Overview of C++ Memory Management

There are 4 major memory segments in C++:

1. Global: variables outside stack, heap
2. Code (a.k.a. text): the compiled program
3. Heap: dynamically allocated variables
4. Stack: parameters, automatic and temporary variables

There is no automatic garbage collection: user program must explicitly free dynamic memory.

Global

- Global segment is fixed in size
- Variables are scoped to the entire lifetime of the program
- Includes global scope and static variables, static class and namespace members

Code

- Code segment is fixed
- Code segment is "read-only"

Heap and stack are used in dynamic memory allocation. Memory allocation can be divided into two main categories: frame (or stack) allocations and heap allocations.

Stack (Frame)

- With frame allocation you typically work with the actual memory block itself
- Frame objects are automatically deleted

Allocation on the frame takes its name from the "stack frame" that is set up whenever a function is called. The stack frame is an area of memory that temporarily holds the arguments to the function as well as any variables that are defined local to the function. Frame variables are often called "automatic" variables because the compiler automatically allocates the space for them.

There are two key characteristics of frame allocations. First, when you define a local variable, enough space is allocated on the stack frame to hold the entire variable, even if it is a large array or data structure. Second, frame variables are automatically deleted when they go out of scope:

```
void MyFunction( )
{
```

```
    // Local object created on the stack
    CString strName;
    ...
    // Object goes out of scope and is deleted as function ends
}
```

For local function variables, this scope transition happens when the function exits, but the scope of a frame variable can be smaller than a function if nested braces are used. This automatic deletion of frame variables is very important. In the case of simple primitive types (such as int or byte), arrays, or data structures, the automatic deletion simply reclaims the memory used by the variable. Since the variable has gone out of scope, it cannot be accessed anyway. In the case of C++ objects, however, the process of automatic deletion is a bit more complicated.

When an object is defined as a frame variable, its constructor is automatically invoked at the point where the definition is encountered. When the object goes out of scope, its destructor is automatically invoked before the memory for the object is reclaimed. This automatic construction and destruction can be very handy, but you must be aware of the automatic calls, especially to the destructor.

The key advantage of allocating objects on the frame is that they are automatically deleted. When you allocate your objects on the frame, you don't have to worry about forgotten objects causing memory leaks. (For details on memory leaks, see the article Diagnostics: Detecting Memory Leaks.) A disadvantage of frame allocation is that frame variables cannot be used outside their scope. Another factor in choosing frame allocation versus heap allocation is that for large structures and objects, it is often better to use the heap instead of the stack for storage since stack space is often limited.

Heap

- With heap allocation you are always given a pointer to the memory block
- heap objects must be explicitly deleted by the programmer

The heap is reserved for the memory allocation needs of the program. It is an area apart from the program code and the stack. Typical C programs use the functions malloc and free to allocate and deallocate heap memory. C++ programs use new and delete to allocate and deallocate objects in heap memory.

When you use new and delete instead of malloc and free, you are able to take advantage of the class library's memory-management debugging enhancements, which can be useful in detecting memory leaks. Note that

the total size of objects allocated on the heap is limited only by your system's available virtual memory.

Examples:

Allocation of an Array of Bytes

To allocate an array of bytes on the frame:

Define the array as shown by the following code. The array is automatically deleted and its memory reclaimed when the array variable exits its scope.

```
{
    const int BUFF_SIZE = 128;
    // Allocate on the frame
    char myCharArray[BUFF_SIZE];
    int myIntArray[BUFF_SIZE];
    // Reclaimed when exiting scope
}
```

To allocate an array of bytes (or any primitive data type) on the heap:

Use the new operator with the array syntax shown in this example:

```
const int BUFF_SIZE = 128;
// Allocate on the heap
char* myCharArray = new char[BUFF_SIZE];
int* myIntArray = new int[BUFF_SIZE];
```

To deallocate the arrays from the heap:

Use the delete operator as follows:

```
delete [] myCharArray;
delete [] myIntArray;
```

Allocation of a Data Structure

To allocate a data structure on the frame:

Define the structure variable as follows:

struct MyStructType { int topScore;};

void SomeFunc(void)

```
{
    // Frame allocation
    MyStructType myStruct;
    // Use the struct
    myStruct.topScore = 297;
    // Reclaimed when exiting scope
}
```

The memory occupied by the structure is reclaimed when it exits its scope.

To allocate data structures on the heap:

Use new to allocate data structures on the heap and delete to deallocate them, as shown by the following examples:

```
// Heap allocation
MyStructType* myStruct = new MyStructType;
// Use the struct through the pointer ...
myStruct->topScore = 297;
delete myStruct;
```

Allocation of an Object

To allocate an object on the frame:

Declare the object as follows:

```
{
CPerson myPerson; // Automatic constructor call here
myPerson.SomeMemberFunction(); // Use the object
}
```

The destructor for the object is automatically invoked when the object exits its scope.

To allocate an object on the heap:

Use the new operator, which returns a pointer to the object, to allocate objects on the heap. Use the delete operator to delete them.

The following heap and frame examples assume that the CPerson constructor takes no arguments.

```
// Automatic constructor call here
CPerson* myPerson = new CPerson;
myPerson->SomeMemberFunction(); // Use the object
delete myPerson; // Destructor invoked during delete
```

If the argument for the CPerson constructor is a pointer to char, the statement for frame allocation is:

```
CPerson myPerson( "Joe Smith" );
```

The statement for heap allocation is:

```
CPerson* MyPerson = new CPerson( "Joe Smith" );
```