace std;
03
04 void func() {
05     Disp p('X', 2);
06     cout << p.val << endl;
07 }
```

disp.h:

```
01
02 struct Disp {
03     char val;
04     int x, y;
05
06     Disp(char c, int x=0, int y
07         =0) {
08         val = c;
09         this->x = x;
10         this->y = y;
11     };
12 }
```

disp.cc:

```
01 #include "disp.h"
02 using namespace std;
03
04 void func() {
05     Disp p{'X', 2};
06     cout << p.val << endl;
07 }
```

disp.h:

```
01
02 struct Disp {
03     char val;
04     int x, y;
05
06     Disp(char c, int x=0, int y
07         =0)
08         : val(c), x(x), y(y) {}
09 };
```

disp.cc:

```
01 #include "disp.h"
02 using namespace std;
03
04 void func() {
05     Disp p{'X', 2};
06     cout << p.val << endl;
07 }
```

disp.h:

```
01
02 struct Disp {
03     char val;
04     int x, y;
05
06     Disp(char c, int x=0, int y
07         =0)
08         : val(c), x(x), y(y) {}
09
10     Disp(int p) : val('X'),
11         x(p % 80), y(p / 80) {}
12 };
```

disp.cc:

```
01 #include "disp.h"
02 using namespace std;
03
04 void func() {
05     Disp p{2};
06     cout << p.val << endl;
07 }
```


disp.h:

```
01
02 struct Disp {
03     char val;
04     int x, y;
05
06     Disp(char c, int x=0, int y
07         =0)
08         : val(c), x(x), y(y) {}
09
10     Disp(int p)
11         : Disp('X', p % 80, p / 80)
12         {}
13 };
```

disp.cc:

```
01 #include "disp.h"
02 using namespace std;
03
04 void func() {
05     Disp p{2};
06     cout << p.val << endl;
07 }
```

disp.h:

```
01 int count = 0;
02 struct Disp {
03     char val;
04     int x, y;
05
06     Disp(char c, int x=0, int y
07         =0)
08         : val(c), x(x), y(y) {
09         count++;
10
11     Disp(int p)
12         : Disp('X', p % 80, p / 80)
13         {}
14
15     ~Disp() { count--; }
};
```

disp.cc:

```
01 #include "disp.h"
02 using namespace std;
03
04 void func() {
05     Disp p{2};
06     cout << p.val << "-"
07         << count << endl;
08 }
```

disp.h:

```
01 struct Disp {
02     static int count;
03     char val;
04     int x, y;
05
06     Disp(char c, int x=0, int y
07           =0)
08         : val(c), x(x), y(y) {
09         count++;
10     }
11
12     Disp(int p)
13         : Disp('X', p % 80, p / 80)
14         {}
15 };
16
17 ~Disp() { count--; }
```

disp.cc:

```
01 #include "disp.h"
02 using namespace std;
03
04 void func() {
05     Disp p{2};
06     cout << p.val << "-"
07          << Disp::count << endl;
08 }
09
10 int Disp::count = 0;
```

disp.h:

```
01 struct Disp {
02     static int count;
03     char val;
04     int x, y;
05
06     Disp(char c, int x=0, int y
07         =0);
08
09     Disp(int p)
10         : Disp('X', p % 80, p / 80)
11         {}
12
13     ~Disp();
14 };
```

disp.cc:

```
01 #include "disp.h"
02 using namespace std;
03
04 void func() {
05     Disp p{2};
06     cout << p.val << "-"
07         << Disp::count << endl;
08 }
09
10 int Disp::count = 0;
11
12 Disp::Disp(char c, int x, int y)
13     : val(c), x(x), y(y) {
14     count++;
15 }
16 Disp::~Disp() { count--; }
```

disp.h:

```
01 struct Disp {
02     private:
03         static int count;
04         char val;
05         int x, y;
06
07     public:
08         Disp(char c, int x=0, int y
09             =0);
10         Disp(int p);
11         ~Disp();
12 };
```

disp.cc:

```
01 #include "disp.h"
02 using namespace std;
03
04 void func() {
05     Disp p{2};
06     // Übersetzerfehler bei:
07     cout << p.val << endl;
08 }
09
10 int Disp::count = 0;
11
12 Disp::Disp(char c, int x, int y)
13     : val(c), x(x), y(y) {
14     count++;
15 }
16 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`

Syntax: `class Abgeleitet: Vererbungsart Basis`

```
01 class Foo {  
02     int n;  
03     protected:  
04     int f1(int i){  
05         return i * n;  
06     }  
07 };
```

Syntax: class Abgeleitet: Vererbungsart Basis

```
01 class Foo {  
02     int n;  
03     protected:  
04     int f1(int i){  
05         return i * n;  
06     }  
07 };
```

```
01 class Bar : Foo {  
02     public:  
03     int n; // eigene Var  
04     int f2(int i){  
05         return f1(i) * n;  
06     }  
07 };
```

Vererbung

Syntax: `class Abgeleitet: Vererbungsart Basis`

```
01 class Foo {  
02     int n;  
03     protected:  
04     int f1(int i){  
05         return i * n;  
06     }  
07 };
```

```
01 class Bar : Foo {  
02     public:  
03     int n; // eigene Var  
04     int f2(int i){  
05         return f1(i) * n;  
06     }  
07 };
```

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

```
01 class FooBaz: public Foo, protected Baz
02 {
03     // ...
04 }
```

```
01 class FooBaz: public Foo, protected Baz
02 {
03     // ...
04 }
```

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

```
01 class Foo {
02     public:
03         void f1() { cout << "Foo::f1" << endl; };
04         virtual void f2() { cout << "Foo::f2" << endl; };
05         virtual void f3() = 0;
06 };
07 class Bar : public Foo {
08     public:
09         void f2() { cout << "Bar::f2" << endl; };
10         void f3() { cout << "Bar::f3" << endl; };
11 };
12 class Baz : public Foo {
13     public:
14         void f1() { cout << "Baz::f1" << endl; };
15         void f3() { cout << "Baz::f3" << endl; };
16 };
```

Virtuelle Methoden

```
01 class Foo {
02     public:
03         void f1() { cout << "Foo::f1" << endl; };
04         virtual void f2() { cout << "Foo::f2" << endl; };
05         virtual void f3() = 0;
06 };
07 class Bar : public Foo {
08     public:
09         void f2() override { cout << "Bar::f2" << endl; };
10         void f3() override { cout << "Bar::f3" << endl; };
11 };
12 class Baz : public Foo {
13     public:
14         void f1() { cout << "Baz::f1" << endl; };
15         void f3() override { cout << "Baz::f3" << endl; };
16 };
```

Virtuelle Methoden

```
01 class Foo {
02     public:
03         void f1() {...};
04         virtual void f2() {...};
05         virtual void f3() = 0;
06 };
07 class Bar : public Foo {
08     public:
09         void f2() override {...}
10         void f3() override {...}
11 };
12 class Baz : public Foo {
13     public:
14         void f1() {...}
15         void f3() override {...}
16 };
```

```
01 Foo foo;
02 // Nicht erlaubt
03 // Übersetzerfehler
```

Virtuelle Methoden

```
01 class Foo {
02     public:
03         void f1() {...};
04         virtual void f2() {...};
05         virtual void f3() = 0;
06 };
07 class Bar : public Foo {
08     public:
09         void f2() override {...}
10         void f3() override {...}
11 };
12 class Baz : public Foo {
13     public:
14         void f1() {...}
15         void f3() override {...}
16 };
```

```
01 Bar bar;
02 bar.f1();
03 // "Foo::f1"
04 bar.f2();
05 // "Bar::f2"
06 bar.f3();
07 // "Bar::f3"
```

Virtuelle Methoden

```
01 class Foo {
02     public:
03         void f1() {...};
04         virtual void f2() {...};
05         virtual void f3() = 0;
06 };
07 class Bar : public Foo {
08     public:
09         void f2() override {...}
10         void f3() override {...}
11 };
12 class Baz : public Foo {
13     public:
14         void f1() {...}
15         void f3() override {...}
16 };
```

```
01 Bar bar;
02 bar.f1();
03 // "Foo::f1"
04 bar.f2();
05 // "Bar::f2"
06 bar.f3();
07 // "Bar::f3"

01 Baz baz;
02 baz.f1();
03 // "Baz::f1"
04 baz.f2();
05 // "Foo::f2"
06 baz.f3();
07 // "Baz::f3"
```

