**Constants**

Preferred to eliminate the use of preprocessor #defines for value substitution. Also used for pointers, function arguments, return types, class objects and member functions.

# Value substitution

C++ uses const for value substitution into the domain of the compiler.

const int bufsize = 100;

You can use bufsize any place where the compiler must know the value at compile time.

The compiler can use bufsize to perform constant folding, which means the compiler will reduce a complicated constant expression to a simple one by performing the necessary calculations at compile time. This is especially important in array definitions:

char buf[bufsize];

You can use const for all the built-in types (char, int, float, and double) and their variants (as well as class objects, as you’ll see later in this chapter). Because of subtle bugs that the preprocessor might introduce, you should always use const instead of #define value substitution.

## const in header files

You can place the definition for a const in a header file and distribute it to translation units by including the header file.

A const in C++ defaults to internal linkage; that is, it is visible only within the file where it is defined and cannot be seen at link time by other translation units. You must always assign a value to a const when you define it, except when you make an explicit declaration using extern.

extern const int bufsize;

Normally, the C++ compiler avoids creating storage for a const, but instead holds the definition in its symbol table.

When you use extern with const, however, you force storage to be allocated. Storage must be allocated because extern says “use external linkage,” which means that several translation units must be able to refer to the item, which requires it to have storage.

The goal of never allocating storage for a const also fails with complicated structures. Whenever the compiler must allocate storage, constant folding is prevented.

Because the compiler cannot always avoid allocating storage for a const, const definitions must default to internal linkage, that is, linkage only within that particular translation unit. Otherwise, linker errors would occur with complicated consts because they cause storage to be allocated in multiple cpp files. The linker would then see the same definition in multiple object files, and complain.

## Safety consts

If you initialize a variable with a value that is produced at runtime and you know it will not change for the lifetime of that variable, make it a const so the compiler will give you an error message if you accidentally try to change it.

#include <iostream>

using namespace std;

const int i = 100; // Typical constant

const int j = i + 10; // Value from const expr

long address = (long)&j; // Forces storage

char buf[j + 10]; // Still a const expression

int main() {

cout << "type a character & CR:";

const char c = cin.get(); // Can't change

const char c2 = c + 'a';

//c2 = 'B'; // CE: assignment of read-only variable 'c2'

cout << c2;

return 0;

}

Input:

H

Output:

type a character &amp; CR:©

## Aggregates

It’s possible to use const for aggregates, but you’re virtually assured that the compiler will not be sophisticated enough to keep an aggregate in its symbol table, so storage will be allocated. In these situations, const means “a piece of storage that cannot be changed.” However, the value cannot be used at compile time because the compiler is not required to know the contents of the storage at compile time.

#include <iostream>

using namespace std;

// Constants and aggregates

const int i[] = { 1, 2, 3, 4 };

//float f[i[3]]; // CE: array bound is not an integer constant before ']' token

struct S { int i, j; };

const S s[] = { { 1, 2 }, { 3, 4 } };

//double d[s[1].j]; // CE: array bound is not an integer constant before ']' token

int main() {}

## Differences with C

const in C, means an ordinary variable that cannot be changed.

1. In C, a const always occupies storage and its name is global.
2. The C compiler cannot treat a const as a compile-time constant.

const int bufsize = 100;

char buf[bufsize]; // Illegal

1. In C, statement

const int bufsize;

is valid, but not in C++, and the C compiler accepts it as a declaration indicating there is storage allocated elsewhere.

Because C defaults to external linkage for const and C++ defaults to internal linkage for const.

1. In C++, you must explicitly change the linkage to external using extern:

extern const int bufsize; // Declaration only

This line also works in C.

1. In C++, a const doesn’t necessarily create storage. In C a const always creates storage.

The C approach to const is not very useful, and if you want to use a named value inside a constant expression (one that must be evaluated at compile time), C almost forces you to use #define in the preprocessor.

# Pointers

Pointers can be made const. The compiler will still endeavor to prevent storage allocation and do constant folding when dealing with const pointers.

