Chapter 7

FUNCTION OVERLOADING AND

DEFAULT ARGUMENTS

***Listing 7-1. Function Overloading***

**//: C07:Stash3.h**

**// Function overloading**

**#ifndef STASH3\_H**

**#define STASH3\_H**

**class Stash {**

**int size; // Size of each space**

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

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

**// Dynamically allocated array of bytes:**

**unsigned char\* storage;**

**void inflate(int increase);**

**public:**

**Stash(int size); // Zero quantity**

**Stash(int size, int initQuantity);**

**~Stash();**

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

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

**int count();**

**};**

**#endif // STASH3\_H ///:~**

***Listing 7-2. More Function Overloading***

**//: C07:Stash3.cpp {O}**

**// Function overloading**

**#include "Stash3.h" // To be INCLUDED from Header FILE above**

**#include "../require.h" // To be INCLUDED from Header FILE in *Chapter 3***

**#include <iostream>**

**#include <cassert>**

**using namespace std;**

**const int increment = 100;**

**Stash::Stash(int sz) {**

**size = sz;**

**quantity = 0;**

**next = 0;**

**storage = 0;**

**}**

**Stash::Stash(int sz, int initQuantity) {**

**size = sz;**

**quantity = 0;**

**next = 0;**

**storage = 0;**

**inflate(initQuantity);**

**}**

**Stash::~Stash() {**

**if(storage != 0) {**

**cout << "freeing storage" << endl;**

**delete []storage;**

**}**

**}**

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

**if(next >= quantity) // Enough space left?**

**inflate(increment);**

**// Copy element into storage,**

**// starting at next empty space:**

**int startBytes = next \* size;**

**unsigned char\* e = (unsigned char\*)element;**

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

**storage[startBytes + i] = e[i];**

**next++;**

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

**}**

**void\* Stash::fetch(int index) {**

**require(0 <= index, "Stash::fetch (-)index");**

**if(index >= next)**

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

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

**return &(storage[index \* size]);**

**}**

**int Stash::count() {**

**return next;**

**// Number of elements in CStash**

**}**

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

**assert(increase >= 0);**

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

**int newQuantity = quantity + increase;**

**int newBytes = newQuantity \* size;**

**int oldBytes = quantity \* size;**

**unsigned char\* b = new unsigned char[newBytes];**

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

**b[i] = storage[i];**

**// Copy old to new**

**delete [](storage);**

**// Release old storage**

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

**quantity = newQuantity; // Adjust the size**

**} ///:~**

***Listing 7-3. The Test Program***

/**/: C07:Stash3Test.cpp**

**//{L} Stash3**

**// Function overloading**

**#include "Stash3.h"**

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

**#include <fstream>**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**int main() {**

**Stash intStash(sizeof(int));**

**for(int i = 0; i < 100; i++)**

**intStash.add(&i);**

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

**cout << "intStash.fetch(" << j << ") = "**

**<< \*(int\*)intStash.fetch(j)**

**<< endl;**

**const int bufsize = 80;**

**Stash stringStash(sizeof(char) \* bufsize, 100);**

**ifstream in("Stash3Test.cpp");**

**assure(in, "Stash3Test.cpp");**

**string line;**

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

**stringStash.add((char\*)line.c\_str());**

**int k = 0;**

**char\* cp;**

**while((cp = (char\*)stringStash.fetch(k++))!=0)**

**cout << "stringStash.fetch(" << k << ") = "**

**<< cp << endl;**

**} ///:~**

***Listing 7-4. Unions***

**//: C07:UnionClass.cpp**

**// Unions with constructors and member functions**

**#include <iostream>**

**using namespace std;**

**union U {**

**private: // Access control too!**

**int i;**

**float f;**

**public:**

**U(int a);**

**U(float b);**

**~U();**

**int read\_int();**

**float read\_float();**

**};**

**U::U(int a) { i = a; }**

**U::U(float b) { f = b;}**

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

