

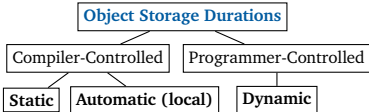
Fundamentals and Machine Model

Machine/Memory Model and the Function Call Stack

Object: a piece of data that's stored at a particular location in memory during runtime.

Variable: a *name* in source code that is associated with an object at compile time.

- Not all objects are associated with variables; e.g. dynamically-stored objects and string literals are not.
- The value stored by a variable's memory object may change, but the association between a variable and an object itself can only change when the variable goes out of **scope**.



Static objects "live" for essentially a program's runtime. Local objects' lifetimes are tied to scope (e.g. a block of code or pair of curly braces). Dynamic objects are manually created/destroyed.

- Objects declared in a loop body (between the {} are created/destroyed each time the loop repeats.

Atomic (primitive) types: types whose objects can't be subdivided into smaller objects; includes `int`, `double`, `bool`, `float`, `char`, and all pointer types. Atomic objects are default-initialized to undefined values.

```
1 // Four different ways to initialize an int to 5
2 int a = 5; int b(5); int c{5}; int d = {5};

1 // Explicitly cast an int 'd' to a double 'e'
2 double e = static_cast<double>(d);
```

Objects in C++ are **statically-typed**. Although an object may evaluate to a different type in an expression, the type of an object itself cannot change (class objects obey this rule too).

The memory allocated to store a function's parameters and local variables during runtime is called a **stack frame** or activation record. The memory frame for the most-recently called function is added to the "top" of the **function call stack** and is destroyed when the function returns ("Last In First Out" ordering).

Procedural Abstraction and Program Design

Procedural Abstraction involves using functions to break down a complex procedure into sub-tasks and separate the interface of a procedure (what it does) from implementation (how it works).

Interface examples: declarations in `.h` files, valid/invalid inputs, RME statements, *signature* (function name and parameter types), return type, and ADT representation invariants.

Implementation examples: definitions in `.cpp` files and code/comments inside function bodies.

Pointers and Arrays

A **pointer** is a type of object that stores another object's memory address as its value.

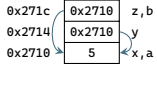
- An `int*` pointer variable can *only* point to an `int`; an `int**` pointer variable can *only* point to an `int*`; and so on. (E.g. attempting to make an `int*` pointer point to a `double` will lead to a compile error.)

Dereferencing a pointer: getting the object at an address. Note that the star * operator is used both to declare pointers and to dereference them (similarly, the & operator is used both to get an object's address and to declare a reference).

```
1 int x = 3; int y = 4;
2 int *ptr = &x; // ptr initialized to x's address
3 cout << *ptr; // dereferences ptr/prints 3
4 ptr = &y; // no star... assigns y's address to ptr
5 *ptr = 6; // dereferences ptr/assigns 6 to y
```

- Assigning `ptr = ptr2` copies the address stored by `ptr2` to `ptr` (subsequently changing `ptr2` wouldn't change `ptr`).
- A reference to a reference is really another reference for the "original" object.

```
1 int x = 5;
2 int* y = &x; // creates pointer to x
3 int* z = y; // creates another pointer to x
4 int &a = x; // creates reference to x
5 int* &b = z; // creates reference to z
6 cout << &b << endl; // Prints 5
7 cout << &y << endl; // prints 0x2714
8 cout << y << endl; // prints 0x2710
9 cout << &*z << endl; // equiv. to cout << &x
10 cout << &*z << endl; // equiv. to cout << z
```



Null pointer: a pointer that holds address `0x0` (which no object can be located at) and implicitly converts to `false`. Any pointer can be nulled; to do so, set it equal to `nullPtr` (0 or `NULL` also work but are bad style).

Common Pointer Bugs/Errors

- Dereferencing a default-initialized pointer results in undefined behavior, as (like all atomic objects) pointers that aren't explicitly initialized are default-initialized to an undefined value (*not* `nullPtr`).
- Dereferencing a null pointer also leads to undefined behavior (almost always a program crash).
- If a function returns a pointer or reference to one of its local variables (which die when the function returns), using the reference or dereferencing the pointer produces undefined behavior.

```
Number Swap Function
1 void swap_pointed(int *x, int *y) {
2     int tmp = *x;
3     *x = *y;
4     *y = tmp;
5 }
6
7 int main() {
8     int a = 1216, b = 1261;
9     swap_pointed(&a, &b);
10 }
```

Pointers vs References

References and pointers both enable working between stack frames (scopes) and indirection. Some ways they're different:

- References *must* be explicitly initialized (unlike pointers). This is because references are aliases for existing objects.
- Pointers must be dereferenced to access the objects they point at, while references are used "as-is".
- You can change the object that a (non-const) pointer points to, while a reference's binding to an object can't be changed.

