OSCI Exercises

C++ crash course

Winter term 2026/27

Manuel Vögele & Benedict Herzog

Bochum Operating Systems and System Software Group (BOSS)

RUHR UNIVERSITÄT BOCHUM

RUB

BOCHUM OPERATING SYSTEMS & SYSTEM SOFTWARE

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^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

ANSI C is (almost) valid C++

```c
#include <stdio.h>


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

```
#include <cstdio>


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

```cpp
#include <iostream>


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

```cpp
#include <iostream>


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

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

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

Namespaces

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

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

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

References

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

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

References

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

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

Differences to Pointers

- initialization at definition
- automatically dereferenced
- cannot be changed
- cannot be NULL

References as function parameters

Constant references

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

References as function parameters

Constant references

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

mutable references

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

int foo = 23;
inc(foo);

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

Defaut Parameter Values

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

int foo = 23;
inc(foo);

std::cout << foo << std::endl; // Output: 24

inc(foo, 2);

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

Function Overloading

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

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

Function Overloading

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

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

Overload resolution is not trivial!

struct & class


```
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
// }
```

struct & class

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;
}
```

struct & class

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;
}
```

struct & class

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;
}
```

struct & class

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;
}
```

struct & class

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;
```

struct & class

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};
    // Compiler error at:
    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--; }
```

```
class Disp {
  private:
    static int count;
    char val;
    int x, y;

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

```
class Disp {
    static int count;
    char val;
    int x, y;

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

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()’
```

→ Frequent error dubbed „most vexing parse“!

Inheritance

Syntax: `class <Derived>: <inheritance type> <Base>`

Inheritance

Syntax: `class <Derived>: <inheritance type> <Base>`

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

Inheritance

Syntax: `class <Derived>: <inheritance type> <Base>`

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

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

Inheritance

Syntax: `class <Derived>: <inheritance type> <Base>`

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

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

inheritance type	Elements from **Base**
public	element's visibility unchanged *Default if* **Derived** *is a struct*
protected	public elements from **Base** become protected
private	public and protected elements from **Base** become private *Default if* **Derived** *is a class*

Multiple Inheritance

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

Multiple Inheritance

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

If Foo and Baz have the same base class → Diamond-Problem

Virtual Methods

Methodselection

non-virtual at compile time based on the static type

virtual at runtime based on the „actual“ type

Virtual Methods

```
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; }
};
```

Virtual Methods

```
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; }
};
```

Virtual Methods

```
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
```

Virtual Methods

```
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"
```

Virtual Methods

```
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"
```

Virtual Methods

```
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"
```

Operator Overloading & friends

```
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;
```

Operator Overlading – usage in StuBS?

Stream operators (z.B. `cout`, derived from `std::ostream`)

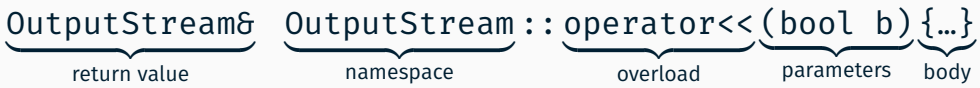
```
std::cout << "The␣answer␣is␣" << std::hex << 66 << std::endl;
```

Operator Overlading – usage in StuBS?

Stream operators (z.B. `cout`, derived from `std::ostream`)

```
std::cout << "The␣answer␣is␣" << std::hex << 66 << std::endl;
```

Equivalent in StuBS: `OutputStream`

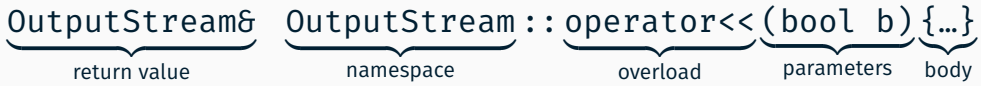


Operator Overlading – usage in StuBS?

Stream operators (z.B. `cout`, derived from `std::ostream`)

```
std::cout << "The␣answer␣is␣" << std::hex << 66 << std::endl;
```

Equivalent in StuBS: `OutputStream`



Manipulators:

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

Casting & Type Conversions

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

Adds or removes the attributes `const` or `volatile`

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

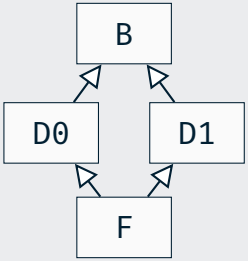
Casting & Type Conversions

const_cast | **static_cast** | dynamic_cast | reinterpret_cast

Type check at compile time:

```
D0 d{};
B *x = &d;
D0 *a = static_cast<D0*>(x); ✓
D1 *b = static_cast<D1*>(x); ↯
// Runtime: undefined behavior

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



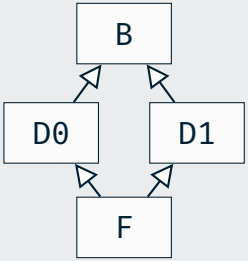
Casting & Type Conversions

const_cast | **static_cast** | dynamic_cast | reinterpret_cast

Type check at compile time:

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

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

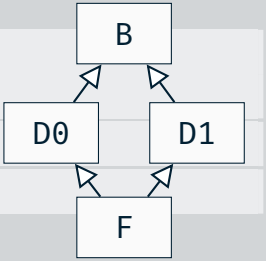


Casting & Type Conversions

const_cast | static_cast | **dynamic_cast** | reinterpret_cast

Type check at runtime (only polymorphic classes):

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



Casting & Type Conversions

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

„reinterpret“ the bit values

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

Casting & Type Conversions

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

„reinterpret“ the bit values

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

C-style casts (`int x=(int) malloc(42)`) are supported in C++, but **should not be used**.

Learning by Doing:
Exercise 0 (C++ Streams)