**Initialization & Cleanup**

When it comes to safety, however, there’s a lot more the compiler can do for us than C provides.

Two of these safety issues are initialization and cleanup. A large segment of C bugs occur when the programmer forgets to initialize or clean up a variable.

# Guaranteed initialization with the constructor

In C++, the class designer can guarantee initialization of every object by providing a special function called the constructor. If a class has a constructor, the compiler automatically calls that constructor at the point an object is created, before client programmers can get their hands on the object. The constructor call isn’t even an option for the client programmer; it is performed by the compiler at the point the object is defined. The name of the constructor is the same as the name of the class.

# Guaranteed cleanup with the destructor

In C++, cleanup is as important as initialization and is therefore guaranteed with the destructor. The syntax for the destructor is similar to that for the constructor: the class name is used for the name of the function. However, the destructor is distinguished from the constructor by a leading tilde (~). In addition, the destructor never has any arguments because destruction never needs any options. The destructor is called automatically by the compiler when the object goes out of scope.

**Both the constructor and destructor are very unusual types of functions: they have no return value.**

# Elimination of the definition block

In general, C++ will not allow you to create an object before you have the initialization information for the constructor. Because of this, the language wouldn’t be feasible if you had to define variables at the beginning of a scope. In fact, the style of the language seems to encourage the definition of an object as close to its point of use as possible. In C++, any rule that applies to an “object” automatically refers to an object of a built-in type as well.

This means that any class object or variable of a built-in type can also be defined at any point in a scope. It also means that you can wait until you have the information for a variable before defining it, so you can always define and initialize at the same time.

In general, you should define variables as close to their point of use as possible, and always initialize them when they are defined. (This is a stylistic suggestion for built-in types, where initialization is optional.) This is a safety issue. By reducing the duration of the variable’s availability within the scope, you are reducing the chance it will be misused in some other part of the scope. In addition, readability is improved because the reader doesn’t have to jump back and forth to the beginning of the scope to know the type of a variable.

## for loops

In C++, you will often see a for loop counter defined right inside the for expression:

for(int j = 0; j < 100; j++) {

cout << "j = " << j << endl;

}

However, some confusion may result if you expect the lifetime of the variable j to extend beyond the scope of the for loop – they do not.

## Storage allocation

Although the storage is allocated at the beginning of the block, the constructor call doesn’t happen until the sequence point where the object is defined because the identifier isn’t available until then.

The compiler even checks to make sure that you don’t put the object definition (and thus the constructor call) where the sequence point only conditionally passes through it, such as in a switch statement or somewhere a goto can jump past it.

Uncommenting the statements in the following code will generate a warning or an error:

// Can't jump past constructors

class X {

public:

X();

};

X::X() {}

void f(int i) {

if(i < 10) {

//! goto jump1; // Error: goto bypasses init

}

X x1; // Constructor called here

jump1:

switch(i) {

case 1 :

X x2; // Constructor called here

break;

//! case 2 : // Error: case bypasses init

X x3; // Constructor called here

break;

}

}

int main() {

f(9);

f(11);

return 0;

}

# Aggregate initialization

An aggregate is just what it sounds like: a bunch of things clumped together. This definition includes aggregates of mixed types, like structs and classes. An array is an aggregate of a single type.

Initializing aggregates can be error-prone and tedious. C++ aggregate initialization makes it much safer. When you create an object that’s an aggregate, all you must do is make an assignment, and the initialization will be taken care of by the compiler.

This assignment comes in several flavors, depending on the type of aggregate you’re dealing with, but in all cases the elements in the assignment must be surrounded by curly braces. For an array of built-in types this is quite simple:

int a[5] = { 1, 2, 3, 4, 5 };

If you try to give more initializers than there are array elements, the compiler gives an error message. But what happens if you give fewer initializers? For example:

int b[6] = {0};

Here, the compiler will use the first initializer for the first array element, and then use zero for all the elements without initializers. Notice this initialization behavior doesn’t occur if you define an array without a list of initializers. So the expression above is a succinct way to initialize an array to zero, without using a for loop, and without any possibility of an off-by-one error (Depending on the compiler, it may also be more efficient than the for loop.)

A second shorthand for arrays is automatic counting, in which you let the compiler determine the size of the array based on the number of initializers:

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

Now if you decide to add another element to the array, you simply add another initializer.

But how do you determine the size of the array? The expression sizeof c / sizeof \*c (size of the entire array divided by the size of the first element).

In the examples above, the initializers are assigned directly to the elements of the aggregate, but constructors are a way of forcing initialization to

occur through a formal interface. Here, the constructors must be called to perform the initialization. So if you have a struct that looks like this,

struct Y {

float f;

int i;

Y(int a);

};

You must indicate constructor calls. The best approach is the explicit one as follows:

Y y1[] = { Y(1), Y(2), Y(3) };

You get three objects and three constructor calls.

# Default constructors

A default constructor is one that can be called with no arguments. A default constructor is used to create a “vanilla object,” but it’s also important when the compiler is told to create an object but isn’t given any details.

The default constructor is so important that if (and only if) there are no constructors for a structure (struct or class), the compiler will automatically create one for you.

// Automatically-generated default constructor

class V {

int i; // private

}; // No constructor

int main() {

V v, v2[10];

}

If any constructors are defined, however, and there’s no default constructor, the instances of V above will generate compile-time errors.

You might think that the compiler-synthesized constructor should do some intelligent initialization, like setting all the memory for the object to zero. But it doesn’t – that would add extra overhead but be out of the programmer’s control. If you want the memory to be initialized to zero, you must do it yourself by writing the default constructor explicitly.

Although the compiler will create a default constructor for you, the behavior of the compiler-synthesized constructor is rarely what you want. In general, you should define your constructors explicitly and not allow the compiler to do it for you.

# END