Arrays and Pointer Arithmetic

Arrays: fixed-size containers that store objects of the same type (and same size) in contiguous memory.

```
1 // Valid array declarations
2 int arr[3] = {1,2}; // {1,2,0}
3 int zeroArr[3] = {}; // {0,0,0}

1 // Valid array declarations
2 int arr[] = {4,5,6};
3 int mat[2][2] = {1,2,3,4};

1 // INVALID array declarations
2 int junk[]; // Undefined items
3 int err[2][1] = {5,6,7,8}; // No
```

Array decay: using an array in a context where a value is required causes the compiler to convert the array into a pointer to its first element. Array decay is why it's necessary to pass an array's size separately from the array to a function (or to indicate the end of an array with a *sentinel character* like C-strings do).

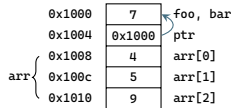
- Dereferencing a pointer that goes past the bounds of an array results in undefined behavior. But merely using a pointer that goes just past the end of an array without dereferencing it is well-defined.

```
1 void add_five(int arr[], int size) {
2     for (int i = 0; i < size; i++) { arr[i] += 5; }
3     // arr[i] += 5 is equiv. to *(arr + i) += 5
4 }
5
6 int main() {
7     int arr[] = { 10, 20, 30 };
8     add_five(arr, (sizeof(arr)) / sizeof(*arr));
9     cout << arr[1] << endl; // prints 25
10    // 1[arr] is equiv. to arr[1], but bad style
```

Passing `arr` by value passes a pointer to `arr[0]` by value. Also, `arr[i]` is shorthand for pointer arithmetic followed by a dereference, i.e., `arr[i] = *(arr + i)`.

- The `sizeof` operator returns the size of an object in *bytes*. In this example, `sizeof(arr)` alone would return 12, not 3.

```
1 int foo = 7;
2 int& bar = foo; // value of bar is foo (7)
3 int* ptr = &foo;
4 int arr[3] = { 4, 5, 9 };
5 cout << arr << endl; // prints 0x1008 (decay)
6 cout << (arr + 2) << endl; // prints 0x1010
7 cout << (&foo + 1) << endl; // prints 0x1004
```



Pointer arithmetic: adding an integer `n` to a pointer yields a pointer that is `n` objects forward in memory.

Pointer subtraction: Subtracting two pointers of the same type yields an integer (possibly a negative one) equal to the *number of objects* between them.

Pointer comparison: comparing pointers of the same type compares the addresses they store.

```
Pointer Operations
1 // Mainly for pointers into the same array
2 double arr[4] = { 2.5, 5.0, 8.0, 7.0 };
3 double* ptr1 = &arr[0], *ptr2 = &arr[3];
4 cout << &arr << endl; // prints 2.5
5 cout << (ptr2 - ptr1) << endl; // prints 3
6 cout << (ptr1 - ptr2) << endl; // prints -3
7 (ptr1 > ptr2); // equates to false (0)
8 ptr1 += 2; // ptr1 now points at arr[2]
```

Using the & operator on an array produces a pointer to the entire array, not a pointer to the first element or a pointer to a pointer (& does not require a value, so it doesn't cause decay).

```
1 int arr[4] = { 1, 2, 3, 4 };
2 int (&arr_ptr)[4] = &arr; // pointer to entire array
3 cout << (&arr_ptr)[2] << endl; // prints 3
4 // ++arr_ptr would increment by the size of 4 ints
```

Traversal By Pointer: arrays can be traversed by pointer (mostly used with C-strings and iterators).

```
Traversal By Pointer: Pattern 1
1 int const SIZE = 3;
2 int arr[SIZE] = {1, 7, 2};
3 int *ptr = arr;
4 int *end = arr + SIZE;
5 // int end is just past the end of arr
6 while (ptr < end) {
7     cout << *ptr << endl;
8     ++ptr; // "Walk" ptr across arr
9 } // Alternative to while loop below
1 for (; ptr < end; ++ptr) { ... }
```

```
Traversal by Pointer: Pattern 2 (C-String Sanitization)
1 void sanitize_username(Account *acc, char to_remove) {
2     char *ptr_a = acc->username, *ptr_b = acc->username;
3     while (*ptr_a && *ptr_b) { // while not '\0'
4         if (*ptr_b != to_remove) {
5             *ptr_a = *ptr_b;
6             ++ptr_a; // ++ptr_a only when a char gets copied
7         }
8         ++ptr_b; // ++ptr_b every time the loop executes
9     }
10    *ptr_a = '\0'; // null-terminate string when done
11 }
```

The const Keyword

The `const` type qualifier prevents objects from being modified after initialization (attempting to do so causes a compile error). `const` scalars must be explicitly-initialized to compile.

const pointers: pointers that can modify what they point at but cannot be re-pointed to different objects.

