Moving to Modern C++:
New Features for Class and Hierarchy Design

Steve Dewhurst
Dan Saks

1

Modern C++

The C++ programming language is defined by a formal international standard specification. That standard was updated in 2011 and again in 2014. Modern C++ is the language as specified by these recent standards.

Compared to the earlier standards, Modern C++ introduces a significant number of new language and library features. This course focuses primarily on the language features of Modern C++ and programming techniques that use those features.

2

Saks & Associates
393 Leander Dr.
Springfield, OH 45504-4906 USA
+1-937-324-3601
dan@dansaks.com
www.dansaks.com

3

4

Legal Stuff

- These notes are Copyright © 2017 by Stephen C. Dewhurst and Daniel Saks.
- If you have attended this course, then:
  - You may make copies of these notes for your personal use, as well as backup copies as needed to protect against loss.
  - You must preserve the copyright notices in each copy of the notes that you make.
  - You must treat all copies of the notes — electronic and printed — as a single book. That is,
    - You may lend a copy to another person, as long as only one person at a time (including you) uses any of your copies.
    - You may transfer ownership of your copies to another person, as long as you destroy all copies you do not transfer.

5

More Legal Stuff

- If you have not attended this course, you may possess these notes provided you acquired them directly from Saks & Associates, or:
  - You have acquired them, either directly or indirectly, from someone who has (1) attended the course, or (2) paid to attend it at a conference, or (3) licensed the material from Saks & Associates.
  - The person from whom you acquired the notes no longer possesses any copies.
- If you would like permission to make additional copies of these notes, contact Saks & Associates.

6

About Steve Dewhurst

Steve Dewhurst is the cofounder and president of Semantics Consulting, Inc. He is the author of the critically-acclaimed books *C++ Common Knowledge* and *C++ Gotchas*, and the co-author of *Programming in C++*. He has written numerous technical articles on C++ programming techniques and compiler design.

As a Member of Technical Staff at AT&T Bell Laboratories, Steve worked with C++ designer Bjarne Stroustrup on the first public release of the C++ language and cfront compiler. He was lead designer and implementer of AT&T's first non-cfront C++ compiler. As a compiler architect at Glockenspiel, Ltd., he designed and implemented a second C++ compiler. He has also written C, COBOL, and Pascal compilers.

Steve served on both the ANSI/ISO C++ standardization committee and the ANSI/IEEE Pascal standardization committee.

7

About Steve Dewhurst

Steve has consulted for projects in areas such as compiler design, embedded telecommunications, e-commerce, and derivative securities trading. He has been a frequent and highly-rated speaker at industry conferences such as *Software Development* and *Embedded Systems*. He was a Visiting Scientist at CERT and a Visiting Professor of Computer Science at Jackson State University.

Steve was a contributing editor for *The C/C++ User's Journal*, an editorial board member for *The C++ Report*, and a cofounder and editorial board member of *The C++ Journal*.

Steve received an A.B. in Mathematics and an Sc.B. in Computer Science from Brown University in 1980 and an M.S. in Engineering/ Computer Science from Princeton University in 1982.

8

About Dan Saks

Dan Saks is the president of Saks & Associates, which offers training and consulting in C and C++ and their use in developing embedded systems.

Dan is a contributing editor for *embedded.com* online. He has written columns for numerous print publications including *The C/C++ Users Journal*, *The C++ Report*, *Software Development*, and *Embedded Systems Design*. With Thomas Plum, he wrote *C++ Programming Guidelines*, which won a *1992 Computer Language Magazine Productivity Award*. He has also been a Microsoft MVP.

Dan has taught C and C++ to thousands of programmers around the world. He has presented at conferences such as *Software Development*, *Embedded Systems*, and *C++ World*. He has served on the advisory boards of the *Embedded Systems* and *Software Development* conferences.

9

About Dan Saks

Dan served as secretary of the ANSI and ISO C++ standards committees and as a member of the ANSI C standards committee. More recently, he contributed to the *CERT Secure C Coding Standard* and the *CERT Secure C++ Coding Standard*.

Dan collaborated with Thomas Plum in writing and maintaining *Suite++™, the Plum Hall Validation Suite for C++*, which tests C++ compilers for conformance with the international standard. Previously, he was a Senior Software Engineer for Fischer and Porter (now ABB), where he designed languages and tools for distributed process control. He also worked as a programmer with Sperry Univac (now Unisys).

Dan earned an M.S.E. in Computer Science from the University of Pennsylvania, and a B.S. with Highest Honors in Mathematics/Information Science from Case Western Reserve University.

10

Past C++ Standards

- ***1998***: “C++98”
  - the first international C++ standard (ISO [1998])
- ***2003***: “C++03”
  - a revised international C++ standard (ISO [2003])
  - bug fixes
  - nothing else new
- ***2005***: “TR1”
  - Library “Technical Report 1” (ISO [2005])
  - proposals for library extensions
  - not a new standard

11

Modern C++ Standards

- ***2011***: “C++11”
  - a new international C++ standard (ISO [2011a])
  - significant new language features
  - most of TR1, plus more library components
- ***2014***: “C++14”
  - the latest international C++ standard (ISO [2014])
  - mostly improvements to C++11 features
  - a few new features, too

12

Class Design

13

Special Member Functions

- C++ regards certain member functions as special.
- In C++11, the ***special member functions*** are:

Special Member Function	Typical Form for Class T
default constructor	`T();`
copy constructor	`T(T const &);`
copy assignment operator	`T &operator =(T const &);`
destructor	`~T();`
move constructor	`T(T &&);`
move assignment operator	`T &operator =(T &&);`

- Move constructors and move assignment operators are new additions to this list.

14

Special Member Functions

- What make these functions special is that:
  - C++ implicitly declares these member functions for some class types when the program doesn't explicitly declare them.
  - The compiler implicitly defines a special member function if the program actually uses it.
  - A program can't explicitly define an implicitly-declared special member function.

15

Implicitly…

- These functions are implicitly declared by the compiler to be public and inline.
- They are implicitly defined when they are used.
- If the implicit definition satisfies the conditions to be constexpr, the implicitly-defined function is constexpr.
- Generated special member functions are implicitly declared to be `noexcept`, unless one of the functions they call is `noexcept(false)`.

16

Edge Cases

- This is similar to the case of implicitly-declared copy operations, whose argument type is reference to non-const (!) if one of the copy operations invoked has such an argument.

```
template <typename T>
class X {
    auto_ptr<T> p; // everyone loves auto_ptr!
public:
    // compiler-generated copy operations:
    // X(X &that) : p(that.p) {}
    // X &operator =(X &rhs) { p = rhs.p; return *this; }
};
```

17

Implicitly-Declared Functions

- In general:
  - If you want to provide an operation for a C++ class, define a public member function that will do it.
  - If you want to prevent an operation on that class, do nothing.
- Unfortunately, this approach fails for special member functions.
- Over the years, C++ programmers have adopted different approaches to “turn off” implicitly-declared functions.
- For example…

18

Preventing Copy Operations

- You can declare the copy constructor and copy assignment operator as private members:

```
class widget {
    ~~~
private:
    widget(widget const &);
    widget &operator=(widget const &);
};
```

- Hence, any attempt to copy one `widget` object to another will produce a compile-time access violation.
- Well, almost...

19

Preventing Copy Operations

- Actually, calls to these copy functions from `widget`'s members and friends will compile.
- However, they won't link:
  - The linker message might not be clear or timely, but…
  - It prevents the error from making its way into the executable program.

20

Documenting Implicit Copy Operations

- How would someone reading the code know whether the definitions are missing on purpose or by accident?
  - One common practice is to declare the copy operations and then comment them out.

```
class widget {
public:
    // widget(widget const &);
    // widget &operator=(widget const &);
    ~~~
};
```

- This is nearly as much trouble as implementing them, but it makes clear that they're implicitly defined on purpose.

