

Best Practices for Modern C++



CORE GUIDELINES EXPLAINED



RAINER GRIMM

C++ Core Guidelines Explained

C.146

Use `dynamic_cast` where class hierarchy navigation is unavoidable

It's the job of a `dynamic_cast` to navigate in a class hierarchy.

```
struct Base {    // an interface
    virtual void f();
    virtual void g();
};

struct Derived : Base {    // a wider interface
    void f() override;
    virtual void h();
};

void user(Base* pb) {
    if (Derived* pd = dynamic_cast<Derived*>(pb)) {
        // ... use Derived's interface ...
    }
    else {
        // ... make do with Base's interface ...
    }
}
```

To detect the right type for `pb` during run time, a `dynamic_cast` is necessary: `dynamic_cast<Derived*>(pb)`. If the cast fails, you get a null pointer.

A downcast can also be performed with `static_cast`, which avoids the cost of the run-time check. `static_cast` is only safe if the object is definitely `Derived`.

The following rules are two options you have for `dynamic_cast`.

C.147

Use `dynamic_cast` to a reference type when failure to find the required class is considered an error

and

C.148

Use `dynamic_cast` to a pointer type when failure to find the required class is considered a valid alternative

To make it short: You can apply a `dynamic_cast` to a pointer or to a reference. If the `dynamic_cast` fails, you get back a null pointer in the case of a pointer and a `std::bad_cast` exception in the case of a reference. Consequently, use a `dynamic_cast` to a pointer if a failure is a valid option; if a failure is not a valid option, use a reference.

The program `badCast.cpp` shows both cases.

```
// badCast.cpp

struct Base {
    virtual void f() {}
};
struct Derived : Base {};

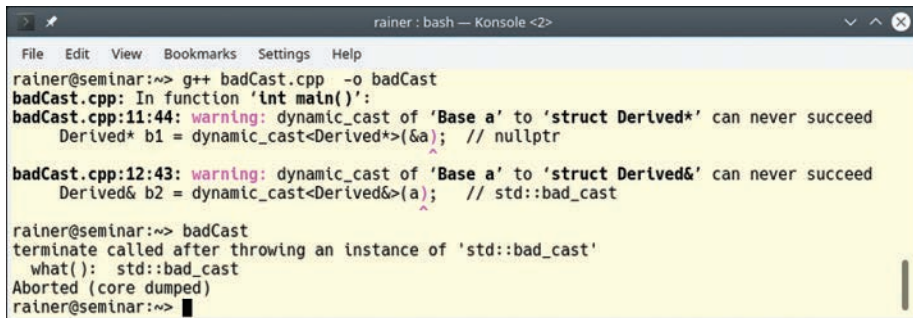
int main() {

    Base a;

    Derived* b1 = dynamic_cast<Derived*>(&a); // nullptr
    Derived& b2 = dynamic_cast<Derived&>(a);   // std::bad_cast

}
```

The `g++` compiler complains about both `dynamic_casts` at compile time. At run time, the program throws the expected exception `std::bad_cast` for the reference (see Figure 5.15).



```
rainer@seminar:~$ g++ badCast.cpp -o badCast
badCast.cpp: In function 'int main()':
badCast.cpp:11:44: warning: dynamic_cast of 'Base a' to 'struct Derived*' can never succeed
    Derived* b1 = dynamic_cast<Derived*>(&a); // nullptr
                                ^
badCast.cpp:12:43: warning: dynamic_cast of 'Base a' to 'struct Derived&' can never succeed
    Derived& b2 = dynamic_cast<Derived&>(a); // std::bad_cast
                                ^
rainer@seminar:~$ badCast
terminate called after throwing an instance of 'std::bad_cast'
what(): std::bad_cast
Aborted (core dumped)
rainer@seminar:~$
```

Figure 5.15 *dynamic_cast causes a `std::bad_cast` exception*

C.152

Never assign a pointer to an array of derived class objects to a pointer to its base

This may not happen very often, but when it happens, the consequences are terribly bad. The result may be an invalid object access or memory corruption. The code snippet shows the invalid object access.

```
struct Base { int x; };
struct Derived : Base { int y; };

Derived a[] = {{1, 2}, {3, 4}, {5, 6}};
Base* p = a; // Bad: a decays to &a[0] which is converted to a Base*
p[1].x = 7; // overwrite Derived[0].y
```

The last assignment should update the Base member `x` of the second array element, but due to pointer arithmetic, it points to the second `int` after `p[0].x`. This happens to be memory of `a[0].y`! The reason is that `Base*` was assigned a pointer to an array of derived objects `Derived`. During this assignment (`Base* p = a;`), the array `a` decays to `&a[0]`, which is converted to a `Base*`.

Decay is the name of an implicit conversion that applies lvalue-to-rvalue, array-to-pointer, and function-to-pointer conversions, removing `const` and `volatile` qualifiers. This means that you can call a function accepting `Derived*` with an array of `Derived`s. Necessary information such as the length of the array of `Derived`s is lost.