Pointer-to-const: read-only pointers; pointers that can be re-bound but can't modify what they point at.

- A `const` pointer must be initialized to compile, but a pointer-to-`const` doesn't need to be.

Reference-to-const: a read-only alias.

const array: an array of `const` elements. Note that the positioning of `const` matters for arrays of pointers.

const Conversions and Passing

The compiler treats every pointer-to-`const` as if they point to a `const` object and every reference-to-`const` as if they're aliased to a `const` object. It won't allow conversions that could bypass existing `const` protections (so, for example, you can assign a `const` pointer to a pointer-to-`const`, but the converse is *not* true).

```
1 int foo(int* a) { ... }
2 int bar(int b) { ... }
3 int func(const int* c) { ... }
4 const int x = 3;
5 bar(x); func(&x); // both ok
6 foo(&x); // ERROR
```

```
1 const int x = 3;
2 int y = x; // OK
3 const int* cptr = &x; // OK
4 const int& cref = x; // OK
5 int* ptr = cptr; // ERROR 1
6 int& ref = cref; // ERROR 2

1 int x = 2, y = 5;
2 const int *x_ptr = &x;
3 int *y_ptr = &y;
4 *y_ptr = *x_ptr; // OK
5 y_ptr = x_ptr; /* ERROR (even though x isn't const) */
```

- Pass by **pointer/reference**: if you need to modify the original object (as opposed to a local copy).
- Pass by **value**: if an object is small (e.g., an `int`) and you can't/don't need to modify the original.
- Pass by **pointer/reference-to-const**: if you want to pass a large object without modifying it.

Strings, Streams and I/O

Creating and Using C-Strings and Strings

```
1 const char* cstr = "abcd"; // Only works for string literals; use .c_str() on string variables
2 char color[] = "00274C"; // Create 7-element array (including \0) and copy a string literal to it
3 // Note: '\0' is the only char that evaluates to false (useful for traversal-by-pointer loops).
4 cout << cstr << " " << *cstr << " " << &cstr[0] << endl; // prints "abcd a bcd"
5 cout << (cstr + 1) << " " << *(cstr + 1) << " " << *(cstr + 1); // prints "bcd b 98" ('a' == 97)
6 string xyz = string(cstr); // Explicitly copy cstring to a string (implicit copy would work too)
```

	Length	Copy Value	Index	Concatenate	Compare
<cstring>	strlen(cstr);	strcpy(cstr1, cstr2);	cstr[i];	strcat(cstr1, cstr2);	strcmp(cstr1, cstr2);
<string>	str.length();	str1 = str2;	str[i];	str1 += str2;	str1 != str2;

Streams and File I/O

Input redirection	Output redirection	Pipeline	Combined redirection
./main.exe < input.txt	./main.exe > output.txt	./output.exe input.exe	./main.exe < input.in > output.out

File I/O Example: Print Lines From File

```
1 #include <fstream> // defines (if/of)stream objects
2 int main() {
3     ifstream inFS;
4     inFS.open("file.txt"); // valid
5     if (!inFS.is_open()) { return 1; }
6     string my_string; // initialized to empty string
7     while (getline(inFS, my_string)) {
8         cout << my_string << endl;
9     } // could close inFS manually via inFS.close();
10    // inFS also closes when scope ends/main returns
```

Ex: Copy One File's Contents to Another

```
1 #include <fstream>
2 int main() {
3     ifstream inFS("input.txt"); // Also valid
4     ofstream outFS("output.txt");
5     string my_string;
6     // newline and space both "delimit" words
7     while (inFS >> my_string) {
8         outFS << my_string << '\n';
9     } // '\n' is the newline char
10 }
```

istreamstream: a stream that "simulates" input from a hardcoded string.

ostreamstream: a stream that captures output and stores it in a string (use `.str()` to get the string).

- `ifstream`, `istreamstream`, and `cin` can all be passed to a function with an `istream&` parameter. Likewise, `ofstream`, `ostreamstream`, and `cout` can all be passed to a function with an `ostream&` parameter.

```
1 string input = "abc";
2 istreamstream inSS(input);

1 ostreamstream outSS; // i/o streams are defined in <sstream>
2 Mat_print(mat, outSS); // Capture output
```

Command-Line Arguments

`argc`: an `int` parameter of `main` representing the number of a command's arguments.