21

Preventing Copy Operations

- Another idiom for banishing the copy operations is to derive the class from an uncopyable base class:

```
class widget: uncopyable {   // a private base by default
    ~~~
};
```

- Any attempt to copy a `widget` object will trigger a compile error when it attempts to copy its uncopyable base class subobject.
- The Boost library (*www.boost.org*) provides an uncopyable base class called `noncopyable`.

22

Uncopyable Base Classes

- An uncopyable base class has private copy operations and protected default constructor and destructor:

```
class uncopyable {
protected:
    uncopyable() {}
    ~uncopyable() {}
private:
    uncopyable(uncopyable const &);
    uncopyable &operator =(uncopyable const &);
};
```

- The protected members must be called from derived class constructors and destructors.

23

Deleted Functions

- Modern C++ lets you prevent the compiler from implicitly declaring a member function by placing ***= delete*** at the end of the function declaration.
- A function so defined is a ***deleted function***.
- For example, you can disable the generated copy operations for class `widget` by writing:

```
class widget {
public:
    widget(widget const &) = delete;
    widget &operator=(widget const &) = delete;
    ~~~
};
```

24

Extended Deletion

- Note that deleting copy operations implicitly deletes move operations…
- …and vice versa:

```
class widget {
public:
    widget(widget &&) = delete;
    widget &operator=(widget &&) = delete;
    // copy operations are also deleted
    ~~~
};
```

25

Implicit Deleting

- Declaring explicit versions of copying also prevents implicit moves, and vice versa:

```
class gadget {  // no move operations: conventional
public:
    gadget(gadget const &);
    gadget &operator=(gadget const &);
};

class widget {  // no copy operations: not unreasonable
public:
    widget(widget &&);
    widget &operator=(widget &&);
};
```

26

Say What You Mean

- If you want move operations but not copy operations, say so:

```
class widget {
public:
    widget(widget &&);
    widget &operator=(widget &&);
    widget(widget const &) = delete;
    widget &operator=(widget const &) = delete;
    ~~~
};
```

- If you want copy operations and no move operations, it's common to define the copy operations only.
- In that case, copy and move have the same meaning.

27

Defaulted Functions

- Implicitly-declared special members are public and inline.
- Sometimes, you want them to be protected or private or non-inline:

```
class widget {
public:
    virtual ~widget();  // widget's a polymorphic base
    ~~~
protected:  // its copy operations should be protected
    widget(widget const &);
    widget &operator=(widget const &);
    ~~~
private:
    T m, n, p;
};
```

28

Defaulted Functions

- The compiler won't implicitly define an explicitly-declared function.
- Thus, if you declare a special member function, such as a constructor, you have to define it as well.
- This invites errors:

```
widget::widget(widget const &w):
    m (w.m), n (w.n) {          // oops: forgot member p
}
```

- Moreover, compilers don't always optimize explicitly-defined functions as well as they optimize compiler-generated ones.

29

Defaulted Functions

- Modern C++lets you specify default semantics for a special member function by placing `= default` at the end of the function declaration.
- A function so defined is a ***defaulted function***.
- For example, you can implement the generated copy operations as protected members of class `widget` by writing simply:

```
class widget {
    ~~~
protected:
    widget(widget const &) = default;
    widget &operator=(widget const &) = default;
    ~~~
};
```

30

Defaulted Functions

- A class has an implicitly-declared default constructor ***only if*** the class has no explicitly-declared constructors.
- Also, implicitly-declared destructors are non-virtual by default.
- Thus, this class has no default constructor and a non-virtual destructor:

```
class widget {
    ~~~
protected:
    widget(widget const &) = default;
    widget &operator=(widget const &) = default;
    ~~~
};
```

31

Defaulted Functions

- You can use defaulted function definitions to restore or modify default behaviors:

```
class widget {
    ~~~
public:
    widget() = default;
    virtual ~widget() = default;
    explicit widget(widget const &) = default;
protected:
    widget &operator=(widget const &) = default;
    ~~~
};
```

32

Explicit Copy Constructor?

- Well, they're certainly not common, but they do restrict the form of initialization to direct initialization only.

```
widget a;
widget b (a);     // OK
widget c = a;     // error!

void f(widget);
void g(widget &);
f(a);             // error!
g(a);             // OK

widget h()
    { return a; } // error!
```

33

Delayed Defaults

- `= default` need not appear on the first declaration of a function:

```
class widget {
public:
    widget();                    // 1st declaration
    virtual ~widget();           // 1st declaration
    ~~~
};
~~~
widget::widget() = default;      // 2nd declaration
widget::~widget() = default;     // 2nd declaration
```

- This lets you define a defaulted function as a non-inline in a source file instead of a header.

34

Deleted Member Functions

- You can inhibit dynamic allocation for objects of a class type by declaring operators `new` and `delete` as deleted in that class:

```
class widget {
public:
    void *operator new(size_t) = delete;
    void *operator new [](size_t) = delete;
    ~~~
};
~~~
widget *p = new widget;          // compile error
widget *x = new widget [10];     // compile error
```

- Little-known fact: Deleted functions are implicitly inline.

35

Deleted Non-Member Functions

- Non-member functions may also be deleted.
- For example, you may want to `doit` only to rather small objects provided they are not of type `int`:

```
template <typename T>
enable_if_t<(sizeof(T) < 8)> doit(T const &a) { ~~~ }

template <typename T>
enable_if_t<(sizeof(T) >= 8)> doit(T const &) = delete;

void doit(int) = delete;
```

36

One Use For T `const &&`

- You can combine "rvalue reference to const" and deletion in useful ways:

```
void doit(X const &);   // do it to const or non-const Xs
void doit(X const &&) = delete; // but not to temporaries
```

- Constant temporaries are rare, but not unheard of:

```
X const operator +(X const &lhs, X const &rhs);
~~~
X a, b;
doit(a+b);      // compile error!
```

37

Implicitly-Declared Functions

- When does a class T have implicitly-declared member functions?
  - Default Constructor: When T declares no constructor at all.
  - Copy Constructor: When T declares no copy constructor.
    - If T declares an explicit move operation, the constructor is defined as deleted.
    - Otherwise, it's defaulted.
  - Copy Assignment: When T declares no copy assignment.
  - Destructor: When T declares no destructor.
  - Move Constructor: When T declares none of: copy constructor, copy assignment, move assignment, and destructor.
  - Move Assignment: When T declares none of: copy constructor, copy assignment, move constructor, and destructor.

38

Complexity Implies Convention

- The rules for implicit declaration and definition of special members functions are actually a bit more arcane and involved than the previous slide's description.
- To deal with complexity in the past, we heeded

✓ *Jim Coplien's "Orthodox Canonical Form"*

✓ *Andy Koenig and Barb Moo's "Rule of Three"*

- The increased complexity imposed by modern C++ implies:

✓ *Michael Caisse's "Rule of Zero"*

  - That is, do your own thinking!
- Thinking is hard and error-prone, and should be substituted with more reliable convention where possible.
- Let's look at a few conventional cases…

39

Counted Pointer: Copy, Move as Copy

- A simple counted pointer maintains a reference count on a shared object.

