Übung zu Betriebssystembau

C++ Crashkurs

Oktober 2025

Alexander Krause

Arbeitsgruppe Systemsoftware
Technische Universität Dortmund

(Mit Material vom Lehrstuhl 4 der FAU)

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++

```c
#include <stdio.h>


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

```
1 #include <cstdio>
2
3
4 int main() {
5     const char * str = "Informatik";
6     int n = 4;
7     printf("%s %d\n", str, n);
8     return 0;
9 }
```

```
1 #include <iostream>
2
3
4 int main() {
5     const char * str = "Informatik";
6     int n = 4;
7     std::cout << str << ' ' << n << std::endl;
8     return 0;
9 }
```

```cpp
#include <iostream>


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

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

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

Namensräume

```
1  namespace Foo {
2      int getNum() {
3          return 23;
4      }
5  }
6
7  namespace Bar {
8      int getNum() {
9          return 42;
10     }
11 }
12
13 std::cout << Foo::getNum() << std::endl
14           << Bar::getNum() << std::endl;
```

Referenzen

```
1 int foo = 23;
2 int& bar = foo;
3 std::cout << bar << std::endl; // Ausgabe: 23
4
5 bar = 42;
6 std::cout << foo << std::endl; // Ausgabe: 42
```

Referenzen

```
1 int foo = 23;
2 int& bar = foo;
3 std::cout << bar << std::endl; // Ausgabe: 23
4
5 bar = 42;
6 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

```
1 void dump(std::ostream &os, const Complex &c) {
2     os << c.real << " + " << c.img << "i\n";
3 }
```

Referenzen als Funktionsparameter

Konstante Referenzparameter

```
1 void dump(std::ostream &os, const Complex &c) {
2     os << c.real << " + " << c.img << "i\n";
3 }
```

Nicht-konstante Referenzparameter

```
1 void inc(int& i) { i++; }
2
3 int foo = 23;
4 inc(foo);
5
6 std::cout << foo << std::endl; // Ausgabe: 24
```

Standardparameter

```
1  void inc(int& i, int n = 1) {
2      i += n;
3  }
4
5  int foo = 23;
6  inc(foo);
7
8  std::cout << foo << std::endl; // Ausgabe: 24
9
10 inc(foo, 2);
11
12 std::cout << foo << std::endl; // Ausgabe: 26
```

Überladen von Funktionen

```
1 bool isZero(int i){
2     return i == 0;
3 }
4
5 bool isZero(double i){
6     const double eps = 0.00001;
7     return i < eps && i > -eps;
8 }
```

Überladen von Funktionen

```
1 bool isZero(int i){
2     return i == 0;
3 }
4
5 bool isZero(double i){
6     const double eps = 0.00001;
7     return i < eps && i > -eps;
8 }
```

Overload resolution ist nicht trivial!

```
1
2 struct Disp {
3     char val;
4     int x, y;
5 };
```

disp.h:

```
1
2 struct Disp {
3     char val;
4     int x, y;
5 };
```

disp.cc:

```
1  #include "disp.h"
2  using namespace std;
3
4  void func() {
5      struct Disp p;
6      p.val = 'X';
7      p.x = 2;
8      p.y = 0;
9      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:

```
1
2  struct Disp {
3      char val;
4      int x, y;
5
6      Disp() {
7          val = 'X';
8          this->x = 2;
9          this->y = 0;
10     }
11 };
```

disp.cc:

```
1 #include "disp.h"
2 using namespace std;
3
4 void func() {
5     Disp p;
6     cout << p.val << endl;
7 }
```

disp.h:

```
1
2  struct Disp {
3      char val;
4      int x, y;
5
6      Disp(char c, int x=0, int y=0) {
7          val = c;
8          this->x = x;
9          this->y = y;
10     }
11 };
```

disp.cc:

```
1  #include "disp.h"
2  using namespace std;
3
4  void func() {
5      Disp p('X', 2);
6      cout << p.val << endl;
7  }
```

disp.h:

```
1
2  struct Disp {
3      char val;
4      int x, y;
5
6      Disp(char c, int x=0, int y=0) {
7          val = c;
8          this->x = x;
9          this->y = y;
10     }
11 };
```

