```
 1: // $Id: smart_ptr.cpp,v 1.3 2015-01-27 17:50:27-08 - - $
 2:
 3: //
 4: // Smart pointer class encapsulating a pointer, like shared_ptr.
 5: //
 6:
 7: #include <iostream>
 8: using namespace std;
 9:
10: ////////////////////////////////////////////////////////////////////
11: // pointer.h
12: ////////////////////////////////////////////////////////////////////
13:
14: template <typename Type>
15: class pointer {
16:       template <typename U>
17:       friend ostream& operator<< (ostream& , const pointer<U>& );
18:
19:    private:
20:
21:       // Invariant: ref_count == obj_ptr == nullptr
22:       //            ref_count->count on heap, obj_ptr->object itself
23:
24:       size_t* ref_count;
25:       Type* obj_ptr;
26:
27:       // Auxiliary functions.
28:
29:       inline void increment_count() {
30:          if (ref_count) ++*ref_count;
31:       }
32:
33:       inline void copy_that (size_t* ref, Type* obj) {
34:          if (ref_count and --*ref_count == 0) {
35:             delete ref_count;
36:             delete obj_ptr;
37:          }
38:          ref_count = ref;
39:          obj_ptr = obj;
40:       }
41:
42:       inline void clear_that (pointer& that) const {
43:          that.ref_count = nullptr;
44:          that.obj_ptr = nullptr;
45:       }
46:
```

```
47:
48:    public:
49:       // Replace implicitly generated functions.
50:
51:       pointer(): ref_count (nullptr), obj_ptr (nullptr) {
52:       }
53:
54:       pointer (const pointer& that): ref_count (that.ref_count),
55:                                      obj_ptr (that.obj_ptr) {
56:          increment_count();
57:       }
58:
59:       pointer (pointer&& that): ref_count (that.ref_count),
60:                                 obj_ptr (that.obj_ptr) {
61:          clear_that (that);
62:       }
63:
64:       pointer& operator= (const pointer& that) {
65:          if (this == &that) return *this;
66:          copy_that (that.ref_count, that.obj_ptr);
67:          increment_count();
68:          return *this;
69:       }
70:
71:       pointer& operator= (pointer&& that) {
72:          if (this == &that) return *this;
73:          copy_that (that.ref_count, that.obj_ptr);
74:          clear_that (that);
75:          return *this;
76:       }
77:
78:       ~pointer() {
79:          copy_that (nullptr, nullptr);
80:       }
81:
82:       // Other constructors.
83:
84:       pointer (Type* p_obj_ptr): ref_count (new size_t (1)),
85:                                  obj_ptr (p_obj_ptr) {
86:       }
87:
88:       // Mutators (non-const functions).
89:
90:       inline Type& operator*() { return *obj_ptr; }
91:
92:       inline Type* operator->() { return obj_ptr; }
93:
```

```
 94:
 95:       // Accessors (const functions).
 96:
 97:       inline const Type& operator*() const { return *obj_ptr; }
 98:
 99:       inline const Type* operator->() const { return obj_ptr; }
100:
101:       inline operator bool() const { return obj_ptr; }
102:
103:       inline bool operator== (const pointer& that) const {
104:          return obj_ptr == that.obj_ptr;
105:       }
106:
107:       inline bool operator!= (const pointer& that) const {
108:          return not (*this == that);
109:       }
110:
111:       size_t users() const { return ref_count ? *ref_count : 0; }
112:
113:       size_t unique() const { return users() == 1; }
114:
115:       size_t empty() const { return users() == 0; }
116: };
117:
118: template <typename Type>
119: ostream& operator<< (ostream& out, const pointer<Type>& that) {
120:    out << that.obj_ptr << "[" << *that.ref_count << "]";
121:    return out;
122: }
123:
```

```
124:
125: //////////////////////////////////////////////////////////////////
126: // Main program.
127: //////////////////////////////////////////////////////////////////
128:
129: struct node {
130:    string str;
131:    pointer<node> link;
132:    node (const char* s, pointer<node> p = nullptr): str (s), link (p) {
133:    }
134: };
135:
136: int main (int argc, char** argv) {
137:    pointer<node> head, tail;
138:    for (int i = 0; i < argc; ++i) {
139:       pointer<node> tmp = pointer<node> (new node (argv[i]));
140:       if (head == nullptr) head = tmp;
141:                       else tail->link = tmp;
142:       tail = tmp;
143:    }
144:    for (pointer<node> p = head; p != nullptr; p = p->link) {
145:       cout << p << "->\"" << p->str << "\"" << endl;
146:    }
147:    return 0;
148: }
149:
150: //TEST// alias grind='valgrind --leak-check=full --show-reachable=yes'
151: //TEST// grind smart_ptr hello world foo bar baz >smart_ptr.out 2>&1
152: //TEST// mkpspdf smart_ptr.ps smart_ptr.cpp* smart_ptr.out
153:
```

```
    1: @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ mkc: starting smart_ptr.cpp
    2: smart_ptr.cpp:
    3:       $Id: smart_ptr.cpp,v 1.3 2015-01-27 17:50:27-08 - - $
    4: g++ -g -O0 -Wall -Wextra -rdynamic -std=gnu++11 smart_ptr.cpp -o smart_p
tr -lglut -lGLU -lGL -lX11 -lrt -lm
    5: rm -f smart_ptr.o
    6: @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ mkc: finished smart_ptr.cpp
```

```
    1: ==9184== Memcheck, a memory error detector
    2: ==9184== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
    3: ==9184== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright in
fo
    4: ==9184== Command: smart_ptr hello world foo bar baz
    5: ==9184==
    6: 0x4e7d0e0[2]->"smart_ptr"
    7: 0x4e7d2a0[2]->"hello"
    8: 0x4e7d450[2]->"world"
    9: 0x4e7d600[2]->"foo"
   10: 0x4e7d7b0[2]->"bar"
   11: 0x4e7d960[3]->"baz"
   12: ==9184==
   13: ==9184== HEAP SUMMARY:
   14: ==9184==     in use at exit: 0 bytes in 0 blocks
   15: ==9184==   total heap usage: 38 allocs, 38 frees, 531 bytes allocated
   16: ==9184==
   17: ==9184== All heap blocks were freed -- no leaks are possible
   18: ==9184==
   19: ==9184== For counts of detected and suppressed errors, rerun with: -v
   20: ==9184== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```