`argv`: functionally, an array of the arguments. Technically, `argv` is passed to `main` as a pointer to an array of pointers to C-strings. So `argv[0]` is a pointer to a C-string that represents the name of the program.

```
1 #include <iostream>
2 #include <string> // includes stoi()/stod()
3 int main(int argc, char* argv[]) { // char** argv also OK
4     if (string(argv[1]) == "add") {
5         int sum = 0;
6         for (int i = 2; i < argc; i++) { sum += stoi(argv[i]); }
7         cout << "Sum: " << sum << ", argc: " << argc << endl;
8     } // pay attention to where the "actual" arguments start
9     // Also remember to use stoi()/stod() when needed
```

```
Terminal
hugokin@ubuntu:~$ ./main.exe add 7 2
Sum: 9, argc: 4
hugokin@ubuntu:~$ ./main.exe add 1 2 3
Sum: 6, argc: 5
hugokin@ubuntu:~$ _
```

ADTs, Structs and Classes

C-Style Structs and ADTs

A **struct** is a class-type object composed of member subobjects (heterogeneous data). They're passed by value by default, and they support assignment and initialization via the `=` operator. A **struct** or **class** object can also be declared as `const`, which prevents it and all of its data members from being modified.

- `const` class-type objects must have their data members initialized (or a runtime error will occur).

- You can't call non-`const` member functions on a `const` instance of a `class` or `struct`. Also, a `const` member function can only call other `const` member functions.

Arrow -> operator: shorthand for pointer dereferencing followed by member access. `(*ptr).x` == `ptr->x`;

- Without parentheses, the dot and arrow operators have greater precedence than dereferencing.

Abstract Data Type: a data type that separates its behavior and implementation. ADTs encompass both data and behaviors/functions that act upon it. Not all structs are ADTs, some are "plain old data".

- Avoid accessing the member data of an ADT directly (even in tests) because it breaks the interface.

C++ Classes

In C++, the only real difference between classes and structs are that classes have private member access and private inheritance by default (structs default to public access/inheritance).

Constructors

- The compiler implicitly creates a default ctor if there are no user-defined ctors.
- The order in which members are declared in a class is *always* the order they're initialized in.
- Initialization values from a member init. list over-write initializations made during declarations.
- Data members that aren't included in a ctor's member initializer list or initialized at declaration get default-initialized/constructed.
- A delegating ctor must contain a call to the other ctor (and nothing else) in its member init. list.

```
Constructor Definition Example
1 class Animal {
2 private: string name;
3 public:
4   Animal(string name_in) // Non-default ctor
5   : name(name_in) { // Member init. list
6   Animal() // Default ctor (no arguments)
7   : Animal("Blank") { // ctor delegation
8   }; // Note the semicolon here!
9
10  class Bird : public Animal {
11 private: bool has_wings;
12 public: Bird(string name, bool wings_in)
13       : Animal(name), has_wings(wings_in) { }
14 }; // Derived class ctors must call base ctor
15
16 class Duck : public Bird {
17 private: string color;
18 public: Duck(string name, bool wings, string rgb)
19       : Bird(name, wings), color(rgb) { }
20 }; // Calling Bird ctor also calls Animal ctor
21
22 // This is how to define a ctor OUTSIDE of body
23 Duck::Duck(string name, bool wings, string rgb)
24 : Bird(name, wings), color(rgb) { }
```

Nested Classes and Constructors

To initialize a nested class object, initialize it with a valid argument for the nested class's ctor.

