Memory Allocation/Deallocation in C++

References:

The material in this handout is collected from the following references:

- Sections $12.1.2$ and 12.2 of the text book C++ Primer.
- Section 11.2 of C++ Programming Language.
- Items 3 and 5 of Effective C++.

Synopsis

This is a compact organization of the most important details of memory allocation and deallocation in C++ programs:

- C++ programs can dynamically allocate memory from a region of program memory called the *heap* using C standard library functions `std::malloc`, `std::calloc`, and `std::realloc`. The memory allocated by these functions is deallocated using standard C library function `std::free`. These functions are declared in `<cstdlib>`.
- C++ provides better alternatives to allocate and deallocate memory.
- The C++ way of allocating memory *for an object* is to use operator `new`. The memory is allocated from a region of program memory called *free store*. Operator `delete` is the counterpart of operator `new` and is used to return the previously allocated memory back to the free store.
- The C++ way of allocating memory for an array of objects is to use operator `new[]`. Just like operator `new`, operator `new[]` allocates memory from a region of program memory called *free store*. Operator `delete[]` is the counterpart of operator `new[]` and is used to return the previously allocated memory back to the free store.
- Why is the C++ way of allocating/deallocating memory better than the C way? The reason is that C++ operators do everything that standard C library functions do plus more. For user-defined types, operator `new` first allocates memory for an object and then calls a constructor (a function defined by the type) to initialize the object. To return the memory back to the free store operator `delete` is called. `delete` will first call a destructor (a function defined by the type) to perform cleanup actions and then return the memory back to free store. For user-defined types, operator `new[]` first allocates memory for an array of objects and then initializes each object in the array by calling a constructor. To return the memory back to the free store, operator `delete[]` is called. `delete[]` will first call a destructor for each object in the array and then return the memory allocated by `new[]` back to free store.
- Although the way of allocating memory has changed with C++, the use of that memory is identical to C.
- Once more: You use the dynamically-allocated memory *exactly* as you would use it in C. There is absolutely no difference in how you *use* the memory.

C++ syntax	Explanation
`new T`	Allocates and default-initializes a new object of type `T` on the free store and returns a pointer to the object.
`new T (args)` `new T {args}`	Same as above except that object is initialized using `args`.
`delete p`	Destroys object `*p` points and frees memory used to hold `*p`. `p` must point at an object that was dynamically allocated.
`new T [n]`	Allocates and default-initializes array of `n` new objects of type `T` on the free store. Returns pointer to initial element in array.
`new T [n]` `{initializers}`	Same as above except that array elements are initialized with comma-separated initializers.
`delete[] p`	Destroys each object in array to which `p` points and then frees the memory used to hold the array. `p` must point to initial element of an array that was dynamically allocated.

Operators `new`, `delete`, `new[]`, and `delete[]` are discussed in greater detail in the following sections.

Dynamic storage duration

An object's *storage duration* determines the memory management associated with the object. There are three types of memory management for objects that occur inside a typical C++ program: automatic storage duration, static storage duration, and dynamic storage duration. Before discussing dynamic storage duration, let's review the other two types of memory management.

Automatic storage duration is associated with *local* or *internal* variables that are defined at function scope. A local variable occupies *stack memory* that the system allocates when it encounters the variable's definition during execution of the function (in whose scope the variable is defined). The system automatically deallocates that memory at the end of the block that contains the definition. Once a variable has been deallocated, any pointers or references to it become invalid. It is the programmer's responsibility to avoid using such invalid pointers or references. For example,

```cpp
// this function deliberately yields an invalid pointer
// it is intended as a negative example - don't do this
int* invalid_pointer() {
  int x;
  // do stuff with x
  return &x; // instant disaster
}

// this function deliberately yields an invalid reference
// it is intended as a negative example - don't do this
int& invalid_reference() {
  int x;
  // do stuff with x
  return x; // instant disaster
}
```

The first function returns the address of the local variable `x`. Unfortunately, when the function returns, doing so ends execution of the block that contains the definition of `x`, which deallocates `x`. The pointer that `&x` created is now invalid, but the function has returned the address anyway. Similarly, the second function returns a reference of a local variable. What happens when either function returns is anybody's guess. In particular, C++ implementations are not required to diagnose the error - you get what you get.

