This topic contains guidelines and recommendations about common memory profiling scenarios, including some heuristics about memory usage patterns that might indicate a memory problem. Of course, every application is different, so these guidelines only outline strategies for memory profiling; to apply these strategies you will need a good understanding of your application.

- Finding a memory leak (page 1)
- Checking that a memory leak is fixed (page 4)
- Finding out what is using most memory (page 5)
- Checking for memory problems (page 5)

More information about memory management

When you are investigating memory usage, background information about .NET memory management may be useful. Some good starting points:

Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework
(Memory Management in .NET
http://www.c-sharpcorner.com/UploadFile/tkagarwal/MemoryManagementInNet.aspx)

CLR Inside Out: Large Object Heap Uncovered

Dangers of the Large Object Heap

Finding a memory leak

The following examples outline some approaches to finding a memory leak for some common scenarios.
We recommend that you make a note of the steps you take when you are looking for a memory leak, so that you can perform the same actions later to check that you have fixed the leak.

The snapshots and analysis you perform depend on the functionality of your application.

Example A: opening and closing a dialog box

This scenario is the most straightforward, so it is recommended as the preferred way of finding a memory leak.

1. Start ANTS Memory Profiler and start profiling your application. Get the application into the state in which you are interested in its memory.

2. Take two snapshots, so you can compare memory usage before and after the action that you believe leaks memory:
   a. Take the first snapshot.
   b. Perform an action that you believe causes a leak; then perform the actions that should clean up any objects created by the first action.
      For example, open a dialog box, change some settings, and then close the dialog box.
   c. Take a second snapshot.

   Objects created by the action should be cleaned up before the second snapshot, so any new objects created in the second snapshot are likely to indicate a memory leak.

3. Apply the **Only new objects** filter to show only the classes with new instances in the second snapshot.

4. On the class list, look for classes with a high positive value in the **Instance Diff** or **Size Diff** column. These values indicate the classes responsible for increased memory usage in your second snapshot, so they are good indicators of the likely cause of a memory leak. At this stage, we recommend that you look at *all* classes (not just the classes you recognize): although your own classes may be responsible for the memory leak, the symptoms of the leak may be increased usage in other classes, such as `System.String`.

5. Look at instances of classes with unexpectedly high growth in size or number of instances:
   * If the class that looks interesting is one you recognize, look at instances on the object list.
   * If the class that looks interesting is not one you recognize, use the class graph to navigate along the chain of references to objects in this class, until you reach a class you recognize. Next, look at instances of that class on the object list.

6. On the object list, look for objects with a high value in the **Distance from GC Root** column.

   Often, leaked objects are found at a greater distance from their nearest GC root because all the obvious, shorter chains of reference from a GC root to an object have been broken already.
7. Show the object graph for the object that looks interesting. Follow chains of references up the graph to identify objects keeping your object in memory unexpectedly.

Example B: populating and clearing a list

1. Start ANTS Memory Profiler and start profiling your application. Get the application into the state in which you are interested in its memory.

2. Take two snapshots, so you can compare memory usage before and after the action that you believe leaks memory:
   a. Perform the action that you believe causes a leak. For example, populate a list with data.
   b. Take the first snapshot during or immediately after this action - that is, before any clean-up happens.
   c. Perform the action that should clean up the objects created by the first action. For example, clear the data from your list.
   d. Take a second snapshot.
   Objects should be cleaned up between snapshots, so any remaining objects were probably created by the first action.

3. Apply the **Only surviving objects** filter to show only classes that exist in both snapshots.

4. On the class list, look for classes with a high value in the **Live Instances** or **Live Size** column. These values indicate the classes responsible for memory usage in your snapshots. At this stage, we recommend that you look at all classes (not just the classes you recognize): although your own classes may be responsible for the memory leak, the symptoms of the leak may be increased usage in other classes, such as `System.String`.

5. Look at instances of the class with unexpectedly high size or number of instances:
   - If the class that looks interesting is one you recognize, look at instances on the object list.
   - If the class that looks interesting is not one you recognize, use the class graph to navigate along the chain of references to objects in this class, until you reach a class you recognize. Next, look at instances of that class on the object list.

6. On the object list, look for objects with a high value in the **Distance from GC Root** column. Often, leaked objects are found at a greater distance from their nearest GC root because all the obvious, shorter chains of reference from a GC root to an object have been broken already.