```
template <typename T>
class Cptr {
public:
    explicit Cptr(T *p);
    ~Cptr();
    Cptr(Cptr const &) noexcept;
    Cptr &operator =(Cptr const &) noexcept;
    ~~~
private:
  T *p_;
  size_t *c_;
};
```

40


```
template <typename T>
Cptr<T>::Cptr(Cptr const &that) noexcept
  : p_(that.p_), c_(that.c_)
    { ++*c_; }

template <typename T>
Cptr<T> &Cptr<T>::operator =(Cptr const &rhs) noexcept {
  if (this != &rhs) {
    if (!--*c_) {
      delete p_;
      delete c_;
    }
    p_ = rhs.p_;
    c_ = rhs.c_;
    ++*c_;
  }
  return *this;
}
```

41

```
template <typename T>
class Cptr {
public:
    explicit Cptr(T *p) : p_(p), c_(new size_t(1)) {}
    ~Cptr() { if (!--*c) { delete p_; delete c_; } }
    T *operator ->() const noexcept { return p_; }
    T &operator *() const noexcept { return *p_; }
    Cptr(Cptr const &) noexcept;
    Cptr &operator =(Cptr const &) noexcept;
    explicit operator bool() const noexcept
        { return p_ != nullptr; }
private:
    T *p_;
    size_t *c_;
};
```

42

Auto Pointer: Move Only

- Here's an auto pointer that uses actual move operations:

```
template <typename T>
class Aptr {
public:
    explicit Aptr(T *p);
    ~Aptr();
    Aptr(Aptr const &) = delete;              // no copy
    Aptr &operator =(Aptr const &) = delete;
    Aptr(Aptr &&) noexcept;                  // move only
    Aptr &operator =(Aptr &&) noexcept;
    ~~~
private:
  T *p_;
};
```

43

```
template <typename T>
Aptr<T>::Aptr(Aptr &&that) noexcept
  : p_(that.p_)
    { that.p_ = nullptr; }

template <typename T>
Aptr<T> &Aptr<T>::operator =(Aptr &&rhs) noexcept {
  delete p_;
  p_ = rhs.p_;
  rhs.p_ = nullptr;
  return *this;
}
```

- Note that it is recommended, but not necessary, to define the copy operations as deleted.
- The compiler would automatically have defined them as deleted in the presence of explicit move operations.

44


```
template <typename T>
class Aptr {
public:
    explicit Aptr(T *p) : p_(p) {}
    ~Aptr() { delete p_; }
    T *operator ->() const { return p_; }
    T &operator *() const { return *p_; }
    Aptr(Aptr const &) = delete;
    Aptr &operator =(Aptr const &) = delete;
    Aptr(Aptr &&) noexcept;                  // move only
    Aptr &operator =(Aptr &&) noexcept;    // move only
    explicit operator bool() const
        { return p_ != nullptr; }
private:
    T *p_;
};
```

45

Scoped Pointer: No Copy or Move

- A scoped pointer just holds a pointer to an object and deletes it on scope exit.

```
template <typename T>
class Sptr {
public:
    explicit Sptr(T *p);
    ~Sptr();
    Sptr(Sptr const &) = delete;        // no copy or move
    Sptr &operator =(Sptr const &) = delete;
    ~~~
private:
  T *p_;
};
```

46

```
template <typename T>
class Sptr {
public:
    explicit Sptr(T *p) : p_(p) {}
    ~Sptr() { delete p_; }
    T *operator ->() const noexcept { return p_; }
    T &operator *() const noexcept { return *p_; }
    Sptr(Sptr const &) = delete;       // no copy or move
    Sptr &operator =(Sptr const &) = delete;
    explicit operator bool() const noexcept
        { return p_ != nullptr; }
private:
    T *p_;
};
```

47

Clone Pointer: Copy and Move

- A cloning pointer has different copy and move operations.

```
template <typename T>
class Nptr {
public:
    explicit Nptr(T *p);
    ~Nptr();
    Nptr(Nptr const &);                // copy
    Nptr &operator =(Nptr const &);
    Nptr(Nptr &&) noexcept;            // and move
    Nptr &operator =(Nptr &&) noexcept;
    ~~~
private:
  T *p_;
};
```

48

```
template <typename T>
Nptr<T> &Nptr<T>::operator =(Nptr const &rhs) {
    T *temp = rhs.p_->clone();
    delete p_;
    p_ = temp;
    return *this;
}

template <typename T>
Nptr<T> &Nptr<T>::operator =(Nptr &&rhs) noexcept {
    delete p_;
    p_ = rhs.p_;
    rhs.p_ = nullptr;
    return *this;
}
```

49

```
template <typename T>
Nptr<T>::Nptr(Nptr const &that)
    : p_(that.p_ ? that.p_->clone() : nullptr) {}

template <typename T>
Nptr<T>::Nptr(Nptr &&that) noexcept
    : p_(that.p_)
    { that.p_ = nullptr; }
```

50

```
template <typename T>
class Nptr {
public:
    explicit Nptr(T *p) : p_(p) {}
    ~Nptr() { delete p_; }
    T *operator ->() const noexcept { return p_; }
    T &operator *() const noexcept { return *p_; }
    Nptr(Nptr const &);                          // copy
    Nptr &operator =(Nptr const &);
    Nptr(Nptr &&) noexcept;                      // and move
    Nptr &operator =(Nptr &&) noexcept;
    explicit operator bool() const noexcept
        { return p_ != nullptr; }
private:
    T *p_;
};
```

51

A Future Issue

- The following code is valid:

```
class C {
public:
    ~C();
   ~~~
};
```

- Note that C has an explicitly-declared destructor and an implicitly-declared copy constructor.
- The implicit copy constructor is deprecated in this case.
- It's also deprecated if there is an explicit copy assignment.

52

Advice

- This is a very “detailed” area of the standard. Keep it simple.
- ✓ *Remember the “Rule of Three”: Always think about copy initialization, copy assignment, and destruction as a unit.*
- ✓ *Augment the Rule of Three with copy vs. move semantics:*
  - Copyable and movable
  - Copyable only
  - Movable only
  - Neither copyable nor movable
- ✓ *Prefer to be conventional.*
- ✓ *Otherwise, prefer explicit declarations for special members, and `default` or `delete` them as needed.*

53

Effect of Member Template Copy/Move

- Note that a templated constructor or assignment operator will never be used to generate copy or move operations.

```
class Nptr {
public:
    // Oops!  Compiler-generated copy operations
    // Oops!  Compiler-generated move operations,
    //        unless there's a destructor...

    template <typename S>      // copy-like, not copy
        Nptr(Nptr<S> const &);
    template <typename S>
        Nptr &operator =(Nptr<S> const &);
    ~~~
}
```

54

```
class Nptr {
public:
    ~~~
    Nptr(Nptr const &);                    // copy
    Nptr &operator =(Nptr const &);
    template <typename S>                  // copy-like
        Nptr(Nptr<S> const &);
    template <typename S>
        Nptr &operator =(Nptr<S> const &);
    Nptr(Nptr &&) noexcept;                // and move
    Nptr &operator =(Nptr &&) noexcept;
    template <typename S>                  // and move-like
        Nptr(Nptr<S> &&) noexcept;
    template <typename S>
        Nptr &operator =(Nptr<S> &&) noexcept;
    ~~~
};
```

55

In-Class Initializers

- Constructors that perform nearly identical initializations on data members can be hard to maintain.
- Consider this class with three ordinary data members:

```
class Widget {
public:
    ~~~
private:
    string id_;
    size_t seq_;
    bool synced_;
};
```

56

In-Class Initializers

- It has two very similar, yet distinct, constructors:

```
class Widget {
public:
    Widget():
       id_ ("unset"), seq_ (sequence()),
       synced_ (do_sync()) {
    }
    Widget(string const &id):
       id_ (id), seq_ (sequence()),
       synced_ (do_sync()) {
    }
    ~~~
};
```

57

In-Class Initializers

- Modern C++ lets you specify a default initial value for each data member.
- Any constructor that doesn't explicitly initialize a member in its initialization list will use the default value:

```
class Widget {
public:
    Widget() {}                            // 3 defaults
    Widget(string const &id): id_ (id) {}  // 2 defaults
    ~~~
private:
    string id_ = "unset";
    size_t seq_ = sequence();
    bool synced_ = do_sync();
};
```

58

Order Of Initialization

- Note that, as always, data members are initialized in the order that are declared within the class.
- Later initializations may depend on earlier ones.

```
class Widget {
public:
    ~~~
private:
    string id_ = "unset";
    size_t seq_ = id_.size();        // order dependency!
    bool synced_ = id_ != "unset";   // order dependency!
};
```

✓ *Avoid initialization order dependencies.*

59

Syntax of In-Class Initializers

- An in-class initializer can use any of three syntaxes:

```
class Widget {
    ~~~
    string id_ = "unset";   // (1) copy initialization
    string id_ {"unset"};   // (2) direct initialization
    string id_ = {"unset"}; // (3) copy initialization
    string id_ ("unset");   // syntax error...
};
```

- Prefer direct initialization (2) to copy initialization (1 or 3).
  - It looks weird, but Stroustrup recommends it.
- There's no harm in using (1) where direct and copy initialization are the same, but to be consistent it's probably best to use (2).

