The C++ Programming Language

Dynamic Memory Management Techniques

Outline

Dynamic Memory Management
Interaction with Malloc and Free
Interaction with Arrays
Interaction with Constructors and Destructors
Object Placement Syntax
Overloading Global Operator New
Assignment to **this**
Class Specific **new** and **delete**
Interaction with Inheritance
Interaction with Overloading
Class Specific **new** and **delete** Example
Summary
Error Handling

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 ints!
  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.

Dynamic Memory Management (cont'd)

- **operator new** can be either a globally defined function or a member of class T or a base class of T.
  - Here is a minimal example of a global definition of **operator new**:

    ```
    extern "C" void *malloc (size_t);
    void *operator new (size_t sz) {
        return malloc (sz);
    }
    ```

- There must be only one global **operator new** (with these particular argument types) in an executable.
  - Note, it is possible to overhead **operator new**!
  - If you do not supply your own, there is one in the C++ run-time library that's only a little more complicated than this one.

Dynamic Memory Management (cont'd)

- **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**:

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

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

```
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.*,

```
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...

- 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

- Most (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 {
    int *bar;
public:
    Foo (void) { bar = new int (100); }
    // ...
    ~Foo (void);
};
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.
  - Thus, performing a resize operation may lead to increased overhead.

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:

```
void f (void) {
    T x;
}
```

- Executing f causes the following to happen:

```
Allocate enough memory to hold a T;
Construct the T in that memory;
Destroy the T;
Deallocate the memory.
```

- Similarly, the next line has the following effects:

```
T *tp = new T;
Allocate enough memory to hold a T;
If allocation was successful,
    construct a T in that memory;
Store the address of the memory in tp
```

- Finally, the following happens on deletion:

```
delete tp;
If tp is non-zero,
    destroy the T in the memory addressed by tp;
    free the memory addressed by tp.
```


Interaction with Constructors and Destructors (cont'd)

- How can a programmer control the memory allocated for objects of type T?
  - The answer lies in the allocation process, not the construction process.
  - C++ provides fine-grained control over what it means to "allocate enough memory to hold a T."
- *e.g.*,

```
T *tp = new T;
first set tp = operator new (sizeof (T))
then call constructor for class T at location 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:

```
void *operator new (size_t, void *p) { return p; }
```

- Now you can do something like this:

```
// 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:

```
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.*,

```
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...

```
/* 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 often be tuned for a particular problem.
  - *e.g.*, assume you never want to **delete** any allocated memory:

```
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 = malloc (buf_size))
               buf_end = buf_start + buf_size;
          else
               return 0;
     }
     temp = buf_start;
     buf_start += size;
     return temp;
}
```


Assignment to this

- Typical memory management approach:

```
typedef char ELEMENT;
class Stack_1 {
private:
     int top_, max_;
     ELEMENT *stack_;
public:
     enum {DEFAULT_SIZE = 100};
     Stack_1 (int size = Stack_1::DEFAULT_SIZE) {
          this->stack_ = new ELEMENT[size];
          this->max_ = size;
          this->top_ = 0;
     }
     // ...push, pop, top, is_empty, is_full, etc.
     ~Stack_1 (void) { delete this->stack_; }
};
Stack_1 *s1 = new Stack_1; // defaults to 100
```

- However, this has three problems:

(1) Extra allocation per dynamic object creation.
(2) Extra indirection on every access to **this**->stack;
(3) Extra "allocation wrapper" overhead in the constructor.

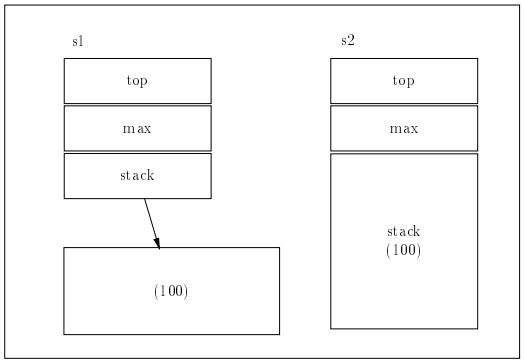
Assignment to this (cont'd)

- In C++ releases prior to 2.0, if a user wanted to eliminate

the problems describe above, the only way was to a

```
typedef char ELEMENT;
class Stack_2 {
private:
     int top_, max_;
     ELEMENT stack_[1];
     ~Stack_2 (void) { delete (void *) this; this = 0; }
     // the cast is essential!
public:
     enum {DEFAULT_SIZE = 100};
     Stack_2 (int size = Stack_2::DEFAULT_SIZE) {
          int nbytes = sizeof *this +
               (size - 1) * sizeof *this->stack_;
          this = (Stack_2 *) new char[nbytes];
          this->max_ = size;
          this->top_ = 0;
     }
     // ...push, pop, top, is_empty, is_full, etc.
     void dispose (void) {
          this->Stack_2::~Stack_2 ();
     }
};
```

Assignment to this (cont'd)



- *e.g.*,

```
Stack_1 s1;
Stack_2 s2;
```


Assignment to this (cont'd)

- However, assigning to **this** is now considered a language anachronism for several reasons:
  1. It is easy to make a mistake...
  2. It is not transparent, *e.g.*,
     - Using assignment to **this** also means that all objects must be allocated on the free store.
     - Declaring the destructor **private** allows the compiler to ensure all objects come off the free store, *e.g.*,