7. Show the object graph for the object that looks interesting. Follow chains of references up the graph to identify objects keeping your object in memory unexpectedly.
Example C: general strategy recommendation

This way of finding a memory leak is recommended:

- if you are not sure what functionality or actions are causing the memory leak
- for applications where you expect memory to be constant, but instead memory usage increases slowly
- for applications that do not have functionality that you can manually execute and then clean up

1. Start ANTS Memory Profiler and start profiling your application.
2. Run your application and monitor memory usage on the timeline. When memory usage starts to increase, take several snapshots (the frequency and number of snapshots you need to take depends on your application and how rapidly memory increases).
3. Select two snapshots to compare.
4. If there are classes that you expect to be large or increasing in size, apply the **Never referenced by an instance of class** filter to remove these classes from the results.
5. On the class list, look for classes with a high value in the **Size Diff** column. This value indicates the growing classes, so it is a good indicator of a memory leak. At this stage, we recommend that you look at all classes (not just the classes you recognize): although your own classes may be responsible for the memory leak, the symptoms of the leak may be increased usage in other classes, such as `System.String`.
6. Look at instances of the class with the largest difference in size between snapshots:
   - If the class that looks interesting is one you recognize, look at instances of the object list.
   - If the class that looks interesting is not one you recognize, use the class graph to navigate along the chain of references to objects in this class, until you reach a class you recognize. Next, look at instances of that class on the object list.
7. On the object list, apply the **Only surviving objects** filter to show only the classes that exist in both snapshots, and then look for high values in the **Live Size** column. This identifies the largest objects, which have stayed in memory for longest, so may indicate the cause of a leak.
8. Show the object graph for the largest object that should not be in memory. Follow chains of references up the graph to identify objects keeping your object in memory unexpectedly.

Checking that a memory leak is fixed

The following steps outline how to check that a memory leak you previously identified is now fixed:

1. Repeat the steps you used to find the memory leak (page 1). If you are looking for a particular class or object, it may be useful to use the find box to locate the class or object you are interested in.
2. If you have fixed the leak, the objects should not be in memory.
If an unexpected instance does seem to still be in memory, display the object on the object graph, and check whether it is on the finalizer queue (clear the Hide finalizer queue GC roots option on the bar above the graph). If your object is on the finalizer queue, take another snapshot and check again: the object may be removed when the garbage collector runs.

Finding out what is using most memory

The following steps outline how to find out what classes are using most memory in your application.

1. Start ANTS Memory Profiler and start profiling your application. Get the application into the state in which you are interested in its memory, and then take a memory snapshot.

2. On the class list, look for the largest classes, or the classes with the highest number of instances. For this analysis, you are going to explore memory usage by following references to a class, so it is not important if you do not recognize these classes.

3. Select a class and display the class graph. Use the explorer to look at what is keeping your selected class in memory.

   Start by looking at the class with the highest number of direct references to objects in your selected class (this is the class at the top of the graph, to the left of your selected class). Click on the class to show classes that refer to it, and then continue to follow the chain of references to understand what is keeping the instances of your selected class in memory.

   For example, System.String is often the largest class; using this class graph you can find out what is keeping strings in memory.

Checking for memory problems

The following steps outline how to carry out a systematic check of memory usage, to determine whether there are any memory problems.

1. Before you start, work out a plan for what functionality or states you want to check. You will need to be methodical about taking snapshots at the appropriate times to check these states, and you will need to have a good understanding of the expected memory usage - so you can identify unexpected memory usage.

2. Start ANTS Memory Profiler and start profiling your application. Take a snapshot of the application in each of the states you want to compare. Depending on what you are trying to discover, it may be sufficient to take a snapshot before and after performing actions, or you may need to take additional snapshots during use, so that you can analyze memory usage throughout.

3. It may help with your analysis if you change the name of the snapshots so that they are easier to recognize.

4. For each state that you want to check on, select a baseline and current snapshot in the Snapshots bar. Alternatively, you can check on memory usage in a single state in isolation: from the Current list, select the snapshot you are interested in; from the Baseline list, select No baseline.

   ♦ Look at each snapshot to understand where memory is being used in each state.
Find out more about how to identify what is using most memory (page 5)

- Compare pairs of snapshots to check for memory leaks.
  Read suggested strategies for finding memory leaks (page 1)
- Use the object filters to check for common indicators of memory leaks.
  Find out more about using filters
- Look for fragmentation problems on the large object heap
  Read tips on identifying fragmentation problems on the large object heap