Dynamic Memory

- Describe the two kinds of system memory available
- Allocate and deallocate memory dynamically
- Identify common issues associated with dynamic memory

> "Avoid allocating and deallocating in different modules" **Sutter, Alexandrescu, 2005.**

Various components of object-oriented programs can be reused by other applications. An important aspect of object designing is including flexibility in their memory requirements to enhance reusability. Objects are more reusable by different clients if they account for their own memory needs internally. These memory requirements may depend on problem size, which might not even be known approximately at compile-time. Programming languages address this aspect of reusability by supporting dynamic memory allocation.

This chapter introduces the C++ syntax for allocating and deallocating memory dynamically. The chapter entitled Classes and Resources augments this material with the details required to code classes that manage dynamic memory internally.

Static and Dynamic Memory

The memory accessible by a C++ program throughout its execution consists of static and dynamic components. After the user starts an application, the operating system loads its executable into RAM and transfers control to that executable's entry point (the `main()` function). The loaded executable only includes the memory allocated at compile time. During execution, the application may request more memory from the operating system. The system satisfies such requests by allocating more memory in RAM. After the application terminates and returns control to the operating system, the system recovers all of the memory that the application has used.

Static Memory

The memory that the operating system allocates for the application at load time is called *static memory*. Static memory includes the memory allocated for program instructions and program data. The compiler determines the amount of static memory that each translation unit requires. The linker determines the amount of static memory that the entire application requires.

The application's variables and objects share static memory amongst themselves. When a variable or object goes out of scope its memory becomes available for newly defined variables or objects. The lifetime of each local variable and object concludes at the closing brace of the code block within which it has been defined:

```
// lifetime of a local variable or object

for (int i = 0; i < 10; i++) {
    double x = 0;      // lifetime of x starts here
    // ...
}                      // lifetime of x ends here

for (int i = 0; i < 10; i++) {
    double y = 4;      // lifetime of y starts here
    // ...
}                      // lifetime of y ends here
```

Since the variable `x` goes out of scope before the variable `y` is declared, the two variables may occupy the same memory location. This system of sharing memory amongst local variables and objects ensures that each application minimizes its use of RAM.

Static memory requirements are determined at compile-link time and do not change during execution. This memory is fast, fixed in its amount and allocated at load time.

Dynamic Memory

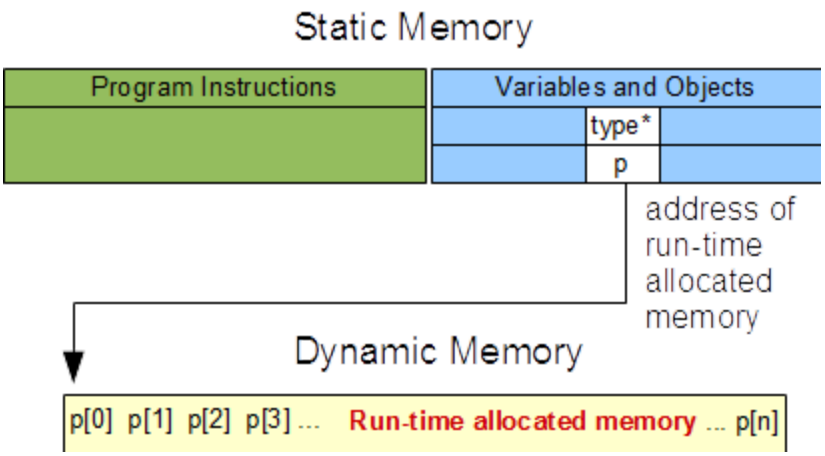
The memory that an application obtains from the operating system during execution is called `dynamic memory.`

Dynamic memory is distinct from the static memory. While the operating system allocates static memory for an application at load time, the system reserves dynamic memory, allocates it and deallocates it at run-time.

Scope Considerations

To keep track of an application's dynamic memory, we store the address of each allocated region in a pointer variable. We allocate memory for this pointer itself in static memory. This pointer variable must remain in scope as long as we need access to the data in the allocated region of dynamic memory.

Consider allocating dynamic memory for an array of n elements. We store the array's address in a pointer, `p`, in static memory as illustrated below. We allocate memory for the elements of the array dynamically and store the data in those elements starting at address `p`.



Lifetime

The lifetime of any dynamically allocated memory ends when the pointer holding its address goes out of scope. The application must explicitly deallocate the allocated region of dynamic memory within this scope. If the application neglects to deallocate the allocated region, that memory becomes inaccessible and irrecoverable until the application returns control to the operating system.

Unlike variables and objects that have been allocated in static memory, those in dynamic memory do not automatically go of out scope at the closing brace of the code block within which they were defined. We must manage their deallocation explicitly ourselves.

Dynamic Allocation

The keyword `new` followed by `[n]` allocates contiguous memory dynamically for an array of `n` elements and returns the address of the array's first element.

Dynamic allocation of arrays takes the form

```
pointer = new Type[size];
```

where `Type` is the type of the array's elements.

For example, to allocate dynamic memory for an array of `n` `Student`s, we write

```
int n;                      // the number of students
Student* student = nullptr; // the address of the dynamic array

cout << "How many students in this section? ";
cin >> n;

student = new Student[n];   // allocates dynamic memory
```

The `nullptr` keyword identifies the address pointed to as the null address. This keyword is an implementation constant. Initialization to `nullptr` ensures that `student` is not pointing to any valid dereferencable address. The size of the array is a run-time variable and not an integer constant or constant expression. Note that the size of an array allocated in static memory must be an integer constant or constant expression.

