

736

How to: Understand and Use `const`

D

An important factor, both for and against C++, was the willingness of the C++ community to acknowledge C++'s many imperfections.

Bjarne Stroustrup

The Design and Evolution of C++, p. 178

The key word `const` in C++ is used in many contexts. Using classes that support object “const-ness” is straightforward, but developing classes that support constness requires some care in design and implementation and some knowledge of often overlooked C++ features that facilitate designing with `const`.

D.1 Why `const`?

Many C++ programmers rely on object const-ness to combine efficiency and safety. For example, passing parameters by value (the default mechanism in C++ and the only parameter passing mechanism in C and Java) creates a copy of the passed argument.¹ For example consider passing a copy of a string:

```
void verse(int bottleCount, string beverage)
{
    cout << bottleCount << " bottles of "
         << beverage << " on the wall" << endl;
    cout << bottleCount << " bottles of "
         << beverage << endl;
    // and so on
}
```

This function might be called several times in a loop as shown here.

```
// illustration of function call and argument/parameter copy
string bev;
cout << "enter a beverage ";
cin >> bev;
for(int k = 100; k > 0; k--)
{
    verse(k,bev);
}
```

¹In Java everything is a pointer (or a reference, depending on your viewpoint), so making copies isn't expensive. In C nearly everything is a pointer so making copies isn't expensive. In C++ value semantics mean “make a copy,” so copies are expensive.

In this example, the one-hundred function calls create one-hundred copies of the variable `bev`: one per call.² If the prototype of the function `verse` is changed to use a const-reference parameter as shown below, then no copies are made:

```
void verse(int bottleCount, const string & beverage)
{
    // function here
}
```

The pass-by-reference (indicated by the `&` in the parameter) means that no copy is made in passing an argument. The `const` modifier means that the parameter `beverage` cannot be modified within the body of `verse`. The reference is for efficiency and the `const` is for safety. Since many C++ programmers rely on passing const-reference parameters, class designers should know how to support this style of programming.

D.1.1 Literal Arguments

In the example above, it's possible to use pass-by-reference without the `const` modifier to achieve efficiency without regard to safety.³ However, in C++ if the parameter `beverage` is a reference parameter, but *not* a const reference parameter, then the function call below will not compile:

```
verse(99, "orange juice");
```

Here, the second parameter is the string literal `"orange juice"`. Literals must be passed by value or by const reference. In the latter case a temporary object will be constructed to hold the literal. The C++ standard requires a const reference parameter in this case; a reference parameter without the `const` modifier won't support literal (constant) arguments.

D.2 `const` Member Functions

How does the compiler determine what functions are safe to call when a parameter is defined as `const`? For example, in the `verse` function above, suppose the programmer adds the line below to convert the first character of the beverage to its upper-case equivalent (`toupper` is accessible via the header file `<cctype>`, see How to F).

```
void verse(int bottleCount, const string & beverage)
{
    beverage[0] = toupper(beverage[0]);
    // verse output here
}
```

²The compiler might be able to re-use the same copy, but not necessarily.

³This is typical, for example, in Pascal programs where arrays are passed as var parameters to avoid the overhead of copying the array (e.g., consider a binary-search function that searches in $O(\log n)$ time but takes $O(n)$ time to copy the array; not the paradigm of efficiency we'd like).



Compiling this code under Visual C++ 5.0 yields the error message

```
error C2106: left operand must be l-value
```

That's an "ell," where an l-value (for left-hand-side of an assignment value) is a value that can be assigned to. In the code above, it is not possible to assign to `beverage[0]` since the parameter `beverage` is `const`. How does the compiler determine this?

