Chapter 13

DYNAMIC OBJECT CREATION

***Listing 13-1. malloc() with Class Objects***

**//: C13:MallocClass.cpp**

**// Malloc with class objects**

**// What you'd have to do if not for "new"**

**#include "../require.h" // To be INCLUDED from *Chapter 9***

**#include <cstdlib> // malloc() & free()**

**#include <cstring> // memset()**

**#include <iostream>**

**using namespace std;**

**classObj {**

**int i, j, k;**

**enum { sz = 100 };**

**charbuf[sz];**

**public:**

**void initialize() { // Can't use constructor**

**cout << "initializing Obj" << endl;**

**i = j = k = 0;**

**memset(buf, 0, sz);**

**}**

**void destroy() const { // Can't use destructor**

**cout << "destroying Obj" << endl;**

**}**

**};**

**int main() {**

**Obj \*obj = (Obj\*)malloc(sizeof(Obj));**

**require(obj != 0);**

**obj->initialize();**

**// ... sometime later:**

**obj->destroy();**

**free(obj);**

**} ///:~**

***Listing 13-2. Illustrating new and delete***

**//: C13:Tree.h**

**#ifndef TREE\_H**

**#define TREE\_H**

**#include <iostream>**

**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;**

**}**

**};**

**#endif // TREE\_H ///:~**

**//: C13:NewAndDelete.cpp**

**// Simple demo of new & delete**

**#include "Tree.h" // Header FILE to be INCLUDED from above**

**using namespace std;**

**int main() {**

**Tree \*t = new Tree(40);**

**cout << t;**

**delete t;**

**} ///:~**

***Listing 13-3. Illustrating a Case of Bad void Pointer Deletion***

**//: C13:BadVoidPointerDeletion.cpp**

**// Deleting void pointers can cause memory leaks**

**#include <iostream>**

**using namespace std;**

**class Object {**

**void \*data; // Some storage**

**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,**

**// no destructor calls are necessary**

**}**

**};**

**int main() {**

**Object\* a = new Object(40, 'a');**

**delete a;**

**void\* b = new Object(40, 'b');**

**delete b;**

**} ///:~**

***Listing 13-4. Header File for “pointer Stash”***

**//: C13:PStash.h**

**// Holds pointers instead of objects**

**#ifndef PSTASH\_H**

**#define PSTASH\_H**

**class PStash {**

**int quantity; // Number of storage spaces**

**int next; // Next empty space**

**// Pointer storage:**

**void\*\* storage;**

**void inflate(int increase);**

**public:**

**PStash() : quantity(0), storage(0), next(0) {}**

**~PStash();**

**int add(void\* element);**

**void\* operator[](int index) const; // Fetch**

**// Remove the reference from this PStash:**

**void\* remove(int index);**

**// Number of elements in Stash:**

**int count() const { return next; }**

**};**

**#endif // PSTASH\_H ///:~**

***Listing 13-5. Implementation of “pointer Stash”***

**//: C13:PStash.cpp {O}**

**// Pointer Stash definitions**

**#include "PStash.h" // To be INCLUDED from above**

**#include "../require.h"**

**#include <iostream>**

**#include <cstring> // 'mem' functions**

**using namespace std;**

**int PStash::add(void\* element) {**

**const int inflateSize = 10;**

**if(next >= quantity)**

**inflate(inflateSize);**

**storage[next++] = element;**

**return(next - 1); // Index number**

**}**

**// No ownership:**

**PStash::~PStash() {**

**for(int i = 0; i < next; i++)**

**require(storage[i] == 0,**

**"PStash not cleaned up");**

**delete []storage;**

**}**

**// Operator overloading replacement for fetch**

**void\* PStash::operator[](int index) const {**

**require(index >= 0,**

**"PStash::operator[] index negative");**

**if(index >= next)**

**return 0; // To indicate the end**

**// Produce pointer to desired element:**

**return storage[index];**

**}**

**void\* PStash::remove(int index) {**

**void\* v = operator[](index);**