disp.cc:

```
1  #include "disp.h"
2  using namespace std;
3
4  void func() {
5      Disp p{'X', 2};
6      cout << p.val << endl;
7  }
```

disp.h:

```
1
2 struct Disp {
3     char val;
4     int x, y;
5
6     Disp(char c, int x=0, int y=0)
7       : val(c), x(x), y(y) {}
8 };
```

disp.cc:

```
1 #include "disp.h"
2 using namespace std;
3
4 void func() {
5     Disp p{'X', 2};
6     cout << p.val << endl;
7 }
```

disp.h:

```
1
2  struct Disp {
3      char val;
4      int x, y;
5
6      Disp(char c, int x=0, int y=0)
7        : val(c), x(x), y(y) {}
8
9      Disp(int p) : val('X'),
10         x(p % 80), y(p / 80) {}
11 };
```

disp.cc:

```
1 #include "disp.h"
2 using namespace std;
3
4 void func() {
5     Disp p{2};
6     cout << p.val << endl;
7 }
```

disp.h:

```
1
2  struct Disp {
3      char val;
4      int x, y;
5
6      Disp(char c, int x=0, int y=0)
7        : val(c), x(x), y(y) {}
8
9      Disp(int p)
10       : Disp('X', p % 80, p / 80) {}
11 };
```

disp.cc:

```
1 #include "disp.h"
2 using namespace std;
3
4 void func() {
5     Disp p{2};
6     cout << p.val << endl;
7 }
```

disp.h:

```
1  int count = 0;
2  struct Disp {
3      char val;
4      int x, y;
5
6      Disp(char c, int x=0, int y=0)
7        : val(c), x(x), y(y) {
8          count++;
9      }
10
11     Disp(int p)
12       : Disp('X', p % 80, p / 80) {}
13
14     ~Disp() { count--; }
15 };
```

disp.cc:

```
1  #include "disp.h"
2  using namespace std;
3
4  void func() {
5      Disp p{2};
6      cout << p.val << "-"
7           << count << endl;
8  }
```

disp.h:

```
1  struct Disp {
2      static int count;
3      char val;
4      int x, y;
5
6      Disp(char c, int x=0, int y=0)
7        : val(c), x(x), y(y) {
8          count++;
9      }
10
11     Disp(int p)
12       : Disp('X', p % 80, p / 80) {}
13
14     ~Disp() { count--; }
15 };
```

disp.cc:

```
1  #include "disp.h"
2  using namespace std;
3
4  void func() {
5      Disp p{2};
6      cout << p.val << "-"
7           << Disp::count << endl;
8  }
9
10 int Disp::count = 0;
```

disp.h:

```
1  struct Disp {
2      static int count;
3      char val;
4      int x, y;
5
6      Disp(char c, int x=0, int y=0);
7
8      Disp(int p)
9        : Disp('X', p % 80, p / 80) {}
10
11     ~Disp();
12 };
```

disp.cc:

```
1  #include "disp.h"
2  using namespace std;
3
4  void func() {
5      Disp p{2};
6      cout << p.val << "-"
7           << Disp::count << endl;
8  }
9
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:

```
1  struct Disp {
2    private:
3      static int count;
4      char val;
5      int x, y;
6
7    public:
8      Disp(char c, int x=0, int y=0);
9      Disp(int p);
10     ~Disp();
11 };
```

disp.cc:

```
1  #include "disp.h"
2  using namespace std;
3
4  void func() {
5      Disp p{2};
6      // Übersetzerfehler bei:
7      cout << p.val << endl;
8  }
9
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!