60

In-Class Initializers and Copying

- Note that compiler-generated copy and move constructors do not use in-class initializers.
- As usual, they use member-by member copy/move.

```
class Widget {
public:
    Widget(string const &id): id_ (id) {}  // 2 defaults
    Widget(Widget const &that) = default;  // no defaults
    ~~~
private:
    string id_ {"unset"};
    size_t seq_ {sequence()};
    bool synced_ {do_sync()};
};
```

61

In-Class Initializers and Aggregates

- In C++11 it was unclear whether in-class initializers prevented a class from being an aggregate.

```
struct Point {          // no declared constructors
    short x {0};
    short y {0};
};
~~~
Point origin;           // OK: calls default constructor
Point p { 17, 211 };    // compiler error in C++11?
```

- p's definition may be an error because `Point` has no constructor that takes two arguments, nor is it an aggregate type.
- In-class initializers are permitted for a C++14 aggregate.

62

Heaps

- Let's design a (quick!) heap container type.
- A heap is a partially-ordered, space-efficient data structure that has several nice properties.
  - You can create a heap from an unordered sequence in linear time.
  - Insertion (push) and erasure (pop) are logarithmic time.
- We'll use the STL heap algorithms to implement our heap…

63

```
template <typename T, typename Comp = less<T>>
class Heap {
public:
    ~Heap() {}
    Heap(Heap const &) = delete;
    Heap &operator =(Heap const &) = delete;
    // constructors go here...
    ~~~
    void pop();
    void push(T const &val);
    T const &top() const;
    bool empty() const;
private:
    vector<T> cont_;
    Comp comp_;
};
```

64

```
void pop() {
    pop_heap(begin(cont_), end(cont_), comp_);
    cont_.pop_back();
}

void push(T const &val) {
    cont_.push_back(val);
    push_heap(begin(cont_), end(cont_), comp_);
}

T const &top() const {
    return cont_.front();
}

bool empty() const {
    return cont_.empty();
}
```

65

Heap Constructors

- Let’s give the heap some constructors:

```
template <typename T, typename Comp = less<T>>
class Heap {
public:
    ~~~
    Heap(Comp comp = Comp());                         // #1
    Heap(size_t n, T const &v, Comp comp = Comp()); // #2
    Heap(size_t n, Comp comp = Comp());             // #3
    ~~~
};
```

66

Some Simplification

- Let's over-simplify for now by initializing the comparator member:

```
template <typename T, typename Comp = less<T>>
class Heap {
public:
    Heap();                         // #1
    Heap(size_t n, T const &v);     // #2
    Heap(size_t n);                 // #3
    ~~~
private:
    vector<T> cont_;
    Comp comp_ {Comp()};
};
```

67

Initializing Heaps

- With this interface in place, we can create some heap objects:

```
Heap<int> h1;                       // #1, default
Heap<int> h2 (5, 0);                // #2
Heap<int> h3 (12);                  // #3

Heap<int> h5 {};                    // #1, default
Heap<int> h6 {5, 0};                // #2
Heap<int> h7 {12};                  // #3
Heap<int> h9 {1, 2, 3};             // error!
```

68

Adding An Initializer List Constructor

- Let's add an initializer list constructor to the mix:

```
template <typename T, typename Comp = less<T>>
class Heap {
public:
    ~~~
    Heap();                        // #1
    Heap(size_t n, T const &v);    // #2
    Heap(size_t n);                // #3
    Heap(initializer_list<T> i);   // #4
    ~~~
};
```

69

Initializing Heaps, Redux

- With this augmented interface, some declarations have changed meaning.

```
Heap<int> h1;                       // #1, default
Heap<int> h2 (5, 0);                // #2
Heap<int> h3 (12);                  // #3

Heap<int> h5 {};                    // #1, default
Heap<int> h6 {5, 0};                // #4, was #2
Heap<int> h7 {12};                  // #4, was #3
Heap<int> h9 {1, 2, 3};             // #4, was error
```

70

Initializer Lists and Overload Resolution

- Overload resolution prefers initalizer list parameters over other types.
- Adding the initializer list constructor changed the behavior of these declarations:

```
Heap<int> h6 {5, 0};    // was 2-arg constructor
Heap<int> h7 {12};      // was 1-arg constructor
```

- Now both use the constructor that accepts an `initializer_list`.

✓ *You should think very carefully before adding a constructor with an `initializer_list` parameter to an existing class.*

71

Good Advice Nobody Likes

- As a class user, consider using parenthesized direct initialization in preference to braced direct initialization unless you intend to call an `initializer_list` constructor:

```
Heap<int> h2 (5, 0);    // always calls 2-arg constructor
Heap<int> h2 {5, 0};    // depends...
```

- While this approach may cause fewer surprises going forward, it unfortunately means abandoning some of the benefits of braced initialization:
  - Arguments may have their values truncated.
  - C++'s "most vexing parse" may appear.
  - It's démodé…

72

Avoiding Future Problems

- As a class designer, you should consider declaring a deleted `initializer_list` constructor if you think you may implement such a constructor in the future:

```
template <typename T, typename Comp = less<T>>
class Heap {
public:
    Heap();
    Heap(size_t n, T const &v);
    Heap(size_t n);
    Heap(initializer_list<T> i) = delete;
    ~~~
};
```

73

Deleted Placeholders

- The deleted "placeholder" will help to avoid surprises if the `initializer_list` constructor is implemented in the future:

```
Heap<int> h6 {5, 0};    // won't compile
Heap<int> h2 (5, 0);    // must use this syntax
Heap<int> h7 {12};      // won't compile
Heap<int> h3 (12);      // must use this syntax
```

- The disadvantages are similar to those mentioned earlier.
- Your users will be forced to avoid braced initialization even if an initializer list constructor is never part of the class interface.
- Your interface—and possibly you—will be criticized.

74

Advice Nobody Likes

- There is as yet no standard practice for when to use braced initialization vs. the traditional forms.
- However, in general we suggest…

✓ *Use braced initialization only or primarily to invoke a constructor that takes an initializer-list.*

✓ *Otherwise, use parenthesized direct initialization to initialize class objects.*

✓ *Use copy initialization to initialize objects of predefined type.*

75

Videlicet

✓ *Use braced initialization only or primarily to invoke a constructor that takes an initializer-list.*

```
vector<int> v {1, 3, 3, 7};
```

✓ *Otherwise, use parenthesized direct initialization to initialize class objects.*

```
string s ("Honi soit qui mal thus puns.");
vector<int> v2 (14, 53);
```

✓ *Use copy initialization to initialize objects of predefined type.*

```
int x = 1376;
```

76

Another Constructor Overload Issue

- Let's look at another common problem with constructor overloading that doesn't necessarily involve `intializer_list`:

```
template <typename T, typename Comp = less<T>>
class Heap {
public:
    Heap();                            // #1
    Heap(size_t n, T const &v);        // #2
    Heap(size_t n);                    // #3
    // Heap(initializer_list<T> i); // #4, now removed
    template <typename In>
    Heap(In b, In e);                  // #5, new range init
    ~~~
};
```

77

Constructor Overload Code Smell

- The range initialization member template may give surprising results:

```
Heap<int> h2 (5, 0);     // #5 chosen (an error), was #2
Heap<int> h6 {5, 0};     // #5 chosen (an error), was #2
```

- The member template is a better match than the non-template two-argument constructor.
- Why?
  - The template is an exact match; `In` is deduced to be `int`.
  - The non-template requires a conversion on the first argument from `int` to `size_t`.

78

One Fix

- One common way to fix this issue is to basically say, "Choose the range constructor only if the arguments are STL iterators."
- Earlier, we used alias templates to generate a number of compile time queries on STL iterators:

```
IsRand<Iter>    // is Iter a random-access STL iterator?
IsIn<Iter>      // is Iter an input STL iterator?
```

- We can use these queries with the `enable_if` to restrict the range constructor arguments to STL input iterators only.
- That is, we can use `enable_if` to cause template argument deduction to fail unless a compile-time condition is met…

79

Disabling the Constructor

```
template <typename T, typename Comp = less<T>>
class Heap {
public:
    ~~~
    template <
        typename In,
        typename =
            typename enable_if<IsIn<In>::value>::type
    > Heap(In b, In e);
};
```

- Here, the required condition is that `In` be an input iterator.

80

Disabling With Style

- We can use an alias template and constexpr function to clean up the syntax a bit:

```
template <bool cond, typename T = void>
using enable_if_t = typename enable_if<cond, T>::type;

template <typename Iter>
constexpr bool is_in() { return IsIn<Iter>::value; }
~~~
template <
    typename In, typename = enable_if_t<is_in<In>()>
> Heap(In b, In e);
```

- The definition of enable_if_t above is part of C++14.

81

Another Fix

- An easier fix in our case is to recognize that some constructors don't make a lot of sense for a heap:

```
template <typename T, typename Comp = less<T>>
class Heap {
public:
    Heap();                          // #1
    Heap(size_t n, T const &v);      // #2
    Heap(size_t n);                  // #3
    template <typename In>
    Heap(In b, In e);                // #5, range init
    ~~~
};
```

- Problem solved.

82

Similar Constructor Declarations

- Let's reintroduce our initializer list constructor.

```
template <typename T, typename Comp = less<T>>
class Heap {
public:
    Heap();                           // #1, default
    Heap(initializer_list<T> init);  // #4, back again
    template <typename In>
    Heap(In b, In e);                 // #5, range init
    ~~~
}
```

- The constructor implementations are similar…

83

Similar Constructor Implementations

```
template <typename In, typename = ~~~>
Heap(In b, In e): cont_ (b, e)
    { make_heap(cont_.begin(), cont_.end(), comp_); }

Heap(initializer_list<T> init): cont_ (init)
    { make_heap(cont_.begin(), cont_.end(), comp_); }
```

- The member initializer lists are similar, and the constructor bodies are identical.
- Duplicated code violates our DRY (Don't Repeat Yourself) design principle, leading to bugs, death, and perdition.

84

Traditional Factoring

- Traditionally, identical code would be factored into an "init" function.
- Of course, an "init" member function can't initialize any class members, only assign to them:

```
template <typename In>
void Heap::common_init(In b, In e) {
    cont_.assign(b, e);        // container assignment
    make_heap(cont_.begin(), cont_.end(), comp_);
}
```

85

Factoring Issues

- Each similar constructor would invoke the (typically protected) common implementation:

```
template <typename In, typename = ~~~>
Heap(In b, In e)
    { common_init(b, e); }

Heap(initializer_list<T> init)
    { common_init(init.begin(), init.end()); }
```

- Note, however, that constructor initializations can't be moved into the implementation function.
- In this case, that may result in inefficiency, in other cases it may not be possible to assign to a member.

86

Delegating Constructors

- A better alternative may be to use a “delegating” or “forwarding” constructor.
- In Modern C++, we can invoke one constructor from another:

```
template <typename In, typename = ~~~>
Heap(In b, In e): cont_ (b, e) // container init
    { make_heap(cont_.begin(), cont_.end(), comp_); }

Heap(initializer_list<T> init):
    Heap (init.begin(), init.end()) {}
```

- The syntax is similar to invoking a base class constructor.
- Note that a forwarded-to constructor may also forward to another constructor, provided that no cycle is produced.

87

No Re-Initializations Allowed

- If a constructor forwards to another, there may be no other initializations on the member-initialization-list.
- The forwarded-to constructor will already have performed any required initializations, and it’s not possible to re-initialize an object.

```
Heap(initializer_list<T> init):
    Heap (init.begin(), init.end()), comp_ (Comp()) {}
            // error: re-initialization ^

Heap(initializer_list<T> init):
    Heap (init.begin(), init.end()) {
    comp_ = Comp();         // OK: assignment
}
```

88

Interaction With Exceptions

- Ordinarily, when an exception leaves a constructor, only initialized subobjects are destroyed.
- The destructor for the object as a whole is not called, because the object does not exist until its constructor exits normally.

```
template <typename In, typename = ~~~>
Heap(In b, In e): cont_ (b, e) {
    make_heap(cont_.begin(), cont_.end(), comp_);
    do_something_that_throws();     // no call to ~Heap
}
```

- In this case, the destructors for `cont_` and `comp_` will be called, but not the `Heap` destructor itself.

89

Interaction With Exceptions

- However, if a delegated-to constructor completes and returns to the delegating constructor, the object is considered to exist.

```
Heap(initializer_list<T> init):
    Heap(init.begin(), init.end()) {
    // at this point, the Heap object exists
    do_something_that_throws();     // will call ~Heap
}
```

- In short: Once any constructor for the *complete* object exits normally, the object exists:
  - No further initialization may be performed on its members.
  - The destructor for the complete object may be invoked.

90

Converting Constructors

- A constructor that can be called with a single argument is a "converting constructor" because it can perform an implicit conversion from the argument to the class:

```
class Rational {
public:
    // both default and converting constructor
    Rational(int num = 0, int denom = 1);
    ~~~
};
~~~
Rational r (1, 7);
r += 3;     // implicit conversion of int to Rational
```

91

Unfortunate Implicit Conversions

- However, most implicit conversions obscure rather than clarify.
- Here's a carelessly-designed container:

```
template <typename T>
class Cont {
public:
    Cont(size_t n);
    friend          // Making New Friends idiom...
    bool operator ==(Cont const &lhs, Cont const &rhs) {
        ~~~
    }
};
```

92

Unfortunate Implicit Conversions

- The `Cont` constructor allows an implicit conversion from `size_t` to `Cont`.

```
Cont<float> a (10);
Cont<float> b (12);

if (a == b) ~~~  // OK, fine...
if (a == 12) ~~~ // compiles, probably not fine
```

- The implicit conversion of `12` to an anonymous `Cont` object is probably not intended.

93

Explicit Constructors

- Declaring the converting constructor `explicit` prevents implicit conversions:

```
template <typename T>
class Cont {
public:
    explicit Cont(size_t n);
    ~~~
};
~~~
Cont<float> a (10);
Cont<float> b (12);
if (a == b) ~~~     // OK
if (a == 12) ~~~    // error! no implicit conversion
if (a == Cont(12))  // OK, you may shoot yourself
```

94

More Opportunities to be Explicit

- Newer language features have given us additional opportunities to decide whether a constructor should be declared explicit.

```
template <
    typename T,
    typename = enable_if<is_integral<T>::value>::type
> class Rational {
public:
    explicit Rational(T, T = 1);
    explicit Rational(initializer_list<T>);
    template <typename... Ts>
    explicit Rational(Rational<Ts> const &...);
    ~~~
};
```

95

The Deadly Conversion Operator

- Most experienced C++ designers have a healthy fear of conversion operators.

```
template <typename T>
class Ptr {
public:
    explicit Ptr(T const *);
    operator bool() const noexcept;
    ~~~
};
~~~
Ptr<T> p1 (new T), p2 (new T);
if (p1) ~~~             // OK, that's clear and proper...
int result = p1 + p2;   // now hold on!
```

96

Safer But Not Safe Conversions

- C++ programmers being… the way they are, they'll start using our smart pointer as an arithmetic type.
- We can improve safety by returning a type that's not arithmetic, but that is convertible to `bool`.

