Chapter 16

INTRODUCTION TO TEMPLATES

***Listing 16-1. Illustrating a Simple Integer Stack***

**//: C16:IntStack.cpp**

**// Simple integer stack**

**//{L} fibonacci**

**#include "fibonacci.h" // SEE ahead in this Section**

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

**#include <iostream>**

**using namespace std;**

**class IntStack {**

**enum { ssize = 100 };**

**int stack[ssize];**

**int top;**

**public:**

**IntStack() : top(0) {}**

**void push(int i) {**

**require(top < ssize, "Too many push()es");**

**stack[top++] = i;**

**}**

**int pop() {**

**require(top > 0, "Too many pop()s");**

**return stack[--top];**

**}**

**};**

**int main() {**

**IntStack is;**

**// Add some Fibonacci numbers, for interest:**

**for(int i = 0; i < 20; i++)**

**is.push(fibonacci(i));**

**// Pop & print them:**

**for(int k = 0; k < 20; k++)**

**cout << is.pop() << endl;**

**} ///:~**

***Listing 16-2. Header File for The Fibonacci Number Generator***

**//: C16:fibonacci.h**

**// Fibonacci number generator**

**int fibonacci(int n); ///:~**

Listing 16-3 is the implementation.

***Listing 16-3. The Implementation of The Fibonacci Number Generator***

**//: C16:fibonacci.cpp {O}**

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

**int fibonacci(int n) {**

**const int sz = 100;**

**require(n < sz);**

**static int f[sz]; // Initialized to zero**

**f[0] = f[1] = 1;**

**// Scan for unfilled array elements:**

**int i;**

**for(i = 0; i < sz; i++)**

**if(f[i] == 0) break;**

**while(i <= n) {**

**f[i] = f[i-1] + f[i-2];**

**i++;**

**}**

**return f[n];**

**} ///:~**

***Listing 16-4. Illustrating The Template Syntax***

**//: C16:Array.cpp**

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

**#include <iostream>**

**using namespace std;**

**template<class T>**

**class Array {**

**enum { size = 100 };**

**T A[size];**

**public:**

**T& operator[](int index) {**

**require(index >= 0 && index < size,**

**"Index out of range");**

**return A[index];**

**}**

**};**

**int main() {**

**Array<int> ia;**

**Array<float> fa;**

**for(int i = 0; i < 20; i++) {**

**ia[i] = i \* i;**

**fa[i] = float(i) \* 1.414;**

**}**

**for(int j = 0; j < 20; j++)**

**cout << j << ": " << ia[j]**

**<< ", " << fa[j] << endl;**

**} ///:~**

***Listing 16-5. Illustrating The Non-Inline Template/Function Definition***

**//: C16:Array2.cpp**

**// Non-inline template definition**

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

**template<class T>**

**class Array {**

**enum { size = 100 };**

**T A[size];**

**public:**

**T& operator[](int index);**

**};**

**template<class T>**

**T& Array<T>::operator[](int index) {**

**require(index >= 0 && index < size,**

**"Index out of range");**

**return A[index];**

**}**

**int main() {**

**Array<float> fa;**

**fa[0] = 1.414;**

**} ///:~**

***Listing 16-6. Illustrating A Simple Integer Stack Template***

**//: C16:StackTemplate.h**

**// Simple stack template**

**#ifndef STACKTEMPLATE\_H**

**#define STACKTEMPLATE\_H**

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

**template<class T>**

**class StackTemplate {**

**enum { ssize = 100 };**

**T stack[ssize];**

**int top;**

**public:**

**StackTemplate() : top(0) {}**

**void push(const T& i) {**

**require(top < ssize, "Too many push()es");**

**stack[top++] = i;**

**}**

**T pop() {**

**require(top > 0, "Too many pop()s");**

**return stack[--top];**

**}**

**int size() { return top; }**

**};**

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

***Listing 16-7. Testing Out The Integer Stack Template in Listing 16-6***

**//: C16:StackTemplateTest.cpp**

**// Test simple stack template**

**//{L} fibonacci**

**#include "fibonacci.h"**

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

**#include <iostream>**

**#include <fstream>**

**#include <string>**

**using namespace std;**

**int main() {**

**StackTemplate<int> is;**

**for(int i = 0; i < 20; i++)**

**is.push(fibonacci(i));**

**for(int k = 0; k < 20; k++)**

**cout << is.pop() << endl;**

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

