Dynamic Memory Management

In C++, the `new()` and `delete()` operators provide built-in language support for dynamic memory allocation and deallocation.

This feature has several benefits:

- Reduces common programmer errors: it is easy to forget to multiply the number of objects being allocated by `sizeof` when using `malloc()`, e.g.,
  ```
  // oops, only 2 1/2 int's!
  int *a = (int *) malloc (10);
  ```
- Enhances source code clarity: generally, there is no need to: (1) declare operator `new()` and `delete()`, (2) explicitly use casts, or (3) explicitly check the return value.
- Improves run-time efficiency: (1) users can redefine operator `new()` and `delete()` globally and also define them on a per-class basis and (2) calls can be inlined.

Operator `new()`, be it local or global, is only used for “free store” allocation

Therefore, the following does not involve any direct invocation of operator `new()`:

```
Xa;
X f (void) { X b; /* ... */ return b; }
```

Note, an object allocated from the free store has a lifetime that extends beyond its original scope,

```
int *f (int i) {
  int *ip = new() int[i];
  // ...
  return ip;
}
```
Error Handling

- By default, if operator `new()` cannot find memory it calls a pointer to function called `_new_handler()`, e.g.,

```cpp
void *operator new() (size_t size) {
    void *p;
    while ((p = malloc (size)) == 0)
        if (_new_handler)
            (*_new_handler());
        else
            return 0;
    return p;
}
```

- if `_new_handler()` can somehow supply memory for `malloc()` then all is fine - otherwise, an exception is thrown

- Note, `_new_handler()` can be set by users via the `set_new_handler()` function, e.g., `set_new_handler (::abort);`

Interaction with Malloc and Free

- All C++ implementations also permit use of C `malloc()` and `free()` routines. However:
  1. Don’t intermix `malloc()`/`delete()` and `new()`/`free()`.
  2. Be careful not to use these to allocate C++ class objects with constructors or destructors, e.g.,

```
class Foo {
    public:
        Foo (void) { foo_ = new() int (100); }
    // ...
    ~Foo (void);
    private:
        int *foo_;
    }

    Foo *bar = new() Foo; // OK, calls constructor
    Foo *baz = malloc (sizeof *baz); // ERROR, constructor not called
    free (bar); // Error, destructor not called!
```

- Note, C++ does not supply a `realloc()`-style operator.

Interaction with Arrays

- The global `new()` and `delete()` operators are always used for allocating and deallocating arrays of class objects.
- When calling `delete()` for a pointer to an array, use the `[]` syntax to enabled destructors to be called, e.g.,

```
class Foo {
    public:
        Foo (void);
        ~Foo (void);
    }

    Foo *bar = new() Foo[100];
    Foo *baz = new() Foo;
    // ...
    delete [] bar; // must have the []
    delete baz; // must not have the []
```

Interaction with Constructors and Destructors

- Allocation and deallocation are completely separate from construction and destruction
  - construction and destruction are handled by constructors and destructors
  - Allocation and deallocation are handled by operator `new()` and operator `delete()`
- Note, at the time a constructor is entered, memory has already been allocated for the constructor to do its work
- Similarly, a destructor does not control what happens to the memory occupied by the object it is destroying
Interaction with Constructors and Destructors (cont’d)

- Here’s a simple case:
  ```c
  void f (void) {
    T x;
  }
  ```
- Executing `f()` causes the following to happen:
  1. Allocate enough memory to hold a `T`;
  2. construct the `T` in that memory;
  3. Destroy the `T`;
  4. Deallocate the memory.

Similarly, the next line has the following effects:
```c
T *tp = new() T;
```
1. Allocate enough memory to hold a `T`;
2. if allocation was successful,
3. construct a `T` in that memory;
4. Store the address of the memory in `tp`

Finally, the following happens on deletion:
```c
delete() tp;
```
if `tp` is non-zero, destroy the `T` in the memory addressed by `tp` and then deallocate the memory addressed by `tp`.

Object Placement Syntax

- The C++ memory allocation scheme provides a way to construct an object in an arbitrary location via an object placement syntax. Merely say:
  ```c
  void *operator new() (size_t, void *p) { return p; }
  ```
- Now you can do something like this:
  ```c
  // Allocate memory in shared memory
  void *vp = shm_malloc (sizeof (T));
  T *tp = new() (vp) T; // construct a T there.
  ```
- Because it is possible to construct an object in memory that has already been allocated, there must be a way to destroy an object without deallocating its memory. To do that, call the destructor directly:
  ```c
  tp->T::~T (); // Note, also works on built-in types!
  shm_free (tp);
  ```
Object Placement Syntax (cont’d)