Virtuelle Methoden

```
01 class Foo {
02     public:
03         void f1() {...};
04         virtual void f2() {...};
05         virtual void f3() = 0;
06 };
07 class Bar : public Foo {
08     public:
09         void f2() override {...}
10         void f3() override {...}
11 };
12 class Baz : public Foo {
13     public:
14         void f1() {...}
15         void f3() override {...}
16 };
```

```
01 Foo * foo = &bar;
02 foo->f1();
03 // "Foo::f1"
04 foo->f2();
05 // "Bar::f2"
06 foo->f3();
07 // "Bar::f3"

01 foo = &baz;
02 foo->f1();
03 // "Foo::f1"
04 foo->f2();
05 // "Foo::f2"
06 foo->f3();
07 // "Baz::f3"
```

Operatorüberladung & Freundschaft

```
01 class Complex {
02     int real, img;
03     public:
04     Complex(int real, int img) : real(real), img(img) { }
05
06     Complex operator+(const Complex &o) {
07         return Complex{real + o.real, img + o.img};
08     }
09
10     friend Complex operator-(Complex l, Complex r);
11 };
12
13 Complex operator-(Complex l, Complex r) {
14     return Complex{l.real - r.real, l.img - r.img};
15 }
16
17 std::cout << Complex{4,2} - Complex{2,1} << std::endl;
```

Übung zu Betriebssystembau - C++ Crashkurs (WiSe23/24)

Operatorüberladung – wofür in STUBS?

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

```
01 std::cout << "Die_Antwort_ist_" << std::hex << 66 << std::endl;
```


Operatorüberladung – wofür in STUBS?

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

```
01 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`)

```
01 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`

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

Casting & Typkonvertierungen

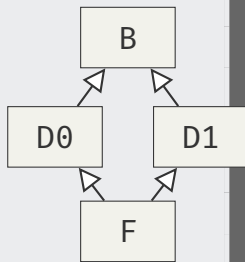
const_cast

static_cast

dynamic_cast

reinterpret_cast

```
01 D0 d{};  
02 B *x = &d;  
03 D0 *a = static_cast<D0*>(x); ✓  
04 D1 *b = static_cast<D1*>(x); ⚡  
05 // Laufzeit: Undef. Verhalten  
06 D1 *c = static_cast<D1*>(a); ⚡  
07 // Compiler: invalid static_cast from  
08 // type 'D1*' to type 'D0*'
```



Casting & Typkonvertierungen

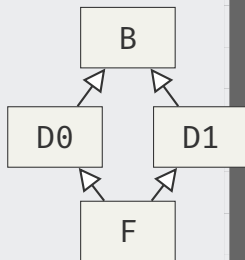
const_cast

static_cast

dynamic_cast

reinterpret_cast

```
01 F f{};
02 B *x = &f;
03 D0 *a = static_cast<D0*>(x); ✓
04 D1 *b = static_cast<D1*>(x); ✓
05 // Okay: F erbt von D0 und D1
06
07 D1 *c = static_cast<D1*>(a); ⚡
08 // Compiler: invalid static_cast from
09 // type 'D1*' to type 'D0*'
```



Casting & Typkonvertierungen

const_cast

static_cast

dynamic_cast

reinterpret_cast

Typprüfung zur Laufzeit (nur polymorphe Klassen):

01

```
D0 d{};
```

02

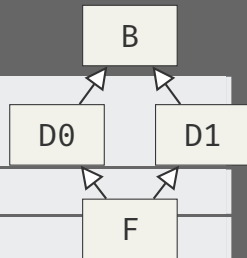
```
B *x = &d;
```

03

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

04

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



Casting & Typkonvertierungen

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

„Reinterpretieren“ der Bitwerte

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

Casting & Typkonvertierungen

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

„Reinterpretieren“ der Bitwerte

```
01 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)**