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

**string line;**

**StackTemplate<string> strings;**

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

**strings.push(line);**

**while(strings.size() > 0)**

**cout << strings.pop() << endl;**

**} ///:~**

***Listing 16-8. Illustrating use of Built-in Types as Template Arguments***

**//: C16:Array3.cpp**

**// Built-in types as template arguments**

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

**#include <iostream>**

**using namespace std;**

**template<class T, int size = 100>**

**class Array {**

**T array[size];**

**public:**

**T& operator[](int index) {**

**require(index >= 0 && index < size,**

**"Index out of range");**

**return array[index];**

**}**

**int length() const { return size; }**

**};**

**class Number {**

**float f;**

**public:**

**Number(float ff = 0.0f) : f(ff) {}**

**Number& operator=(const Number& n) {**

**f = n.f;**

**return \*this;**

**}**

**operator float() const { return f; }**

**friend ostream&**

**operator<<(ostream& os, const Number& x) {**

**return os << x.f;**

**}**

**};**

**template<class T, int size = 20>**

**class Holder {**

**Array<T, size>\* np;**

**public:**

**Holder() : np(0) {}**

**T& operator[](int i) {**

**require(0 <= i && i < size);**

**if(!np) np = new Array<T, size>;**

**return np->operator[](i);**

**}**

**int length() const { return size; }**

**~Holder() { delete np; }**

**};**

**int main() {**

**Holder<Number> h;**

**for(int i = 0; i < 20; i++)**

**h[i] = i;**

**for(int j = 0; j < 20; j++)**

**cout << h[j] << endl;**

**} ///:~**

***Listing 16-9. Illustrating Creation of The Stack as a Template***

**//: C16:TStack.h**

**// The Stack as a template**

**#ifndef TSTACK\_H**

**#define TSTACK\_H**

**template<class T>**

**class Stack {**

**struct Link {**

**T\* data;**

**Link\* next;**

**Link(T\* dat, Link\* nxt):**

**data(dat), next(nxt) {}**

**}\* head;**

**public:**

**Stack() : head(0) {}**

**~Stack(){**

**while(head)**

**delete pop();**

**}**

**void push(T\* dat) {**

**head = new Link(dat, head);**

**}**

**T\* peek() const {**

**return head ? head->data : 0;**

**}**

**T\* pop(){**

**if(head == 0) return 0;**

**T\* result = head->data;**

**Link\* oldHead = head;**

**head = head->next;**

**delete oldHead;**

**return result;**

**}**

**};**

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

***Listing 16-10. Testing Out the Template Stack in Listing 16-9***

**//: C16:TStackTest.cpp**

**//{T} TStackTest.cpp**

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

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

**#include <fstream>**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**class X {**

**public:**

**virtual ~X() { cout << "~X " << endl; }**

**};**

**int main(int argc, char\* argv[]) {**

**requireArgs(argc, 1); // File name is argument**

**ifstream in(argv[1]);**

**assure(in, argv[1]);**

**Stack<string> textlines;**

**string line;**

**// Read file and store lines in the Stack:**

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

**textlines.push(new string(line));**

**// Pop some lines from the stack:**

**string\* s;**

**for(int i = 0; i < 10; i++) {**

**if((s = (string\*)textlines.pop())==0) break;**

**cout << \*s << endl;**

**delete s;**

**} // The destructor deletes the other strings.**

**// Show that correct destruction happens:**

**Stack<X> xx;**

**for(int j = 0; j < 10; j++)**

**xx.push(new X);**

**} ///:~**

***Listing 16-11. Illustrating the Templatized Pointer Stash***

**//: C16:TPStash.h**

**#ifndef TPSTASH\_H**

**#define TPSTASH\_H**

**template<class T, int incr = 10>**

**class PStash {**

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

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

**T\*\* storage;**

**void inflate(int increase = incr);**

**public:**

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

**~PStash();**

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

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

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