```
int main (int argc, char *argv[]) {
    int size = atoi (argv[1]);
    Stack_2 s (size); // error, private destructor!
    Stack_2 *sp = new Stack_2 (size);
    sp->dispose ();
}
```


Class Specific new and delete

- With Release 2.0, it is possible to overload the allocation/deallocation operators **operator new** and **delete** for class X:

```
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`:

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

Class Specific new and delete (cont'd)

- It is possible to approximate the "assignment to this" optimization using class specific **operator new**:

```
class Stack {
private:
    int top_, max_;
    char stack_[1];
    ~Stack (void) { delete (void *) this; this = 0;}
public:
    enum {DEFAULT_SIZE = 100};
    Stack (void) { }
    void *operator new (size_t nbytes,
                        int nelems = Stack::DEFAULT_SIZE) {
        int size = nbytes + (nelems - 1) * sizeof (char);
        Stack *sp = (Stack *) new char[size];
        sp->top_ = 0;
        sp->size_ = nelems;
        return sp;
    }
    // ...push, pop, top, is_empty, is_full, etc.
    void dispose (void) {
        this->Stack::~Stack ();
    }
};
```

Class Specific new and delete (cont'd)

- However, overloading **operator new** is not entirely transparent, compared with assignment to **this**.

```
int main (int argc, char *argv[]) {
      int size = atoi (argv[1]);
      Stack *sp = new (size) Stack;
      // note operator new now acts as constructor!
      sp->dispose ();
}
```

- One way to make this look more "normal" is to define a new helper function:

```
class Stack {
public:
      static Stack *new_stack (int size) {
            return new (size) Stack;
      }
      // ...
};
```

- Then the line in the main program becomes:

```
Stack *sp = Stack::new_stack (size);
```

- This change is not transparent, of course!

Interaction with Inheritance

- Operators **new** and **delete** are inherited:

```
// class Y objects also allocated using X::operator new
class Y : X { /* ...*/ };
```

- In this case `nbytes` contains the size (in bytes) of the derived class object.
  - Note, it is also possible to redefine **operator new** and **delete** in **class** Y as well.

- Note, if you want to supply an allocator/deallocator pair that works correctly for derived classes you *must* supply a virtual destructor in the base class or refrain from using the `size_t` argument in the deallocator.
  - Otherwise incorrect sizes will be used.

Interaction with Overloading

- **operator new** can take additional arguments of any type that it can use as it wishes, *e.g.*,

```
enum Mem_Speed {SLOW, NORM, FAST, DEFLT};
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**,

```
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**
  - Therefore, no **operator new** can be found for T that does not require a second argument.

Interaction with Overloading (cont'd)

- There are three ways to solve the above problem.
  1. The class definition for T might contain an explicit declaration:

```
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:

```
T *tp = ::new T;
```

  3. Finally, give a default value to class specific **operator new**, *e.g.*,

```
void *operator new (size_t, Mem_Speed = DEFLT);
```

Interaction with Overloading (cont'd)

- It is not possible to overload **operator delete**. 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 new and delete Example

- Class specific **new** and **delete** operators are useful for homogeneous container classes.
  - *e.g.*, linked lists or binary trees, where the size of each object is fixed.
- This permits both *eager* allocation and *lazy* deallocation strategies that amortize performance, in terms of time and space utilization.
- It is possible to become quite sophisticated with the allocation strategies.
  - *e.g.*, trading off transparency for efficiency, etc.

Class Specific new and delete Example (cont'd)

- Here's an example that illustrates how we can use **operator new** and **operator delete** to reduce overhead from a dynamically allocated stack ADT.
- // File stack.h

```
#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);
```


Class Specific new and delete Example (cont'd)

- // File stack.h (cont'd)

```
private:
      static int chunk_size;
      static int memory_exhausted;
      class Stack_Chunk {
      friend class Stack;
      private:
            int top;
            int chunk_size;
            Stack_Chunk *link;
            T stack_chunk[1];
            static Stack_Chunk *free_list;
            static Stack_Chunk *spare_chunk;
            void *operator new (size_t, int = 1,
                                Stack_Chunk * = 0);
            void operator delete (void *);
      } *stack;
};
```

Class Specific new and delete Example (cont'd)

- // File stack.C

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


Class Specific new and delete Example (cont'd)

- // File stack.C

```
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;
}
```


Class Specific new and delete Example (cont'd)

- // File stack.C

```
Stack::~Stack (void) {
    for (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;
        this->stack = this->stack->link;
        delete temp;
    }
    return temp;
}
```


Class Specific new and delete Example (cont'd)

- // File stack.C

```
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;
}
```

- // File main.C

```
#include <stream.h>
#include <stdlib.h>
#include "stack.h"
const int DFLT_SIZE = 10;
const int CHUNK_SIZE = 40;
int main (int argc, char *argv[]) {
    int size = argc == 1 ? DFLT_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 ();
            cout << "top = " << t << '\n';
        }
        else if (!stack.is_empty ()) {
            t = stack.pop ();
            cout << "pop = " << t << '\n';
        }
        else
            cout << "stack is currently empty!\n";
    while (!stack.is_empty ()) {
        t = stack.pop ();
        cout << "pop = " << t << '\n';
    }
    return 0;
}
```


Summary

- *e.g.*,

```
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
}
```