```
operator void *() const;
```

- This is what the `<iostream>` library does, and it's better.
- But it's not perfect:

```
cin >> aVar;
if (cin) ~~~                     // fine...
cout << cin << cerr << cout;     // compiles, but ???
```

97

Bjorn Karlsson's Safe Bool Idiom

- Karlsson [2004] recommends the "Safe Bool" idiom, which tries to be as obscure as possible to avoid hanky-panky:

```
template <typename T>
class Ptr {
public:
    ~~~
    struct PtrBool_ { int member_; };
    typedef int PtrBool_::*PtrBoolType_;
    operator PtrBoolType_() const
        { return p_ != 0 ? &PtrBool_::member_ : 0; }
private:
    T *p_;
};
```

98

Explicit Conversion Operators

- In Modern C++, we have it easy.
- You can declare a conversion operator **explicit**.
- The compiler can apply an explicit conversion operator implicitly only for direct initializations.
- Copy initializations require an explicit cast.

```
template <typename T>
class Ptr {
public:
    explicit Ptr(T const *);
    explicit operator bool() const noexcept;
    ~~~
};
```

99

Explicitly Direct

- Our earlier examples and counter examples are treated properly:

```
Ptr<T> p1 (new T), p2 (new T);
if (p1) ~~~               // OK
int result = p1 + p2;     // error, thankfully
bool isNull = p1;         // error: copy initialization
bool isVeryNull (p1);     // OK: direct initialization

void func(bool);
func(p1);                 // error: copy initialization
func(bool(p1));           // OK: explicit conversion
```

100

Contextual Conversions

- If explicit conversion operators are applied implicitly only for direct initialization, why is it applied in an if-statement's condition?
- Answer: The condition is "contextually converted" to `bool`.
- Converting *expr* to `bool` in this context is valid:

```
if (expr) ~~~
```

if this direct initialization is valid:

```
bool var (expr);
```

101

Braced Initialization For Constructors

- The member-initializer-list of a constructor may employ either the traditional "parens" form of initialization or braced initialization:

```
Name::Name(char const *first, char const *last):
    first_ (first), last_ (last) {}     // traditional

Name::Name(char const *first, char const *last):
    first_ {first}, last_ {last} {}     // trendy
```

- In this context, either is a direct initialization, so there's little *technical* merit in doing it one way or another.
  - Bjarne Stroustrup is trendy.
  - Scott Meyers and yours truly are traditional.

102

Constructors and Argument Passing

- Let's rewrite Name:

```
class Name {
public:
    Name(string const &fst, string const &lst):
        f_ (fst), l_ (lst) {}
    ~~~
private:
    string f_, l_;
};
```

- Passing a string argument by "reference to const" is traditional, but inefficient for rvalue arguments:

```
Name stooge ("Joe", "Besser"); // inefficient
```

103

Special-Casing for Efficiency

- It's common practice to provide special cases for lvalues and rvalues:

```
class Name {
public:
    void change_last(string const &lst) // #1, for lval
        { l_ = lst; }
    void change_last(string &&lst)      // #2, for rval
        { l_ = move(lst); }
    ~~~
};
~~~
string Smith ("Smith");
stooge.change_last(Smith);              // #1, lvalue
stooge.change_last("Smith");            // #2, rvalue
```

104

Special-Casing For Multiple Arguments

- But constructors typically introduce combinatorial issues:

```
class Name {
public:
    Name(string const &fst, string const &lst):
        f_ (fst), l_ (lst) {}              // both lval
    Name(string const &fst, string &&lst):
        f_ (fst), l_ (move(lst)) {}        // lval, rval
    Name(string &&fst, string const &lst):
        f_ (move(fst)), l_ (lst) {}        // rval, lval
    Name(string &&fst, string &&lst):
        f_ (move(fst)), l_ (move(lst)) {}  // both rval
    ~~~
};
```

105

Pass By Value

- An alternative is to pass by value:

```
class Name {
public:
    Name(string fst, string lst):   // no combinatorics
        f_ (move(fst)), l_ (move(lst)) {}
    ~~~
};
```

- If the argument is an lvalue, it's ***copied*** to the parameter.
- If the argument is an rvalue, it's ***moved*** to the parameter.
- In either case, the parameter is then moved to the data member.

106

Cost vs. Complexity

- For lvalues:
  - cost of(pass by value) == copy + move
  - cost of(pass by "lvalue reference to const") == copy
- For rvalues:
  - cost of(pass by value) == move + move
  - cost of(pass by rvalue reference) == move
- If moving is very efficient, this can be an effective way to reduce combinatorial complexity.

107

Restraint!

- Most experts take pains to point out that this use of pass by value should be used thoughtfully and with restraint.
  - Meyers: "*Consider* pass by value for copyable parameters that are *cheap to move* and *always copied*." [Meyers 2015, emphasis added]
  - Sutter: "…'too cute' & probably just an antipattern…" [Sutter, 2014]
- Most experts seem to recommend its use only in multi-argument constructors.
  - Sutter: "…*except for one case*…" [ibid.]

108

What About Universal References?

- We could instead consider having a single function that uses universal references:

```
class Name {
public:
    template <typename S>
    Name(S &&fst, S &&lst)
        : f_(forward<S>(fst)), l_(forward<S>(lst)) {}
private:
    string f_, l_;
};
```

- This will still produce the same explosion of functions.
- But the main problem is that it introduces complexity.

109

Deduction Problems

- The interface introduces problems with argument deduction.

```
template <typename S>
Name(S &&fst, S &&lst);
~~~
Name exstooge ("Joe", Smith);  // #1, error!
Name stooge ("Joe", "Besser"); // #2, error!
```

- In #1, the type name `S` can't be deduced in the first case `"Joe"` and `Smith` have different types.
- The same is true of #2.
- We're passing by reference, so the first argument is `char(&)[4]` and the second is `char(&)[7]`. Deduction fails.

110

Fixing Deduction Problems

- We can fix that by adding a little complexity:

```
class Name {
public:
    template <typename S1, typename S2>
    Name(S1 &&fst, S2 &&lst);
    ~~~
};
```

- We still have too much freedom:

```
Name stooge (12, 32.6); // ???
```

111

Reigning In Creativity

- We can address this problem with a little SFINAE:

```
template <typename T>
using Stringish
    = enable_if_t<is_convertible<T, string>::value>;

class Name {
public:
    template <typename S1, typename S2,
              typename = Stringish<S1>,
              typename = Stringish<S2>>
    Name(S1 &&fst, S2 &&lst);
    ~~~
};
```

112

KISS

- This approach is “clever” but it still doesn’t address the potential problem of a combinatorial explosion of specializations.
- It’s also complex and possibly incorrect.
- Compare this interface

```
template <typename S1, typename S2,
          typename = Stringish<S1>,
          typename = Stringish<S2>>
Name(S1 &&fst, S2 &&lst);
```

- with this one:

```
Name(string fst, string lst);
```

113

114

115

116

Hierarchy Design

117

Preventing Derivation, Traditionally

- Traditional C++ interview question: “How do you prevent derivation from a class?”
- Answer: “Let the class be the only friend of a virtual base class that has private constructors.”