Nested class objects in a const class object are also const.

```
1 class Book {
2 public:
3   Book(double price_in)
4   : price(price_in) { }
5 // Note: no default Book ctor
6 private:
7   double price;
8 };
9
10 class Person {
11 public:
12   Person(string& n, double p)
13   : name(n), favBook(p) { }
14 private:
15   string name;
16   Book favBook;
17 };
18
19 class Book {
20 public:
21   Book(double price_in)
22   : price(price_in) { }
23 private:
24   double price;
25 };
26
27 class Person {
28 public:
29   Person(string& n, double p)
30   : name(n), favBook(p) { }
31 private:
32   string name;
33   Book favBook;
34 };
35
36 class Book {
37 public:
38   Book(double price_in)
39   : price(price_in) { }
40 private:
41   double price;
42 };
43
44 class Person {
45 public:
46   Person(string& n, double p)
47   : name(n), favBook(p) { }
48 private:
49   string name;
50   Book favBook;
51 };
52
53 class Book {
54 public:
55   Book(double price_in)
56   : price(price_in) { }
57 private:
58   double price;
59 };
60
61 class Person {
62 public:
63   Person(string& n, double p)
64   : name(n), favBook(p) { }
65 private:
66   string name;
67   Book favBook;
68 };
69
70 class Book {
71 public:
72   Book(double price_in)
73   : price(price_in) { }
74 private:
75   double price;
76 };
77
78 class Person {
79 public:
80   Person(string& n, double p)
81   : name(n), favBook(p) { }
82 private:
83   string name;
84   Book favBook;
85 };
86
87 class Book {
88 public:
89   Book(double price_in)
90   : price(price_in) { }
91 private:
92   double price;
93 };
94
95 class Person {
96 public:
97   Person(string& n, double p)
98   : name(n), favBook(p) { }
99 private:
100  string name;
101  Book favBook;
102 };
103
104 class Book {
105 public:
106   Book(double price_in)
107   : price(price_in) { }
108 private:
109   double price;
110 };
111
112 class Person {
113 public:
114   Person(string& n, double p)
115   : name(n), favBook(p) { }
116 private:
117   string name;
118   Book favBook;
119 };
120
121 class Book {
122 public:
123   Book(double price_in)
124   : price(price_in) { }
125 private:
126   double price;
127 };
128
129 class Person {
130 public:
131   Person(string& n, double p)
132   : name(n), favBook(p) { }
133 private:
134   string name;
135   Book favBook;
136 };
137
138 class Book {
139 public:
140   Book(double price_in)
141   : price(price_in) { }
142 private:
143   double price;
144 };
145
146 class Person {
147 public:
148   Person(string& n, double p)
149   : name(n), favBook(p) { }
150 private:
151   string name;
152   Book favBook;
153 };
154
155 class Book {
156 public:
157   Book(double price_in)
158   : price(price_in) { }
159 private:
160   double price;
161 };
162
163 class Person {
164 public:
165   Person(string& n, double p)
166   : name(n), favBook(p) { }
167 private:
168   string name;
169   Book favBook;
170 };
171
172 class Book {
173 public:
174   Book(double price_in)
175   : price(price_in) { }
176 private:
177   double price;
178 };
179
180 class Person {
181 public:
182   Person(string& n, double p)
183   : name(n), favBook(p) { }
184 private:
185   string name;
186   Book favBook;
187 };
188
189 class Book {
190 public:
191   Book(double price_in)
192   : price(price_in) { }
193 private:
194   double price;
195 };
196
197 class Person {
198 public:
199   Person(string& n, double p)
200   : name(n), favBook(p) { }
201 private:
202   string name;
203   Book favBook;
204 };
205
206 class Book {
207 public:
208   Book(double price_in)
209   : price(price_in) { }
210 private:
211   double price;
212 };
213
214 class Person {
215 public:
216   Person(string& n, double p)
217   : name(n), favBook(p) { }
218 private:
219   string name;
220   Book favBook;
221 };
222
223 class Book {
224 public:
225   Book(double price_in)
226   : price(price_in) { }
227 private:
228   double price;
229 };
230
231 class Person {
232 public:
233   Person(string& n, double p)
234   : name(n), favBook(p) { }
235 private:
236   string name;
237   Book favBook;
238 };
239
240 class Book {
241 public:
242   Book(double price_in)
243   : price(price_in) { }
244 private:
245   double price;
246 };
247
248 class Person {
249 public:
250   Person(string& n, double p)
251   : name(n), favBook(p) { }
252 private:
253   string name;
254   Book favBook;
255 };
256
257 class Book {
258 public:
259   Book(double price_in)
260   : price(price_in) { }
261 private:
262   double price;
263 };
264
265 class Person {
266 public:
267   Person(string& n, double p)
268   : name(n), favBook(p) { }
269 private:
270   string name;
271   Book favBook;
272 };
273
274 class Book {
275 public:
276   Book(double price_in)
277   : price(price_in) { }
278 private:
279   double price;
280 };
281
282 class Person {
283 public:
284   Person(string& n, double p)