**// "Remove" the pointer:**

**if(v != 0) storage[index] = 0;**

**return v;**

**}**

**void PStash::inflate(int increase) {**

**const int psz = sizeof(void\*);**

**void\*\* st = new void\*[quantity + increase];**

**memset(st, 0, (quantity + increase) \* psz);**

**memcpy(st, storage, quantity \* psz);**

**quantity += increase;**

**delete []storage; // Old storage**

**storage = st; // Point to new memory**

**} ///:~**

***Listing 13-6. Test Program for “pointer Stash”***

**//: C13:PStashTest.cpp**

**//{L} PStash**

**// Test of pointer Stash**

**#include "PStash.h"**

**#include "../require.h"**

**#include <iostream>**

**#include <fstream>**

**#include <string>**

**using namespace std;**

**int main() {**

**PStash intStash;**

**// 'new' works with built-in types, too. Note**

**// the "pseudo-constructor" syntax:**

**for(int i = 0; i < 25; i++)**

**intStash.add(new int(i));**

**for(int j = 0; j < intStash.count(); j++)**

**cout << "intStash[" << j << "] = "**

**<< \*(int\*)intStash[j] << endl;**

**// Clean up:**

**for(int k = 0; k < intStash.count(); k++)**

**delete intStash.remove(k);**

**ifstream in ("PStashTest.cpp");**

**assure(in, "PStashTest.cpp");**

**PStash stringStash;**

**string line;**

**while(getline(in, line))**

**stringStash.add(new string(line));**

**// Print out the strings:**

**for(int u = 0; stringStash[u]; u++)**

**cout << "stringStash[" << u << "] = "**

**<< \*(string\*)stringStash[u] << endl;**

**// Clean up:**

**for(int v = 0; v < stringStash.count(); v++)**

**delete (string\*)stringStash.remove(v);**

**} ///:~**

***Listing 13-7. Handling a Case of Running Out of Memory***

**//: C13:NewHandler.cpp**

**// Changing the new-handler**

**#include <iostream>**

**#include <cstdlib>**

**#include <new>**

**using namespace std;**

**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**

**}**

**} ///:~**

***Listing 13-8. Overloading the Global new and delete***

**//: C13:GlobalOperatorNew.cpp**

**// Overload global new/delete**

**#include <cstdio>**

**#include <cstdlib>**

**using namespace std;**

**void\* operator new(size\_t sz) {**

**printf("operator new: %d Bytes\n", sz);**

**void\* m = malloc(sz);**

**if(!m) puts("out of memory");**

**return m;**

**}**

**void operator delete(void\* m) {**

**puts("operator delete");**

**free(m);**

**}**

**class S {**

**int i[100];**

**public:**

**S() { puts("S::S()"); }**

**~S() { puts("S::~S()"); }**

**};**

**int main() {**

**puts("creating & destroying an int");**

**int\* p = new int(47);**

**delete p;**

**puts("creating & destroying an s");**

**S \*s = new S;**

**delete s;**

**puts("creating & destroying S[3]");**

**S \*sa = new S[3];**

**delete []sa;**

**} ///:~**

***Listing 13-9. Overloading the Local (for a Class) new and delete***

**//: C13:Framis.cpp**

**// Local overloaded new & delete**

**#include <cstddef> // Size\_t**

**#include <fstream>**

**#include <iostream>**

**#include <new>**

**using namespace std;**

**ofstream out("Framis.out");**

**class Framis {**

**enum { sz = 10 };**

**char c[sz]; // To take up space, not used**

**static unsigned char pool[];**

**static bool alloc\_map[];**

**public:**

**enum { psize = 100 }; // framis allowed**

**Framis() { out << "Framis()\n"; }**

**~Framis() { out << "~Framis() ... "; }**

**void\* operator new(size\_t) throw(bad\_alloc);**

**void operator delete(void\*);**

**};**

**unsigned char Framis::pool[psize \* sizeof(Framis)];**

**bool Framis::alloc\_map[psize] = {false};**

**// Size is ignored -- assume a Framis object**

**void\***

**Framis::operator new(size\_t) throw(bad\_alloc) {**

**for(int i = 0; i < psize; i++)**

**if(!alloc\_map[i]) {**

**out << "using block " << i << " ... ";**

**alloc\_map[i] = true; // Mark it used**

**return pool + (i \* sizeof(Framis));**

**}**

**out << "out of memory" << endl;**

**throw bad\_alloc();**

**}**

**void Framis::operator delete(void\* m) {**

**if(!m) return; // Check for null pointer**

**// Assume it was created in the pool**

**// Calculate which block number it is:**

**unsigned long block = (unsigned long)m**

**- (unsigned long)pool;**

**block /= sizeof(Framis);**

**out << "freeing block " << block << endl;**

**// Mark it free:**

**alloc\_map[block] = false;**

**}**

**int main() {**

**Framis \*f[Framis::psize];**

**try {**

**for(int i = 0; i < Framis::psize; i++)**

**f[i] = new Framis;**

**new Framis; // Out of memory**

**} catch(bad\_alloc) {**

**cerr << "Out of memory!" << endl;**

**}**

**delete f[10];**

**f[10] = 0;**

**// Use released memory:**

**Framis \*x = new Framis;**

**delete x;**

**for(int j = 0; j < Framis::psize; j++)**

**delete f[j]; // Delete f[10] OK**

**} ///:~**

***Listing 13-10. Using operator new() for Arrays***

**//: C13:ArrayOperatorNew.cpp**

**// Operator new for arrays**

**#include <new> // Size\_t definition**

**#include <fstream>**

**using namespace std;**

**ofstream trace("ArrayOperatorNew.out");**

**class Widget {**

**enum { sz = 10 };**

**int i[sz];**

**public:**

**Widget() { trace << "\*"; }**

**~Widget() { trace << "~"; }**

**void\* operator new(size\_t sz) {**

**trace << "Widget::new: "**

**<< sz << " bytes" << endl;**

**return ::new char[sz];**

**}**

**void operator delete(void\* p) {**

**trace << "Widget::delete" << endl;**

**::delete []p;**

**}**

**void\* operator new[](size\_t sz) {**

**trace << "Widget::new[]: "**

**<< sz << " bytes" << endl;**

**return ::new char[sz];**

**}**

**void operator delete[](void\* p) {**

**trace << "Widget::delete[]" << endl;**

**::delete []p;**

**}**

**};**

**int main() {**

**trace << "new Widget" << endl;**

**Widget\* w = new Widget;**

**trace << "\ndelete Widget" << endl;**

**delete w;**

**trace << "\nnew Widget[25]" << endl;**

**Widget\* wa = new Widget[25];**

**trace << "\ndelete []Widget" << endl;**

**delete []wa;**

**} ///:~*****Listing 13-11. Illustrating that the Constructor Doesn’t Come into Play in Case new Fails***

**//: C13:NoMemory.cpp**

**// Constructor isn't called if new fails**

**#include <iostream>**

**#include <new> // bad\_alloc definition**

**using namespace std;**

**class NoMemory {**

**public:**

**NoMemory() {**

**cout << "NoMemory::NoMemory()" << endl;**

**}**

**void\* operator new(size\_t sz) throw(bad\_alloc){**

**cout << "NoMemory::operator new" << endl;**

**throw bad\_alloc(); // "Out of memory"**

**}**

**};**

**int main() {**

**NoMemory\* nm = 0;**

**try {**

**nm = new NoMemory;**

**} catch(bad\_alloc) {**

**cerr << "Out of memory exception" << endl;**

**}**

**cout << "nm = " << nm << endl;**

**} ///:~*****Listing 13-12. Illustrating a Case of Placement with operator new()***

**//: C13:PlacementOperatorNew.cpp**

**// Placement with operator new**

**#include <cstddef> // Size\_t**

**#include <iostream>**

**using namespace std;**

**class X {**

**int i;**

**public:**

**X(int ii = 0) : i(ii) {**

**cout << "this = " << this << endl;**

**}**

**~X() {**

**cout << "X::~X(): " << this << endl;**

**}**

**void\* operator new(size\_t, void\* loc) {**

**return loc;**

**}**

**};**

**int main() {**

**int l[10];**

**cout << "l = " << l << endl;**

**X\* xp = new(l) X(47); // X at location l**

**xp->X::~X(); // Explicit destructor call**

**// ONLY use with placement!**

**} ///:~**