

Const as a Promise

(corrected)

Dan Saks
Saks & Associates
www.dansaks.com

Copyright © 2019 by Dan Saks

1

Uses for const

- To define *symbolic constants*?
 - Yes, but...
 - In Modern C++, `constexpr` is often better.

- To define *immutable (never changing) data*?
 - Yes, but...
 - Again, `constexpr` is often better.

- To *prevent modifying a potentially modifiable operand*?
 - Yes!
 - We see this mostly when passing arguments and returning values *by pointer or reference*.
 - This is the primary use for `const` in Modern C++.

6

Using const is Good Hygiene

- Using **const** turns *potential run-time bugs into compile-time errors*.
 - As compile-time errors, the bugs are impossible to ignore.
- Consequently, using const properly helps make *interfaces*:
 - *easier to use correctly*, and
 - *harder to use incorrectly*.
- Using `constexpr` helps turn run-time computations into compile-time computations.

7

Be Proactive

- Unfortunately, too many programmers use const *reactively*...
 - ...only in response to compiler complaints.
- Conscientious programmers use const *proactively*...
 - ...as they design and code.
- Be conscientious...
 - ✓ *Use const proactively.*

8

What Const Sorta Means

- You can use `const` to define objects of built-in or user-defined types:

```
int const number = 8675309;    // const int
char const msg[] = "hello";    // array of const char
```

- A `const` object is non-modifiable... sorta.**
- That is, you can read from it but not write to it:

```
int n = number;                // OK: can read number
number = 2 * n + 1;            // no: can't write to number
char c = msg[1];               // OK: can read msg
msg[0] = 'H';                 // no: can't write to msg
```

9

Mandated Initialization

- You can't write to an existing `const` object.
- Your only chance to give it a value is when you create it.
- Thus, C++ insists that you **must initialize every `const` object**:

```
namespace example {
    int const upper_bound; // error: missing initializer
    extern int const limit; // OK: not a definition
    int const level = 42;  // OK: explicit initializer
}
```

10

Constant Expressions

- In C++, the dimension in an array object definition must be an *integer constant expression*:

```
void foo(size_t n) {
    int x[n];        // no: dimension must be constant
    int y[17];      // OK: dimension is constant
```

- This is also true elsewhere in C++:

```
struct foo {
    int bf: W;      // field width, W, must be constant
    ~~~
};
```

11

Constant Expressions

- A *constant expression*:
 - can have operators and multiple operands, but
 - *must be evaluated at compile time.*
- In C++, an integer constant object *initialized with a constant expression* is an integer constant expression:

```
int const level = 42; // constant initializer
~~~
int x[2 * level + 1]; // OK: dimension is constant
```

- Surprisingly, a *const object isn't always a constant expression...*

12

Constant Expressions

- C++ lets you initialize a const object with a **non-constant expression**:

```
int n = 42;           // initializer is constant
                    // but n itself isn't
~ ~ ~
int const level = n; // OK: non-constant initializer
```

- The program **might** initialize level at run time.
- In this case, level is **not a constant expression**:

```
int x[2 * Level + 1]; // error: level is non-constant
```

13

Constant Expressions

- You can use constexpr to guarantee compile-time evaluation.
- A **constexpr object** must be initialized with a constant expression:

```
int n = 42;           // non-constant
~ ~ ~
constexpr int max = n; // no: non-constant initializer
~ ~ ~
constexpr int min = 2; // OK: constant initializer
```

- “constexpr is conster than const.” — Steve Dewhurst

✓ **Prefer constexpr to const for defining symbolic constants.**

14

CV-Qualifiers

- Anywhere you can use `const`, you actually can use either:
 - `const`, or
 - `volatile`, or
 - both (in either order).
- Collectively, `const` and `volatile` are *cv-qualifiers*.
 - Type conversions involving `volatile` are very similar to those involving `const`.
- `constexpr` is *not* a cv-qualifier.
 - You can't use `constexpr` *everywhere* that you can use `const`.
 - And vice versa.

15

Key Insights

- `const` objects of arithmetic type are pretty straightforward.
- `const` is more useful when you combine it with pointers or references.
 - It also gets more complicated.
- As with much of C++, understanding `const` is difficult if you can't get past the syntax.
- Here are key insights to help you understand the syntax...

16

The Structure of Declarations

💡 *Insight: Every object and function declaration has two main parts:*

- a sequence of one or more **declaration specifiers**
- a **declarator** (or a sequence thereof, separated by commas)

▪ For example:

static unsigned long int *x[N]
↙ ↖
declaration specifiers declarator

▪ The **name declared** in a declarator is the **declarator-id**.

17

Declaration Specifiers and Declarators

- A **declaration specifier** can be:
 - a **type specifier**:
 - a keyword such as int, unsigned, long, or double
 - a user-defined type, such as string or vector<int>
 - a **non-type specifier**:
 - a keyword such as extern, static, inline, or typedef

18

Declarator Operators

- A **declarator** is a *declarator-id*, possibly surrounded by operators.

💡 *Insight: In a declarator, the operators group according to the same precedence as when they appear in an expression.*

| Precedence | Operator | Meaning |
|------------|----------|------------------|
| Highest | () | grouping |
| | [] | array |
| | () | function |
| Lowest | unary * | pointer |
| | unary & | lvalue reference |
| | unary && | rvalue reference |

19

Declarator Operators

*x[N]

- How do you know whether:
 - x is “a pointer to an array”?
 - x is “an array of pointers”?
- [] has higher precedence than unary *.
- So the winner is...
 - x is “an **array of pointers**”!
- More precisely, x is an “array with N elements of type pointer”.

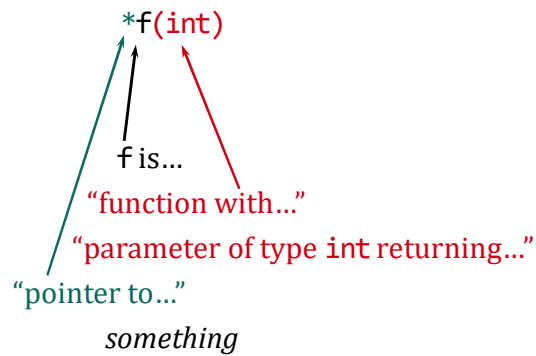
20

Parentheses in Declarators

- Parentheses serve two roles in declarators:
 - As the **function call operator**:
 - These ()s **follow** the declarator-id.
 - They have the same precedence as [].
 - As **grouping**:
 - These ()s **enclose** the declarator-id.
 - They have the highest precedence of all.
- For example...

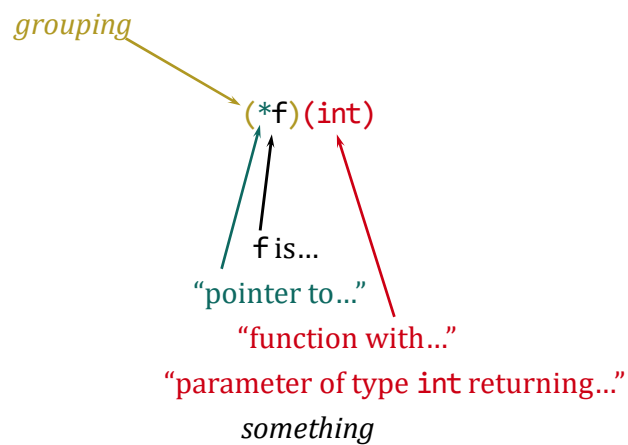
21

Parentheses in Declarators



22

Parentheses in Declarators



23

Type vs. Non-Type Specifiers

💡 *Insight: Type specifiers modify other type specifiers.*

💡 *Insight: Non-type specifiers apply directly to the declarator-id.*



- Here, `unsigned`, `long`, and `int` are type specifiers.
 - They form the type to which the pointers in array `x` point.
- `static` is a non-type specifier that applies directly to `x`.

24

Declaration Specifier Order

💡 *The order of the declaration specifiers doesn't matter to the compiler.*

- These two declarations mean the same thing:

```
unsigned long ul;           // unsigned long
long unsigned ul;         // same thing
```

- So do these three:


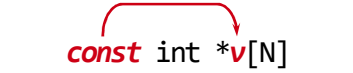


```
const unsigned long cul;   // const unsigned long
long unsigned const cul;   // same thing
unsigned const long cul;   // same, and we're not amused
```

25

const is a Type Specifier

💡 *const is a type specifier, much like Long or unsigned.*

💡 *const modifies the other **type** specifier(s) in the same declaration.*

| <i>right interpretation</i> | <i>wrong interpretation</i> |
|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
|  |  |
|  |  |

- v is an object of type “array of N pointers to const int”.

26

const in Declarators

💡 *Insight: `const` and `volatile` are the only symbols (in C++) that can appear either as declaration specifiers or in declarators.*

- In both of these, `const` is a type specifier:

```
const int | *v[N] // const modifies int
int const | *v[N] // same thing
```

- Here, `const` appears in the declarator:

```
int | *const v[N] // const modifies the * (the pointer)
```

27

const in Declarators

💡 *`const` appearing to the immediate right of a `*` in a declarator turns the pointer into a “const pointer”.*

```
widget *const cpw // const pointer to...
widget *const *pcpw // pointer to const pointer to...
widget **const cppw // const pointer to pointer to...
```

28

const in Declarators

- * and const are separate tokens.
- The spacing around the * doesn't matter to the compiler:

```
widget*const cpw      // const pointer to...
widget* const cpw     // const pointer to...
widget *const cpw     // const pointer to...
widget * const cpw    // const pointer to...
```

- However, * followed by const is effectively a single operator — the “*const pointer*” operator.
- ***const** has the *same operator precedence* as just *****.

29

Declarations That Mean What You Intend

- There's a simple way to ensure that you're placing const (or volatile) where you want it in a declaration:
 - First, write the declaration as it would be without const (or volatile).
 - Then...
- ✓ *Place const (or volatile) to the immediate right of the type specifier or operator that you want it to modify.*

30

Declarations That Mean What You Intend

- For example, suppose we want x to be:
 - “array of N *const pointers* to *volatile uint32_t*”.
- Start by writing the declaration for:
 - “array of N *pointers* to *uint32_t*”...

```
uint32_t *x[N];
```

31

Declarations That Mean What You Intend

- Here it is again, with room for the cv-qualifiers:


```
uint32_t      *      x[N];
```
- Next, add *const* to the immediate right of the *:


```
uint32_t      *const x[N];
```
- Finally, add *volatile* to the immediate right of *uint32_t*:


```
uint32_t volatile *const x[N];
```
- Bob’s your uncle!
 - x is an “array of N *const pointers* to *volatile uint32_t*”.

32

So What About constexpr?

- Again, these are equivalent:

```

constexpr char | *p
char constexpr | *p
    
```

- constexpr is a declaration specifier.
- So these are equivalent, too:

```

constexpr char | *p
char constexpr | *p
    
```

33

So What About constexpr?

- Surprisingly, these are *not equivalent*:

```

char constexpr | *p
char const | *p
    
```

- These *are*:

```

char constexpr | * p // constexpr pointer to char
char | *const p // const pointer to char
    
```

34

Type vs. Non-Type Specifiers

- Syntactically, `constexpr` is *not* a type specifier.
- It behaves more like a *non-type specifier*:

`constexpr` unsigned long int ***x**[N]



- Here, `constexpr` modifies `x`, not the other type specifiers.
- `x`'s type is as if it were declared as:

unsigned long int ***const** x[N]

- But the initializer must be a constant expression.

35

Address-Of

- The address-of operator, `&`, preserves constness:

```
int i;
int const ci = 42;
~
&i           // yields "pointer to [non-const] int"
&ci          // yields "pointer to const int"
```

36

Array-to-Pointer Conversions

- In various contexts, arrays implicitly convert to pointers.
- The *array-to-pointer conversion* preserves constness:
 - “array of [non-const] T” converts to “pointer to [non-const] T”.
 - “array of const T” converts to “pointer to const T”.
- String literals, such as "xyzy", have type “array of const char”.
 - They convert to “pointer to const char”.
- Now, let’s look at each way you can place const in a pointer declaration...

37

const Placement and Meaning

```
const T *p      // [non-const] pointer to const T
T const *p      // same
```

- In this case:
 - p is a “[non-const] pointer to const T”
 - *p is a “const T”.

- Meaning:

```
T const x = something;
p = &x;      // OK: can modify pointer itself
*p = x;      // no: can't modify T that p points to
```

38

const Placement and Meaning

```
T *const p           // const pointer to [non-const] T
```

- In this case:
 - p is a “const pointer to [non-const] T”.
 - *p is a “[non-const] T”.

- Meaning:

```
T x, y;
p = &x;    // no: can't modify pointer itself
*p = y;    // OK: can modify T that p points to
```

39

const Placement and Meaning

```
const T *const p     // const pointer to const T
T const *const p     // same
```

- In either case:
 - p is a “const pointer to const T”.
 - *p is a “const T”.

- Meaning:

```
T const x = something;
p = &x;    // no: can't modify pointer itself
*p = x;    // no: can't modify T that p points to
```

40

Conversions Involving const

- Consider these declarations:

```
T *p;           // pointer
T const *pc;   // pointer to const
~~~
void wp(T *q);  // wants pointer
void wpc(T const *qc); // wants pointer to const
```

- Clearly, these calls are OK:

```
wp(p);        // argument and parameter types match exactly
wpc(pc);      // here again
```

41

Conversions Involving const

- Here are the declarations, again:

```
T *p;           // pointer
T const *pc;   // pointer to const
~~~
void wp(T *q);  // wants pointer
void wpc(T const *qc); // wants pointer to const
```

- Now, are these calls OK?

```
wp(pc); // convert "pointer to const" into "pointer"?
wpc(p); // convert "pointer" into "pointer to const"?
```

- To answer this...

42

const as a Promise

✓ *Think of const as a promise.*

- You can apply ethical reasoning to the semantics of const.
- Imagine a conversation between:
 - **K**: the current *Keeper of X*, and
 - **B**: the *Borrower of X*...

43

const as a Promise

- **K**: “If I give you access to *X*, can I trust that you won’t change it?”
- **B**: “Yes, I promise I won’t change *X*.”
- **K**: “You understand that I *might* still be able to change *X*, and give others permission to do so as well?”
- **B**: “Yes, my promise not to change *X* doesn’t necessarily mean no one else can change *X*.”
- **K**: “You also understand that you can’t weasel out by asking others to break the promise for you?”
- **B**: “Yes, I do. I won’t ask others to break my promise.”

44

const as a Promise

- Let's apply this to:

```
T const *qc          // pointer to const
```

- The promise is that:
 - The program won't use any value **obtained** from qc — directly or indirectly — to alter any T objects.
- This promise doesn't necessarily apply to any other pointer...
- ...even if that pointer happens to have the same value as qc.

45

const as a Promise

- Now, let's apply this reasoning to calling wp(pc):

```
T const *pc;          // pc is the keeper of access
~~~~
void wp(T *q);        // wp borrows access via q, but
~~~~                 // q makes no promise
wp(pc);              // error: compiler won't trust wp
```

- Calling wp(pc) provokes a compile error:
 - It's an invalid pointer conversion that loses constness.
- If it compiled, it might allow code within wp to violate the promise in pc's declaration.

46

const as a Promise

- In contrast, calling `wpc(p)` is OK:

```
T *p;           // p is the keeper of access
~~~~          // it makes no promise
void wpc(T const *qc); // wpc borrows access via qc
~~~~          // promising to not write to *qc
wpc(p);        // OK: violates no promises
```

- Passing `p` to `wpc` (as `qc`) doesn't violate any promises.
- The call `wpc(p)` involves a particular kind of type conversion...

47

Qualification Conversion

- A **qualification conversion** adds cv-qualifiers to the type to which the converted pointer points.
- More precisely, a qualification conversion converts:
 - an object of type "pointer to CV_1 T"
 - into type "pointer to CV_2 T",
- where:
 - CV_1 is either empty, const, volatile, or const volatile, and
 - CV_2 is **more cv-qualified** than CV_1 .
 - That is, CV_2 has every qualifier in CV_1 , plus at least one more.

48

Qualification Conversion

- Qualification conversions apply in assignments as well as parameter passing, as in:

```
T *p;
T const *pc;
T volatile *pv;
~~~
pc = p;      // OK: adds const
p = pc;      // error: loses const
pv = p;      // OK: adds volatile
p = pv;      // error: loses volatile
pv = pc;     // error: adds volatile, but loses const
pc = pv;     // error: adds const, but loses volatile
```

49

Reference Initialization

- Reference initialization behaves similarly to qualification conversions for pointers:

```
int i = 37;           // int
int const ci = 42;   // const int
~~~
void wi(int &r);      // wants int
void wci(int const &rc); // wants const int
```

- Clearly, these calls are OK:

```
wi(i);      // binds "ref to int" to "int"
wci(ci);    // binds "ref to const int" to "const int"
```

50

Reference Initialization

- A reference initialization *can increase* the constness of the referenced object.

```
int i = 37;           // int
int const ci = 42;   // const int
~~~
void wi(int &r);      // wants int
void wci(int const &rc); // wants const int
```

- Thus, this is OK, too:

```
wci(i); // binds "ref to const int" to "int"
```

- It doesn't break any promises.

51

Reference Initialization

- A reference initialization *can't decrease* the constness of the referenced object.

```
int i = 37;           // int
int const ci = 42;   // const int
~~~
void wi(int &r);      // wants int
void wci(int const &rc); // wants const int
```

- This is not OK:

```
wi(ci); // no! can't bind "ref to int" to "const int"
```

- Binding parameter r to ci might break ci's promise.

52

Iterators

- A standard library *iterator* is a *generalization of a pointer*.
- It's an object that behaves in many ways just like a pointer:
 - An iterator might actually be a pointer.
 - Or, it might be a class object with overloaded operators that let it act like a pointer, primarily:
 - unary `*`
 - `++`

53

Iterators

- Every standard container defines member types:
 - *iterator* is the type for an object that can “point” to an element in a container.
 - *const_iterator* is the type for an object that can “point” to an element in a const-qualified container.
- A `const_iterator` really behaves like a “*pointer to const*”...
 - Not a “const pointer”.

54

Iterators

- Conversions between `iterator` and `const_iterator` types mimic qualification conversions:

```
#include <deque>
using namespace std;
~~~
deque<int>::iterator i;
deque<int>::const_iterator ci;
~~~
i = ci;           // no: drops const from "pointed to" type
ci = i;          // OK: adds const to "pointed to" type
```

55

Using const in Parameter Declarations

- Some ways of placing `const` are more useful than others.
- Suppose, for some types `R` and `T`, you have a function declared as:

```
R foo(T *p);
```

- A typical call to `foo` passes an “array of `T`”:

```
T x[N];           // for some positive integer constant N
~~~
foo(x);
```

- Let’s consider different placements for `const` in the parameter list...

56

Using const in Parameter Declarations

```
R foo(T const *p);           // (1)
```

```
~~~~
```

```
foo(x);
```

- foo can alter p and use p to inspect any element of x:

```
for (; *p != v; ++p) {       // OK
    if (*p == one_thing) {   // OK
        }
    }
}
```

- However...

57

Using const in Parameter Declarations

```
R foo(T const *p);           // (1)
```

```
~~~~
```

```
foo(x);
```

- foo can't use p to change any element of x:

```
for (; *p != v; ++p) {       // OK
    if (*p == one_thing) {   // OK
        *p = another_thing;  // error
    }
}
```

- This is a **meaningful constraint** on foo's behavior.
- If this is the behavior you want, then this is the way to get it.