The compiler will tell you if you attempt to change a const pointer.

When using const with pointers, you have two options:

1. const can be applied to what the pointer is pointing to
2. const can be applied to the address stored in the pointer itself.

## Pointer to const

The const specifier binds to the thing it is “closest to.” So if you want to prevent any changes to the element you are pointing to, you write a definition like this:

const int\* u;

It reads “u is a pointer, which points to a const int.”

Here, no initialization is required because you’re saying that u can point to anything (that is, it is not const), but the thing it points to cannot be changed.

**Second form:**

int const\* v; // effect is the same as the previous definition

It reads “v is an ordinary pointer to an int that happens to be const.” // Correct

~~“v is a const pointer to an int.”~~ // Wrong

To prevent any confusion stick to the first form.

## const pointer

To make the pointer itself a const, you must place the const specifier to the right of the \*, like this:

int d = 1;

int\* const w = &d;

It reads: “w is a pointer, which is const, that points to an int.”

The pointer itself is now the const, the compiler requires that it be given an initial value that will be unchanged for the life of that pointer.

It’s OK, however, to change what that value points to by saying

\*w = 2;

## const pointer to a const object

use either of two legal forms:

int d = 1;

const int\* const x = &d; // (1)

int const\* const x2 = &d; // (2)

Now neither the pointer nor the object can be changed.

## Formatting

The ‘\*’ in fact binds to the identifier, not the type. It can be placed anywhere between the type name and the identifier. So you could do this:

int \*u = &i, v = 0;

which creates an int\* u and a non-pointer int v.

## Assignment and type checking

You can assign the address of a non-const object to a const pointer because you’re simply promising not to change something that is OK to change.



You can’t assign the address of a const object to a non-const pointer because then you’re saying you might change the object via the pointer.



Of course, you can always use a cast to force such an assignment, but this is bad programming practice because you are then breaking the constness of the object, along with any safety promised by the const.

#include <iostream>

using namespace std;

int d = 1;

const int e = 2;

int\* u = &d; // OK

//int\* v = &e; // CE error: invalid conversion from 'const int\*' to 'int\*'

int\* w = (int\*)&e; // OK but Bad Programming

int main() { return 0; }

## Character array literals

The place where strict constness is not enforced is with character array literals.

char\* cp = "howdy";

The compiler will accept it without complaint.

This is technically an error because a character array literal (“howdy” in this case) is created by the compiler as a constant character array, and the result of the quoted character array is its starting address in memory.

Modifying any of the characters in the array is a runtime error, although not all compilers enforce this correctly.

However, if you try to change the values in a character array literal, the behavior is undefined, although it will probably work on many machines.

If you want to be able to modify the string, put it in an array:

char cp[] = "howdy";

#include <iostream>

using namespace std;

int main(void) {

char \* cp = "Ratnesh"; // CW warning: ISO C++ forbids converting a string constant to 'char\*'

char ca[] = "Ratnesh";

cout << cp << endl;

cout << ca << endl;

cout << \*(cp + 3) << endl;

cout << ca[3] << endl;

//\*(cp + 3) = 'N'; // Runtime Error : SIGSEGV

ca[3] = 'N';

cout << \*(cp + 3) << endl;

cout << ca[3] << endl;

return 0;

}

Output:

Ratnesh

Ratnesh

n

n

n

N

# Function arguments & return values

If you are passing objects by value, specifying const has no meaning to the client (it means that the passed argument cannot be modified inside the function).

If you are returning an object of a user-defined type by value as a const, it means the returned value cannot be modified.

If you are passing and returning addresses, const is a promise that the destination of the address will not be changed.

## Passing by const value

You can specify that function arguments are const when passing them by value, such as

void fun(const int i) {

i++; // CE error: increment of read-only parameter 'i'

}

Because the argument is passed by value, you immediately make a copy of the original variable, so the promise to the client is implicitly kept.