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

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

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

**};**

**template<class T, int incr>**

**int PStash<T, incr>::add(T\* element) {**

**if(next >= quantity)**

**inflate(incr);**

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

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

**}**

**// Ownership of remaining pointers:**

**template<class T, int incr>**

**PStash<T, incr>::~PStash() {**

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

**delete storage[i]; // Null pointers OK**

**storage[i] = 0; // Just to be safe**

**}**

**delete []storage;**

**}**

**template<class T, int incr>**

**T\* PStash<T, incr>::operator[](int index) const {**

**require(index >= 0,**

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

**if(index >= next)**

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

**require(storage[index] != 0,**

**"PStash::operator[] returned null pointer");**

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

**return storage[index];**

**}**

**template<class T, int incr>**

**T\* PStash<T, incr>::remove(int index) {**

**// operator[] performs validity checks:**

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

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

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

**return v;**

**}**

**template<class T, int incr>**

**void PStash<T, incr>::inflate(int increase) {**

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

**T\*\* st = new T\*[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**

**}**

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

**(of the Templatized Pointer Stash)*Listing 16-12. Testing Out the Ownership Control***

***(of the Templatized Pointer Stash)***

**//: C16:AutoCounter.h**

**#ifndef AUTOCOUNTER\_H**

**#define AUTOCOUNTER\_H**

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

**#include <iostream>**

**#include <set> // Standard C++ Library container**

**#include <string>**

**class AutoCounter {**

**static int count;**

**int id;**

**class CleanupCheck {**

**std::set<AutoCounter\*> trace;**

**public:**

**void add(AutoCounter\* ap) {**

**trace.insert(ap);**

**}**

**void remove(AutoCounter\* ap) {**

**require(trace.erase(ap) == 1,**

**"Attempt to delete AutoCounter twice");**

**}**

**~CleanupCheck() {**

**std::cout << "~CleanupCheck()"<< std::endl;**

**require(trace.size() == 0,**

**"All AutoCounter objects not cleaned up");**

**}**

**};**

**static CleanupCheck verifier;**

**AutoCounter() : id(count++) {**

**verifier.add(this); // Register itself**

**std::cout << "created[" << id << "]"**

**<< std::endl;**

**}**

**// Prevent assignment and copy-construction:**

**AutoCounter(const AutoCounter&);**

**void operator=(const AutoCounter&);**

**public:**

**// You can only create objects with this:**

**static AutoCounter\* create() {**

**return new AutoCounter();**

**}**

**~AutoCounter() {**

**std::cout << "destroying[" << id**

**<< "]" << std::endl;**

**verifier.remove(this);**

**}**

**// Print both objects and pointers:**

**friend std::ostream& operator<<(**

**std::ostream& os, const AutoCounter& ac){**

**return os << "AutoCounter " << ac.id;**

**}**

**friend std::ostream& operator<<(**

**std::ostream& os, const AutoCounter\* ac){**

**return os << "AutoCounter " << ac->id;**

**}**

**};**

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

***Listing 16-13. Implementing the AutoCounter in Listing 16-12***

**//: C16:AutoCounter.cpp {O}**

**// Definition of static class members**

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

**AutoCounter::CleanupCheck AutoCounter::verifier;**

**int AutoCounter::count = 0;**

**///:~**

With **AutoCounter** in hand, you can now test the facilities of the **PStash**. Listing 16-14 not only shows that the **PStash** destructor cleans up all the objects that it currently owns, but it also demonstrates how the **AutoCounter** class detects objects that haven’t been cleaned up.