58

Using const in Parameter Declarations

```
R foo(T *const p);           // (2)
~ ~ ~
foo(x);
```

- Using this declaration for p, foo can't alter p itself.
- However, it can use p to alter the value of x[0].
- This const is pretty *useless, if not deceptive*:
 - It's reasonable to expect that const in a parameter list affects the function's outward behavior.
 - But this const doesn't...

59

Using const in Parameter Declarations

```
R foo(T *const p);           // (2)
~ ~ ~
foo(x);
```

- This const constrains foo's implementation.
 - foo can't change its own copy of x's address.
- It doesn't affect foo's outward behavior.
 - foo can still change the contents of x using...

60

Using const in Parameter Declarations

```
R foo(T *const p) {
    T *q = p;           // OK: copying const object
    while (~) {
        *q++ = something; // OK: doesn't break promises
    }
}
```

- Here's another way to do the same thing:

```
R foo(T *const p) {
    size_t i = 0;
    while (~) {
        p[i++] = something; // OK: doesn't break promises
    }
}
```

61

Using const in Parameter Declarations

```
R foo(T const *const p); // (3)
~
foo(x);
```

- This is overkill.
- From the caller's perspective, it's the same as:

```
R foo(T const * p); // (1)
```

- The rightmost const has no affect on foo's outward behavior.

62

Using const in Parameter Declarations

- ✓ *Declare a pointer parameter as “pointer to const” if the function shouldn’t alter the “pointed to” object(s).*
- Using const in this way imposes a meaningful constraint on the function’s outward behavior.
- The guideline for reference parameters is similar...

63

Using const in Parameter Declarations

- ✓ *Declare a reference parameter as “reference to const” if the function shouldn’t alter the referenced object.*

- For example, the source operand of a copy should be “reference to const”:

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

64

Top-Level CV-Qualifiers

- Types in C++ can have one or more levels of composition.
- For example, type “pointer to char” has two levels:
 - 1) “pointer to”
 - 2) “char”
- Type “array of pointer to int” has three levels:
 - 1) “array of”
 - 2) “pointer to”
 - 3) “int”
- Type “string” has just one level:
 - 1) “string”

67

Top-Level CV-Qualifiers

- A cv-qualifier on the first level of a type is called a ***top-level cv-qualifier***.
- For example, these declarations have top-level cv-qualifiers:

```
T *const p           // top-level is const
T const *volatile q  // top-level is volatile
```

- These don't:

```
T x                   // no top-level cv-qualifier
T const volatile *p  // no top-level cv-qualifier
```

68

Top-Level const in Parameter Declarations

- The top-level const in this declaration is pretty useless:

```
int g(int const i);    // useless const
```

- With or without the const, calls to this function *pass by value*:

```
int n;
~
g(n);                // copies n to parameter i
```

- Since the call copies n to i, g can't alter n.
- Declaring i as const has no effect on n, only on g's use of i.

69

Top-Level const in Parameter Declarations

- Here's a common technique used to implement the standard memcpy function:

```
void *memcpy(void *d, void const *s, size_t n) {
    ~
    for (; n > 0; --n, ~) {    // OK
        ~
    }
    return d;
}
```

- It tracks the number of bytes remaining to copy by decrementing parameter n.

70

Top-Level const in Parameter Declarations

- Declaring parameter `n` as `const` precludes using `n` as a downward counter:

```
void *memcpy(void *d, void const *s, size_t const n) {
    ~~~
    for (; n > 0; --n, ~~~) { // error: n is const
        ~~~
    }
    return d;
}
```

- What do you gain by declaring `n` `const`?

71

Top-Level const in Parameter Declarations

- Many places where you can use `const` are not very useful:

```
void *const memcpy(
    void *const d, void const *const s, size_t const n
);
```



- There's only one really useful `const` in this declaration.