A variable has *static storage duration* if it is a *global* or *external* variable. Such a variable is defined at file scope or because it is an internal variable defined using `static` storage specifier. Variables with static storage duration are given storage in the *data* (for variables with initializers) or *bss* region (for uninitialized variables) of program memory and are alive throughout the duration of program execution. We can return the address of statically allocated variables:

```
// it is legitimate to return address of statically allocated variable
int* pointer_to_static() {
  static int x;
  // do stuff with x
  return &x;
}

// this function is completely legitimate
int& reference_to_static() {
  static int x;
  // do stuff with x
  return x;
}
```

By saying that `x` is `static`, we're saying that we want to allocate `x` once, and only once, at some point before the first time that `pointer_to_static` or `reference_to_static` is ever called, and that we don't want to deallocate it as long as the program runs. There is nothing wrong with returning the address of a `static` variable; the pointer will be valid as long as the program runs.

> ***The sizes of data and bss sections of your program can be determined with the*** `size` ***command in Linux.***

However, static allocation has the potential disadvantage that every call to `pointer_to_static` will return a pointer to the same object! Suppose we want to define a function such that each time we call it, we get a pointer to a brand new object, which stays around until we decide that we no longer want it. To do so, objects with dynamic storage duration are required and we use dynamic allocation and deallocation, which we request by using `new` and `delete` keywords.

Allocating/deallocating an object

If `T` is the type of an object, `new T` is an expression that allocates an object of type `T` in a region of program memory called the *free store*, *default-initializes* (this is initialization that happens by default when an initial value is not specified) the object, and yields a pointer to this object. As an example:

```
// this int object is uninitialized - just like when an
// automatic int variable is defined without an initializer
int *pi { new int };
```

will allocate an unnamed object of type `int` on the free store whose value is unspecified.

> ***The C++ ISO standard uses the term free store to describe the heap.***

It is possible to give a specific value to use when initializing the object by executing `T(args)` or `T{args}`. The object allocated on the free store is said to have *dynamic storage duration* because it can stay around until the program either ends or executes `delete pi` (whichever happens first), where `pi` is a (copy of) the pointer returned by `new`. In order to delete a pointer, the pointer must point to an object that was allocated by `new`, or be equal to `nullptr`. Deleting a `nullptr` has no effect.

As an example,

```
int *pi { new int{35} };
```

will allocate an object of type `int` on the free store, initialize the object to `35`, and cause `pi` to point to that object. We can affect the value of the object by executing statements such as

```
++*pi; // *pi is now 36
```

after which the object has value `36`. When we're done with the object, we can execute

```
delete pi;
```

after which the space occupied by `*pi` is freed and `pi` becomes an invalid pointer, with a value that we can no longer use until we've assigned a new value to it.

As another example, we might write a function that allocates an `int` object, initializes it, and returns a pointer to it:

```
int* pointer_to_dynamic() {
  return new int{0};
}
```

which imposes on its caller the responsibility of freeing the object at an appropriate time.

In addition to built-in types, we can allocate memory for user-defined types:

```
struct Sprite { // on-screen graphic
  double x, y;
  int weight, level;
  std::string name;
};

void f2() {
  Sprite s1; // allocated on the stack (handled by compiler)
  // dynamically allocate object on free store (handled by programmer)
  Sprite *s2 {new Sprite}; // members of object *s2 are not initialized
  // members of object *s3 are initialized
  Sprite *s3 {new Sprite {11.1, 22.2, 33, 44, "cpp"}};

  s1.level  = 1; // members of s1 were not initialized
  s2->level = (*s3).level; // members of s3 were initialized

  // other stuff ...