***Listing 16-14. Testing Out The Templatized Pointer Stash Using AutoCounter***

**//: C16:TPStashTest.cpp**

**//{L} AutoCounter**

**#include "AutoCounter.h"**

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

**#include <iostream>**

**#include <fstream>**

**using namespace std;**

**int main() {**

**PStash<AutoCounter> acStash;**

**for(int i = 0; i < 10; i++)**

**acStash.add(AutoCounter::create());**

**cout << "Removing 5 manually:" << endl;**

**for(int j = 0; j < 5; j++)**

**delete acStash.remove(j);**

**cout << "Remove two without deleting them:"**

**<< endl;**

**// ... to generate the cleanup error message.**

**cout << acStash.remove(5) << endl;**

**cout << acStash.remove(6) << endl;**

**cout << "The destructor cleans up the rest:"**

**<< endl;**

**// Repeat the test from earlier chapters:**

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

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

**PStash<string> 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 << "] = "**

**<< \*stringStash[u] << endl;**

**} ///:~**

***Listing 16-15. Demonstrating a Stack with Runtime Controllable Ownership***

**//: C16:OwnerStack.h**

**// Stack with runtime controllable ownership**

**#ifndef OWNERSTACK\_H**

**#define OWNERSTACK\_H**

**template<class T> class Stack {**

**struct Link {**

**T\* data;**

**Link\* next;**

**Link(T\* dat, Link\* nxt)**

**: data(dat), next(nxt) {}**

**}\* head;**

**bool own;**

**public:**

**Stack(bool own = true) : head(0), own(own) {}**

**~Stack();**

**void push(T\* dat) {**

**head = new Link(dat,head);**

**}**

**T\* peek() const {**

**return head ? head->data : 0;**

**}**

**T\* pop();**

**bool owns() const { return own; }**

**void owns(bool newownership) {**

**own = newownership;**

**}**

**// Auto-type conversion: true if not empty:**

**operator bool() const { return head != 0; }**

**};**

**template<class T> T\* Stack<T>::pop() {**

**if(head == 0) return 0;**

**T\* result = head->data;**

**Link\* oldHead = head;**

**head = head->next;**

**delete oldHead;**

**return result;**

**}**

**template<class T> Stack<T>::~Stack() {**

**if(!own) return;**

**while(head)**

**delete pop();**

**}**

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

***Listing 16-16. Testing Out the Ownership of the Stack in Listing 16-15***

**//: C16:OwnerStackTest.cpp**

**//{L} AutoCounter**

**#include "AutoCounter.h"**

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

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

**#include <iostream>**

**#include <fstream>**

**#include <string>**

**using namespace std;**

**int main() {**

**Stack<AutoCounter> ac; // Ownership on**

**Stack<AutoCounter> ac2(false); // Turn it off**

**AutoCounter\* ap;**

**for(int i = 0; i < 10; i++) {**

**ap = AutoCounter::create();**

**ac.push(ap);**

**if(i % 2 == 0)**

**ac2.push(ap);**

**}**

**while(ac2)**

**cout << ac2.pop() << endl;**

**// No destruction necessary since**

**// ac "owns" all the objects**

**} ///:~**

***Listing 16-17. Illustrating Holding of Objects by Value Using Templates***

**//: C16:ValueStack.h**

**// Holding objects by value in a Stack**

**#ifndef VALUESTACK\_H**

**#define VALUESTACK\_H**

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

**template<class T, int ssize = 100>**

**class Stack {**

**// Default constructor performs object**

**// initialization for each element in array:**

**T stack[ssize];**

**int top;**

**public:**

**Stack() : top(0) {}**

**// Copy-constructor copies object into array:**

**void push(const T& x) {**

**require(top < ssize, "Too many push()es");**

**stack[top++] = x;**

**}**

**T peek() const { return stack[top]; }**

**// Object still exists when you pop it;**

**// it just isn't available anymore:**

**T pop() {**

**require(top > 0, "Too many pop()s");**

**return stack[--top];**

**}**

**};**

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

***Listing 16-18. Making the ValueStack in Listing 16-17 Work Using SelfCounter***

**//: C16:SelfCounter.h**

**#ifndef SELFCOUNTER\_H**

**#define SELFCOUNTER\_H**

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

**#include <iostream>**

**class SelfCounter {**

**static int counter;**

**int id;**

**public:**

**SelfCounter() : id(counter++) {**

**std::cout << "Created: " << id << std::endl;**

**}**

**SelfCounter(const SelfCounter& rv) : id(rv.id){**

**std::cout << "Copied: " << id << std::endl;**

**}**

**SelfCounter operator=(const SelfCounter& rv) {**

**std::cout << "Assigned " << rv.id << " to "**

**<< id << std::endl;**

**return \*this;**

**}**

**~SelfCounter() {**

**std::cout << "Destroyed: "<< id << std::endl;**

**}**

**friend std::ostream& operator<<(**

**std::ostream& os, const SelfCounter& sc){**

**return os << "SelfCounter: " << sc.id;**

**}**

**};**

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

**//: C16:SelfCounter.cpp {O}**

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

