Bindings, Initialization, Scope, and Lifetime

by: Maria Taku & Eric Powders

(mat2185@columbia.edu)

(ejp2127@columbia.edu)

Columbia University

COMS E6998 – Advanced Topics in Programming Languages & Compilers

Fall 2012

Table of Contents

1. Declarations and Definitions 3

1.1 Background 3

1.2 Declarations and Definitions in Classes....4

2. Scope and Lifetime....5

3. Initialization 7

3.1 Background 7

3.2 Initialization of Class Data Members....7

3.3 Initialization with Multiple Inheritance....9

3.4 Static Data Member Initialization 10

4. Binding Lifetime 11

5. Object Lifetime....12

5.1 Background 12

5.2 Static Memory Allocation....13

5.3 Stack Memory Allocation 14

5.4 Heap Memory Allocation 15

5.4.1 Heap Memory Management – Free Lists....16

5.4.2 Heap Memory Management – Garbage Collections 18

1. Declarations and Definitions

1.1 Background

A declaration makes known to the world the type and name of a variable. A definition, on the other hand, allocates storage for the variable and might also specify its initial value. A definition is what makes the variable "usable." If something has been declared but not yet defined then we can't yet use that variable because storage hasn't yet been allocated for it. In many languages a declaration, by default, is also a definition. Often it is acceptable to have many declarations for the same variable, although typically only one definition is permitted; of course, all declarations must agree on the variable's type in a strongly-typed language. The same thing can apply to functions; it is often permissible to declare a function numerous times so long as the number and types of its arguments, and its return type, remain the same. Typically, however, it is only permissible to *define* a function once in the same scope, otherwise the compiler wouldn't know which definition to use. Function overloading is, of course, an exception, in which case it is permissible to both declare and define a function with the same name yet with different parameters than previously seen; of course, this creates an entirely new function altogether, which is why it is permitted.

In some languages, you *can* declare a variable without yet defining it, such as by preceding the declaration with the keyword *extern* in C++. This is valuable, for example, when you wish to reference a variable whose definition will appear later (such as further down in the current translation unit, or perhaps in another translation unit). Additionally, some languages permit function declarations to appear inside other functions, in which case the scope of the *inner* function is typically constrained to be only while inside the *outer* function.

1.2 Declarations and Definitions in Classes

When developing a language with classes, the language creator must make decisions about bindings, scope, and lifetime within a class. For example, when defining a class member function, should the user be permitted to reference *any* name in the class, even names that haven't yet been seen? In C++, this is permitted; it is ok to reference any other name in the class, even if its declaration and/or definition appear textually after this function's definition. This is accommodated because first a class's member *declarations* are compiled in their entirety, and only *then* are the *definitions* compiled; therefore, it is even okay for class member functions' definitions to cross-reference each other. In a nested class in C++, the member declarations of the outer class *and* the member declarations of all of the nested classes are first compiled, *then* all of the definitions are compiled. This, again, prevents compilation errors when referencing names not yet textually seen.

Another question for the language designer is when a class name can be used. For example, is a class considered declared once the class name has been introduced, or not until the class has been defined? For example, in C++, a class is considered *declared* once the name has been seen, but a class isn't considered *defined* until the *entire* class body is complete. One implication of this is that the class can't have data members of its own type, but *can* have data members that are references or pointers to its own type. This is important, for example, when creating a linked list class. Each node needs to be able to hold one (or more) pointers to other node(s), which are typically objects of its same type. We wouldn't expect a class to be able to hold data members of its own type, as this becomes an infinitely recursive class definition. Of course, a class *can* have static data members of its own type, because static data members aren't part of the instantiated class objects themselves; they are stored separately in static memory. Therefore, a class object holding a static data member of its own type merely references one static address in memory, which isn't a recursive definition.

Another question that language designers must consider is whether or not class functions can utilize other class members as default function arguments. In C++, for example, class functions may only utilize, as default function arguments, *static* class members, because static class members will have already been defined before the class object itself has been instantiated. Using a non-static class member as a default function argument leads to the possibility that this class member may not yet have been defined.

One additional item worth mentioning is that, in C++, when defining a namespace the user cannot, in fact, reference any other name in the namespace whose declaration or definition hasn't yet appeared. Therefore, a known technique is for the user to first *declare* all of the names *inside* the namespace definition, and then to later *define* all of the names *outside* of the namespace definition; this permits access to *all* of the namespace names when writing such definitions.