✓ *Use `const` proactively.*

- But don't clutter up your code with useless `const`.

✓ *Avoid using `const` at the top level of parameter declarations.*

72

Const and Class Design

- C++ programs can declare class objects as `const`.
- Such objects are useful only if the class is designed to support `const` objects.
- As an example, consider a rudimentary class that implements variable-length character strings:

```
string s = "hello";  
s += ", world";  
if (s.size() > 10)  
    ~~~  
    s.clear();
```

73

Const and Class Design

- The class definition might look in part like:

```
class string {  
public:  
    ~~~  
    size_t size();  
    void clear();  
    ~~~  
private:  
    char *text;  
    size_t stored_size;  
};
```

74

Const and Class Design

- When you pass a `string` as an argument to a function, passing by reference is typically cheaper than passing it by value:
 - Passing by reference merely passes the address.
 - Passing by value uses the `string`'s copy constructor to make a copy of the entire string.
- ✓ ***Declare a reference parameter as “reference to const” if the function shouldn’t alter the referenced object.***

75

Const and Class Design

```
R foo(string const &s) {
    ~~~
    for (size_t i = 0; i < s.size(); ++i) {
        ~~~
    }
}
```

- Given the `string` class as written, this function won't compile...

76

Const and Class Design

```
R foo(string const &s) {
    ~~~
    for (size_t i = 0; i < s.size(); ++i) { // error
        ~~~
    }
```

- foo's parameter s is declared as a "reference to const string".
 - This is a promise that foo won't change the value of s.
- The compiler complains because it doesn't see a promise...
- ...a promise that calling s.size() won't modify s.

77

Const and Class Design

- Remember, every non-static class member function has an implicitly-declared parameter named this:

```
class string {
public:
    size_t size(string *this); // implicitly declared
    ~~~
}
```

- If this were explicitly-declared, the size function could make the promise by adding const:

```
size_t size(string const *this);
```

- Here's how you actually do it...

78

Const Member Functions

- By declaring `size` as a *const member function*:

```
class string {
public:
    size_t size() const;    // const member function
    ~~~
};
```

- Syntactically, `const` modifies the function call `()`s to its left.

79

Const Member Functions

- A const member function won't compile if it tries to modify a member of `*this`, as in:

```
size_t string::size() const {
    return ++stored_size;    // error
}
```

- This is good.
- It keeps the promise.

80

Const Member Functions

- The `string::clear` function must remain a non-const member.
 - It can't do its job unless it alters data in a `string`.
- If you try to define it as a const member function, it won't compile:

```
void string::clear() const {
    delete [] text;
    text = nullptr;    // no: can't modify text
    stored_size = 0;  // no: can't modify stored_size
}
```

- This is as it should be.

81

Const Member Functions

- It follows from the rules for qualification conversions that:
 - A program can apply a const member function to a non-const object as well as a const object.
 - A program can apply a non-const member function **only** to a non-const object.
- Consequently:
 - Declaring a member function as const **doesn't force** others to declare objects as const.
 - However, failure to declare a member function as const **may prevent** others from declaring objects as const.

✓ ***Declare a member function as const whenever meaningful.***

82

The End

Thanks for Listening

83

Not Quite

There's More

84

A Curious Flaw

- Let's add a `[]` operator to our string class.
- It should be a *const member* so we can use it with a const string:

```
R foo(string const &s) {
    ~~~
    for (size_t i = 0; i < s.size(); ++i) {
        if (s[i] == something) {
            ~~~
        }
    }
}
```

- The compiler *interprets* `s[i]` as `s.operator[](i)`.
- The member function definition looks like...

85

A Curious Flaw

```
class string {
public:
    ~~~
    char &operator[](size_t i) const {
        return text[i];
    }
    ~~~
private:
    char *text;
    size_t stored_size;
};
```

- Unfortunately, this operator lets us modify a const string...

