Unit 8 constants

Section 1 Value substitution

1. List the steps to create a preprocessor substitution.

#define BUFSIZE 100

**BUFSIZE** is a name that only exists during preprocessing, therefore it doesn’t occupy storage and can be placed in a header file to provide a single value for all translation units that use it. It’s very important for code maintenance to use value substitution instead of so-called “magic numbers.”

1. List the steps to use a const in a header file.

you must be able to place **const** definitions inside header files as you can with **#define.**

 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:**

**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.

1. Describe why const does not work with aggregates.

 the value cannot be used at compile time because the compiler is not required to know the contents of the storage at compile time.

array definition, the compiler must be able to generate code that moves the stack pointer to accommodate the array.

n both of the illegal definitions above, the compiler complains because it cannot find a constant expression in the array definition.

1. List at least two differences between C and C++ in the handling of const.

In C

somehow it came to mean for them “an ordinary variable that cannot be changed.”

a **const** always occupies storage and its name is global.

The C compiler cannot treat a **const** as a compile-time constant.

 In C a **const** always creates storage.

In C++

You can optionally say

const int bufsize;

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

C++ defaults to internal linkage for **consts** so if you want to accomplish the same thing in C++, you must explicitly change the linkage to external using **extern:**

extern const int bufsize;// declaration only

In C++, a **const** doesn’t necessarily create storage.

Whether or not storage is reserved for a **const** in C++ depends on how it is used.

if a **const** is used simply to replace a name with a value (just as you would use a **#define),** then storage doesn’t have to be created for the **const.**

In C++, a **const** that is outside all functions has file scope (i.e., it is invisible outside the file).

 it defaults to internal linkage.

a **const** in C++ defaults to internal linkage, you can’t just define a **const** in one file and reference it as an **extern** in another file.

in C++ means that the definition exists elsewhere (again, this is not necessarily true in C).

You can now see why C++ requires a **const** definition to have an initializer: the initializer distinguishes a declaration from a definition (in C it’s always a definition, so no initializer is necessary). With an **extern const** declaration, the compiler cannot do constant folding because it doesn’t know the value.

**Section 2 Pointers**

1. List two options to making a pointer const.

**const** can be applied to what the pointer is pointing to

**const** can be applied to the address stored in the pointer itself.

1. Explain the use of pointer to const.

is to read it starting at the identifier and work your way ou

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;

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

int const\* v;

“**v** is a **const** pointer to an **int.”**  “**v** is an ordinary pointer to an **int** that happens to be **const.”**

That is, the **const** has bound itself to the **int** again,

 you should probably stick to the first form.

1. Explain the use of const pointer.

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

int\* const w = &d;

“**w** is a pointer, which is **const,** that points to an **int.”**

\*w = 2;

You can also make a **const** pointer to a **const** object using 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.

 putting one pointer definition on a line, and initializing each pointer at the point of definition whenever possible.

int\* u = &i;

So you could do this:

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

which creates an **int\* u,** as before, and a non-pointer **int v**. Because readers often find this confusing, it is best to follow the form shown in this book

1. List at least two places where type checking is enforced when using const.

you can’t assign the address of a **const** object to a non**-const** pointer

1. When is type checking not enforced when using const?

 with character array literals. You can say

char\* cp = ″howdy″;

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

character array literals are actually constant character arrays.

 lets you get away with treating them as non-**const**because there’s so much existing C code that relies on this.

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.**

the behavior is undefined, although it will probably work on many machines.

Section 3 Function arguments and Return values

1. Explain the use of passing by const value.

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.

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

void f1(const int i) {

    i++; // Illegal -- compile-time error

}

a reference is like a constant pointer that is automatically dereferenced, so it has the effect of being an alias to an object.

Inside the function, the **const** takes on meaning: the argument cannot be changed. So it’s really a tool for the creator of the function, and not the caller

void f2(int ic) {

    const int& i = ic;

    i++; // Illegal -- compile-time error

}

1. Explain the use of returning by const value.

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

const int g ();

you are promising that the original variable (inside the function frame) will not be modified.

For built-in types, it doesn’t matter whether you return by value as a **const,**so you should avoid confusing the client programmer and leave off the **const** when returning a built-in type by value.

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).

