Memory Management in C++

Chapter 5

Objectives

- **Explain use of static, automatic (stack) and heap memory.**
- **Use *new* and *delete* to manage memory.**
- **Provide constructors and destructors to support dynamic objects.**
- **Discuss techniques for handling memory allocation errors.**
- **Hide details of memory management in a class.**
- **Implement a dynamic string class.**
- **Gain experience through code walk-throughs and lab exercises.**

  - The example programs are in the **chapter directory.**
  - Labs located in Labs/Lab5

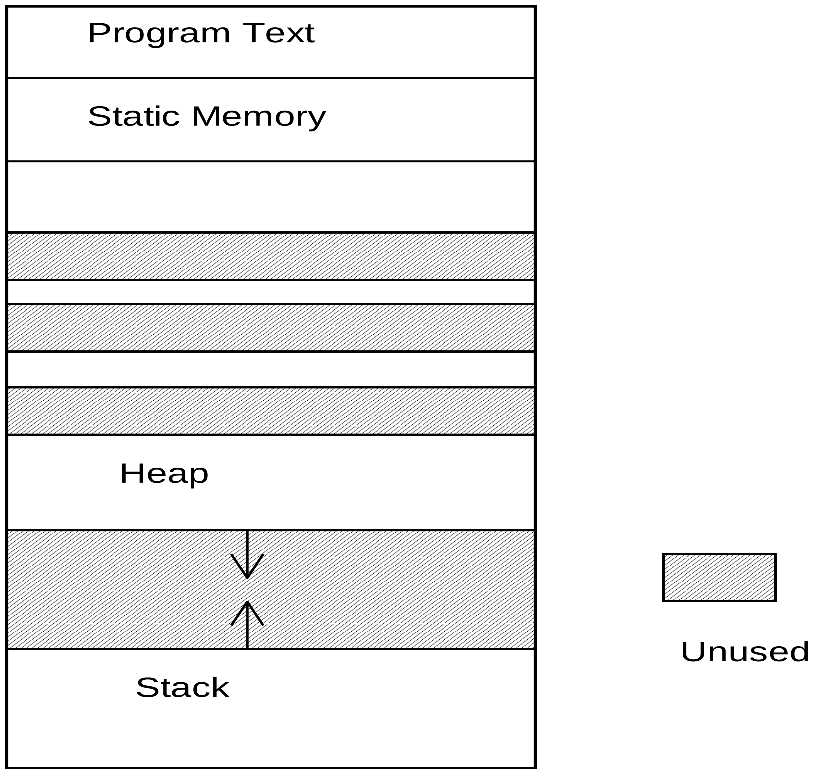
Why Is Memory Management Important?

- **OOP designs can transparently use a lot of memory:**
  - Declaring an object
  - Assigning an object
  - Call-by-value argument passing (copy)
  - Inheritance
- **Using dynamic memory can be more efficient in memory usage and offer more flexibility.**
- **C++ enables a class to hide many details of memory management from users of the class.**

Choices for an Object's Memory

- **Static:**
    - Defined outside any function, in main, or with keyword **static**.
    - Lifetime is duration of program.
- **Automatic:**
    - Local variable inside a function.
    - Comes into being when function is entered and ceases to exist when function is exited.
    - Typically stored on program's stack.
- **Dynamic:**
    - Created at run-time by explicit statement.
    - Exists until explicitly destroyed.
    - Stored in the heap or free store.

Typical Memory Layout



Free Store Allocation

```
char     *buf;              // pointer to buffer
int size;                   // size of buffer
.
size = 1000;
buf = new char [size];      // allocate storage

for (int i=0; i<size; ++i)
   buf[i] = 0;              // initialize
.
delete [] buf;              // when no longer needed
```

new Operator

- **Allocates memory for an object or array of objects of type-name from the free store**
- **Returns a suitably typed, nonzero pointer to the object.**

```
T    *p;              // T is any type, built-in or user defined
p = new T;            // single object
p = new T[10];        // array of 10 objects
p = new T(a,b);       // passes arguments to constructor
```

Allocation Errors

- It is important for your program to allow for possible failure of memory allocation by *new*.
- You do that by providing a *try* block for the allocation and a *catch* block for error handling.
  - An allocation error will throw an exception of type **bad_alloc**.
  - A "new handler" can also be provided.

Demo

- The folder BadAllocation contains a file BadAllocation.cpp that demonstrates memory allocation errors.
- Create a new project and add the file to the project.
- Build and run the program.

new vs. malloc

- **Allocate an array of 100 long integers:**
- **Using *new*:**
  - long *array = new long[100];
- **Using *malloc*:**
  - **malloc** must be told number of bytes to allocate.
  - **malloc** returns a void pointer which must be cast to the appropriate type (in C++).
  - long *array;
  - array = (long *)malloc (sizeof(long)*100);

Advantages of *new*

- Don't need to compute number of bytes.
- Don't need to type cast.
- Applies to user defined types.
- In allocating and deallocating an array of user defined objects, constructors and destructors get invoked properly.

delete Operator

- ***delete*** **invokes a destructor when deleting an object of a user defined type.**
- **Deleting a null pointer is a no-op, but deleting a pointer twice is a serious error. Why?**
  - It is a good practice to set a pointer to null (or zero) after deleting it.
- **The reason for the [] in deleting an array of objects is to cause the destructor to be invoked for each element of an array of a user defined type.**

```
delete p;                   // deletes object pointed to by p
p = NULL;                   // good practice after delete
delete [] a;                // deletes each element of array pointed to by a
a = nullptr;                // nullptr can be used at all places where NULL is expected
```

Demo

- **The folder NewDelete has an application that demonstrates use of new, delete, and nullptr.**

Destructor (Review)

- **Name is class name preceded by a tilde (~).**
- **Automatically invoked when an object goes out of scope.**
- **Explicitly invoked when *delete* is applied to a pointer to a class object.**
- **Allows class to control object destruction.**

Hiding Memory Management

- **A class can hide details of memory management:**
  - Declare a pointer in private section.
  - Member functions manage pointer and storage.
  - Users of the class are freed of details of memory management.
- **Consider null terminated strings in C:**
  - User has to carefully manage storage for the characters, including terminating null byte.
- **We will build a examine version of our String class:**
  - Storage allocation is handled by the class transparently to the user.
  - ANSI C++ provides a standard library String class.

Demo

- **The folder String contains a partially complete example of the String class.**
- **Notice the *PrintStrings* function uses pointers as arguments.**
- **This is to avoid the “hidden constructor” problem.**

Summary

- **Three kinds of memory for program objects are *static*, *automatic* and *heap*.**
- ***new* and *delete* enable the managing of heap memory.**
- **Dynamic objects belonging to user classes can be created and destroyed.**
- **A programmer can provide for dynamic objects by including appropriate constructors and destructors with class definition.**
- **You should always check the value returned by *new*, unless you provide a new handler function.**
- **A class can hide details of memory management.**
- **A dynamic string class in C++ can make string handling easier and more robust than in C.**