Dynamic Deallocation

The keyword `delete` followed by `[]` and the address of a dynamically allocated region of memory deallocates the memory that the corresponding `new[]` operator had allocated.

Dynamic deallocation of arrays takes the form

```
delete [] pointer;
```

where `pointer` holds the address of the dynamically allocated array.

For example, to deallocate the memory allocated for the array of `n` `Student`s above, we write

```
delete [] student;
student = nullptr;  // optional
```

The `nullptr` assignment ensures that student now holds the null address. This optional assignment eliminates the possibility of deleting the original address a second time, which is a serious run-time error. Deleting the `nullptr` address has no effect.

Note that omitting the brackets in a deallocation expression deallocates the first element of the array, leaving the other elements inaccessible.

Deallocation does not return dynamic memory to the operating system. The deallocated memory remains available for subsequent dynamic allocations. The operating system only reclaims all of the dynamically

allocated memory once the application has returned control to the system.

A Complete Example

Consider a simple program in which the user enters a number and the program allocates memory for that number of `Student`s. The user then enters data for each student. The program displays the data stored, deallocates the memory and terminates:

```
// Dynamic Memory Allocation
// dynamic.cpp

#include <iostream>
#include <cstring>
using namespace std;

struct Student {
    int no;
    float grade[2];
};

int main( ) {
    int n;
    Student* student = nullptr;

    cout << "Enter the number of students : ";
    cin >> n;
    student = new Student[n];

    for (int i = 0; i < n; i++) {
        cout << "Student Number: ";
        cin  >> student[i].no;
        cout << "Student Grade 1: ";
        cin  >> student[i].grade[0];
        cout << "Student Grade 2: ";
        cin  >> student[i].grade[1];
    }

    for (int i = 0; i < n; i++) {
        cout << student[i].no << ": "
        << student[i].grade[0] << ", " << student[i].grade[1]
        << endl;
    }

    delete [] student;
```

```
    student = nullptr;
}
```

Memory Issues

Issues regarding dynamic memory allocation and deallocation include:

- Memory leaks
- Insufficient memory

Memory Leaks

Memory leaks are one of the most important bugs in object-oriented programming. A memory leak occurs if an application loses the address of dynamically allocated memory before that memory has been deallocated. This may occur if:

- The pointer to dynamic memory goes out of scope before the application deallocates that memory
- The pointer to dynamic memory changes its value before the application deallocates the memory starting at the address stored in that pointer

Memory leaks are difficult to find because they often do not halt execution immediately. We might only become aware of their existence indirectly through subsequently incorrect results or progressively slower execution.

Insufficient Memory

On small platforms where memory is severely limited, a realistic possibility exists that the operating system might not be able to provide the amount of dynamic memory requested. If the operating system cannot dynamically allocate the requested memory, the application may throw an exception and stop executing. The topic of exception handling is beyond the scope of these notes.

Single Instances (Optional)

Although dynamic memory is often allocated for data structures like arrays, we can also allocate dynamic memory for single instances of any type. The allocation and deallocation syntax is similar to that for arrays. We simply remove the brackets.

Allocation

The keyword new without brackets allocates dynamic memory for a single variable or object.

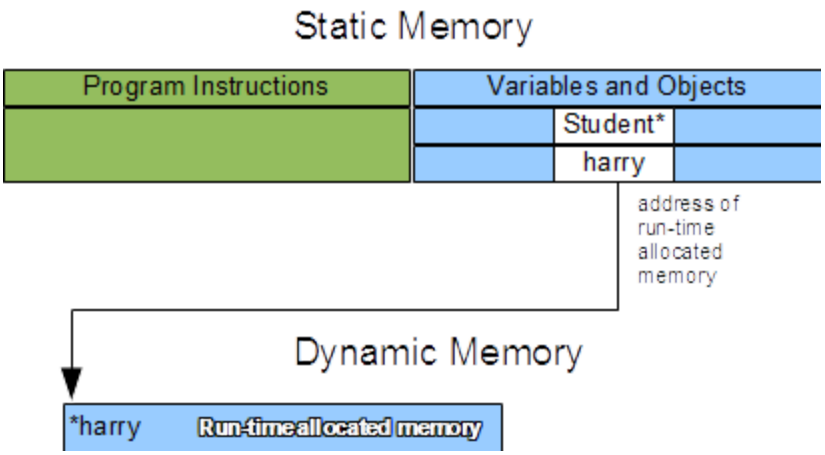
A dynamic allocation statement takes the form

```
pointer = new Type;
```

For example, to store one instance of a Student in dynamic memory, we write

```
Student* harry = nullptr;   // a pointer in static memory
harry = new Student;        // points to a Student in dynamic memory

// we must deallocate harry later!
```



Deallocation

The keyword `delete` without brackets deallocates dynamic memory at the address specified.

A dynamic deallocation statement takes the form

```
delete pointer;
```

`delete` takes the address that was returned by the `new` operator.

For example, to deallocate the memory for `harry`, we write

```
delete harry;
harry = nullptr;  // good programming style
```

Summary

- The memory available to an application at run-time consists of static memory and dynamic memory
- Static memory lasts the lifetime of the application
- The linker determines the amount of static memory used by the application
- The operating system provides dynamic memory to an application at run-time upon request
- The keyword `new[]` allocates a contiguous region of dynamic memory and returns its starting address
- We store the address of dynamically allocated memory in static memory
- `delete[]` deallocates contiguous memory starting at the specified address
- Allocated memory must be deallocated within the scope of the pointer that holds its address