```
class Finalize {
    Finalize() {}                   // private member
    friend class Final;
};

class Final: virtual Finalize {     // private base
public:
    ~~~
};
```

118

Preventing Derivation

- A virtual base subobject of a complete object is initialized by a constructor of the complete object.
- However, only `Final` has access to `Finalize`'s constructor.

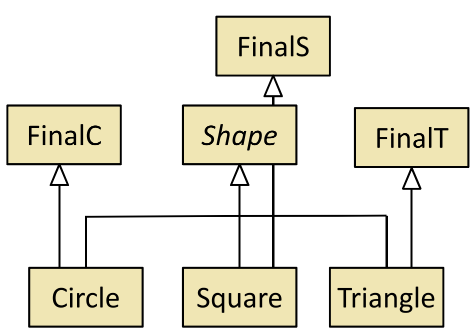
```
class MoreFinal: public Final {
public:
    MoreFinal() {}  // error! call to private constructor
};
```

- The implicit call from the derived class constructor to the remote virtual base class constructor causes a compile-time access violation error.

119

Traditional Problems

- This technique is cute, but not of much use in production code.
  - A virtual base class usually incurs a penalty in both space and time.
  - It doesn't scale well for polymorphic hierarchies, which are usually best designed with abstract base classes and concrete leaves.



120

Modern Version of the Idiom

- Modern C++ augments friend declarations to improve the idiom slightly:

```
template <typename T>
class Finalize {
    Finalize() {}
    friend T;       // OK in C++11, but in not C++03
};
~~~
class Circle: public Shape, virtual Finalize<Circle> {
    ~~~
};
```

- Note: Trendy use of the Curiously-Recurring Template Pattern (CRTP).

121

Final Classes

- In Modern C++, declaring a class to be `final` is both clearer and more efficient:

```
class Final final {
public:
    ~~~
};
~~~
class Finaler: public Final {   // error! that's final!
public:
    ~~~
};
```

- Only class definitions (not declarations) can be declared final.

122

Abstract Bases, Concrete Leaves

- We've always been able to indicate when a class is intended to be a polymorphic base class — it has a public virtual destructor:

```
class Shape {
public:
    virtual ~Shape();   // I'm a polymorphic base class!
    virtual void draw() const = 0;  // and I'm abstract!
    ~~~
};
```

- Now we can specify concrete leaf classes just as clearly:

```
class Circle final: public Shape {  // I'm a leaf.
    ~~~
};
```

123

Abstract Base Classes

- We have a strong preference for abstract base classes in polymorphic hierarchies:

```
class Command {                     // abstract base
public:
    virtual ~Command();
    virtual void operator ()() = 0;
};
```

124

Concrete Base Classes

- There are exceptions…

```
class Decorator: public Command {   // concrete base
public:
    void operator ()() {
        decorated_->operator ()();
        do_something_after();
    }
    ~~~
private:
    Command *decorated_;
};
```

125

```
class Command {                     // abstract base
   ~~~
};

class Decorator: public Command {   // concrete base
    ~~~
};

class WrapDecorator final:
    public Decorator {              // concrete leaf
public:
    void operator ()() {
        do_something_first();
        Decorator::operator()();
    }
    ~~~
};
```

126

Abstract Leaf Classes

- It’s also *possible* to have a `final` abstract class.

```
class ExperimentalShapeBase final: public Shape {
    // No! No! No! Don’t use this yet!
    // When it's ready I’ll take off the final!
public:
    void draw() const = 0;
    ~~~
};
```

- This is not necessarily a recommendation…

127

Final and Standalone Classes

- Doesn’t this irritate you?

```
class DumpableCollection: public vector<Dumpable *> {
public:
    void dump() const {
        ~~~
    }
};
```

- If only…

```
template <typename T, typename A = allocator<T>>
class vector final { ~~~ };
```

128

Standalone Safety: The Final Word

- Polymorphic base classes and standalone classes have very different design idioms.
- It's nice to be able to indicate that in the code:

```
template <
    typename charT,
    typename traits = char_traits<charT>,
    typename A = allocator<charT>
> class basic_string final { ~~~ };
using string = basic_string<char>;
~~~
class MyDangerousString: public string {    // error!
```

- But this is wishful thinking; `basic_string` is not `final`.

129

Should You Take A Stand?

- Should you declare standalone classes `final`?

```
struct Decommission final {
    void operator ()(Widget *p) const {
        decommission(p);
    }
};
```

- It's hard to argue against the practice, but virtually nobody does it.

130

Overriding and the Virtual Keyword

- A derived class member function overrides a base class member function if:
  - the base class member is a virtual function, and
  - the derived class member has the same name and signature as the base class member.
- If the derived class member overrides the base class member, its return type must be compatible (identical or covariant with) that of the base class member.
- If a derived class member overrides a base class member, it is also a virtual function.
- In this case, use of the `virtual` keyword is unnecessary.
  - The declaration has *exactly* the same meaning with or without the keyword.

131

Using Virtual

- Some experts recommend using the `virtual` keyword in derived classes for documentation:

```
class Command {
public:
    virtual void operator ()() = 0;
    virtual Command *clone() = 0;
};

class Macro final: public Command {
public:
    virtual void operator ()(); // overrides
    virtual Macro *clone();     // overrides
    ~~~
};
```

132

Using Virtual

- However, changes to the base class member function can leave the derived class with three virtual functions:

```
class Command {
public:
    virtual void operator ()() = 0;
    virtual Command *clone() const = 0;
};
class Macro final: public Command {
public:
    virtual void operator ()(); // overrides
    virtual Macro *clone();     // no override...
    ~~~                         // but virtual
};
```

133

Not Using Virtual

- Other experts recommend not using the virtual keyword:

```
class Command {
public:
    virtual void operator ()() = 0;
    virtual Command *clone() const = 0;
};
class Macro final: public Command {
public:
    void operator ()(); // overrides, still virtual
    Macro *clone();     // no override, not virtual
    ~~~
};
```

- Neither approach is very attractive.

134

Override

- In Modern C++, you can declare overriding derived class member functions as `override`:

```
class Command {
public:
    virtual void operator ()() = 0;
    virtual Command *clone() const = 0;
};
class Macro final: public Command {
public:
    void operator ()() override;    // OK: overrides
    Macro *clone() override;        // error! ...
    ~~~                             // doesn't override
};
```

135

Remote Overriding

- `override` helps to document remote overrides:

```
class Level1 {
    virtual void f() = 0;
};
class Level2: public Level1 {
};
~~~
class Level4: public Level3 {
    void f() override;  // clearer when base is remote
};
```

- Note that using `virtual` is still optional.

136

Unusual Overriding

- It also helps document unusual overrides.
- Using `virtual` here might confuse a reader into thinking the "re-pured" virtual function was the first declaration of the function in the hierarchy:

```
class Level1 {
    virtual void f() = 0;
};
class Level2: public Level1 {
    void f() override;
};
class Level3: public Level2 {
    void f() override = 0; // regain purity
};
```

137

Overriding Advice

- ✓ *Omit `virtual` when unnecessary; use `override` instead, not in addition.*
- ✓ *Use `virtual` only for the initial declaration.*

138

Final

- Sometimes a base class wants to fix a part of its implementation, but allow other parts to be modified by derived classes.
- Therefore the whole class can't be `final`.
- By convention, non-virtual functions are "invariants over the hierarchy."
  - You can't override them, and you should not hide a non-virtual function.
- Somewhat less commonly, an intermediate base class that overrides an inherited virtual function may wish to prevent further customization of the function by its own derived classes.
- For example, a base class may wish to treat an inherited, overridden virtual function as a Template Method…

139

Final

```
class Creature {
public:
    virtual void move_to(Point) = 0;
    ~~~
};

class Biped: public Creature {
public:
    void move_to(Point p) override; // Template Method
    ~~~
protected:
    virtual void prepare_left() = 0;
    virtual void prepare_right() = 0;
};
```

140

Bipeds Move Similarly

- We want all bipeds to use the same overall algorithm to move:

```
void Biped::move_to(Point p) {
    while (position_ != p) {
        prepare_left();
        move_left();
        prepare_right();
        move_right();
    }
}
```

- A specific biped is supposed to customize this algorithm in a limited way, through overriding of `prepare_left` and `prepare_right`.