Only the non**-const** return value can be used as an lvalue. Thus, it’s important to use **const** when returning an object by value if you want to prevent its use as an lvalue.

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). Only when you’re returning objects of user-defined types by value does it become an issue

1. Explain the use of temporary objects.

They require storage and they must be constructed and destroyed.

The difference is that you never see them – the compiler is responsible for deciding that they’re needed and the details of their existence.

In the above example, **f5( )** returns a non-**const X** object. But in the expression:

f7 (f5() ) ;

the compiler must manufacture a temporary object to hold the return value of **f5( )** so it can be passed to **f7( )**.

**However, f7( ) takes its argument** *by reference,* which means in this example takes the address of the temporary **X**

it must create a temporary to hold that return value. So in both expressions the temporary object is being modified, and as soon as the expression is over the temporary is cleaned up.

As a result, the modifications are lost so this code is probably a bug

Expressions like these are simple enough for you to detect the problem, but when things get more complex it’s possible for a bug to slip through these cracks.

he way the **const**ness of class objects is preserved is shown later in the chapter.

They’re automatically **const.**

1. Explain the use of passing addresses.

If you pass or return an address (either a pointer or a reference), it’s possible for the client programmer to take it 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

a **const** depends on what you want to allow your client programmer to do with it

This is important, review this content in the cpp ConstPointers Unit 8

1. Explain the use of standard argument passing.

, passing an address is virtually always more efficient than passing an entire class object, and if you pass by **const** reference it means your function will not change the destination of that address,

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.

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.

Section 4 Classes

1. List at least two uses of const in a class.

One of the places you’d like to use a **const** for constant expressions is inside classes.

You might assume that the logical choice is to place a **const** inside the class. This doesn’t produce the desired result. Inside a class, **const** partially reverts to its meaning in C.

 The use of **const** inside a class means “This is constant for the lifetime of the object.”

when you create an ordinary (non-**static**) **const** inside a class, you cannot give it an initial value.

 This initialization must occur in the constructor, of course, but in a special place in the constructor.

1. Define constructor initializer list.

Special initialization point is called the *constructor initializer list,* occurs only in the definition of the constructor

“constructor calls” that occur after the function argument list and a colon, but before the opening brace of the constructor body

 This is to remind you that the initialization in the list occurs before any of the main constructor code is executed. This is the place to put all **const** initializations.

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

built-in types (which simply means assignment)

**float pi(3.14159)**

1. List at least two examples of a “constructor” for a built-in type.

It’s often useful to encapsulate a built-in type inside a class to guarantee initialization with the constructor.

Integer(int ii =0) , integer(int ii)

1. Explain the use of the keyword static const.

 The **static** keyword, in this situation, means “there’s only one instance, regardless of how many objects of the class are created,”

 a member of a class which is constant, and which cannot change from one object of the class to another.

Thus, a **static const** of a built-in type can be treated as a compile-time constant.

you must provide the initializer at the point of definition of the **static const.**

Since **size** is used to determine the size of the array **stack,** it is indeed a compile-time constant, but one that is hidden inside the class.

See exercise StringStack.cpp to see syntasis

This meant that **const** was useless for constant expressions inside classes.

 untagged **enum** with no instances

static const int size = 100;

would be instead:

enum { size = 100 };

1. Explain the use of the term const object.

A **const** object is defined the same for a user-defined type as a built-in type.

const int i = 1;

const blob b(2);

Here, **b** is a **const** object of type **blob.**

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**

See example Quotes.cpp

up. The **quote( )** member function also cannot be **const** because it modifies the data member **lastquote** (see the **return** statement). However, **lastQuote( )** makes no modifications, and so it can be **const** and can be safely called for the **const** object **cq.**

**Section 5 – Volatile**

1. **Explain the use of the keyword volatile**

**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)

**volatile** tells the compiler not to make any assumptions about that data, especially during optimization.

 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 rather than optimizing the code to remove what would normally be a redundant read.

1. **Explain the use of the keyword const volatile**

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.

As with **const,** you can use **volatile** for data members, member functions, and objects themselves. You can only call **volatile** member functions for **volatile** objects.

The syntax of **volatile** is identical to **const,** so discussions of the two are often treated together. The two are referred to in combination as the *c-v qualifier.*