285   : name(n), favBook(p) { }
286 private:
287   string name;
288   Book favBook;
289 };
290
291 class Book {
292 public:
293   Book(double price_in)
294   : price(price_in) { }
295 private:
296   double price;
297 };
298
299 class Person {
300 public:
301   Person(string& n, double p)
302   : name(n), favBook(p) { }
303 private:
304   string name;
305   Book favBook;
306 };
307
308 class Book {
309 public:
310   Book(double price_in)
311   : price(price_in) { }
312 private:
313   double price;
314 };
315
316 class Person {
317 public:
318   Person(string& n, double p)
319   : name(n), favBook(p) { }
320 private:
321   string name;
322   Book favBook;
323 };
324
325 class Book {
326 public:
327   Book(double price_in)
328   : price(price_in) { }
329 private:
330   double price;
331 };
332
333 class Person {
334 public:
335   Person(string& n, double p)
336   : name(n), favBook(p) { }
337 private:
338   string name;
339   Book favBook;
340 };
341
342 class Book {
343 public:
344   Book(double price_in)
345   : price(price_in) { }
346 private:
347   double price;
348 };
349
350 class Person {
351 public:
352   Person(string& n, double p)
353   : name(n), favBook(p) { }
354 private:
355   string name;
356   Book favBook;
357 };
358
359 class Book {
360 public:
361   Book(double price_in)
362   : price(price_in) { }
363 private:
364   double price;
365 };
366
367 class Person {
368 public:
369   Person(string& n, double p)
370   : name(n), favBook(p) { }
371 private:
372   string name;
373   Book favBook;
374 };
375
376 class Book {
377 public:
378   Book(double price_in)
379   : price(price_in) { }
380 private:
381   double price;
382 };
383
384 class Person {
385 public:
386   Person(string& n, double p)
387   : name(n), favBook(p) { }
388 private:
389   string name;
390   Book favBook;
391 };
392
393 class Book {
394 public:
395   Book(double price_in)
396   : price(price_in) { }
397 private:
398   double price;
399 };
400
401 class Person {
402 public:
403   Person(string& n, double p)
404   : name(n), favBook(p) { }
405 private:
406   string name;
407   Book favBook;
408 };
409
410 class Book {
411 public:
412   Book(double price_in)
413   : price(price_in) { }
414 private:
415   double price;
416 };
417
418 class Person {
419 public:
420   Person(string& n, double p)
421   : name(n), favBook(p) { }
422 private:
423   string name;
424   Book favBook;
425 };
426
427 class Book {
428 public:
429   Book(double price_in)
430   : price(price_in) { }
431 private:
432   double price;
433 };
434
435 class Person {
436 public:
437   Person(string& n, double p)
438   : name(n), favBook(p) { }
439 private:
440   string name;
441   Book favBook;
442 };
443
444 class Book {
445 public:
446   Book(double price_in)
447   : price(price_in) { }
448 private:
449   double price;
450 };
451
452 class Person {
453 public:
454   Person(string& n, double p)
455   : name(n), favBook(p) { }
456 private:
457   string name;
458   Book favBook;
459 };
460
461 class Book {
462 public:
463   Book(double price_in)
464   : price(price_in) { }
465 private:
466   double price;
467 };
468
469 class Person {
470 public:
471   Person(string& n, double p)
472   : name(n), favBook(p) { }
473 private:
474   string name;
475   Book favBook;
476 };
477
478 class Book {
479 public:
480   Book(double price_in)
481   : price(price_in) { }
482 private:
483   double price;
484 };
485
486 class Person {
487 public:
488   Person(string& n, double p)
489   : name(n), favBook(p) { }
490 private:
491   string name;
492   Book favBook;
493 };
494
495 class Book {
496 public:
497   Book(double price_in)
498   : price(price_in) { }
499 private:
500   double price;
501 };
502
503 class Person {
504 public:
505   Person(string& n, double p)
506   : name(n), favBook(p) { }
507 private:
508   string name;
509   Book favBook;
510 };
511
512 class Book {
513 public:
514   Book(double price_in)
515   : price(price_in) { }
516 private:
517   double price;
518 };
519
520 class Person {
521 public:
522   Person(string& n, double p)
523   : name(n), favBook(p) { }
524 private:
525   string name;
526   Book favBook;
527 };
528
529 class Book {
530 public:
531   Book(double price_in)
532   : price(price_in) { }
533 private:
534   double price;
535 };
536
537 class Person {
538 public:
539   Person(string& n, double p)
540   : name(n), favBook(p) { }
541 private:
542   string name;
543   Book favBook;
544 };
545
546 class Book {
547 public:
548   Book(double price_in)
549   : price(price_in) { }
550 private:
551   double price;
552 };
553
554 class Person {
555 public:
556   Person(string& n, double p)
557   : name(n), favBook(p) { }
558 private:
559   string name;
560   Book favBook;
561 };
562
563 class Book {
564 public:
565   Book(double price_in)
566   : price(price_in) { }
567 private:
568   double price;