141

Customizing a Template Method

- Here's a correct customization:

```
class Judoka: public Biped {
    ~~~
    void prepare_left() override;
    void prepare_right() override;
};
```

142

Customizing a Template Method

- Here's the type of creativity we'd like to avoid:

```
class Drunkard: public Biped {
    ~~~
    void move_to(Point p) override {
        random_walk_to(p);
    }
    void prepare_left() override {}
    void prepare_right() override {}
    ~~~
};
```

143

Final to the Rescue

- We can use `final` to disallow further overriding of a virtual function in derived classes:

```
class Biped: public Creature {
public:
    void move_to(Point p) final;
    ~~~
protected:
    virtual void prepare_left() = 0;
    virtual void prepare_right() = 0;
};
```

144

Final vs. Override

- You can use `final` and `override` together:

```
void move_to(Point p) final override;
```

- It's possible to declare the first declaration of a virtual function to be `final`, but then it must be explicitly declared `virtual` and can't be declared `override`:

```
virtual void move_to(Point p) final = 0;
```

- In any overriding declaration of a virtual function, `final` implies `override`.

145

Clarity Is Good

- There is no convention yet as to whether one should use just `final` or `final override` (or `override final`).
- However, using `virtual`, `override`, and `final` systemically provides a clear thread from the original declaration to the last override.
- For example…

146

Possible Convention

```
class Level1 {
    virtual void f() = 0;
};
class Level2 : public Level1 {
    void f() override;
};
class Level3 : public Level2 {
    void f() override = 0;
};
class Level4 : public Level3 {
    void f() final override;
};
```

147

Identifiers with "Special Meaning"

- The identifiers `final` and `override` are not keywords.
- They take on the function of keywords in specific contexts.
- Meyers calls them "contextual keywords," a pretty descriptive term.
- Elsewhere, they behave just as they did in C++03.
- This is for backward-compatibility with existing code, where the identifiers `final` and `override` may already be in use.
- Of course, contextual keywords open new vistas in obfuscated C++…

148

Don't Do This…

- …except at obfuscated C++ competitions.

```
class initial {
    virtual void override() const = 0;
    virtual void override(int) = 0;
    virtual void final() = 0;
};
```

contextual keyword

```
class final final : public initial {
    void override() override;
    final *override(int) final override;
};
```

identifier

149

Augmented Commands

- Let's augment our command hierarchy to have an id and priority:

```
class Command {
public:
    Command(string const &id, int priority):
        id_ (id), priority_ (priority) {}
    Command(string const &id): Command (id, 0) {}
    Command(): Command ("Unk", 0) {}
    virtual ~Command();
    virtual void operator ()() = 0;
private:
    string id_;
    int priority_;
};
```

150

Derived Drudgery

- Concrete derived classes do not inherit base class constructors.
- Typically, the derived class has to repeat the base class interface even if it adds little to it:

```
class Launch: public Command {
public:
    Launch(string const &id, int priority):
        Command (id, priority) {}
    Launch(string const &id): Command (id) {}
    // Launch();    // we don't want a default for Launch
    void operator ()() override;
};
```

151

Inheriting Constructors

- In this case, each derived class constructor:
  - invokes the corresponding base class constructor, and
  - sets the virtual pointer to the derived class vtable.
- An alternative is to employ a using declaration to inherit a base class's constructors:

```
class Launch: public Command {
public:
    using Command::Command; // inherit all the base ctors
    Launch() = delete;      // ...but hide the default
    void operator ()() override;
};
```

152

Inheriting Constructors

- The effect appears to allow the base constructor to be called when initializing a derived class object:

```
Launch launch ("SuzyQ", -3);
```

- Actually, the effect is identical to our previous derived class implementation.
- The derived class has an implicitly-declared constructor that invokes the corresponding base class constructor.
- In the case of Launch, its virtual functions will properly call the overriding derived-class versions.
- Note that these derived class constructors will be noexcept if the inherited base constructors are.

153

Inheriting Constructors

✓ *Use inheriting constructors feature with caution.*

- It's hazardous if the class adds additional data members:

```
class Apologize: public Command {
public:
    using Command::Command; // inherit all the base ctors
    void operator ()() override;
private:
    char const *apology_text_;          // oops, no init
    bool apology_is_genuine_ {false};   // OK
};
```

- Combining inherited constructors with member initialization can get complex. Keep it simple.

154

Setting Properties of the this Pointer

- The `const` and `volatile` qualifiers can be used to set properties of arguments to functions, including the properties of the `this` pointer.

```
class X {
public:
    // a is non-const, so is *this...
    void operation(T &a);
    // a is const, so is *this...
    void operation(T const &a) const;
    // a is volatile, so is *this...
    void operation(T volatile &a) volatile;
    // yep...
    void operation(T const volatile &) const volatile;
};
```

155

Overloading on Constness of *this

- In a const member function, `this` is a "pointer to const".
- In a non-const member function, `this` is a "pointer to non-const".
- It's common to overload on the constness of a member function:

```
class X {
public:
    T       *find(string const &);          // X *
    T const *find(string const &) const;    // X const *
    ~~~
};
X a;
X const &b = a;
T *p = a.find("this");          // call non-const member
T const *q = b.find("that");    // call const member
```

156

Overloading on Lvalue/Rvalue

- Similarly, we can specify whether an argument can accept only lvalues or only rvalues:

```
class X {
public:
    void op1(T const &arg);     // arg refers to lvalue
    void op1(T &&arg);          // arg refers to rvalue
};

X a;
T t;
a.operation(t);                 // lvalue version
a.operation(T());               // rvalue version
```

157

Overloading on Lvalue/Rvalue of *this

- Using a ref-qualifier, we can specify that the object that this refers to is either an lvalue or an rvalue.

```
template <typename T>
class X {
public:
    void op1(T &arg);   // arg refers to lvalue
    void op1(T &&arg);  // arg refers to rvalue
    void op2() &;       // this refers to lvalue
    void op2() &&;      // this refers to rvalue
};

X<int> a;
a.op2();        // a is lvalue: call lvalue version
X<int>().op2(); // X() is rvalue: call rvalue version
```

158

Overloading on Lvalue/Rvalue of *this

- The property of the object used to call a member function (whether it's an lvalue or an rvalue) can dictate the return type of the function:

```
class X {
public:
    T &get() &; // this refers to lvalue; return lvalue
    T get() &&; // this refers to rvalue; return rvalue
};

X a;
T t;
t = a.get();    // lvalue version, copy assignment
t = X().get();  // rvalue version, move assignment
```

159

160

Bibliography

- Abrahams [2010]. David Abrahams, Rani Sharoni, Doug Gregor, N3050=10-0040
- ISO [1998]. *ISO/IEC Standard 14882:1998, Programming languages—C++*.
- ISO [2003]. *ISO/IEC 14882:2003: Programming languages — C++*.
- ISO [2005]. *ISO/IEC TR 19768, C++ Library Extensions*.
- ISO [2011a]. *ISO/IEC 14882:2011: Programming languages — C++*.
- ISO [2011b]. *ISO/IEC 9899:2011: Programming languages — C*.
- ISO [2014]. *ISO/IEC Standard 14882:2014, Programming languages—C++*.
- Karlsson [2004]. Bjorn Karlsson, “The Safe Bool Idiom”. *The C++ Source*. www.artima.com/cppsource/safebool.html

161

Bibliography

- Meyers [2015]. Scott Meyers, Effective Modern C++. O'Reilly.
- Stroustrup [2013]. Bjarne Stroustrup, *The C++ Programming Language, 4th ed*. Addison-Wesley.
- Sutter [2013]. Herb Sutter, “GotW #94 Solution: AAA Style (Almost Always Auto)”, *Sutter’s Mill*. herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/
- Sutter [2013a]. Herb Sutter, “GotW #91: herbsutter.com/2013/06/05/gotw-91-solution-smart-pointer-parameters/
- Sutter [2014]. Herb Sutter, “Back to the Basics! Essentials of Modern C++ Style”, *CppCon*. www.youtube.com/watch?v=xnqTKD8uD64
- Vandevoorde [2003]. David Vandevoorde and Nicolai Josuttis, *C++ Templates*. Addison-Wesley.

162

163

164