```

```
  delete s2; // deallocate object *s2 (must be done by programmer)
  delete s3; // deallocate object *s3 (must be done by programmer)
} // s1 goes out of scope and the memory is automatically deallocated
```

Allocating and deallocating an array

If `T` is a type and `n` is a non-negative integral value, `new T[n]` allocates an array of `n` objects of type `T` and returns a pointer of type `T*` to the initial element of the array. Each object is default-initialized, meaning that if `T` is a built-in type and the array is allocated at local scope, then the objects are uninitialized. If `T` is a class type (that is, an user-defined type), then each element is initialized by running its default constructor.

When `T` is a class type, there are two important implications of this initialization process: First, if the class doesn't allow default-initialization, then the compiler will reject the program. Second, each of the `n` elements in the array is initialized, which can be a substantial execution overhead. The standard library has a type `vector<T>` that provides a more flexible mechanism for dynamically allocating arrays. It is often preferable to use that mechanism, rather than `new[]`, when dynamically allocating an array.

Consider this example:

```
// allocate space for array of 10 chars and 10 ints
char *p1 = new char[10]; // array of 10 chars that are default-initialized
int *p2 = new int[10](); // array of 10 ints that are value-initialized

// use p1, p2 ...

// release the memory (programmer)
delete [] p1; // C++ only (array delete)
delete [] p2; // C++ only (array delete)
```

Initialization in which built-in types are initialized to zero and class types are initialized by their default constructors. Note the use of `delete[]` in this example. The brackets are necessary to tell the system to deallocate an entire array, rather than a single element. An array allocated by `new[]` stays around until the program ends or until the program executes `delete[] p1` and `delete[] p2`, where `p1` and `p2` are (copies of) the pointer that `new[]` yielded. Before deallocating the array, the system destroys each element, in reverse order.

As an example, here is a function that takes a pointer to the initial character of a null-terminated character array such as a string literal, copies all the characters in the array (including the null character at the end) into a newly allocated array, and returns a pointer to the initial element of the new array:

```
#include <cstring>
#include <algorithm>

char* duplicate_chars(char const *ps) {
  // allocate enough space; remember to add one for the null
  size_t length {std::strlen(ps) + 1};
  char *result { new char [length] };
  // copy into our newly allocated space and return pointer to 1st element
  std::copy(ps, ps+length, result);
  return result;
}
```

Recall that `std::strlen` returns the number of characters in a null-terminated array, excluding the null character at the end. We therefore add `1` to the result of `std::strlen` to account for the null, and allocate that many characters. Because pointers are iterators, we can use the `std::copy` algorithm to copy characters from the array denoted by `ps` into the array denoted by `result`. Because `length` includes the null character at the end of the array, the call to `copy` copies that character as well as the ones before it.

As before, this function imposes on its caller the obligation to free the memory that it allocated. In general, finding an opportune time to free dynamically allocated memory is far from easy.

The following code fragment illustrates the use of dynamically allocated arrays to read `double`s values from a data text file. The first value in the file representing the number of values in the file:

```
std::ifstream ifs("some_array.dat");
int size;
ifs >> size; // get number of values to read from user
double *pd { new double [size] }; // allocate array on free store
for (int i {0}; i < size; ++i) { // read values from file
  ifs >> pd[i];
}
```

Again, it is the programmer's responsibility to release the memory when not needed anymore. This is done by

```
delete [] pd;
```

In addition to allocating array of built-in types, we can also allocate array of user-defined types:

```
void f3() {
  // allocated array of 10 Sprites on the stack (handled by compiler)
  Sprite s1[10];
  // allocate array of 10 Sprites on free store (handled by programmer)
  Sprite *s2 {new Sprite[10]}; // elements are un-initialized

  s1[0].level = 1; // s1[0] is a Sprite struct
  s2[0].level = 2; // s2[0] is a Sprite struct
  s2->level = 4; // does this work?

  // other stuff ...

  // deallocate memory for 10 Sprite objects
  delete [] s2; // (must be handled by programmer)
} // s1 goes out of scope and its memory is released automatically
```