**int SelfCounter::counter = 0; ///:~**

**//: C16:ValueStackTest.cpp**

**//{L} SelfCounter**

**#include "ValueStack.h"**

**#include "SelfCounter.h"**

**#include <iostream>**

**using namespace std;**

**int main() {**

**Stack<SelfCounter> sc;**

**for(int i = 0; i < 10; i++)**

**sc.push(SelfCounter());**

**// OK to peek(), result is a temporary:**

**cout << sc.peek() << endl;**

**for(int k = 0; k < 10; k++)**

**cout << sc.pop() << endl;**

**} ///:~**

***Listing 16-19. Illustrating A Simple Integer Stack with Iterators***

**//: C16:IterIntStack.cpp**

**// Simple integer stack with iterators**

**//{L} fibonacci**

**#include "fibonacci.h"**

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

**#include <iostream>**

**using namespace std;**

**class IntStack {**

**enum { ssize = 100 };**

**int stack[ssize];**

**int top;**

**public:**

**IntStack() : top(0) {}**

**void push(int i) {**

**require(top < ssize, "Too many push()es");**

**stack[top++] = i;**

**}**

**int pop() {**

**require(top > 0, "Too many pop()s");**

**return stack[--top];**

**}**

**friend class IntStackIter;**

**};**

**// An iterator is like a "smart" pointer:**

**class IntStackIter {**

**IntStack& s;**

**int index;**

**public:**

**IntStackIter(IntStack& is) : s(is), index(0) {}**

**int operator++() { // Prefix**

**require(index < s.top,**

**"iterator moved out of range");**

**return s.stack[++index];**

**}**

**int operator++(int) { // Postfix**

**require(index < s.top,**

**"iterator moved out of range");**

**return s.stack[index++];**

**}**

**};**

**int main() {**

**IntStack is;**

**for(int i = 0; i < 20; i++)**

**is.push(fibonacci(i));**

**// Traverse with an iterator:**

**IntStackIter it(is);**

**for(int j = 0; j < 20; j++)**

**cout << it++ << endl;**

**} ///:~**

***Listing 16-20. Illustrating Nesting of an Iterator Inside the Container***

**//: C16:NestedIterator.cpp**

**// Nesting an iterator inside the container**

**//{L} fibonacci**

**#include "fibonacci.h"**

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

**#include <iostream>**

**#include <string>**

**using namespace std;**

**class IntStack {**

**enum { ssize = 100 };**

**int stack[ssize];**

**int top;**

**public:**

**IntStack() : top(0) {}**

**void push(int i) {**

**require(top < ssize, "Too many push()es");**

**stack[top++] = i;**

**}**

**int pop() {**

**require(top > 0, "Too many pop()s");**

**return stack[--top];**

**}**

**class iterator;**

**friend class iterator;**

**class iterator {**

**IntStack& s;**

**int index;**

**public:**

**iterator(IntStack& is) : s(is), index(0) {}**

**// To create the "end sentinel" iterator:**

**iterator(IntStack& is, bool)**

**: s(is), index(s.top) {}**

**int current() const { return s.stack[index]; }**

**int operator++() { // Prefix**

**require(index < s.top,**

**"iterator moved out of range");**

**return s.stack[++index];**

**}**

**int operator++(int) { // Postfix**

**require(index < s.top,**

**"iterator moved out of range");**

**return s.stack[index++];**

**}**

**// Jump an iterator forward**

**iterator& operator+=(int amount) {**

**require(index + amount < s.top,**

**"IntStack::iterator::operator+=() "**

**"tried to move out of bounds");**

**index += amount;**

**return \*this;**

**}**

**// To see if you're at the end:**

**bool operator==(const iterator& rv) const {**

**return index == rv.index;**

**}**

**bool operator!=(const iterator& rv) const {**

**return index != rv.index;**

**}**

**friend ostream&**

**operator <<(ostream& os, const iterator& it) {**

**return os << it.current();**

**}**

**};**

**iterator begin() { return iterator(\*this); }**

**// Create the "end sentinel":**

**iterator end() { return iterator(\*this, true);}**

**};**

**int main() {**

**IntStack is;**

**for(int i = 0; i < 20; i++)**

**is.push(fibonacci(i));**

**cout << "Traverse the whole IntStack\n";**

**IntStack::iterator it = is.begin();**

**while(it != is.end())**

**cout << it++ << endl;**

**cout << "Traverse a portion of the IntStack\n";**

**IntStack::iterator**

**start = is.begin(), end = is.begin();**

**start += 5, end += 15;**

**cout << "start = " << start << endl;**

**cout << "end = " << end << endl;**

**while(start != end)**

**cout << start++ << endl;**

**} ///:~**

***Listing 16-21. Illustrating A Simple Stack Template with Nested Iterator***

**//: C16:IterStackTemplate.h**

**// Simple stack template with nested iterator**

**#ifndef ITERSTACKTEMPLATE\_H**

**#define ITERSTACKTEMPLATE\_H**

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