You could do this with a pointer, but a nicer syntax is achieved with the reference.

To avoid confusion to the caller, you can make the argument a const inside the function

void fun(int ic) {

const int& i = ic;

i++; // CE error: increment of read-only parameter 'i'

}

## Returning by const value

If you say that a function’s return value is const:

const int g();

This can make the specification of const seem meaningless, because you’re returning it by value, it’s copied so the original value could never be modified via the return value.

The reason const has no meaning when you’re returning a built-in type by value is that the compiler already prevents it from being an lvalue (because it’s always a value, and not a variable).

Returning by value as a const becomes important when you’re dealing with user-defined types.

If a function returns a class object by value as a const, the return value of that function cannot be an lvalue (that is, it cannot be assigned to or otherwise modified).

#include <iostream>

using namespace std;

class X {

int i;

public:

X(int ii = 0);

void modify();

};

X::X(int ii) { i = ii; }

void X::modify() { i++; }

X f5() { return X(); }

const X f6() { return X(); }

void f7(X& x) { x.modify(); } // Pass by non-const reference

int main() {

f5() = X(1);

f5().modify();

//Causes compile-time errors:

//f7(f5()); // CE error: invalid initialization of non-const reference of type 'X&' from an rvalue of type 'X'

//f6() = X(1); // CE error: passing 'const X' as 'this' argument discards qualifiers

//f6().modify(); // CE error: passing 'const X' as 'this' argument discards qualifiers

//f7(f6()); // CE error: invalid initialization of non-const reference of type 'X&' from an rvalue of type 'const X'

return 0;

}

The reason line

f7(f6());

won’t compile in C++ is because of the creation of a temporary.

### Temporaries

Sometimes, during the evaluation of an expression, the compiler must create temporary objects. These are objects like any other: they require storage and they must be constructed and destroyed.

There is one thing about temporaries: they’re automatically const.

## Passing and returning addresses

It’s possible for the client programmer to take a pointer or a reference and modify the original value. If you make the pointer or reference a const, you prevent this from happening. In fact, whenever you’re passing an address into a function, you should make it a const if at all possible.

#include <iostream>

using namespace std;

void t(int\*) {}

void u(const int\* cip) {

//\*cip = 2; // Illegal -- modifies value CE error: assignment of read-only location '\* cip'

int i = \*cip; // OK -- copies value

//int\* ip2 = cip; // Illegal: non-const CE error: invalid conversion from 'const int\*' to 'int\*'

}

const char\* v() {

// Returns address of static character array:

return "result of function v()";

}

const int\* const w() {

static int i;

return &i;

}

int main() {

int x = 0;

int\* ip = &x;

const int\* cip = &x;

t(ip); // OK

//t(cip); // Not OK CE error: invalid conversion from 'const int\*' to 'int\*'

u(ip); // OK

u(cip); // Also OK

//char\* cp = v(); // Not OK CE error: invalid conversion from 'const char\*' to 'char\*'

const char\* ccp = v(); // OK

//int\* ip2 = w(); // Not OK CE error: invalid conversion from 'const int\*' to 'int\*'

const int\* const ccip = w(); // OK

const int\* cip2 = w(); // OK

//\*w() = 1; // Not OK CE error: assignment of read-only location '\* w()'

return 0;

}

### Standard argument passing

Because of the syntax of references (it looks like pass-by-value to the caller) it’s possible to pass a temporary object to a function that takes a const reference, whereas you can never pass a temporary object to a function that takes a pointer with a pointer, the address must be explicitly taken. So passing by reference produces a new situation that never occurs in C: a temporary, which is always const, can have its address passed to a function. This is why, to allow temporaries to be passed to functions by reference, the argument must be a const reference.