We can do more. Not only can you allocate an array of objects of user-defined types, you can also allocate memory for pointer members of these objects:

```
struct Sprite2 { // on-screen graphic, new version
  double x, y;
  int weight, level;
  char *name; // replaced type from std::string to char*
};

void f4(int N) {
  Sprite2 *s { new Sprite2 [N] }; // array is on free store
```

```
  for (int i {1}; i < N; ++i) {
    s[i]->name = new char [20]; // char array is on free store
  }

  // use these N elements to do something useful ...

  // release memory pointed to by name member of each object
  for (int i {1}; i < N; ++i) {
    delete [] s[i]->name;
  }
  delete [] s; // now, deallocate N objects
}
```

Memory exhaustion

Free store is not exhausted when your system runs out of physical main memory. On systems with virtual memory (common in most modern machines), the program will consume a lot of disk space for free store and will take a long time doing so. What happens when `new` and `new[]` can find no contiguous store to allocate? By default, they will throw an exception of type `std::bad_alloc`. If an exception is thrown and not caught anywhere, the program terminates abnormally. The topic of exceptions will be covered at a later point. Here's an example to try out (may take a while):

```
#include <vector>
#include <exception>

void f() { // call this function from main
  std::vector<char*> v;
  try {
    for (;;) {
      char *p = new char[10'000'000]; // acquire some memory
      v.push_back(p); // make sure new memory is referenced
      p[0] = 'x';
    }
  } catch (std::bad_alloc& b) {
    std::cerr << b.what() << " Free store is exhausted\n";
    std::terminate();
  }
}
```

In programs where exceptions must be avoided (the list includes embedded systems and real-time applications), we can use `nothrow` versions of `new` and `delete`. This is similar to `NULL` pointer returned by `malloc` when contiguous memory is exhausted on the heap. For example:

```
void f(int n) {
  int *p = new (std::nothrow) int [n]; // allocate n ints on free store
  if (p == nullptr) {
    std::cerr << "Free store is exhausted\n";
    std::terminate();
  }
  // do stuff with allocated memory ...
  delete [] p; // then, deallocate memory
}
```

Caveat: Don't use `malloc` and `free`!!!

The problem with `malloc`,`calloc`, and `realloc` is simple: they don't know about constructors. Likewise, the problem with `free` is that it doesn't know about destructors. The reason is straightforward: `malloc`, `calloc`, `realloc`, and `free` are C library functions while constructors and destructors are C++ concepts. Consider the following two ways to get space for an array of 10 `string` objects, one using `malloc`, the other using `new`:

```
std::string *str_arr1 =
  static_cast<std::string*>(std::malloc(10 * sizeof(std::string)));
std::string *str_arr2 = new std::string[10];
```

Here `str_arr1` points to enough memory for 10 `string` objects, but no objects have been constructed in that memory. That is, the internal data members of each `string` object are uninitialized because no constructor has been invoked for that object. Furthermore, without jumping through some rather obscure linguistic hoops, you have no way to initialize the objects in the array. In other words, `str_arr1` is pretty useless. In contrast, `str_arr2` points to an array of 10 fully constructed `string` objects, each of which can safely be used in any operation taking a `string`.

Nonetheless, let's suppose you magically managed to initialize the objects in `str_arr1` array. Later on in your program, then, you'd expect to do this:

```
std::free(str_arr1);
std::delete [] str_arr2;
```

The call to `free` will release the memory pointed to by `str_arr1`, but no destructors will be called on the `string` objects in that memory. If the `string` objects themselves allocated memory, as `string` objects are wont to do, all the memory they allocated will be lost. On the other hand, when `delete[]` is called on `str_arr2`, a destructor is called for each object in the array before any memory is released.

> ***Requirement for this module: Because `new`/`delete` and `new[]`/`delete[]` interact properly with constructors and destructors, don't use `malloc` and `free`!!!***

Caveat: Don't mix `new` and `delete` with `malloc` and `free`!!!

Mixing `new` and `delete` with `malloc` and `free` is a bad idea. When you try to call `free` on a pointer you got from `new` or call `delete` on a pointer you got from `malloc`, the results are *undefined*.

