CONTROLLING MEMORY ALLOCATION

Plan For Today

- Overloading `new` and `delete`
- `operator new` and `operator delete` interface
- Placement `operator new` and operator delete expressions
- Designing and implementing memory managers

new Operator

- What is happening when you write code like this

```
string *ps = new string {"memory"};
```

You’re using new operator and like other C++ operators you can’t change its meaning nor behavior

What new operator does is threefold:

1) Allocates enough memory to hold a string object
2) It calls a ctor to initialize that string object
3) Returns a pointer to the string object

new Operator

- new operator calls global function operator new declared in <new> to *only* perform requisite memory allocation
- Given `string *ps = new string {"memory"};` we've

```
// get raw memory for string object
void *raw_memory = operator new(sizeof(string));
// initialize the object in the memory
call string::string("memory")
// make ps point to new memory
string *ps = reinterpret_cast<string*>(raw_memory);
```

operator new

- Plain old function `operator new` has following signature:

```
void* operator new (std::size_t);
```

Function `operator new` takes number of bytes to be allocated as a parameter

new[] Operator

- What is happening when you write code like this

```
string *ps = new string [3] {"a", "b", "c"};
```

You're using new[] operator!!!

What new[] operator does is threefold:

1) Allocates enough memory to hold three `string` objects
2) Calls a ctor to initialize each `string` object
3) Returns a pointer to first `string` object

operator new[]

- new[] operator calls function operator new[] declared in <new> to *only* perform requisite memory allocation for array elements
- Given

```
string *ps = new string [3] {"a", "b", "c"};
```

we've

```
// get raw memory for all array objects
void *memory = operator new[](sizeof(string)*3);
// initialize each string objects
call string::string("a")
call string::string("b")
call string::string("c")
// make ps point to new memory
string *ps = reinterpret_cast<string*>(memory);
```

operator new[]

- Plain old function `operator new[]` has following signature:

```
void* operator new[] (std::size_t);
```

Function `operator  new` takes number of bytes to be allocated as a parameter

delete Operator

- What happens when you write code like this

```
std::string *ps = new std::string {"memory"};
... // use ps
delete ps;
```

You're using `delete` operator and like other C++ operators you can't change its meaning nor behavior

What `delete` operator does is twofold:

1) Calls `string` object's dtor
2) Deallocates memory occupied by the object

`delete` Operator

- `delete` operator calls function `operator delete` declared in `<new>` to perform requisite memory deallocation

```
delete ps;
```

```
// call the object's dtor
ps->~string();
// deallocate memory occupied by the object
operator delete(ps);
```

operator delete

- Plain old function `operator delete` has following signature:

```
void operator delete (void*) noexcept;
```

delete[] Operator

- What is happening when you write code like this

```
string *ps = new string [3] {"a", "b", "c"};
... // using string objects
delete[] ps;
```

You're using delete[] operator!!!

What delete[] operator does is twofold:

1) Calls string dtor for each array element
2) Deallocates memory occupied by array

operator delete[]

- delete[] operator calls function operator delete[] declared in <new> to *only* perform requisite deallocation of memory for array elements

```
string *ps = new string [3] {"a", "b", "c"};
... // using string objects
delete[] ps;
```

```
// call each array element's dtor
ps[0].~string();
ps[1].~string();
ps[2].~string();
// deallocate memory occupied by array
operator delete[](ps);
```

operator delete[]

- Plain old function `operator delete[]` has following signature:

```
void operator delete[] (void*) noexcept;
```

`operator new` And `operator delete` Overloads


```
void* operator new(size_t);
void* operator new[](size_t);
void  operator delete(void*) noexcept;
void  operator delete[](void*) noexcept;
```

- By default, plain-old `operator new` and `operator delete` handle all dynamic memory allocations and deallocations in program
- Therefore, it is sufficient to just redefine plain-old `operator new` and `operator delete`
  - Other overloads will just use redefined plain-old `operator new` and `operator delete`

operator new: Pseudocode


```
// plain-old operator new
void* operator new(size_t size) {
  // handle 0-byte requests as 1-byte requests

  while (1) {
    // attempt to allocate size bytes;
    if (the allocation was successful)
      return (a pointer to the memory);
    // allocation was unsuccessful; find out what the
    // current error-handling [https://bit.ly/3vbBoc1]
    // function is
    std::new_handler the_handler = std::get_new_handler();
    if (the_handler) (*the_handler)();
    else throw std::bad_alloc();
  }
}
```

`operator new[]`: Pseudocode

- Similar to pseudocode for function `operator new`

operator delete: Pseudocode


```
// plain-old operator delete ...
void operator delete(void *raw_memory) noexcept {
  // do nothing if null pointer is being deleted
  if (raw_memory == 0) return;
  deallocate the memory pointed to by raw_memory;
  return;
}
```