#include <iostream>

using namespace std;

class X {};

X f() { return X(); } // Return by value

void g1(X&) {} // Pass by non-const reference

void g2(const X&) {} // Pass by const reference

int main() {

//g1(f()); // NOK : const temporary created by f():

// CE error: invalid initialization of non-const reference of type 'X&' from an rvalue of type 'X'

g2(f()); // OK: g2 takes a const reference

return 0;

}

f() returns an object of class X by value. That means when you immediately take the return value of f() and pass it to another function as in the calls to g1() and g2(), a temporary is created and that temporary is const. Thus, the call in g1() is an error because g1() doesn’t take a const reference, but the call to g2() is OK.

# Classes

The meaning of const is different inside classes.

You must understand the options in order to create const data members of a class.

You can also make an entire object const. But preserving the constness of an object is more complex. The compiler can ensure the constness of a built-in type but it cannot monitor the intricacies of a class. To guarantee the constness of a class object, the const member function is introduced: only a const member function may be called for a const object.

## const in classes

for the array size you may use const instead of #define

Inside a class, const partially reverts to its meaning in C. It allocates storage within each object and represents a value that is initialized once and then cannot change. The use of const inside a class means “This is constant for the lifetime of the object.” However, each different object may contain a different value for that constant. This initialization must occur in a special place in the constructor.

### The constructor initializer list

The special initialization point is called the constructor initializer list, and it was originally developed for use in inheritance.

class Fred {

const int size;

public:

Fred(int sz);

void print();

};

Fred::Fred(int sz) : size(sz) {}

### “Constructors” for built-in types

There are times when it is helpful to make built-in types look like user-defined types. In the constructor initializer list, you can treat a built-in type as if it has a constructor, like this:

class B {

int i;

public:

B(int ii);

void print();

};

B::B(int ii) : i(ii) {}

This is especially critical when initializing const data members because they must be initialized before the function body is entered.

## Compile-time constants in classes

How do you make a compile-time constant inside a class?

**using static**

The static keyword, in this situation, means “there’s only one instance, regardless of how many objects of the class are created". Thus, a static const of a built-in type can be treated as a compile-time constant.

You need to provide the initializer at the point of definition of the static const.

class StringStack {

**static const int size = 100;**

const string\* stack[size];

};

### The “enum hack” in old code

In older versions of C++, staticconst was not supported inside classes.

Typical solution (“enum hack”) was to use an untagged enum with no instances. An enumeration must have all its values established at compile time, it’s local to the class, and its values are available for constant expressions.

class Bunch {

**enum { size = 1000 };**

int i[size];

};

The use of enum here is guaranteed to occupy no storage in the object, and the enumerators are all evaluated at compile time.

There is no overwhelming reason that you must choose static const over the enum hack.

## const objects & member functions

For the compiler to enforce constness of a const object, it must ensure that no data members of the object are changed during the object’s lifetime.

If you declare a member function const, you tell the compiler the function can be called for a const object. A member function that is not specifically declared const is treated as one that will modify data members in an object, and the compiler will not allow you to call it for a const object.

The compiler forces you to reiterate the const specification when defining the function. Then it enforces constness during the function definition by issuing an error message if you try to change any members of the object or call a non-const member function.

**Specify a const function**

To declare const member functions, place the const specifier after the argument list. The const keyword must be repeated in the definition or the compiler sees it as a different function.

Any function that doesn’t modify member data should be declared as const, so it can be used with const objects.

Neither constructors nor destructors can be const member functions because they virtually always perform some modification on the object during initialization and cleanup.

A const member function is safe to call with both const and non-const objects.

#include <iostream>

using namespace std;

class A {

int m;

public:

A(int n) : m(n){ }

~A() { m = 0;}

void fun1(int n);

void fun2(int m) const;

void display() { cout << "m: " << m << endl; }

};

void A::fun1(int n) {

m = n;

}

void A::fun2(int n) const {

//fun1(); // CE error: no matching function for call to 'A::fun1() const'

//m = n; // CE error: assignment of member 'A::m' in read-only object

}

int main() {

A obj(5);

const A cobj(7);

obj.fun1(3);

obj.fun2(9);

obj.display();

//cobj.fun1(3); // CE error: passing 'const A' as 'this' argument discards qualifiers

cobj.fun2(9);

//cobj.display(); // CE error: passing 'const A' as 'this' argument discards qualifiers

return 0;

}

Output:

m: 3

### mutable: bitwise vs. logical const

What if you want to create a const member function, but you’d still like to change some of the data in the object?