In the following code snippet, the function `func` takes its array as a pointer to the first element.

```
void func(Derived* d);
Derived d[] = {{1, 2}, {3, 4}, {5, 6}};

func(d);
```

The array-to-pointer decay is perfectly fine in this `func` case but causes problems in the previous `p[1].x` case.

Overloading and overloaded operators

You can overload functions, member functions, template functions, and operators. You cannot overload function objects, and therefore, you cannot overload lambdas.

The seven rules to overloading and overloaded operators follow one key idea: Build intuitive software systems for your users. Let me rephrase this key idea with a well-known golden rule in software development: Follow the principle of least

astonishment (also known as the principle of least surprise). The principle of least astonishment essentially means that the components of a system should behave in a way that most users will expect them to behave. This principle is very important for overloading and overloaded operators because with great power comes great responsibility.

Although the seven rules address the intuitive behavior of overloading and overloaded operators, they take different perspectives. They address their conventional usage, the implicit conversion of operators, the equivalence of overloaded operations, and the idea that you should overload operators in the namespace of their operands.

Conventional usage

Conventional usage means that the user should not be surprised by unexpected behavior or mysterious side effects of the operators.

C.167

Use an operator for an operation with its conventional meaning

Conventional meaning includes that you use the appropriate operator. For example, here are a few operators that we are used to:

- `==`, `!=`, `<`, `<=`, `>`, and `>=`: comparison operations
- `+`, `-`, `*`, `/`, and `%`: arithmetic operations
- `->`, unary `*`, and `[]`: access of objects
- `=`: assignment of objects
- `<<`, `>>`: input and output operations

C.161

Use nonmember functions for symmetric operators

Conventional meaning includes that your data type should behave like a number if it models a number. This rule is a kind of a generalization of the rule “C.86: Make `==` symmetric with respect to operand types and `noexcept`.”

In general, the implementation of a symmetric operator such as `+` inside the class is not possible.

Assume that you want to implement a type `MyInt`. `MyInt` should support the addition of `MyInt`s and built-in `ints`. Let's give it a try.

```
// MyInt.cpp

struct MyInt {
    MyInt(int v):val(v) {};
    MyInt operator + (const MyInt& oth) const {
        return MyInt(val + oth.val);
    }
    int val;
};

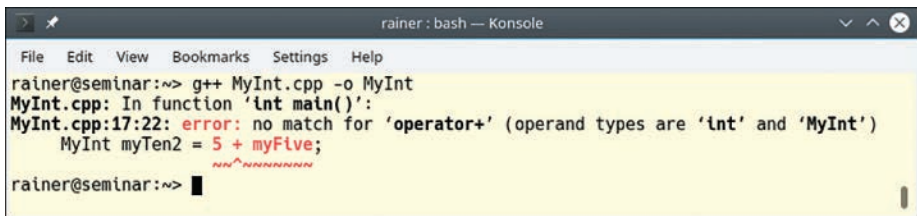
int main() {

    MyInt myFive = MyInt(2) + MyInt(3);
    MyInt myFive2 = MyInt(3) + MyInt(2);

    MyInt myTen = myFive + 5;           // OK
    MyInt myTen2 = 5 + myFive;          // ERROR

}
```

Due to the implicit conversion constructor (`MyInt(int v):val(v)`), the expression `myFive + 5` is valid. Constructors taking one argument are conversion constructors because they take in the concrete case an `int` and return a `MyInt`. In contrast, the last expression `5 + myFive` is not valid because the `+` operator for `int` and `MyInt` is not overloaded (see Figure 5.16).



```
rainer: bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> g++ MyInt.cpp -o MyInt
MyInt.cpp: In function 'int main()':
MyInt.cpp:17:22: error: no match for 'operator+' (operand types are 'int' and 'MyInt')
    MyInt myTen2 = 5 + myFive;
                      ^
rainer@seminar:~>
```

Figure 5.16 Missing overload for `int` and `MyInt`

The small program has many issues:

1. The `+` operator is not symmetric.
2. The `val` variable is public.
3. The conversion constructor is implicit.

It's quite easy to overcome the first two issues with a nonmember operator `+` that is in the class declared as a friend.

```
// MyInt2.cpp

class MyInt2 {
public:
    MyInt2(int v):val(v) {};
    friend MyInt2 operator + (const MyInt2& fir, const MyInt2& sec) {
        return MyInt2(fir.val + sec.val);
    }
private:
    int val;
};

int main() {

    MyInt2 myFive = MyInt2(2) + MyInt2(3);
    MyInt2 myFive2 = MyInt2(3) + MyInt2(2);

    MyInt2 myTen = myFive + 5;    // OK
    MyInt2 myTen2 = 5 + myFive;   // OK

}
```

Now implicit conversion from `int` to `MyInt2` kicks in, and the variable `val` is private. Thanks to the implicit conversion, the `5` in the last line becomes a `MyInt2(5)`.

According to rule “C.46: By default, declare single-argument constructors explicit,” you should not use an implicit conversion constructor.

`MyInt3` has an explicit conversion constructor.

```
// MyInt3.cpp

class MyInt3 {
public:
    explicit MyInt3(int v):val(v) {};
```