Vererbung

Syntax: `class Abgeleitet: Vererbungsart Basis`

Vererbung

Syntax: `class Abgeleitet: Vererbungsart Basis`

```
1 class Foo {
2     int n;
3   protected:
4     int f1(int i){
5         return i * n;
6     }
7 };
```

Vererbung

Syntax: `class Abgeleitet: *Vererbungsart* Basis`

```
1 class Foo {
2     int n;
3   protected:
4     int f1(int i){
5         return i * n;
6     }
7 };
```

```
1 class Bar : Foo {
2   public:
3     int n;  // eigene Var
4     int f2(int i){
5         return f1(i) * n;
6     }
7 };
```

Vererbung

Syntax: `class Abgeleitet: Vererbungsart Basis`

```
1 class Foo {
2     int n;
3   protected:
4     int f1(int i){
5         return i * n;
6     }
7 };
```

```
1 class Bar : Foo {
2   public:
3     int n;  // eigene Var
4     int f2(int i){
5         return f1(i) * n;
6     }
7 };
```

Vererbungsart	Elemente aus `Basis`
`public`	`public` und `protected` bleiben *Standard wenn `Abgeleitet` eine Struktur ist*
`protected`	`public` und `protected` werden zu `protected`
`private`	`public` und `protected` werden zu `private` *Standard wenn `Abgeleitet` eine Klasse ist*

Mehrfachvererbung

```
1 class FooBaz: public Foo, protected Baz
2 {
3     // ...
4 }
```

Mehrfachvererbung

```
1 class FooBaz: public Foo, protected Baz
2 {
3     // ...
4 }
```

Falls `Foo` und `Baz` selbe Basisklasse haben → Diamond-Problem

Virtuelle Methoden

Auswahl der **Methode**

nicht-virtuell zur Übersetzungszeit anhand des statischen Typs

virtuell zur Laufzeit anhand des „tatsächlichen“ Typs

Virtuelle Methoden

```
1  class Foo {
2    public:
3      void f1() { cout << "Foo::f1" << endl; };
4      virtual void f2() { cout << "Foo::f2" << endl; };
5      virtual void f3() = 0;
6  };
7  class Bar : public Foo {
8    public:
9      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

```
1  class Foo {
2    public:
3      void f1() { cout << "Foo::f1" << endl; };
4      virtual void f2() { cout << "Foo::f2" << endl; };
5      virtual void f3() = 0;
6  };
7  class Bar : public Foo {
8    public:
9      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

```
1  class Foo {
2    public:
3      void f1() {...};
4      virtual void f2() {...};
5      virtual void f3() = 0;
6  };
7  class Bar : public Foo {
8    public:
9      void f2() override {...}
10     void f3() override {...}
11 };
12 class Baz : public Foo {
13   public:
14     void f1() {...}
15     void f3() override {...}
16 };
```

```
1 Foo foo;
2 // Nicht erlaubt
3 // Übersetzerfehler
```

Virtuelle Methoden

```
1  class Foo {
2    public:
3      void f1() {...};
4      virtual void f2() {...};
5      virtual void f3() = 0;
6  };
7  class Bar : public Foo {
8    public:
9      void f2() override {...}
10     void f3() override {...}
11 };
12 class Baz : public Foo {
13   public:
14     void f1() {...}
15     void f3() override {...}
16 };
```

```
1  Bar bar;
2  bar.f1();
3  // "Foo::f1"
4  bar.f2();
5  // "Bar::f2"
6  bar.f3();
7  // "Bar::f3"
```

Virtuelle Methoden

```
1  class Foo {
2    public:
3      void f1() {...};
4      virtual void f2() {...};
5      virtual void f3() = 0;
6  };
7  class Bar : public Foo {
8    public:
9      void f2() override {...}
10     void f3() override {...}
11 };
12 class Baz : public Foo {
13   public:
14     void f1() {...}
15     void f3() override {...}
16 };
```

```
1  Bar bar;
2  bar.f1();
3  // "Foo::f1"
4  bar.f2();
5  // "Bar::f2"
6  bar.f3();
7  // "Bar::f3"
```

```
1  Baz baz;
2  baz.f1();
3  // "Baz::f1"
4  baz.f2();
5  // "Foo::f2"
6  baz.f3();
7  // "Baz::f3"
```