This is sometimes referred to as the difference between bitwise constand logical const (also sometimes called memberwise const).

Bitwise const means that every bit in the object is permanent, so a bit image of the object will never change.

Logical const means that, although the entire object is conceptually constant, there may be changes on a member-bymember basis.

However, if the compiler is told that an object is const, it will jealously guard that object to ensure bitwise constness.

Two ways to effect logical constness, change a data member from within a const member function.

1. casting away constness
2. mutable
3. **casting away constness**

You take this and cast it to a pointer to an object of the current type. It would seem that this is already such a pointer. However, inside a const member function it’s actually a const pointer, so by casting it to an ordinary pointer, you remove the constness for that operation.

class Y {

int i;

public:

Y();

void f() const;

};

Y::Y() { i = 0; }

void Y::f() const {

((Y\*)this)->i++; // OK: cast away const-ness

(const\_cast<Y\*>(this))->i++; // // Better: use C++ explicit cast syntax:

}

Drawback: this lack of constness is hidden away in a member function definition, and you have no clue from the class interface that the data of the object is actually being modified unless you have access to the source code.

1. **mutable**

Use mutable keyword in the class declaration to specify that a particular data member may be changed inside a const object.

class Y {

mutable int i;

public:

Y();

void f() const;

};

Y::Y() { i = 0; }

void Y::f() const {

i = i + 1;

}

This way, the user of the class can see from the declaration which members are likely to be modified in a const member function.

### ROMability

If an object is defined as const, it is a candidate to be placed in readonly memory (ROM), which is often an important consideration in embedded systems programming.

Requirements for ROMability are much stricter.

1. Of course, the object must be bitwise-const, rather than logical-const.
2. The class or struct must have no user-defined constructors or destructor.
3. There can be no base classes or member objects with user-defined constructors or destructors.

The effect of a write operation on any part of a const object of a ROMable type is undefined. Although a suitably formed object may be placed in ROM, no objects are ever required to be placed in ROM.

# volatile

The syntax of volatile is identical to that for const, but volatile means “This data may change outside the knowledge of the compiler.”

Somehow, the environment is changing the data (possibly through multitasking, multithreading or interrupts), and volatile tells the compiler not to make any assumptions about that data, especially during optimization.

If the compiler says, “I read this data into a register earlier, and I haven’t touched that register,” normally it wouldn’t need to read the data again. But if the data is volatile, the compiler cannot make such an assumption because the data may have been changed by another process, and it must reread that data.

You create volatile objects using the same syntax that you use to create const objects. You can also create const volatile objects, which can’t be changed by the client programmer but instead change through some outside agency.

#include <iostream>

using namespace std;

class Comm {

const volatile unsigned char byte;

volatile unsigned char flag;

enum { bufsize = 100 };

unsigned char buf[bufsize];

int index;

public:

Comm();

void isr() volatile;

char read(int index) const;

};

Comm::Comm() : index(0), byte(0), flag(0) {}

// Only a demo; won't actually work as an interrupt service routine:

void Comm::isr() volatile {

flag = 0;

buf[index++] = byte;

if(index >= bufsize) index = 0; // // Wrap to beginning of buffer:

}

char Comm::read(int index) const {

if(index < 0 || index >= bufsize)

return 0;

return buf[index];

}

int main() {

volatile Comm Port;

Port.isr(); // OK

// CE error: passing 'volatile Comm' as 'this' argument discards qualifiers

//Port.read(0); // Error, read() not volatile

return 0;

}

You can use volatile for data members, member functions, and objects themselves. You can only call volatile member functions for volatile objects.

const and volatile two are referred to in combination as the c-v qualifier.

# END