**#include <iostream>**

**template<class T, int ssize = 100>**

**class StackTemplate {**

**T stack[ssize];**

**int top;**

**public:**

**StackTemplate() : top(0) {}**

**void push(const T& i) {**

**require(top < ssize, "Too many push()es");**

**stack[top++] = i;**

**}**

**T pop() {**

**require(top > 0, "Too many pop()s");**

**return stack[--top];**

**}**

**class iterator; // Declaration required**

**friend class iterator; // Make it a friend**

**class iterator { // Now define it**

**StackTemplate& s;**

**int index;**

**public:**

**iterator(StackTemplate& st): s(st),index(0){}**

**// To create the "end sentinel" iterator:**

**iterator(StackTemplate& st, bool)**

**: s(st), index(s.top) {}**

**T operator\*() const { return s.stack[index];}**

**T operator++() { // Prefix form**

**require(index < s.top,**

**"iterator moved out of range");**

**return s.stack[++index];**

**}**

**T operator++(int) { // Postfix form**

**require(index < s.top,**

**"iterator moved out of range");**

**return s.stack[index++];**

**}**

**// Jump an iterator forward**

**iterator& operator+=(int amount) {**

**require(index + amount < s.top,**

**" StackTemplate::iterator::operator+=() "**

**"tried to move out of bounds");**

**index += amount;**

**return \*this;**

**}**

**// To see if you're at the end:**

**bool operator==(const iterator& rv) const {**

**return index == rv.index;**

**}**

**bool operator!=(const iterator& rv) const {**

**return index != rv.index;**

**}**

**friend std::ostream& operator<<(**

**std::ostream& os, const iterator& it) {**

**return os << \*it;**

**}**

**};**

**iterator begin() { return iterator(\*this); }**

**// Create the "end sentinel":**

**iterator end() { return iterator(\*this, true);}**

**};**

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

***Listing 16-22. Testing Out the Stack Template in Listing 16-21***

**//: C16:IterStackTemplateTest.cpp**

**//{L} fibonacci**

**#include "fibonacci.h"**

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

**#include <iostream>**

**#include <fstream>**

**#include <string>**

**using namespace std;**

**int main() {**

**StackTemplate<int> is;**

**for(int i = 0; i < 20; i++)**

**is.push(fibonacci(i));**

**// Traverse with an iterator:**

**cout << "Traverse the whole StackTemplate\n";**

**StackTemplate<int>::iterator it = is.begin();**

**while(it != is.end())**

**cout << it++ << endl;**

**cout << "Traverse a portion\n";**

**StackTemplate<int>::iterator**

**start = is.begin(), end = is.begin();**

**start += 5, end += 15;**

**cout << "start = " << start << endl;**

**cout << "end = " << end << endl;**

**while(start != end)**

**cout << start++ << endl;**

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

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

**string line;**

**StackTemplate<string> strings;**

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

**strings.push(line);**

**StackTemplate<string>::iterator**

**sb = strings.begin(), se = strings.end();**

**while(sb != se)**

**cout << sb++ << endl;**

**} ///:~**

***Listing 16-23. Illustrating A Templatized Stack with Nested Iterator***

**//: C16:TStack2.h**

**// Templatized Stack with nested iterator**

**#ifndef TSTACK2\_H**

**#define TSTACK2\_H**

**template<class T> class Stack {**

**struct Link {**

**T\* data;**

**Link\* next;**

**Link(T\* dat, Link\* nxt)**

**: data(dat), next(nxt) {}**

**}\* head;**

**public:**

**Stack() : head(0) {}**

**~Stack();**

**void push(T\* dat) {**

**head = new Link(dat, head);**

**}**

**T\* peek() const {**

**return head ? head->data : 0;**

**}**

**T\* pop();**

**// Nested iterator class:**

**class iterator; // Declaration required**

**friend class iterator; // Make it a friend**

**class iterator { // Now define it**

**Stack::Link\* p;**

**public:**

**iterator(const Stack<T>& tl) : p(tl.head) {}**

**// Copy-constructor:**

**iterator(const iterator& tl) : p(tl.p) {}**

**// The end sentinel iterator:**

**iterator() : p(0) {}**

**// operator++ returns boolean indicating end:**

**bool operator++() {**

**if(p->next)**

**p = p->next;**

**else p = 0; // Indicates end of list**

**return bool(p);**

**}**

**bool operator++(int) { return operator++(); }**

**T\* current() const {**

**if(!p) return 0;**

**return p->data;**

**}**

**// Pointer dereference operator:**

**T\* operator->() const {**

**require(p != 0,**

**"PStack::iterator::operator->returns 0");**

**return current();**

**}**

**T\* operator\*() const { return current(); }**

**// bool conversion for conditional test:**

**operator bool() const { return bool(p); }**

**// Comparison to test for end:**

**bool operator==(const iterator&) const {**

**return p == 0;**

**}**

**bool operator!=(const iterator&) const {**

**return p != 0;**

**}**

**};**

**iterator begin() const {**

**return iterator(\*this);**

**}**

**iterator end() const { return iterator(); }**

**};**

**template<class T> Stack<T>::~Stack() {**

**while(head)**

**delete pop();**

**}**

**template<class T> T\* Stack<T>::pop() {**

**if(head == 0) return 0;**

**T\* result = head->data;**

**Link\* oldHead = head;**

**head = head->next;**

**delete oldHead;**

**return result;**

**}**

**#endif // TSTACK2\_H ///:~*****Listing 16-24. Testing Out the Templatized Stack in Listing 16-23***

**//: C16:TStack2Test.cpp**

**#include "TStack2.h" // To be INCLUDED from above**

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