2. Scope and Lifetime

When designing a language, the designer must decide when names come into existence and how long a name is valid for; this is known as a name's scope and lifetime. Programs often have a number of different scopes for the language designer to consider, such as local scope, global scope, statement scope, class scope, and namespace scope. Additionally, there are a number of different areas of memory in which to store variables and objects, such as the stack, heap (free store), and static memory. Typically items stored in the stack and heap are more directly under the programmer's control, while items stored in static memory might be created once at the start of execution and exist for the duration of execution. Static memory is often used to hold things such as global variables, static variables, and constants.

Order of execution is often an important concept when designing a language. For example, can it be assumed that memory allocation will be performed in the order in which variables are encountered

in a program? What about allocation in static memory that occurs at the start of execution? Can we make any guarantees about the order of variable allocation within a translation unit? What about between different translation units? In C++, for example, a local static object is created and initialized only once, the first time that its definition is encountered during program execution; it then lives until program termination. C++ does guarantee that local static objects are created in the order in which they are encountered, and destroyed in reverse order; exactly when this happens, however, is unspecified by the language. Additionally, there are no guarantees about the order in which this might happen *across* different translation units. Because of this, subsequent definitions in a translation unit *can* reference variables previously defined in *that translation unit*; of course, definitions must not reference variables defined in *other* translation units, as there is no ordering guarantee across translation units. Thus coders must ensure that they do not create dependencies across translation units.

Another scope and lifetime question that language designers face is how to accommodate for the scope and lifetime of composite variables, such as array elements and class data members. In C++, for example, the lifetime of such variables is determined by the object of which they're a part. So, for example, if an array is declared as a global object (in static memory), and a class object is instantiated as a static variable (again, in static memory), then all elements of that array and all data elements of that class will be stored in static memory, and all will be instantiated in accordance with the rules of static memory.

Language designers must also create rules for the scope and lifetime of temporary variables and objects. In C++, for example, a temporary variable that is created as part of the evaluation of an expression will persist until the end of the evaluation of the full expression in which it occurs.

3. Initialization

3.1 Background

Language designers must create rules for initialization; these rules must answer questions such as: When are variables initialized? To what value are variables initialized if the user doesn't specify an initialization value? In order to address such questions and the issues surrounding them, let's take a look at how C++ handles things.

In C++, built-in types and enums in static memory without an initializer specified by the user are initialized to zero. Objects of user-defined types created in static memory will have the default constructor for that object type invoked. Variables *not* created in static memory (so, variables allocated on the stack or the heap), however, will *not* be initialized to well-defined values unless the user specifies an initialization value. The exception to this rule are variables that are part of an array or class that itself is stored in static memory, in which case it will be initialized. Objects of user-defined types will have their default constructors invoked; if the class doesn't have a default constructor, a compiler error will occur. In such cases, the user must either add a default constructor to the class, or supply appropriate arguments to the object's instantiation such that an existing constructor may be invoked by the compiler.

In C++, users can force default initialization via a technique known as value-initialization. Rather than saying "*int* a = new int;*" which will allocate uninitialized memory on the heap, the user can instead say "*int* a = new int();*". The addition of the trailing parentheses are an instruction to the compiler to force-initialize *a* to the default initialization value for an integer, which is zero.

3.2 Initialization of Class Data Members

The same rules apply to class data members. If a class data member isn't specifically initialized at the time the object is instantiated, then it will be default-initialized if the object is in static memory, or

it will be left uninitialized if the object is not allocated in static memory. Thus, if a class data member is itself a user-defined type, then the default constructor for it shall be invoked; if a class data member is an array of user-defined types, then its default constructor shall be invoked for each element of the array. In general, class data members are initialized (or left uninitialized) only when the constructor is called, because an object doesn't have an address until then.

This leads to some interesting consequences. If, for example, a class in C++ contains a constant, objects of this class type cannot be default-initialized because constants must be initialized when they are allocated. If a user tries to instantiate an object of this type, without providing an initializer for the object (such as by calling an appropriate constructor), the user will get a compiler error that the object cannot be initialized. The same thing applies for references. C++ requires that references be initialized when they are created; thus, a default constructor cannot be invoked on a class that contains one or more references. Finally, a class that contains a data member that is itself of a user-defined type without a default constructor cannot itself be default-initialized. The reason is that the inner data member cannot be default-initialized; thus, the containing class itself cannot be default initialized because the compiler will be unable to initialize the inner data member.