In the example above, the compiler knows the prototype/signature of all string member functions. These member functions include two indexing operators: one operator `[]` for `const` strings and one operator `[]` for non-`const` strings. Both prototypes are shown below:

```
char operator[] ( int k ) const; // const strings
char & operator[] ( int k );     // non-const strings
```

Note that the `const` indexing operator returns a `char` which will be a copy of the `k`-th character in the string. The non-`const` function returns a `char&`, a reference to a character in the string. Returning a reference means that the actual character in the string can be modified, (e.g., the code below turns "hello" into "jello" since string `fruit` is not `const`).

```
string fruit = "hello";
fruit[0] = 'j';
```

This code works because the value returned by the indexing operator is a reference (note the return type: `char &`) to a character in the string. Sometimes it helps to realize that the two statements below are equivalent:

```
fruit[0] = 'j';
fruit.operator[] (0) = 'j';
```

At first it may seem strange to see a function call used as an l-value, (i.e., the result returned by the call is assigned to). This is an essential part of how reference return-types are used in C++.

As shown in the example above, some member functions have the word `const` as part of their prototype/signature—the word `const` appears after the parameter list. To see another example, part of the header file for the class *Date* is reproduced below (see *date.h*, Program G.2) with some of the `const` methods shown.



```
class Date
{
public:
    Date(int m,int d,int y);
    // accessor functions

    int Month() const; // return month
    int Day()   const; // return day
    int Year()  const; // return year
    ...
};
```

As shown in the comment in the code above, the terminology often used for `const` member functions is **accessor**, indicating that (private) data is accessed only, not modified. In contrast, non-`const` member functions are often called **mutators**.

Program Tip D.1: A `const` member function is a member function that can be applied to a `const` object. The compiler ensures that `const` member functions do not modify private data. The compiler also ensures that only `const` member functions are called for `const` objects—`const` objects most often occur in programs as `const` reference parameters.

`const` member functions can also be applied to non-`const` objects. As we saw with `operator []` earlier, and as explained in the next section, it's possible to have two versions of a function: one for `const` and one for non-`const` objects.

The key here is that any member function that doesn't modify data should be declared `const` in both the `.h` file and in the `.cpp` file (prototypes of member functions must match declaration and definition, declaration is the `.h` file, definition is the `.cpp` file). Only `const` functions are called on `const` objects, non-`const` objects can call both `const` and non-`const` functions.

D.2.1 Overloading on `const`

Two functions are **overloaded** when they have the same name. Overloaded functions must have different parameter lists so that the compiler can determine which function to call. For example, typically a class has several constructors, all with the same name (the name of the class) but with different parameters. In the `Date` class there are three overloaded constructors.

It's possible to have two member functions with the same name, where one is `const` and the other non-`const`. We saw this in the example of the overloaded indexing operators for the `string` class, which are reproduced below.

```
char  operator[ ]( int k )  const; // const strings
char& operator[ ]( int k );           // non-const strings
```

At first glance these functions may appear to have the same parameter list and thus violate the rule requiring parameter lists of overloaded functions to be different. However, the `const` modifier for a member function really is part of the parameter list—it modifies the parameter `this` that is implicit in every member function and that refers to the object actually passed to the member function. In some sense you can think of all member functions having an implicit first parameter, a parameter of the type of the class to which the member function belongs. The `string` indexing functions would then be rewritten as follows as *non-member* functions (using *self* for *this*.)

```
char  operator[ ](const string& self, int k );
char& operator[ ](string& self, int k );
```

If we actually developed the code like this, instead of using `s.operator[](k)` we would write `operator[](s,k)` where `operator[]` is now a free function with two parameters instead of a member function of the `string` class with one parameter.


D.3 Mutable Data

Sometimes a class is logically const, but not physically const. This means that from a user viewpoint a function doesn't appear to change the class, but internally a change is needed to implement the function. One prototypical instance of this is a group of member functions that iterate over data in a collection. Not all of our iterator classes have been developed in a const-friendly way because we didn't want to discuss the issues raised here, but in more advanced applications you'll want const and non-const iterators. The class `CListIterator` is const friendly, but first we'll discuss another iterator class. Consider a `WordStreamIterator` counting the words in a file:

```
WordStreamIterator ws;
ws.Open("hamlet.txt");
int count = 0;
for(ws.Init(); ws.HasMore(); ws.Next())
{ count++;
}
```

Is the variable `ws` const? It doesn't seem to be since it's reading data from the `ifstream` object it encapsulates. So we don't expect the `WordStreamIterator` methods to be const although `HasMore` is clearly an accessor and should be const.