**#include <iostream>**

**#include <fstream>**

**#include <string>**

**using namespace std;**

**int main() {**

**ifstream file("TStack2Test.cpp");**

**assure(file, "TStack2Test.cpp");**

**Stack<string> textlines;**

**// Read file and store lines in the Stack:**

**string line;**

**while(getline(file, line))**

**textlines.push(new string(line));**

**int i = 0;**

**// Use iterator to print lines from the list:**

**Stack<string>::iterator it = textlines.begin();**

**Stack<string>::iterator\* it2 = 0;**

**while(it != textlines.end()) {**

**cout << it->c\_str() << endl;**

**it++;**

**if(++i == 10) // Remember 10th line**

**it2 = new Stack<string>::iterator(it);**

**}**

**cout << (\*it2)->c\_str() << endl;**

**delete it2;**

**} ///:~**

***Listing 16-25. Illustrating A Templatized PStash with Nested Iterator***

**//: C16:TPStash2.h**

**// Templatized PStash with nested iterator**

**#ifndef TPSTASH2\_H**

**#define TPSTASH2\_H**

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

**#include <cstdlib>**

**template<class T, int incr = 20>**

**class PStash {**

**int quantity;**

**int next;**

**T\*\* storage;**

**void inflate(int increase = incr);**

**public:**

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

**~PStash();**

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

**T\* operator[](int index) const;**

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

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

**// Nested iterator class:**

**class iterator; // Declaration required**

**friend class iterator; // Make it a friend**

**class iterator { // Now define it**

**PStash& ps;**

**int index;**

**public:**

**iterator(PStash& pStash)**

**: ps(pStash), index(0) {}**

**// To create the end sentinel:**

**iterator(PStash& pStash, bool)**

**: ps(pStash), index(ps.next) {}**

**// Copy-constructor:**

**iterator(const iterator& rv)**

**: ps(rv.ps), index(rv.index) {}**

**iterator& operator=(const iterator& rv) {**

**ps = rv.ps;**

**index = rv.index;**

**return \*this;**

**}**

**iterator& operator++() {**

**require(++index <= ps.next,**

**"PStash::iterator::operator++ "**

**"moves index out of bounds");**

**return \*this;**

**}**

**iterator& operator++(int) {**

**return operator++();**

**}**

**iterator& operator--() {**

**require(--index >= 0,**

**"PStash::iterator::operator-- "**

**"moves index out of bounds");**

**return \*this;**

**}**

**iterator& operator--(int) {**

**return operator--();**

**}**

**// Jump interator forward or backward:**

**iterator& operator+=(int amount) {**

**require(index + amount < ps.next &&**

**index + amount >= 0,**

**"PStash::iterator::operator+= "**

**"attempt to index out of bounds");**

**index += amount;**

**return \*this;**

**}**

**iterator& operator-=(int amount) {**

**require(index - amount < ps.next &&**

**index - amount >= 0,**

**"PStash::iterator::operator-= "**

**"attempt to index out of bounds");**

**index -= amount;**

**return \*this;**

**}**

**// Create a new iterator that's moved forward**

**iterator operator+(int amount) const {**

**iterator ret(\*this);**

**ret += amount; // op+= does bounds check**

**return ret;**

**}**

**T\* current() const {**

**return ps.storage[index];**

**}**

**T\* operator\*() const { return current(); }**

**T\* operator->() const {**

**require(ps.storage[index] != 0,**

**"PStash::iterator::operator->returns 0");**

**return current();**

**}**

**// Remove the current element:**

**T\* remove(){**

**return ps.remove(index);**

**}**

**// Comparison tests for end:**

**bool operator==(const iterator& rv) const {**

**return index == rv.index;**

**}**

**bool operator!=(const iterator& rv) const {**

**return index != rv.index;**

**}**

**};**

**iterator begin() { return iterator(\*this); }**

**iterator end() { return iterator(\*this, true);}**

**};**

**// Destruction of contained objects:**

**template<class T, int incr>**

**PStash<T, incr>::~PStash() {**

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

**delete storage[i]; // Null pointers OK**

**storage[i] = 0; // Just to be safe**

**}**

**delete []storage;**

**}**

**template<class T, int incr>**

**int PStash<T, incr>::add(T\* element) {**

**if(next >= quantity)**

**inflate();**

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

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

**}**

**template<class T, int incr> inline**

**T\* PStash<T, incr>::operator[](int index) const {**

**require(index >= 0,**

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

**if(index >= next)**

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

**require(storage[index] != 0,**

**"PStash::operator[] returned null pointer");**

**return storage[index];**

**}**

**template<class T, int incr>**

**T\* PStash<T, incr>::remove(int index) {**

**// operator[] performs validity checks:**

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

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

**storage[index] = 0;**

**return v;**

**}**

**template<class T, int incr>**

**void PStash<T, incr>::inflate(int increase) {**

**const int tsz = sizeof(T\*);**

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