**int U::read\_int() { return i; }**

**float U::read\_float() { return f; }**

**int main() {**

**U X(12), Y(1.9F);**

**cout << X.read\_int() << endl;**

**cout << Y.read\_float() << endl;**

**} ///:~**

***Listing 7-5. A Safe Union***

**//: C07:SuperVar.cpp**

**/ A super-variable**

**#include <iostream>**

**using namespace std;**

**class SuperVar {**

**enum {**

**character,**

**integer,**

**floating\_point**

**} vartype; // Define one**

**union { // Anonymous union**

**char c;**

**int i;**

**float f;**

**};**

**public:**

**SuperVar(char ch);**

**SuperVar(int ii);**

**SuperVar(float ff);**

**void print();**

**};**

**SuperVar::SuperVar(char ch) {**

**vartype = character;**

**c = ch;**

**}**

**SuperVar::SuperVar(int ii) {**

**vartype = integer;**

**i = ii;**

**}**

**SuperVar::SuperVar(float ff) {**

**vartype = floating\_point;**

**f = ff;**

**}**

**void SuperVar::print() {**

**switch (vartype) {**

**case character:**

**cout << "character: " << c << endl;**

**break;**

**case integer:**

**cout << "integer: " << i << endl;**

**break;**

**case floating\_point:**

**cout << "float: " << f << endl;**

**break;**

**}**

**}**

**int main() {**

**SuperVarA('c'), B(12), C(1.44F);**

**A.print();**

**B.print();**

**C.print();**

**} ///:~*Listing 7-6. Managing Blocks of Memory (Header File)***

**//: C07:Mem.h**

**#ifndef MEM\_H**

**#define MEM\_H**

**typedef unsigned char byte;**

**classMem {**

**byte\* mem;**

**int size;**

**void ensureMinSize(int minSize);**

**public:**

**Mem();**

**Mem(int sz);**

**~Mem();**

**int msize();**

**byte\* pointer();**

**byte\* pointer(int minSize);**

**};**

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

***Listing 7-7. Managing Blocks of Memory (Source Code Object cpp File)***

**//: C07:Mem.cpp {O}**

**#include "Mem.h" // To be INCLUDED from Header FILE above**

**#include <cstring>**

**using namespace std;**

**Mem::Mem() { mem = 0; size = 0; }**

**Mem::Mem(int sz) {**

**mem = 0;**

**size = 0;**

**ensureMinSize(sz);**

**}**

**Mem::~Mem() { delete []mem; }**

**int Mem::msize() { return size; }**

**void Mem::ensureMinSize(int minSize) {**

**if(size < minSize) {**

**byte\* newmem = new byte[minSize];**

**memset(newmem + size, 0, minSize - size);**

**memcpy(newmem, mem, size);**

**delete []mem;**

**mem = newmem;**

**size = minSize;**

**}**

**}**

**byte\* Mem::pointer() { return mem; }**

**byte\* Mem::pointer(int minSize) {**

**ensureMinSize(minSize);**

**return mem;**

**} ///:~**

***Listing 7-8. Testing the Mem Class***

**//: C07:MemTest.cpp**

**// Testing the Mem class**

**//{L} Mem**

**#include "Mem.h"**

**#include <cstring>**

**#include <iostream>**

**using namespace std;**

**classMyString {**

**Mem\* buf;**

**public:**

**MyString();**

**MyString(char\* str);**

**~MyString();**

**void concat(char\* str);**

**void print(ostream &os);**

**};**

**MyString::MyString() { buf = 0; }**

**MyString::MyString(char\* str) {**

**buf = new Mem(strlen(str) + 1);**

**strcpy((char\*)buf->pointer(), str);**

**}**

**void MyString::concat(char\* str) {**

**if(!buf) buf = new Mem;**

**strcat((char\*)buf->pointer(**

**buf->msize() + strlen(str) + 1), str);**

**}**

**void MyString::print(ostream &os) {**

**if(!buf) return;**

**os <<buf->pointer() <<endl;**

**}**

**MyString::~MyString() { delete buf; }**

**int main() {**

**MyStrings("My test string");**

**s.print(cout);**

**s.concat(" some additional stuff");**

**s.print(cout);**

**MyString s2;**

**s2.concat("Using default constructor");**

**s2.print(cout);**

**} ///:~**

***Listing 7-9. Managing Blocks of Memory (Modified Header File)***

**//: C07:Mem2.h**

**#ifndef MEM2\_H**

**#define MEM2\_H**

**typedef unsigned char byte;**

**class Mem {**

**byte\* mem;**

**int size;**

**void ensureMinSize(int minSize);**

**public:**

**Mem(int sz = 0);**

**~Mem();**

**int msize();**

**byte\* pointer(int minSize = 0);**

**};**

**#endif // MEM2\_H ///**