Virtuelle Methoden

```
1  class Foo {
2    public:
3      void f1() {...};
4      virtual void f2() {...};
5      virtual void f3() = 0;
6  };
7  class Bar : public Foo {
8    public:
9      void f2() override {...}
10     void f3() override {...}
11 };
12 class Baz : public Foo {
13   public:
14     void f1() {...}
15     void f3() override {...}
16 };
```

```
1  Foo * foo = &bar;
2  foo->f1();
3  // "Foo::f1"
4  foo->f2();
5  // "Bar::f2"
6  foo->f3();
7  // "Bar::f3"
```

```
1  foo = &baz;
2  foo->f1();
3  // "Foo::f1"
4  foo->f2();
5  // "Foo::f2"
6  foo->f3();
7  // "Baz::f3"
```

Operatorüberladung & Freundschaft

```
1  class Complex {
2    int real, img;
3   public:
4    Complex(int real, int img) : real(real), img(img) { }
5
6    Complex operator+(const Complex &o) {
7      return Complex{real + o.real, img + o.img};
8    }
9
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;
```

Operatorüberladung – wofür in StuBS?

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

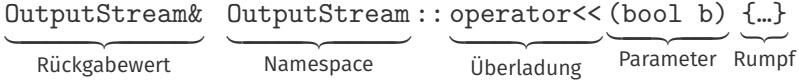
```
1 std::cout << "Die Antwort ist " << std::hex << 66 << std::endl;
```

Operatorüberladung – wofür in StuBS?

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

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

Äquivalent in StuBS: `OutputStream`

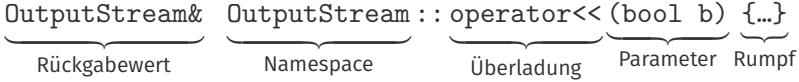


Operatorüberladung – wofür in StuBS?

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

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

Äquivalent in StuBS: `OutputStream`



Manipulatoren:

`OutputStream& hex(OutputStream&){…}`

Casting & Typkonvertierungen

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

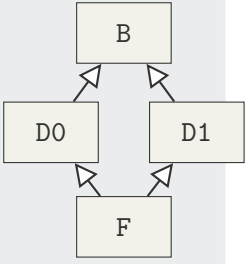
Hinzufügen oder Entfernen der Attribute `const` oder `volatile`

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

Casting & Typkonvertierungen

const_cast | static_cast | dynamic_cast | reinterpret_cast

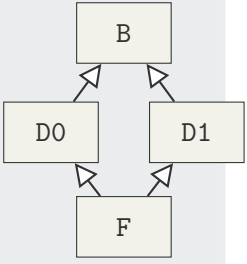
```
1 D0 d{};
2 B *x = &d;
3 D0 *a = static_cast<D0*>(x); ✓
4 D1 *b = static_cast<D1*>(x); ↯
5 // Laufzeit: Undef. Verhalten
6 D1 *c = static_cast<D1*>(a); ↯
7 // Compiler: invalid static_cast from
8 // type 'D1*' to type 'D0*'
```



Casting & Typkonvertierungen

const_cast | static_cast | dynamic_cast | reinterpret_cast

```
1 F f{};
2 B *x = &f;
3 D0 *a = static_cast<D0*>(x); ✓
4 D1 *b = static_cast<D1*>(x); ✓
5 // Okay: F erbt von D0 und D1
6
7 D1 *c = static_cast<D1*>(a); ↯
8 // Compiler: invalid static_cast from
9 // type 'D1*' to type 'D0*'
```

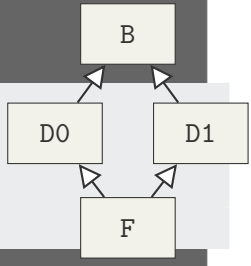


Casting & Typkonvertierungen

const_cast | static_cast | dynamic_cast | reinterpret_cast

Typprüfung zur Laufzeit (nur polymorphe Klassen):

```
1 D0 d{};
2 B *x = &d;
3 D0 *a = dynamic_cast<D0*>(x); ✓
4 D1 *b = dynamic_cast<D1*>(x); ↯ nullptr
```



Casting & Typkonvertierungen

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

„Reinterpretieren“ der Bitwerte

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

Casting & Typkonvertierungen

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

„Reinterpretieren“ der Bitwerte

```
1 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)