- The placement syntax can be used to supply additional arguments to `operator new()`, e.g.,
  ```c++
  new() T; // calls operator new() (sizeof (T))
  new() (2, f) T; // calls operator new() (sizeof (T), 2, f)
  ```
- e.g., provide a C++ interface to vector-resize via realloc...

```c++
// Note, this only works sensibly for built-in types,
// due to constructor/destructor issues...
static inline void *
operator new() (size_t size, void *ptr, size_t new_len) {
  return ptr = = 0 ? malloc (size * new_len)
    : realloc (ptr, new_len * size);
}
// ...char *p = new() (0, 100) char;
p = new() (p, 1000) char;
```

Overloading Global `operator new`

- Memory allocation can be tuned for a particular problem
  - e.g., assume you never want to `delete()` any allocated memory:
    ```c++
    struct align {char x; double d;};
    const int ALIGN = ((char *)((struct align *) 0)->d - (char *) 0);
    void *operator new() (size_t size) {
      static char *buf_start = 0;
      static char *buf_end = 0;
      static int buf_size = 4 * BUFSIZ;
      char *temp;
      size = ((size + ALIGN - 1) / ALIGN) * ALIGN;
      if (buf_start + size >= buf_end) {
        buf_size *= 2;
        buf_size = MAX (buf_size, size);
        if (buf_start + size >= buf_end) {
          buf_size = size;
          buf_end = buf_start + buf_size;
        }
      }
      temp = buf_start;
      buf_start += size;
      return temp;
    }
    ```

Class Specific `new()` and `delete()`

- It is possible to overload the allocation/deallocation operators `operator new()` and `delete()` for an arbitrary class `X`:

```c++
class X {
  public:
    void *operator new() (size_t);
    void operator delete() (void *);
  // ...
};
```
- Now `X::operator new()` will be used instead of the global `operator new()` for objects of class `X`. Note that this does not affect other uses of `operator new()` within the scope of `X`:

```c++
void *X::operator new() (size_t s) {
  return new[] char[s]; // global operator new as usual
}
void X::operator delete() (void *) {
  delete[] p; // global operator delete as usual
}
```

- Note, the version of `operator new()` above will be used only when allocating objects of class `T` or classes derived from `T`
  - i.e., *not* arrays of class objects...
Interaction with Overloading

- Operator new() can take additional arguments of any type that it can use as it wishes, e.g.,

```cpp
definition enum Mem_Speed {SLOW, NORM, FAST, DEFAULT};
void *operator new() (size_t sz, Mem_Speed sp);
```

- Note, operator new() and delete() obey the same scope rules as any other member function
  - if defined inside a class, operator new() hides any global operator new(),
  ```cpp
class T {
public:
  void *operator new() (size_t, Mem_Speed);
};
```
  T* tp = new() T; // Error, need 2 arguments!

- The use of new T is incorrect because the member operator new() hides the global operator new()

There are three ways to solve the above problem.

1. The class definition for T might contain an explicit declaration:
   ```cpp
class T {
public:
  void *operator new() (size_t, Mem_Speed);
  void *operator new() (size_t sz) {
    return ::operator new() (sz);
  }
};
```
2. Alternatively, you can explicitly request the global operator new() using the scope resolution operator when allocating a T:
   ```cpp
   T* tp = ::new() T;
   ```
3. Finally, give a default value to class specific operator new(), e.g.,
   ```cpp
   void *operator new() (size_t, Mem_Speed = DEFAULT);
   ```

Therefore, no operator new() can be found for T that does not require a second argument

Interaction with Overloading (cont'd)

- It is not possible to overload operator delete() with a different signature
- There are several ways around this restriction:
  - Operator delete() can presumably figure out how to delete an object by looking at its address.
    * e.g., obtained from different allocators.
  - Alternatively, operator new() might store some kind of “magic cookie” with the objects it allocates to enable operator delete() to figure out how to delete them.
Class Specific \texttt{new()} and \texttt{delete()} Example

- Class specific \texttt{new()} and \texttt{delete()} operators are useful for homogeneous container classes
  
  - \textit{e.g.}, linked lists or binary trees, where the size of each object is fixed

- This permits both \textit{eager} allocation and \textit{lazy} deallocation strategies that amortize performance, in terms of time and space utilization

- It is possible to become quite sophisticated with the allocation strategies
  
  - \textit{e.g.}, trading off transparency for efficiency, \textit{etc.}

---

File \texttt{Stack.h}

```cpp
#include <new.h>
typedef int T;

class Stack {
public:
  Stack (int csize);
  T pop (void);
  T top (void);
  int push (T new_item);
  int is_empty (void);
  int is_full (void);
  `Stack (void);
static int get_chunk_size (void);
static void set_chunk_size (int size);
static void out_of_memory (int mem_avail);
};
```