Language designers must decide when objects (and their embedded data members) of user-defined type are initialized. In C++, for example, a class's data members are constructed and initialized at the time of object instantiation but before the body of the class constructor is executed. For derived classes, all of the base classes are first constructed and initialized, and then the derived class portion of the object is constructed. Data members are constructed and initialized in the order in which the members are declared in the class definition, *regardless* of the order the user specifies them in the constructor initializer list. All of this must be accounted for by the user when writing code! For example, inside the constructor code, the user can reference any data member in the class (including data

members inherited from the base classes), because all data members are constructed and initialized before the constructor code is run. Additionally, as part of the member initialization list, the user *can* initialize data members from other data members that have already been initialized; the user, of course, must be careful to ensure that only already-initialized data members are used to initialize other data members. The compiler will not report this type of misstep. Finally, in C++, data members are destroyed in reverse order *after* the body of the class's destructor has run; as the final step in the destruction process, the object's memory is released. In objects such as an array or container, the elements are destroyed in reverse order. Thus, the last element in an array is destroyed first, then the next-to-last element, etc, until the first element in the array is finally destroyed.

There is another interesting consequence to the decision by the C++ language designers that all of an object's data members are initialized before the body of the constructor is run. A lot of newcomers to C++ provide default values for data members *inside* the class constructor's body. What these newcomers don't realize is that the data members have *already* been initialized before the constructor body is run! The proper method for data member initialization in C++ is to initialize data members as part of the constructor's member initializer list, which occurs before the body of the constructor is run. In the best case, this prevents dual initialization, as the compiler would default-initialize all of the class data members prior to running the body of the constructor, and then "re-initialize" them in the body of the constructor, per the user's code. In the worst case, the class won't compile if some of the class data members cannot be default-initialized (such as constants or references, as previously mentioned), in which case a C++ newcomer would be confounded as to why he is getting seemingly-incorrect compiler errors.

3.3 Initialization with Multiple Inheritance

If languages permit multiple inheritance, the language designer must think through the consequences of the order of construction and initialization of all of the base class hierarchies. In C++,

for example, there are very precise rules for this. When instantiating an object that is inherited from multiple base classes, the first step the compiler takes is to instantiate all virtual base classes (and their base classes), and initialize all of their data members. The compiler next instantiates all non-virtual base classes (and their base classes) and initializes all of their data members. Thus, the user can know for certain that all virtual bases will be constructed and initialized before non-virtual bases are constructed and initialized. Of course, all of this happens before the derived class is constructed and initialized. An additional decision of consequence made by the C++ designers is that classes are permitted to inherit from the same non-virtual base class multiple times. Thus, the user must recognize that all data members in the non-virtual base classes will be replicated *n* times, where n is the number of times that the derived class has inherited from this base. Finally, object destruction (when the object loses scope) occurs in the precise reverse order of object construction and initialization. C++ is very precise about its definition and enforcement of these rules for object initialization involving multiple inheritance, which can cause confusion for new C++ developers but can prove valuable to advanced C++ developers who can leverage these features.

3.4 Static Data Member Initialization

Static variables in classes (known as static data members) are not part of an instantiated object; instead, static data members are allocated ahead of time at a static address in static memory. All instantiated objects of this class type will share this static data member. This creates an interesting issue: When and how should such static data members be initialized? C++, for example, requires that static data members be defined *somewhere*. A declaration alone won't suffice for static data members. This ensures that space is properly allocated for the static data member by the compiler, since such action will not happen at time of object instantiation (at which point the other, *non*-static data members will be allocated).

4. Binding Lifetime

In the most general sense, *name binding* deals with binding the name of an entity with that entity itself. For example, this can represent an identifier that is bound to a particular object variable or function. Similarly, the *binding time* deals with the period of time in which this binding is valid.

The binding time for a particular entity can occur at different times in an entity's life; and the choices for binding times not only vary greatly among different programming languages, but can also vary a lot within a particular language itself. For example, names can be bound as early as the language design time (e.g., binding of keywords such as FOR and WHILE), to compile time (e.g, statically defined data), and all the way up until runtime (e.g,. dynamic binding).