86

A Curious Flaw

```
R foo(string const &s) {
    ~~~
    for (size_t i = 0; i < s.size(); ++i) {
        s[i] = something;      // oops! compiles anyway
    }
}
```

- It compiles because operator[] returns a “reference to non-const”:

```
char &operator[](size_t i) const {
    return pointer[i];
}
```

- This is illogical...

87

A Curious Flaw

- ***An element selected from a const string should be const.***
- However, operator[] returns the selected element as non-const:

```
char &operator[](size_t i) const {
    return pointer[i];
}
```

- Why does it even compile?
- Shouldn't the return expression provoke a compile error for an invalid reference initialization?
- It doesn't...

88

Const is Shallow

- In a const member function, `this` is a “pointer to const”.
 - It’s actually a “const pointer to const”, but let’s not go there.
- What matters is that...
- A const member function adds const only to the **top type** of each non-static data member of `*this`.
- In other words,...
- ***Const in a const member function is shallow...***

89

What We Get

- Inside `operator[]`, `this` points to a class that appear to be:

```
class string {
    ~~~
private:
    char *const text;           // const pointer to...
                               // ...non-const char
    size_t const stored_size; // const size_t
};
```

- ***The pointer*** to (the initial character in the) the string ***is const***.
- ***The characters*** in the string ***are not***.

90

What We Want

- To prevent modifications, a const string should appear to be:

```
class string {
    ~~~
private:
    char const *const text;    // const pointer to...
                                // ...const char
    size_t const stored_size; // const size_t
};
```

- **The pointer** (the initial character in the) the string **is const**.
- **The characters** in the string **are also**.

91

Correcting the Flaw

- **An element selected from a const string should be const.**
- When operator[] is a const member function, it should return the selected element as “reference to const”:

```
char const &operator[](size_t i) const {
    return pointer[i];
}
```

- This is logically consistent, but not easy to use correctly...

92

Correcting the Flaw

- If string's operator[] is a const member function, it treats *all* strings as const.
- It prevents you from modifying characters in a non-const string:

```
string name {"ben"};
~~~~
name[0] = 'B';           // no! [] yields const char
```

- If you insist on doing the assignment, you must use a const_cast:

```
const_cast<char &>(name[0]) = 'B'; // usually hazardous
```

✓ *As with all casts, use const_cast sparingly.*

93

Overloading on Const

- The better solution is to *overload on const*:

```
class string {
public:
    char      &operator[](size_t i)      {
        return text[i];
    }
    char const &operator[](size_t i) const {
        return text[i];
    }
    ~~~~
};
```

- This yields logically consistent behavior...

94

Overloading on Const

- **An element selected:**
 - *from a non-const string should be non-const.*
 - *from a const string should be const.*
- Overloading member functions on const is a special case...
- You can overload functions with const on any pointer or reference parameter:

```
R f(T *p);           // one pair of...
R f(T const *p);    // ...overloaded functions

R g(T &r);           // another pair of...
R g(T const &r);    // ...overloaded functions
```

95

Overloading on Const

- A **function signature** is the information about a function that participates in overload resolution.
- In general, const appearing in a function's parameter declaration is part of the function's signature.
- For example, each of these functions has a distinct signature:

```
R f(T *p);
R f(T const *p);
```

- However, function signatures **don't include top-level cv-qualifiers** from parameter declarations...

96

When Apparent Overloading Really Isn't

- For example, these two functions have the *same signature*:

```
void f(T const x);    // signature = (T)
void f(T      x);    // same as this
```

- The same is true for:

```
void f(T *const p);  // signature = (T *)
void f(T *      p);  // same as this
```

- In each pair, the second declaration is *not an error*.
- It's just a *redeclaration* of the first function.

97

When Apparent Overloading Really Isn't

- Just to be clear, *this is overloading*:

```
void f(T const *p); // const is not top-level
void f(T      *p);
```

- *This is not*:

```
void f(T *const p); // const is top-level
void f(T *      p);
```

- *The placement of const in or out of a delarator makes all the difference.*

98

The End, for Real

Thanks for Listening

99