CIS 330
C++ and Unix

Lecture 15

Dynamic Object Creation

Object Creation

Remember that….

when an object is created in C++, two events occur

- Storage is allocated (at the beginning of its scope)
- Constructor is called to initialize the object (when the declaration is encountered)

Storage Allocation

Storage allocation can occur in one of several ways

- Before the program begins, allocated in the static area (i.e., exists for the lifetime of the program)
- Allocated in the stack (e.g., opening braces, function, etc.), and released at the end of its scope
- Allocated on the heap

This is similar to how things are done in C

C++

Dynamic memory in C++

- `new` **operator**
  - Not a function (e.g., malloc)
  - `MyType *fp = new MyType(1,2);`
    - Equivalent to `malloc(sizeof(MyType))`, and
    - Its constructor will be called (`MyType(1,2)`)
- `delete` **operator**
  - Not a function (i.e., free)
  - Can be called for any object created with `new`
  - Destructor will first be called, and then the memory will be released
  - Undefined if allocated with `malloc` (or other variations)

Example

```
class Tree {
int height;
public:
  Tree(int treeHeight) : height(treeHeight) {}
  ~Tree() { std::cout << "*"; }
  friend std::ostream&
  operator<<(std::ostream& os, const Tree* t) {
    return os << "Tree height is: "
              << t->height << std::endl;
  }
};
using namespace std;
int main() {
  Tree* t = new Tree(40);
  cout << t;
  delete t;
} ///:~
```

Example

```
Tree height is: 40
*
```

`delete`

Deleting a `void*` is probably a bug

- It is only okay to use for simple data (e.g., does not require a destructor)
- Only the storage is released, and the destructor is **NOT** called (the program does not know which destructor to call, since it can be anything)

Example

```
class Object {
  void* data; const int size; const char id;
public:
  Object(int sz, char c) : size(sz), id(c) {
    data = new char[size];
    cout << "Constructing object " << id
         << ", size = " << size << endl;
  }
  ~Object() {
    cout << "Destructing object " << id << endl;
    delete []data; // OK, just releases storage,
  }
};
int main() {
  Object* a = new Object(40, 'a'); delete a;
  void* b = new Object(40, 'b');   delete b;
} ///:~
```

Example

```
Constructing object a, size = 40
Destructing object a
Constructing object b, size = 40
```

Delete

If you have a memory leak (and you are deleting `new` allocated memory), check the type for the pointer being deleted

When you are using `void* data` to hold different types of objects, cast it to the proper type before using it (and deleting it)

`new` and `delete` for Arrays

To create an array of objects

```
MyType* fp = new MyType[100];
```

- This allocates enough memory on the heap for 100 `MyType` objects

Let's say you also have

```
MyType* fp2 = new MyType;
```

What if you do

```
delete fp2;
delete fp;
```

What happens?

`new` and `delete` for Arrays

```
MyType* fp = new MyType[100];
delete fp;
```

This DOES free up storage for all 100 `MyType` objects, BUT the constructor for only the FIRST element will be called

- Just as in `free()`, the OS keeps track of the memory allocated
- 99 other elements will NOT have their constructors called

Under what scenario could this cause a problem?

`new` and `delete` for Arrays

```
MyType* fp = new MyType[100];
delete [] fp;
// old syntax - delete [100] fp;
```

Running Out of Memory

When `new` cannot find enough memory to allocate, it calls a special function called `new-handler` is called

- More accurately, there is a function pointer, and if the pointer is non-zero, this function is called
- Default behavior is to throw an exception, but if you are using heap memory, it's better to replace it with function that prints a message (e.g., out of memory), and then abort the program

Example

```
int count = 0;
void out_of_memory() {
  cerr << "memory exhausted after " << count
    << " allocations!" << endl;
  exit(1);
}
int main() {
  set_new_handler(out_of_memory);
  while(1) {
    count++;
    new int[1000]; // Exhausts memory
  }
} ///:~
```

Arrays of Objects

With arrays, you can only call the default constructor - what if you need to use a non-default constructor

- Use a vector - the easier solution
- Use an array of pointers - for each element, use `new` with a non-default constructor to create a pointer and save it there

- Placement-new - you can pre-allocate the memory, then initialize it with a non-default constructor by passing in the pre-allocated memory location
- Overload the `new` operator (operator overloading will be covered later)

Example

```
class X {
  int i;
public:
  X(int ii = 0) : i(ii) { cout <<  "this = " << this << endl;
}
  ~X() { cout << "X::~X(): " << this << endl; }
};
int main() {
  int l[10]; // pre-allocate memory
  cout << "l = " << l << endl;
  X* xp = new(l) X(47); // X at location l
  xp->X::~X(); // Explicit destructor call required
}
```

Another Example

```
class Car {
    int _no;
public:
    Car( int no ) :_no( no ) {
    }
};

int main() {
    void* raw_memory = operator new(NUM_CARS * sizeof(Car));
                // actually allocates memory
    Car* ptr = static_cast<Car*>(raw_memory);
    for( int i = 0; i < NUM_CARS; ++i ) {
        new (&ptr[i]) Car( i );
    }
    // destruct in reverse order
    for( int i = NUM_CARS - 1; i >= 0; --i ) {
        ptr[i].~Car();
    }
    operator delete[]( raw_memory ); // actually frees memory
    return 0;
}
```

const

An example where using `const` might make sense

```
int* const q = new int[10];
```

const

An example where using `const` might make sense

```
int* const q = new int[10];
```

- `q` points to the beginning of an array, and you typically do not want to lose this position
- Integers inside `q` can still be modified, but `q` cannot be made to point somewhere else

Questions?