6.2.4 Storage durations of objects

1 An object has a *storage duration* that determines its lifetime. There are three storage durations: static, automatic, and allocated. Allocated storage is described in 7.20.3.

2 The *lifetime* of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address,[25] and retains its last-stored value throughout its lifetime.[26] If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.

3 An object whose identifier is declared with external or internal linkage, or with the storage-class specifier **`static`** has *static storage duration*. Its lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup.

4 An object whose identifier is declared with no linkage and without the storage-class specifier **`static`** has *automatic storage duration*.

5 For such an object that does not have a variable length array type, its lifetime extends from entry into the block with which it is associated until execution of that block ends in any way. (Entering an enclosed block or calling a function suspends, but does not end, execution of the current block.) If the block is entered recursively, a new instance of the object is created each time. The initial value of the object is indeterminate. If an initialization is specified for the object, it is performed each time the declaration is reached in the execution of the block; otherwise, the value becomes indeterminate each time the declaration is reached.

6 For such an object that does have a variable length array type, its lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration.[27] If the scope is entered recursively, a new instance of the object is created each time. The initial value of the object is indeterminate.

Forward references: statements (6.8), function calls (6.5.2.2), declarators (6.7.5), array declarators (6.7.5.2), initialization (6.7.8).

---

25) The term "constant address" means that two pointers to the object constructed at possibly different times will compare equal. The address may be different during two different executions of the same program.

26) In the case of a volatile object, the last store need not be explicit in the program.

27) Leaving the innermost block containing the declaration, or jumping to a point in that block or an embedded block prior to the declaration, leaves the scope of the declaration.

example) without destroying class objects with automatic storage duration.]

6.6.1 The `break` statement [stmt.break]

1 The `break` statement shall occur only in an *iteration-statement* or a `switch` statement and causes termination of the smallest enclosing *iteration-statement* or `switch` statement; control passes to the statement following the terminated statement, if any.

6.6.2 The `continue` statement [stmt.cont]

1 The `continue` statement shall occur only in an *iteration-statement* and causes control to pass to the loop-continuation portion of the smallest enclosing *iteration-statement*, that is, to the end of the loop. More precisely, in each of the statements

```
while (foo) {          do {                for (;;) {
 {                      {                   {
  // ...                  // ...              // ...
 }                      }                   }
contin: ;              contin: ;           contin: ;
}                      } while (foo);      }
```

a `continue` not contained in an enclosed iteration statement is equivalent to `goto contin`.

6.6.3 The `return` statement [stmt.return]

1 A function returns to its caller by the `return` statement.

2 A return statement without an expression can be used only in functions that do not return a value, that is, a function with the return type `void`, a constructor (12.1), or a destructor (12.4). A return statement with an expression of non-void type can be used only in functions returning a value; the value of the expression is returned to the caller of the function. The expression is implicitly converted to the return type of the function in which it appears. A return statement can involve the construction and copy of a temporary object (12.2). Flowing off the end of a function is equivalent to a `return` with no value; this results in undefined behavior in a value-returning function.

3 A return statement with an expression of type "*cv* `void`" can be used only in functions with a return type of *cv* void; the expression is evaluated just before the function returns to its caller.

6.6.4 The `goto` statement [stmt.goto]

1 The `goto` statement unconditionally transfers control to the statement labeled by the identifier. The identifier shall be a label (6.1) located in the current function.

6.7 Declaration statement [stmt.dcl]

1 A declaration statement introduces one or more new identifiers into a block; it has the form

declaration-statement:
block-declaration

If an identifier introduced by a declaration was previously declared in an outer block, the outer declaration is hidden for the remainder of the block, after which it resumes its force.

2 Variables with automatic storage duration (3.7.2) are initialized each time their *declaration-statement* is executed. Variables with automatic storage duration declared in the block are destroyed on exit from the block (6.6).

3 It is possible to transfer into a block, but not in a way that bypasses declarations with initialization. A program that jumps[77)] from a point where a local variable with automatic storage duration is not in scope to a

[77)] The transfer from the condition of a `switch` statement to a `case` label is considered a jump in this respect.

point where it is in scope is ill-formed unless the variable has POD type (3.9) and is declared without an *initializer* (8.5).
[*Example:*

```
void f()
{
    //...
    goto lx;                    // ill-formed: jump into scope of a
    //...
ly:
    X a = 1;
    //...
lx:
    goto ly;                    // OK, jump implies destructor
                                // call for a followed by construction
                                // again immediately following label ly
}
```

—*end example*]

4 The zero-initialization (8.5) of all local objects with static storage duration (3.7.1) is performed before any other initialization takes place. A local object of POD type (3.9) with static storage duration initialized with *constant-expression*s is initialized before its block is first entered. An implementation is permitted to perform early initialization of other local objects with static storage duration under the same conditions that an implementation is permitted to statically initialize an object with static storage duration in namespace scope (3.6.2). Otherwise such an object is initialized the first time control passes through its declaration; such an object is considered initialized upon the completion of its initialization. If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time control enters the declaration. If control re-enters the declaration (recursively) while the object is being initialized, the behavior is undefined. [*Example:*

```
int foo(int i)
{
    static int s = foo(2*i);    // recursive call – undefined
    return i+1;
}
```

—*end example*]

5 The destructor for a local object with static storage duration will be executed if and only if the variable was constructed. [*Note:* 3.6.3 describes the order in which local objects with static storage duration are destroyed.]

6.8 Ambiguity resolution [stmt.ambig]

1 There is an ambiguity in the grammar involving *expression-statement*s and *declaration*s: An *expression-statement* with a function-style explicit type conversion (5.2.3) as its leftmost subexpression can be indistinguishable from a *declaration* where the first *declarator* starts with a `(`. In those cases the *statement* is a *declaration*. [*Note:* To disambiguate, the whole *statement* might have to be examined to determine if it is an *expression-statement* or a *declaration*. This disambiguates many examples. [*Example:* assuming `T` is a *simple-type-specifier* (7.1.5),

```
T(a)->m = 7;                    // expression-statement
T(a)++;                         // expression-statement
T(a,5)<<c;                      // expression-statement
```