In a different context, what about iterating over a `CList` object? (See `clist.h`, Program G.12; recall that `CList` objects cannot change.)



```
void print(const StringList& list)
{
    StringList iter(list);
    for(iter.Init(); iter.HasMore(); iter.Next())
    { cout << iter.current() << endl;
    }
}
```

The problem here is that parameter `list` is const, as it should be, since the `print` function doesn't modify `list`. However, the iterating member functions will need to keep a pointer to the current node of the `CList` list and advance the pointer as needed. For example here's the code from `clist.cpp` for the `Next` iterator function.

```
template <class Type>
void CListIterator<Type>::Next() const
{
    if (HasMore())
    { myCurrent= myCurrent->next;
    }
}
```

Here the iterator function `Next` is labeled as a `const` function, meaning that it cannot modify any of the object's state/instance variables. As a result, if `myCurrent` is declared as a `Node *` pointer, the definition of `Next` above will not compile. If we make `Next` non-`const`, then we cannot support the concept of a `const`-iterator: an iterator over a `const` collection. We'd like to differentiate between `const` collections and non-`const` collections, and have iterators that support both types.

There are two solutions: one is to cast away the `const`ness in the function `Next`. The other is to declare the private variable `myCurrent` as **mutable**. The key word `mutable` is a relatively new addition to C++, but is supported by most recent compilers. A mutable data member can be modified by a `const` function. It's a good idea to keep mutable data to a minimum. However, in some situations where logical `const`ness (the iterator doesn't change the list) and physical `const`ness (the iterator updates a pointer) don't coincide, `mutable` is a nice feature. The declaration for the `CListIterator` class is reproduced below, again see Program G.12 for full details.

```
template <class Type>
class CListIterator
{
public:
    CListIterator(const CList<Type>& list);

    void Init()      const;
    bool HasMore()  const;
    void Next()      const;
    Type Current()  const;

private:
    typedef CList<Type>::TNode Node;
    Node * myFirst;           // front of list
    mutable Node * myCurrent; // current node
};
```

If your compiler doesn't support `mutable` you can cast away `const`ness using either the `const_cast` operator or an old-style cast. Both lead to incredibly ugly code. Since the iterator is `const`, the object `*this` must have its `const`-ness cast away as shown. Since `this` is a pointer to a `const` object (see Section D.4) we must cast so that `*this` isn't `const`; we want to change the object referenced by `this`.

```
template <class Type>
void CListIterator<Type>::Next() const
// post: iterator advanced to next item
{
    if (HasMore())
    { const_cast<CListIterator<Type> *>(this)->myCurrent
      = myCurrent->next;
    }
}
```

For compilers that don't support `const_cast` the following alternative will work.

```
// use an old style cast
(CListIterator<Type> *)(this)->myCurrent = myCurrent->next;
}
```

In both cases, before the cast the pointer `this` has type

```
const CListIterator<Type> *
```

The cast changes `this` to point to a non-const object, so that the object's state can be changed. This non-const reference can be modified since it is an l-value.

Program Tip D.2: It's *not* a good idea to cast away constness. C++ allows this, but you should try to minimize throwing away const since the use of const is for safety (a good thing). Using the keyword `mutable` marks logical constness in a way that is easy to see and easier syntactically than using casts.