In general, earlier binding times results in greater efficiency, since various computations can be performed before runtime of a program. In contrast, later binding times can be associated with greater flexibility and use of the code. For example, the following Java pseudocode illustrates dynamic binding and its flexibility:

```
abstract class Shape {
        protected float length;
        public abstract float area();
}

public class Circle extends Shape {
        Circle(float radius){
                length = radius;
        }
        public float area (){
                return 3.14*(length^2);
        }
}

public class Square extends Shape {
        Square(float side){
                length = side;
        }
        public float area (){
                return (length^2);
```

```
        }
}

public class DynamicBinding{

        public void useMyShape(Shape s){
                s.area();          //dynamic binding!
                .
                .
                .
        }
}
```

As shown by this example, the exact *area()* function that is called in the *useMyShape* method will not be known until runtime. Whether this calls a function to calculate a circle's area or a square's area will depend on whether an object of the Circle subclass (e.g., declared via *Shape circ = new Circle(1);* or such*)* or an object of the Square subclass (e.g., declared via *Shape squ = new Square(1);* or such) is passed as an argument into *useMyShape* respectively. As such, although such dynamic binding may not run as efficiently due to extra runtime processing, it can allow for much greater flexibility and code re-use for a programmer.

5. Object Lifetime

5.1 Background

A concept similar to that of binding lifetime is *object lifetime*. Specifically, this refers to the period of time between an object's creation and its subsequent destruction. This can be compared to the definition of a binding lifetime, which refers to the period of time between a name-to-object's construction and destruction.

At times, the object lifetime and binding lifetime of an entity can directly overlap. However, this is not always the case. For example, this is illustrated by the following Java pseudocode in which variables are passed by reference:

```
foo(Object oRef){
        ...perform various operations via oRef...
}

Object myObject = new Object();
foo(myObject);
```

As can be seen from this example, during the scope of the method *foo*, the identifier *oRef* has a name-to-object binding to *myObject*. However, once *foo* has completed, this name-to-object binding is destroyed, although *myObject* continues to exist.

The above illustrated a situation where the object itself last longer than the name-to-object binding. However, situations may also occur where the name-to-object binding lasts longer than the object. For example, this may occur when a heap variable is destroyed while still having references or pointers to it. These are referred to as *dangling references* and oftentimes indicate an error on the part of the programmer.

As is described in more detail in the following sections, an object's lifetime generally corresponds to its allocation in memory space, whether this is static memory allocation, stack memory allocation, or heap memory allocation.

5.2 Static Memory Allocation

Static memory allocation generally refers to memory that is allocated at compile-time, before the actual execution of a program. Specifically, once a program begins to execute, there should be specific blocks of memory that are set aside to ensure the proper operation of the program; this memory should not be trespassed upon by another program, by the system, or even by the program itself. The addresses and the size of static memory allocations are definitively set during compilation. The term "static" memory allocation receives its name from this feature, since such data does not vary in size or location during the lifetime of the program.

Objects stored in static memory are thus alive and accessible throughout the entire life of a program. Static variables defined within a class also have the added flexibility that a single copy of that data can be shared between all objects of that class.

Examples of items that are placed in static memory include global variables, string and numeric constants, and a variety of compiler-produced tables used in runtime support processes such as garbage collection, dynamic-type checking, exception handling, etc.

5.3 Stack Memory Allocation

The stack region of memory refers to an area of memory where data is added and removed in a Last-In-First-Out manner. In general, when a function executes, it adds relevant material to the stack. The area of the stack in which this "relevant material" is stored is known as that function's *activation record*, and can contain various data such as return addresses, arguments, local variables, bookkeeping information etc. Once the function exits, it is responsible for removing its data from the stack.

Due to its Last-In-First-Out nature, the memory stack is a prime candidate for use in recursive functions. In fact, static memory allocation for recursive functions would not even be an option, since the number of recursive calls is unknown at compile time. The natural nesting of subroutines in recursive calls is easily handled by a stack, however.

As hinted at above, a natural benefit of stack memory is the fact that memory is automatically, and efficiently, reclaimed whenever a function exits. Some languages, such as C++ allow a programmer to manually determine whether objects are allocated to the memory stack or to the memory heap. For example, in C++ a programmer may either declare a new object via:

```
Object c1;                    // Stack allocation.  c1 will automatically delete when it
                              // goes out of scope
Object * c2 = new Object();   // Heap allocation. Regardless of scope, c2 will not be
                              // deleted until "delete c2;" in specifically invoked
```

In the example above, c1 will automatically be deleted once the function exits. This frees the programmer from needing to manually manage the memory allocation of the object. In fact, even if the method containing *c1* is unnaturally aborted, *c1* will still be automatically de-allocated from memory. Especially in the case of recursive functions (e.g., which might re-create *Object c1* every time the function is recursively called), placing objects on the memory stack in this manner may help to reduce destructive memory or resource leaks. In contrast, objects stored in the memory heap must be manually deleted.