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

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

**quantity += increase;**

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

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

**}**

**#endif // TPSTASH2\_H ///:~**

***Listing 16-26. Creating and Testing Two Different Stash Objects***

**//: C16:TPStash2Test.cpp**

**#include "TPStash2.h" // To be INCLUDED from above**

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

**#include <iostream>**

**#include <vector>**

**#include <string>**

**using namespace std;**

**class Int {**

**int i;**

**public:**

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

**cout << ">" << i << ' ';**

**}**

**~Int() { cout << "~" << i << ' '; }**

**operator int() const { return i; }**

**friend ostream&**

**operator <<(ostream& os, const Int& x) {**

**return os << "Int: " << x.i;**

**}**

**friend ostream&**

**operator <<(ostream& os, const Int\* x) {**

**return os << "Int: " << x->i;**

**}**

**};**

**int main() {**

**{ // To force destructor call**

**PStash<Int> ints;**

**for(int i = 0; i < 30; i++)**

**ints.add(new Int(i));**

**cout << endl;**

**PStash<Int>::iterator it = ints.begin();**

**it += 5;**

**PStash<Int>::iterator it2 = it + 10;**

**for(; it != it2; it++)**

**delete it.remove(); // Default removal**

**cout << endl;**

**for(it = ints.begin();it != ints.end();it++)**

**if(\*it) // Remove() causes "holes"**

**cout << \*it << endl;**

**} // "ints" destructor called here**

**cout << "\n-------------------\n";**

**ifstream in("TPStash2Test.cpp");**

**assure(in, "TPStash2Test.cpp");**

**// Instantiate for String:**

**PStash<string> strings;**

**string line;**

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

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

**PStash<string>::iterator sit = strings.begin();**

**for(; sit != strings.end(); sit++)**

**cout << \*\*sit << endl;**

**sit = strings.begin();**

**int n = 26;**

**sit += n;**

**for(; sit != strings.end(); sit++)**

**cout << n++ << ": " << \*\*sit << endl;**

**} ///:~**

***Listing 16-27. Putting It All Together***

**//: C16:Shape.h**

**#ifndef SHAPE\_H**

**#define SHAPE\_H**

**#include <iostream>**

**#include <string>**

**class Shape {**

**public:**

**virtual void draw() = 0;**

**virtual void erase() = 0;**

**virtual ~Shape() {}**

**};**

**class Circle : public Shape {**

**public:**

**Circle() {}**

**~Circle() { std::cout << "Circle::~Circle\n"; }**

**void draw() { std::cout << "Circle::draw\n";}**

**void erase() { std::cout << "Circle::erase\n";}**

**};**

**class Square : public Shape {**

**public:**

**Square() {}**

**~Square() { std::cout << "Square::~Square\n"; }**

**void draw() { std::cout << "Square::draw\n";}**

**void erase() { std::cout << "Square::erase\n";}**

**};**

**class Line : public Shape {**

**public:**

**Line() {}**

**~Line() { std::cout << "Line::~Line\n"; }**

**void draw() { std::cout << "Line::draw\n";}**

**void erase() { std::cout << "Line::erase\n";}**

**};**

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

***Listing 16-28. Using the header file in Listing 16-27***

**//: C16:Drawing.cpp**

**#include <vector> // Uses Standard vector too!**

**#include "TPStash2.h"**

**#include "TStack2.h"**

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

**using namespace std;**

**// A Drawing is primarily a container of Shapes:**

**class Drawing : public PStash<Shape> {**

**public:**

**~Drawing() { cout << "~Drawing" << endl; }**

**};**

**// A Plan is a different container of Shapes:**

**class Plan : public Stack<Shape> {**

**public:**

**~Plan() { cout << "~Plan" << endl; }**

**};**

**// A Schematic is a different container of Shapes:**

**class Schematic : public vector<Shape\*> {**

**public:**

**~Schematic() { cout << "~Schematic" << endl; }**

**};**

**// A function template:**

**template<class Iter>**

**void drawAll(Iter start, Iter end) {**

**while(start != end) {**

**(\*start)->draw();**

**start++;**

**}**

**}**

**int main() {**

**// Each type of container has**

**// a different interface:**

**Drawing d;**

**d.add(new Circle);**

**d.add(new Square);**

**d.add(new Line);**

**Plan p;**

**p.push(new Line);**

**p.push(new Square);**

**p.push(new Circle);**

**Schematic s;**

**s.push\_back(new Square);**

**s.push\_back(new Circle);**

**s.push\_back(new Line);**

**Shape\* sarray[] = {**

**new Circle, new Square, new Line**

**};**

**// The iterators and the template function**

**// allow them to be treated generically:**

**cout << "Drawing d:" << endl;**

**drawAll(d.begin(), d.end());**

**cout << "Plan p:" << endl;**

**drawAll(p.begin(), p.end());**

**cout << "Schematic s:" << endl;**

**drawAll(s.begin(), s.end());**

**cout << "Array sarray:" << endl;**

**// Even works with array pointers:**

**drawAll(sarray,**

**sarray + sizeof(sarray)/sizeof(\*sarray));**

**cout << "End of main" << endl;**

**} ///:~**