D.4 Pointers and const

Many functions have pointer parameters modified by `const`. For example, the `string` constructor from a C-style string:

```
string::string(const char * p)
// post: initialized to C-style string p
```

Since the asterisk follows the type it makes a pointer to, `p` is a *pointer to a constant character*. This means that the object pointed to by `p` cannot be changed, it's constant. You cannot change an object through a pointer declared in this way. Pointers can be modified by `const` in other ways.

```
Date * tptr          = new Date(); // points to today
Date * const cptr    = new Date(); // constant pointer
const Date * coptr   = new Date(); // constant object
```

```
*tptr += 1;          // ok, tomorrow
*cptr += 1;          // ok, object isn't const
cptr = tptr;         // no, cptr is a constant pointer
*coptr += 1;         // no, *coptr is a const object
coptr = tptr;        // ok, pointer isn't const, object is
```

These examples illustrate the differences between a pointer to a constant object: `coptr` in the code above, and a const pointer: `cptr` in the same code.

744

Appendix D How to: Understand and Use `const`



D.5 Summary

Programming with `const` can be painful. It's easy to miss the appearance of `const` in compiler error messages — be sure that you look for it when you get a “member function XXX not implemented” error. You'll usually be told the function signature/prototype causing the error, look for `const` to see if you put a `const` in the header file, but forgot to add the `const` when defining the function.

Some people decide `const` is too painful and never program with `const` reference parameters. However, it's easy to use `const` when you don't have to write the classes, assuming that the class designer and implementer liked `const` too.

So use `const` for safety and learn to design and implement classes that support use of `const` by others.

Bibliography

- [AA85] Donald J. Albers and G.L. Alexanderson. *Mathematical People*. Birkhäuser, 1985.
- [ACM87] ACM. *Turing Award Lectures: The First Twenty Years 1966–1985*. ACM Press, 1987.
- [AS96] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. 2nd ed. MIT Press and McGraw-Hill, 1996.
- [Asp90] William Aspray. *Computing Before Computers*. Iowa State University Press, 1990.
- [Aus98] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1998.
- [Ben86] Jon Bentley. *Programming Pearls*. Addison-Wesley, 1986.
- [Ben88] Jon Bentley. *More Programming Pearls*. Addison-Wesley, 1988.
- [Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.
- [Boo91] Grady Booch. *Object Oriented Design with Applications*. Benjamin Cummings, 1991.
- [Boo94] Grady Booch. *Object Oriented Design and Analysis with Applications*. 2nd ed. Benjamin Cummings, 1994.
- [BRE71] I. Barrodale, F.D. Roberts, and B.L. Ehle. *Elementary Computer Applications in Science Engineering and Business*. John Wiley & Sons Inc., 1971.
- [Coo87] Doug Cooper. *Condensed Pascal*. W.W. Norton, 1987.
- [Dij82] Edsger W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982.
- [DR90] Nachum Dershowitz and Edward M. Reingold. Calendrical calculations. *Software-Practice and Experience*, 20(9):899–928, September 1990.
- [(ed91] Allen B. Tucker (ed.). *Computing Curricula 1991 Report of the ACM/IEEE-CS Joint Curriculum Task Force*. ACM Press, 1991.
- [EL94] Susan Epstein and Joanne Luciano, editors. *Grace Hopper Celebration of Women in Computing*. Computing Research Association, 1994. Hopper-Book@cra.org.

