5 keywords to save your life

Or at least your programming experience

Eric Elfving

Department of Computer and Information Science
Linköping University

LINKÖPING UNIVERSITY

1. `const`
2. `default` and `delete`
3. `virtual` and `override`
4. `auto`
5. Smart pointers

LINKÖPING UNIVERSITY

You might think that you are perfect, but as a human you will make mistakes! When programming C++, `const` exists to catch many of your mistakes.

Create a function with the following contract:

```
Input:  a vector<int>
Output: the smallest number
```

```
int smallest(vector<int> v)
{
    sort(begin(v), end(v));
    return v.front();
}
```

+ Simple, easy to read implementation
+ Uses standard algorithms
- ?

```
int smallest(vector<int> const & numbers)
{
   sort(begin(numbers), end(numbers));
   return numbers.front();
}
```

- `const` gives compile-time error
  g++:
  `error: assignment of read-only location`

```
int smallest(vector<int> const & numbers)
{
   sort(begin(numbers), end(numbers));
   return numbers.front();
}
```

- `const` gives compile-time error
- `const` works as documentation

```
int smallest(vector<int> const & numbers)
{
   sort(begin(numbers), end(numbers));
   return numbers.front();
}
```

- `const` gives compile-time error
- `const` works as documentation
- `const` helps us fulfill our contract

```
int smallest(vector<int> const & numbers)
{
    assert(numbers.size() > 0);
    return *min_element(begin(numbers), end(numbers));
}
```

- `assert` to check preconditions
- Dereferencing okay since we "know" that there are elements in `numbers`

```
class Bad_Vector
{
public:
    size_t size()
    {
        return arr.size();
    }
    int operator[](size_t idx) const
    {
#if defined DEBUG
        assert(idx < size());
#endif
        return arr[idx];
    }
    void push(int i)
    {
        arr.push_back(i);
    }
private:
    vector<int> arr;
};
int main()
{
    Bad_Vector v;
    v.push(12);
    cout << v[0] << endl;
}
```

- Works fine when compiling as usual

- Works fine when compiling as usual
- Gives

  `error: passing ‘const Bad_Vector’ as ‘this’ argument discards qualifiers`

  when compiling with `-DDEBUG`

- Works fine when compiling as usual
- Gives

  `error: passing 'const Bad_Vector' as 'this' argument discards qualifiers`

  when compiling with `-DDEBUG`
- Calling a non-const function from a const function (or any const environment) is forbidden.

1 const
2 default and delete
3 virtual and override
4 auto
5 Smart pointers

LINKÖPING UNIVERSITY

Whenever you create a class type (`class`, `struct` or `union`), you will get a set of special member functions;

- Default constructor
- Destructor
- Copy and move constructor
- Copy and move assignment operator

Default constructor

If there is no user-declared constructor for class X, a constructor having no parameters is implicitly declared as defaulted. [...] The default constructor will be deleted if:

- it has a non-static data member that is const or reference that is missing an *default member initializer*
- it has a subobject (member or base) without default constructor
- it has a subobject with a missing destructor

(adapted from §12.1/4)

Each non-static data member of type T (without initializer) will be default-initialized:

- If T is a class type, the default constructor is called.
- If T is an array type, each element is default-initialized.
- Otherwise, no initialization is performed.

§8.5/7

Destructor

If a class has no user-declared destructor, a destructor is implicitly declared as defaulted. A defaulted destructor for a class X is defined as deleted if:

- X has a subobject with missing destructor

§12.4/4-5

Destructor

After executing the body of the destructor [...] , a destructor for class X calls the destructors for X’s [...] data members [and] the destructors for X’s direct base classes [...]. Bases and members are destroyed in the reverse order of the completion of their constructor
§12.4/8

Copy and move constructor

A constructor for class X taking a first argument of (possibly cv-qualified) type `X &` and either there are no other parameters or else all other parameters have default arguments is a copy constructor.
Type `X &&` gives move constructor.
§12.8/2-3

If the class definition does not explicitly declare a copy constructor, one is declared implicitly. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy constructor is defined as deleted; otherwise, it is defined as defaulted
§12.8/7

If the definition of a class X does not explicitly declare a move constructor, one will be implicitly declared as defaulted iff

- X does not have a user-declared copy constructor,
- X does not have a user-declared copy assignment operator,
- X does not have a user-declared move assignment operator, and
- X does not have a user-declared destructor.

§12.8/9

A defaulted copy/ move constructor for a class X is defined as deleted if X has:

- a potentially constructed subobject type M that cannot be copied/moved because of a missing constructor or an ambiguity,
- any potentially constructed subobject of a type with a destructor that is deleted or inaccessible from the defaulted constructor, or,
- for the copy constructor, a non-static data member of rvalue reference type.