`operator delete[]`: Pseudocode

- Similar to pseudocode for function `operator delete`

`new` Operator And Exceptions

- What happens if plain-old `operator new` allocated memory but ctor throws an exception?
- Memory must first be deallocated by invoking global `void operator delete (void*) noexcept;`
- Then exception is propagated

operator new Overloads

- Other overloads of operator new called *placement* new exist to allow for additional arguments to new

```
// nothrow version of operator new returns nullptr on failure
void* operator new (std::size_t,
                    const std::nothrow_t&) noexcept;
// construct object at specific, preallocated memory address
void* operator new (std::size_t, void*);
// user-defined parameters
void* operator new (std::size_t, ...);
```

You can redefine your own versions of these functions except

```
void* operator new (std::size_t, void*);
```

operator delete Overloads

- Some of operator delete overloads:

```
// users can only call global delete function
void operator delete (void*) noexcept;
// nothrow delete function
void operator delete (void*,
                      const std::nothrow_t&) noexcept;
// delete object at non-allocated memory address
void operator delete (void*, void*) noexcept;
// user-defined parameters
void operator delete (void*, ...);
```

You can redefine your own versions of these functions except

`void* operator delete (void*, void*) noexcept;`

`operator new[]` Overloads

- Other overloads of `operator new[]` called *placement* `new[]` exist to allow for additional arguments to `new[]`

```
// nothrow version of operator new[] returns nullptr on failure
void* operator new[](std::size_t,
                     const std::nothrow_t&) noexcept;
// construct object at specific, preallocated memory address
void* operator new[](std::size_t, void*);
// user-defined parameters
void* operator new[](std::size_t, ...);
```

You can redefine your own versions of these functions except `void* operator new[](std::size_t, void*);`

operator delete[] Overloads

- Some of `operator delete[]` overloads:

```
// users can only call global delete function
void operator delete[](void*) noexcept;
// nothrow delete function
void operator delete[](void*,
                  const std::nothrow_t&) noexcept;
// delete object at non-allocated memory address
void operator delete[](void*, void*) noexcept;
// user-defined parameters
void operator delete[](void*, ...);
```

You can redefine your own versions of these functions except
`void* operator delete[](void*, void*) noexcept;`

`new` Operator And Exceptions

- What happens if nothrow version of `operator new` allocated memory but ctor throws an exception?
- Memory must first be deallocated by invoking global `void operator delete (void*, const std::nothrow_t&) noexcept;`
- Then exception is propagated
- All of this means *nothrow* version of `operator delete` is called only if *nothrow* version of `operator new` throws an exception

Class Interface

- Besides letting programs replace some of global `operator new` and `operator delete` functions, C++ also lets classes provide their own class-specific versions
  - Good idea since redefining global `operator new` and `operator delete` means taking responsibility for all dynamic memory allocations for entire program

Class Interface


```
class X {
public:
  void* operator new(size_t);                                   // 1
  void* operator new(size_t, std::nothrow_t const &) noexcept; // 2
  void* operator new(size_t, void*);                            // 3

  void operator delete(void*) noexcept;                         // 4
  void operator delete(void*, std::nothrow_t const&) noexcept; // 5
  void operator delete(void*, void*) noexcept;                  // 6

  void* operator new[](size_t);                                  // 7
  void* operator new[](size_t, std::nothrow_t const&) noexcept; // 8
  void* operator new[](size_t, void*);                           // 9

  void operator delete[](void*) noexcept;                        // 10
  void operator delete[](void*, std::nothrow_t const&) noexcept; // 11
  void operator delete[](void*, void*) noexcept;                 // 12
// other stuff ...
};
```

Name Hiding Surprise


```
class Base {
public:
  void* operator new(size_t, FastMemory const&); //4
  // ...
};

class Derived : public Base {
  // ...
};

Derived *p1 = new Derived;  // ERROR: no match
Derived* p2 = new (std::nothrow) Derived; // ERROR: no match
size_t arena[100];
void *p3 = reinterpret_cast<void*>(arena);
new (p3) Derived;                         // ERROR: no match
Derived *p4 = new (FastMemory()) Derived; // calls 4
```

Name Hiding Surprise

- Compiler starts in current scope (in `Derived`'s scope), and looks for desired name (here, `operator new`)
- If no instances of name are found, it moves outward to next enclosing scope (in `Base`'s and then global scope) and repeats
- Once it find a scope containing at least one instance of name (in this case, `Base`'s scope), it stops looking and works only with matches it has found, which means that further outer scopes (in this case, the global scope) are not considered and any functions in them are hidden
- Instead, compiler looks at all instances of name it has found, selects a function using overload resolution, and finally checks access rules to determine whether selected function can be called
- Outer scopes ignored even if none of overloads found has compatible signature, meaning that none of them could possibly be right one
- Outer scopes also ignored even if signature-compatible function that's selected isn't accessible

Memory Pooling

- Frequent memory allocation and deallocation can degrade application performance
  - Default memory manager is general-purpose
  - You use memory in very specific way and pay performance penalty for functionality you don’t need
- You could counter that by developing specialized memory managers
- We will understand the basics of writing our own specialized (and highly simplified) memory managers using some examples