There are lots of C libraries based on `malloc` and `free` containing code that is very much worth reusing. When taking advantage of such a library, it's likely you'll end up with the responsibility for `free`ing memory `malloc`ed by the library and/or `malloc`ing memory the library itself will `free`. That's fine. There's nothing wrong with calling `malloc` and `free` inside a C++ program as long as you make sure the pointers you get from `malloc` always get freed by `free` and the pointers you get from `new` eventually find their way to `delete`. The problems start when you get sloppy and try to mix `new` with `free` or `malloc` with `delete`.

> ***Programming tip: Given that `malloc` and `free` are ignorant of constructors and destructors and that mixing `malloc`/`free` with `new`/`delete` can cause undefined behavior, stick to just using `new`s and `delete`s and their array counterparts.***

Caveat: Use the same form in corresponding uses of `new` and `delete`

What's wrong with this code fragment?

```
std::string *str_arr = new std::string[100];

// do something with str_arr

delete str_arr;
```

Everything here appears to be in order - the use of `new[]` is matched with a use of `delete` — but something is still quite wrong: this fragment's behavior is undefined. At the very least, 99 of the 100 `string` objects pointed to by `str_arr` are unlikely to be properly destroyed, because their destructors will probably never be called.

When you use `new`, two things happen. First, memory is allocated. Second, one or more constructors are called for that memory. When you use `delete`, two other things happen: one or more destructors are called for the memory, then the memory is deallocated. The big question for `delete` is this: *how many* objects reside in the memory being deleted? The answer to that determines how many destructors must be called.

Actually, the question is simpler: does the pointer being deleted point to a single object or to an array of objects? The only way for `delete` to know is for you to tell it. If you don't use brackets in your use of `delete`, `delete` assumes a single object is pointed to. Otherwise, it assumes that an array is pointed to:

```
std::string *str_ptr1 = new std::string;
std::string *str_ptr2 = new std::string[100];

// do something with str_ptr1 and str_ptr2

delete str_ptr1;    // delete an object
delete [] str_ptr2; // delete an array of objects
```

What would happen if you used the `[]` form on `str_ptr1`? The result is undefined. What would happen if you didn't use the `[]` form on `str_ptr2`? Well, that's undefined too. Furthermore, it's undefined even for built-in types like `int`s, even though such types lack destructors. The rule, then, is simple:

> ***Programming tip: If you use `[]` when you call `new`, you must use `[]` when you call `delete`. If you don't use `[]` when you call `new`, don't use `[]` when you call `delete`.***

This is a particularly important rule to bear in mind when you are writing a class containing a pointer data member and also offering multiple constructors, because then you've got to be careful to use the *same form* of `new` in all the constructors to initialize the pointer member. If you don't, how will you know what form of `delete` to use in your destructor?

Application example: Dynamically resizing an array

If you've an existing dynamically allocated array, and you want to add more elements, you can't simply append new elements to the old ones. Remember that arrays are stored in a contiguous memory block, and you never know whether or not the memory immediately after the array is already allocated for something else. For that reason, the resizing process takes a few more steps. Here is an example using `int` values. Let's say the original dynamically allocated array consists of `n` `int` elements:

```
int *pi {new int[n]};
```

Suppose you want to resize this array so that there is room for n more values (presumably because the old one is full). There are four main steps.

1. Create an entirely new array of the appropriate type and of the new size. You'll need another pointer for this:

   ```
   int *temp {new int[n + n]};
   ```

2. Copy the data from the old array into the new array:

   ```
   for (int i {0}; i < n; i++) {
     temp[i] = pi[i];
   }
   ```

3. Delete the old array -- you don't need it anymore! (Do as your Mom says, and take out the garbage!)

   ```
   delete [] pi;  // this deletes the array pointed to by pi
   ```

4. Change the pointer. You still want the array to be called `pi`, so assign `pi` the new address in `temp`:

   ```
   pi = temp;
   ```

That's it! The array is now n elements larger than the previous one, and it has the same data in it that the original one had. But, now it has room for n more values.