---

File \texttt{Stack.cpp}

```cpp
#include <stream.h>
#include "stack.h"

int Stack::chunk_size = 0;
int Stack::memory_exhausted = 0;
Stack_Chunk *Stack_Chunk::free_list = 0;
Stack_Chunk *Stack_Chunk::spare_chunk = 0;

void *Stack_Chunk::operator new() (size_t bytes, int size, Stack_Chunk *next) {
  Stack_Chunk *chunk = Stack_Chunk::free_list;
  if (chunk != 0) {
    Stack_Chunk::free_list = chunk->link;
    Stack_Chunk *stack = chunk;
  } else {
    int n_bytes = bytes + (size - 1) * sizeof *chunk->stack_chunk;
    chunk = Stack_Chunk::spare_chunk;
    if (chunk != 0) {
      Stack_Chunk *stack = chunk;
      stack->link = Stack_Chunk::spare_chunk;
      Stack_Chunk::spare_chunk = chunk;
    } else {  ```
if ((chunk = (Stack_Chunk *) new() char[n_bytes])) == 0) {
    chunk = Stack_Chunk::spare_chunk;
    Stack::out_of_memory (1);
} else {
    chunk->chunk_size = size;
    chunk->top = 0;
    chunk->link = next;
    return chunk;
}

void Stack_Chunk::operator delete() (void *ptr) {
    Stack_Chunk *sc = (Stack_Chunk *) ptr;
    if (sc == Stack_Chunk::spare_chunk)
        Stack::out_of_memory (0);
    else {
        sc->link = Stack_Chunk::free_list;
        Stack_Chunk::free_list = sc;
    }
}

int Stack::get_chunk_size (void) {
    return Stack::chunk_size;
}

void Stack::set_chunk_size (int size) {
    Stack::chunk_size = size;
}

void Stack::out_of_memory (int out_of_mem) {
    Stack::memory_exhausted = out_of_mem;
}

Stack::Stack (int csize) {
    Stack::set_chunk_size (csize);
    if (Stack_Chunk::spare_chunk == 0)
        Stack_Chunk::spare_chunk = new() Stack_Chunk;
}

Stack::Stack (void) {
    Stack_Chunk *sc = this->stack; sc != 0; ) {
        Stack_Chunk *temp = sc;
        sc = sc->link;
        delete() (void *) temp;
    }
    for (sc = Stack_Chunk::free_list; sc != 0; ) {
        Stack_Chunk *temp = sc;
        sc = sc->link;
        delete() (void *) temp;
    }
}

T Stack::pop (void) {
    T temp =
    this->stack->stack_chunk[--this->stack->top];
    if (this->stack->top <= 0) {
        Stack_Chunk *temp = this->stack;
    }
Class Specific new() and delete() Example (cont’d)

- File Stack.cpp

```cpp
T Stack::top (void) {
    const int tp = this->stack->top - 1;
    return this->stack->stack_chunk[tp];
}

int Stack::push (T new_item) {
    if (this->stack == 0)
        this->stack = NEW (Stack::get_chunk_size ()) Stack_Chunk;
    else if (this->stack->top >= this->stack->chunk_size)
        this->stack = NEW (Stack::get_chunk_size (),
                        this->stack) Stack_Chunk;
    this->stack->stack_chunk[this->stack->top++] =
        new_item;
    return 1;
}

int Stack::is_empty (void) {
    return this->stack == 0;
}

int Stack::is_full (void) {
    return Stack::memory_exhausted;
}
```

Main program

```cpp
#include <stream.h>
#include <stdlib.h>
#include "Stack.h"

const int DEFAULT_SIZE = 10;
const int CHUNK_SIZE = 40;

int main (int argc, char *argv[]) {
    int size = argc == 1 ? DEFAULT_SIZE : atoi (argv[1]);
    int chunk_size = argc == 2 ? CHUNK_SIZE : atoi (argv[2]);
    Stack stack (chunk_size);
    int t;
    srandom (time (0L));
    for (int i = 0; i < size && !stack.is_full () ; i++)
        if (random () & 01) {
            stack.push (random () % 1000);
            t = stack.top ();
            std::cout << "top = " << t << std::endl;
        } else if (!stack.is_empty ()) {
            t = stack.pop ();
            std::cout << "pop = " << t << std::endl;
        }
    return 0;
}
```
else
    std::cout << "stack is currently empty!\n";
while (!stack.is_empty ()) {
    t = stack.pop ();
    std::cout << "pop = " << t << std::endl;
} return 0;

Copyright ©1997-2003 Vanderbilt University

Summary

class T {
public:
    T (void);
    ~T (void);
    void *operator new (size_t);
    void operator delete() (void);
};

void f (void) {
    T *tp1 = new T; // calls T::operator new
    T *tp2 = ::new T; // calls ::operator new
    T *tp3 = new T[10]; // calls ::operator new
    delete() tp1; // calls T::operator delete()
    ::delete() tp2; // calls ::operator delete()
    delete() [] tp3; // calls ::operator delete()
}