§12.8/11

The implicitly-defined copy/move constructor for a non-union class X performs a memberwise copy/move of its bases and members.
§12.8/15

Copy and move assignment operators

The rules for generating these members mimic the rules for the corresponding constructor quite well. The exact rules can be found in §12.8/19-30

Back to topic...

To remove a generated special member function, use `delete`. To get a special member function that would be removed according to these rules, use `default`:

```
class Y
{
public:
    Y(int); // removes default constructor
    Y() = default;
    Y(Y const &) = delete; // removes move constructor
    Y(Y&&) = default;
};
```

`default` can only be used with special member functions. `delete` is usually used with special member functions, but can be used in other situations as well:

```
struct Z
{
    static void * operator new(size_t) = delete;
};
```

- Removes possibility of allocating objects on heap

```
struct Foo
{
    Foo(int);

    template <typename T>
    Foo(T const &) = delete;
};
```

- Gives access to a constructor taking `int` but not types convertible to `int`
- Works because of overload resolution

Overload resolution of template functions

1. if there is a normal function that exactly matches the call, that function is selected, else
2. if a function template can be instantiated to exactly match the call, that specialization is selected, else
3. if type conversion can be applied to the arguments, allowing a normal function to be used as a unique best match, that function is selected, else
4. overload resolution fails – either no function matches the call, or the call is ambiguous

1 const
2 default and delete
3 virtual and override
4 auto
5 Smart pointers

LINKÖPING UNIVERSITY

Polymorphic behavior does infer a runtime cost $\Rightarrow$ it is not available by default!
There are two requirements to get polymorphic behavior; a pointer or reference to a base class and `virtual` functions.

```
struct Base
{
    virtual void foo() const { cout << "Base::foo" << endl; }
    void bar() const { cout << "Base::bar" << endl; }
};

struct Derived : public Base
{
    void foo() const override { cout << "Derived::foo" << endl; }
    void bar() { cout << "Derived::bar" << endl; }
};

void fun(Base const & b) // remember the &, otherwise you get slicing
{
    b.foo();
    b.bar();
}

int main()
{
    Base b;
    Derived d;
    fun(b);
    fun(d);
}
```

Printout:

```
Base::foo
Base::bar
Derived::foo
Base::bar
```

`bar` is not `virtual` - `Base::bar` is called.

`override` is not needed, but will help you if there are errors in your code:

```
struct B
{
    virtual void fun() const {}
    int size() const {}
};
struct D: B
{
    void fun() override {}
    int size() const override {}
};
```

g++:
error: void D::fun() marked override, but does not override
error: void D::size() marked override, but does not override

Abstract class

To get an abstract class, at least one function has to be pure virtual.

```
struct Foo
{
    virtual bar() = 0;
};
```

Forbids definition of a `Foo` object. A derived class has to implement (override) `bar` to be a concrete class

One VERY important rule: If you are working with dynamic memory allocation of a polymorphic class hierarchy, ALWAYS define a virtual destructor at base level (can be `default`ed).
Otherwise your program is undefined (if you delete a base-class pointer).

OOP is a great tool, but don't overuse it! Use the tool that fits your problem.

1 `const`
2 `default` and `delete`
3 `virtual` and `override`
4 `auto`
5 Smart pointers

LINKÖPING UNIVERSITY

`auto` is great, use it!

- Guarantees that your variable is initialized
- Removes unnecessary conversions
- You get the correct type; now and if stuff changes
- Easier than hard-to-write types

```
// C++98                   C++14                         C++17
int x = 5;                 auto x {5};
double d = 3.3;            auto d {3.3};
// narrowing conversion
int y = d;                 auto y = int(d);
                           // int{d} to get error

int fun();                 auto fun() -> int;
int foo() { ... }          auto foo() { ... }

map<const char, int>::
const_iterator cit = m.begin();
                           auto cit { cbegin(m) };

pair<int, int> p(2, 4);    auto p {make_pair(2, 4)};    auto p {pair{2, 4}};
```

LINKÖPING UNIVERSITY

Not really a keyword, but if you do need pointer behavior and want to represent ownership - prefer usage of smart pointers `shared_ptr` or `unique_ptr` instead of regular pointers!
Regular pointers is still okay to use - just don't use them to represent ownership (see CppCoreGuidelines R.30).

```
shared_ptr<Base> factory(std::string const & name)
{
    auto args = /* some initializer */;
    if ( name == "d1" )
        return make_shared<d1>(args);
    auto d = make_shared<d2>(args);
    if ( /* some check */ )
        throw exception{};
    return d;
}

void handle_resource(Base * b)
{
    if ( !b )
        cout << "No resource";
    else
        b->do_stuff();
}

void get_resource( unique_ptr<Base> p ); // take ownership of p
```

www.liu.se

LINKÖPING
UNIVERSITY