Not all languages allow a programmer to determine whether objects are placed on the memory stack or the memory heap, however. In Java, for example, all objects are always and only placed on the memory heap. Since, unlike C++, Java has automatic garbage collection, however, the potential extra memory wastage of placing all objects on the heap is not so severe.

5.4 Heap Memory Allocation

As mentioned above, the heap is an area of memory used for dynamic memory allocation; for example, data such as lists, sets, linked data structures, or other objects whose size may change during runtime can be stored in the heap.

In the heap, memory is allocated and de-allocated in an arbitrary order. Due to this nature of heaps, there generally must be some system in place in order to manage the space within the heap. For example, consider Figure 1 showing a heap and an allocation request. As shown by this figure, although there is enough total free space to fit the requested data segment, due to the fragmented nature of the free space, the data segment cannot actually be put onto the heap (when data is allocated to the heap, it is generally not split into smaller chunks – the entire segment remains in one contiguous piece).

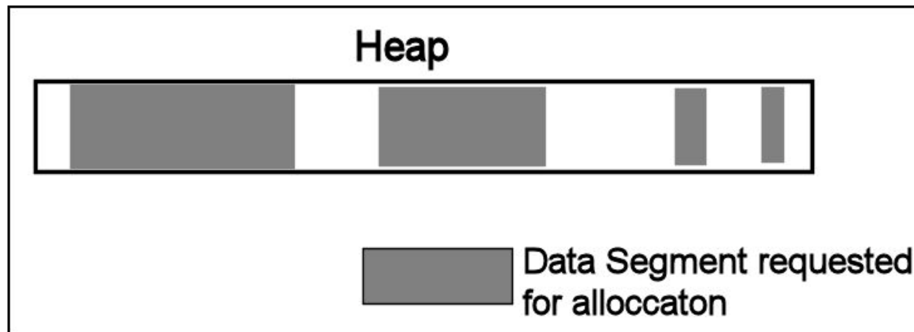

Figure 1: Fragmented heap memory and allocation request.
Gray: Used Memory. White: Free Memory

Figure 1 illustrates a case of external fragmentation, in which the objects stored in the heap have been scattered in such a way that the remaining free space is widely interspersed and essentially unusable – although there might be enough total free space to fit a new object, there is no single piece of free space large enough to actually accept it. Accordingly, there are several ways of dealing with performing heap memory management in order to prevent such fragmentation scenarios.

5.4.1 Heap Memory Management – Free Lists

One method of managing the storage allocation in heap memory deals with using a singly-linked list known as a free list. The free list basically keeps track of all blocks of free space within the heap. Thus, when the heap is initially created, the free list merely consists of a single linked-list entry that comprises the entire heap itself. As memory is allocated to the heap, and the free space is divided into multiple, smaller chunks of memory, the free list is updated such that each linked-list entry indicates one of these free blocks of memory.

In order to allocate a requested chunk of data to the heap, one of two algorithms can be used: the *first fit algorithm* or the *best fit algorithm.*

As the name implies, the first fit algorithm simply selects the first free block of memory in the heap that is large enough to hold the requested data. In other words, the free list is traversed until the first linked-list entry is found that references a free block of data of suitable size. After the data is

placed into the heap, if the chosen block is significantly larger than required, then the free block is divided into its remaining free blocks, and the free list is updated accordingly. However, it's worthwhile to note that if the free block is below some minimum threshold in size, then the entire block (even though it is slightly too large) is allocated to the requested data. This can result in a type of fragmentation known as *internal fragmentation,* where larger-than-necessary blocks of heap memory are allocated to data, thus resulting in wasted space.

The best fit algorithm, on the other hand, looks through the entire heap memory and chooses a free block that "best fits" the requested data. In other words, the best fit algorithm chooses the smallest block of free data that is large enough to satisfy the data allocation request. Thus, the best fit algorithm sacrifices speed (since the entire free list must be traversed during each allocation request) in order to locate free blocks of memory which are most suitable for the requested data allocation.

One interesting question is whether the first fit algorithm or the best fit algorithm results in less fragmentation of the heap memory. At first glance, it may seem that the best fit algorithm will do a better job at managing the heap memory, since it always strives to find the "best fit," whereas the first fit algorithm may somewhat arbitrarily choose free memory blocks in which to insert its data. However, since the best fit algorithm always finds the smallest possible free blocks of memory to store its data, it can result in a lot of very small, leftover free blocks. This can thus hurt the performance of the best fit algorithm in the long run, as it resulting in a lot of tiny free blocks of memory which may be unsuitable for storing data. In general, whether the first fit algorithm or the best fit algorithm works better can vary drastically from case to case – which one results in less fragmentation of the heap can actually depend on the distribution of size requests.