- [Emm93] Michele Emmer, editor. *The Visual Mind: Art and Mathematics*. MIT Press, 1993.
- [G95] Denise W. Güerer. Pioneering women in computer science. *Communications of the ACM*, 38(1):45–54, January 1995.
- [Gar95] Simson Garfinkel. *PGP: Pretty Good Privacy*. O’Reilly & Associates, 1995.
- [GHJ95] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Programming*. Addison-Wesley, 1995.
- [Gol93] Herman H. Goldstine. *The Computer from Pascal to von Neumann*. Princeton University Press, 1993.
- [Gri74] David Gries. On structured programming - a reply to smoliar. *Communications of the ACM*, 17(11):655–657, 1974.
- [GS93] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, 1993.
- [Har92] David Harel. *Algorithmics, The Spirit of Computing*. Addison-Wesley, second edition, 1992.
- [Hoa89] C.A.R. Hoare. *Essays in Computing Science*. Prentice-Hall, 1989. (editor) C.B. Jones.
- [Hod83] Andrew Hodges. *Alan Turing: The Enigma*. Simon & Schuster, 1983.
- [Hor92] John Horgan. Claude e. shannon. *IEEE Spectrum*, April 1992.
- [JW89] William Strunk Jr. and E.B. White. *The Elements of Style*. MacMillan Publishing Co., third edition, 1989.
- [Knu97] Donald E. Knuth. *The Art of Computer Programming*, volume 1 Fundamental Algorithms. Addison-Wesley, third edition, 1997.
- [Knu98a] Donald E. Knuth. *The Art of Computer Programming*, volume 2 Seminumerical Algorithms. Addison-Wesley, third edition, 1998.
- [Knu98b] Donald E. Knuth. *The Art of Computer Programming*, volume 3 Sorting and Searching. Addison-Wesley, third edition 1998.
- [KR78] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978.
- [KR96] Samuel N. Kamin and Edward M. Reingold. *Programming with class: A C++ Introduction to Computer Science*. McGraw-Hill, 1996.
- [Mac92] Norman Macrae. *John von Neumann*. Pantheon Books, 1992.

- [McC79] Pamela McCorduck. *Machines Who Think*. W.H. Freeman and Company, 1979.
- [McC93] Steve McConnell. *Code Complete*. Microsoft Press, 1993.
- [MGRS91] Albert R. Meyer, John V. Gutag, Ronald L. Rivest, and Peter Szolovits, editors. *Research Directions in Computer Science: An MIT Perspective*. MIT Press, 1991.
- [Neu95] Peter G. Neumann. *Computer Related Risks*. Addison Wesley, 1995.
- [Pat96] Richard E. Pattis. *Get A-Life: Advice for the Beginning Object-Oriented Programmer*. Turing TarPit Press, 1999.
- [Per87] Alan Perlis. The synthesis of algorithmic systems. In *ACM Turing Award Lectures: The First Twenty Years*. ACM Press, 1987.
- [PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.
- [RDC93] Edward M. Reingold, Nachum Dershowitz, and Stewart M. Clamen. Calendrical calculations, ii: Three historical calendars. *Software-Practice and Experience*, 23(4):383–404, April 1993.
- [Rie96] Arthur Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.
- [Rob95] Eric S. Roberts. Loop exits and structured programming: Reopening the debate. In *Papers of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education*, pages 268–272. ACM Press, March 1995. SIGCSE Bulletin V. 27 N 1.
- [Rob95] Eric S. Roberts. *The Art and Science of C*. Addison-Wesley, 1995.
- [Sla87] Robert Slater. *Portraits in Silicon*. MIT Press, 1987.
- [Str87] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1987.
- [Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.
- [Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.
- [Mey92] Scott Meyers. *Effective C++*. Addison Wesley, 1992.
- [Mey96] Scott Meyers. *More Effective C++*. Addison-Wesley, 1996.
- [Wei94] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Benjamin Cummings, 1994.
- [Wil56] M.V. Wilkes. *Automatic Digital Computers*. John Wiley & Sons, Inc., 1956.
- [Wil87] Maurice V. Wilkes. Computers then and now. In *ACM Turing Award Lectures: The First Twenty Years*, pages 197–205. ACM Press, 1987.

824

Appendix D How to: Understand and Use `const`

- [Wil95] Maurice V. Wilkes. *Computing Perspectives*. Morgan Kaufmann, 1995.
- [Wir87] Niklaus Wirth. From programming language design to compiler construction. In *ACM Turing Award Lectures: The First Twenty Years*. ACM Press, 1987.