5.4.2 Heap Memory Management – Garbage Collection

Heap memory is not infinite, and at some point data from the heap must be removed in order to make room for new data. In some languages, this erasure of heap memory is done manually. For example, in C the command *free* can be invoked and in C++ the command *delete* can be invoked in order to manually remove data from the memory heap. However, many languages now include an *automatic garbage collection* process that will automatically remove unnecessary data from the heap for the programmer.

The benefits of automatic garbage are, most obviously, the convenience for the programmer. Keeping track of the heap memory and remembering to properly delete unnecessary data can be a large chore for the programmer. Freeing the programmer of this housekeeping allows them to concentrate more on the logic and actual meaning of their programs. Moreover, manually removing unnecessary data from the heap can be extremely error prone, oftentimes resulting in dangling references (e.g., from deleting an object to soon) or memory leaks (e.g., failing to de-allocate data) that can cause costly and dangerous bugs in a program. For example, since data is re-used in the memory heap, dangling references may now have the power to read or write bits to an object that was originally not its intended target. The dangling reference may now unintentionally modify those objects or, if the dangling reference is now pointing to bookkeeping information, it may even corrupt the structure of the heap itself.

On the other hand, automatic garbage collection has the disadvantage that it can greatly sacrifice execution speed. When garbage collection is invoked, it can consume non-insignificant amounts of processing power. Moreover, languages like Java do not allow a programmer to control when garbage collection is actually invoked. Although there are methods like System.gc () and Runtime.gc () which can request of Garbage collection, it's not guaranteed that garbage collection will actually happen. This can be especially harmful in time-sensitive program, such as high frequency

trading programs, where garbage collection invoking at the wrong moment could drastically hurt the program's performance and effectiveness. Nonetheless, despite these faults, garbage-collection algorithms are steadily improving and their run-time overhead is reducing, thus slowing dissolving these disadvantages of automatic garbage collection.

But how does a garbage collector know that data in a heap can be reclaimed? Most simply we know an object is no longer needed when there are no more pointers to that object. Thus, one method of finding reclaimable data is to place a counter in each object that keeps track of the number of pointers to it. If this counter is set to 0, then that object is a candidate for garbage collection. However, such counters may fail to locate circular structures, where two or more objects all point to each other in a circle, yet none of them can be viably reached by the program. One way of locating such circular structures is to start from a valid identifier, and then determining which objects can actually be reached by this identifier. For example, the garbage collector may start outside of the heap, and follow the chain of valid pointers into the heap, keeping track of which objects are actually reachable. Since circular structures will never be reached in such a traversal, they can be successfully marked as reclaimable data.

A brief example of the Java Automatic Garbage Collector is now described. Java's automatic garbage collectors run on the premise that older data is less likely to be unnecessary and reclaimable. As such, the heap is divided into three generations known as the *Young Generation, Tenured Generation, and PermGen.* The Young Generation is furthered divided into three parts known as *Eden, Survivor 1, and Survivor 2.* When an object is first created, it is put into the "youngest" heap area possible – Eden of the Young Generation. A process known as *Minor Garbage Collection* is run in this area. If an object in Eden survives Minor Garbage Collection, then it is moved to Survivor 1. If the object survives Minor Garbage Collection a 2nd time, then it is moved to Survivor 2. Once in this area,

Major Garbage Collection is performed. If an object survives Major Garbage Collection, then it is moved to the Tenured Generation. Major Garbage Collection alone is performed on the Tenured Generation.

The benefit of dividing the garbage collection in this manner is that the garbage collection pauses can potentially be shortened. For example, when garbage collection is triggered (e.g., by the heap reaches a sufficient capacity threshold), then the Minor Garbage Collection is first executed. Since many Eden/Survivor 1 objects are often already dead, Minor Garbage Collection pauses can be short and can also free up a significant portion of heap memory. If enough heap memory is freed, Major Garbage Collection does not even need to be run, and the program can continue immediately. Even though there may be unreachable objects in the Tenured Generation, as long as Minor Garbage Collection has freed up enough heap memory, then there is no reason to reclaim the Tenured Generation space.

Lastly, the PermGem area of the heap memory stores "permanent" information such as metadata about classes and methods that have been loaded, String pool, and class level detail. Garbage collection can occur in this area, but the exact implementation can vary from JVM to JVM.