569 };
570
571 class Person {
572 public:
573   Person(string& n, double p)
574   : name(n), favBook(p) { }
575 private:
576   string name;
577   Book favBook;
578 };
579
580 class Book {
581 public:
582   Book(double price_in)
583   : price(price_in) { }
584 private:
585   double price;
586 };
587
588 class Person {
589 public:
590   Person(string& n, double p)
591   : name(n), favBook(p) { }
592 private:
593   string name;
594   Book favBook;
595 };
596
597 class Book {
598 public:
599   Book(double price_in)
600   : price(price_in) { }
601 private:
602   double price;
603 };
604
605 class Person {
606 public:
607   Person(string& n, double p)
608   : name(n), favBook(p) { }
609 private:
610   string name;
611   Book favBook;
612 };
613
614 class Book {
615 public:
616   Book(double price_in)
617   : price(price_in) { }
618 private:
619   double price;
620 };
621
622 class Person {
623 public:
624   Person(string& n, double p)
625   : name(n), favBook(p) { }
626 private:
627   string name;
628   Book favBook;
629 };
630
631 class Book {
632 public:
633   Book(double price_in)
634   : price(price_in) { }
635 private:
636   double price;
637 };
638
639 class Person {
640 public:
641   Person(string& n, double p)
642   : name(n), favBook(p) { }
643 private:
644   string name;
645   Book favBook;
646 };
647
648 class Book {
649 public:
650   Book(double price_in)
651   : price(price_in) { }
652 private:
653   double price;
654 };
655
656 class Person {
657 public:
658   Person(string& n, double p)
659   : name(n), favBook(p) { }
660 private:
661   string name;
662   Book favBook;
663 };
664
665 class Book {
666 public:
667   Book(double price_in)
668   : price(price_in) { }
669 private:
670   double price;
671 };
672
673 class Person {
674 public:
675   Person(string& n, double p)
676   : name(n), favBook(p) { }
677 private:
678   string name;
679   Book favBook;
680 };
681
682 class Book {
683 public:
684   Book(double price_in)
685   : price(price_in) { }
686 private:
687   double price;
688 };
689
690 class Person {
691 public:
692   Person(string& n, double p)
693   : name(n), favBook(p) { }
694 private:
695   string name;
696   Book favBook;
697 };
698
699 class Book {
700 public:
701   Book(double price_in)
702   : price(price_in) { }
703 private:
704   double price;
705 };
706
707 class Person {
708 public:
709   Person(string& n, double p)
710   : name(n), favBook(p) { }
711 private:
712   string name;
713   Book favBook;
714 };
715
716 class Book {
717 public:
718   Book(double price_in)
719   : price(price_in) { }
720 private:
721   double price;
722 };
723
724 class Person {
725 public:
726   Person(string& n, double p)
727   : name(n), favBook(p) { }
728 private:
729   string name;
730   Book favBook;
731 };
732
733 class Book {
734 public:
735   Book(double price_in)
736   : price(price_in) { }
737 private:
738   double price;
739 };
740
741 class Person {
742 public:
743   Person(string& n, double p)
744   : name(n), favBook(p) { }
745 private:
746   string name;
747   Book favBook;
748 };
749
750 class Book {
751 public:
752   Book(double price_in)
753   : price(price_in) { }
754 private:
755   double price;
756 };
757
758 class Person {
759 public:
760   Person(string& n, double p)
761   : name(n), favBook(p) { }
762 private:
763   string name;
764   Book favBook;
765 };
766
767 class Book {
768 public:
769   Book(double price_in)
770   : price(price_in) { }
771 private:
772   double price;
773 };
774
775 class Person {
776 public:
777   Person(string& n, double p)
778   : name(n), favBook(p) { }
779 private:
780   string name;
781   Book favBook;
782 };
783
784 class Book {
785 public:
786   Book(double price_in)
787   : price(price_in) { }
788 private:
789   double price;
790 };
791
792 class Person {
793 public:
794   Person(string& n, double p)
795   : name(n), favBook(p) { }
796 private:
797   string name;
798   Book favBook;
799 };
800
801 class Book {
802 public:
803   Book(double price_in)
804   : price(price_in) { }
805 private:
806   double price;
807 };
808
809 class Person {
810 public:
811   Person(string& n, double p)
812   : name(n), favBook(p) { }
813 private:
814   string name;
815   Book favBook;
816 };
817
818 class Book {
819 public:
820   Book(double price_in)
821   : price(price_in) { }
822 private:
823   double price;
824 };
825
826 class Person {
827 public:
828   Person(string& n, double p)
829   : name(n), favBook(p) { }
830 private:
831   string name;
832   Book favBook;
833 };
834
835 class Book {
836 public:
837   Book(double price_in)
838   : price(price_in) { }
839 private:
840   double price;
841 };
842
843 class Person {
844 public:
845   Person(string& n, double p)
846   : name(n), favBook(p) { }
847 private:
848   string name;
849   Book favBook;
850 };
851
852 class Book {
853 public:
854   Book(double price_in)
855   : price(price_in) { }
856 private:
857   double price;
858 };
859
860 class Person {
861 public:
862   Person(string& n, double p)
863   : name(n), favBook(p) { }
864 private:
865   string name;
866   Book favBook;
867 };
868
869 class Book {
870 public:
871   Book(double price_in)
872   : price(price_in) { }
873 private:
874   double price;
875 };
876
877 class Person {
878 public:
879   Person(string& n, double p)
880   : name(n), favBook(p) { }
881 private:
882   string name;
883   Book favBook;
884 };
885
886 class Book {
887 public:
888   Book(double price_in)
889   : price(price_in) { }
890 private:
891   double price;
892 };
893
894 class Person {
895 public:
896   Person(string& n, double p)
897   : name(n), favBook(p) { }
898 private:
899   string name;
900   Book favBook;
901 };
902
903 class Book {
904 public:
905   Book(double price_in)
906   : price(price_in) { }
907 private:
908   double price;
909 };
910
911 class Person {
912 public:
913   Person(string& n, double p)
914   : name(n), favBook(p) { }
915 private:
916   string name;
917   Book favBook;
918 };
919
920 class Book {
921 public:
922   Book(double price_in)
923   : price(price_in) { }
924 private:
925   double price;
926 };
927
928 class Person {
929 public:
930   Person(string& n, double p)
931   : name(n), favBook(p) { }
932 private:
933   string name;
934   Book favBook;
935 };
936
937 class Book {
938 public:
939   Book(double price_in)
940   : price(price_in) { }
941 private:
942   double price;
943 };
944
945 class Person {
946 public:
947   Person(string& n, double p)
948   : name(n), favBook(p) { }
949 private:
950   string name;
951   Book favBook;
952 };
953
954 class Book {
955 public:
956   Book(double price_in)
957   : price(price_in) { }
958 private:
959   double price;
960 };
961
962 class Person {
963 public:
964   Person(string& n, double p)
965   : name(n), favBook(p) { }
966 private:
967   string name;
968   Book favBook;
969 };
970
971 class Book {
972 public:
973   Book(double price_in)
974   : price(price_in) { }
975 private:
976   double price;
977 };
978
979 class Person {
980 public:
981   Person(string& n, double p)
982   : name(n), favBook(p) { }
983 private:
984   string name;
985   Book favBook;
986 };
987
988 class Book {
989 public:
990   Book(double price_in)
991   : price(price_in) { }
992 private:
993   double price;
994 };
995
996 class Person {
997 public:
998   Person(string& n, double p)
999   : name(n), favBook(p) { }
1000 private:
1001   string name;
1002   Book favBook;
1003 };
1004
1005 class Book {
1006 public:
1007   Book(double price_in)
1008   : price(price_in) { }
1009 private:
1010   double price;
1011 };
1012
1013 class Person {
1014 public:
1015   Person(string& n, double p)
1016   : name(n), favBook(p) { }
1017 private:
1018   string name;
1019   Book favBook;
1020 };
1021
1022 class Book {
1023 public:
1024   Book(double price_in)
1025   : price(price_in) { }
1026 private:
1027   double price;
1028 };
1029
1030 class Person {
1031 public:
1032   Person(string& n, double p)
1033   : name(n), favBook(p) { }
1034 private:
1035   string name;
1036   Book favBook;
1037 };
1038
1039 class Book {
1040 public:
1041   Book(double price_in)
1042   : price(price_in) { }
1043 private:
1044   double price;
1045 };
1046
1047 class Person {
1048 public:
1049   Person(string& n, double p)
1050   : name(n), favBook(p) { }
1051 private:
1052   string name;
1053   Book favBook;
1054 };
1055
1056 class Book {
1057 public:
1058   Book(double price_in)
1059   : price(price_in) { }
1060 private:
1061   double price;
1062 };
1063
1064 class Person {
1065 public:
1066   Person(string& n, double p)
1067   : name(n), favBook(p) { }
1068 private:
1069   string name;
1070   Book favBook;
1071 };
1072
1073 class Book {
1074 public:
1075   Book(double price_in)
1076   : price(price_in) { }
1077 private:
1078   double price;
1079 };
1080
1081 class Person {
1082 public:
1083   Person(string& n, double p)
1084   : name(n), favBook(p) { }
1085 private:
1086   string name;
1087   Book favBook;
1088 };
1089
1090 class Book {
1091 public:
1092   Book(double price_in)
1093   : price(price_in) { }
1094 private:
1095   double price;
1096 };
1097
1098 class Person {
1099 public:
1100   Person(string& n, double p)
1101   : name(n), favBook(p) { }
1102 private:
1103   string name;
1104   Book favBook;
1105 };
1106
1107 class Book {
1108 public:
1109   Book(double price_in)
1110   : price(price_in) { }
1111 private:
1112   double price;
1113 };
1114
1115 class Person {
1116 public:
1117   Person(string& n, double p)
1118   : name(n), favBook(p) { }
1119 private:
1120   string name;
1121   Book favBook;
1122 };
1123
1124 class Book {
1125 public:
1126   Book(double price_in)
1127   : price(price_in) { }
1128 private:
1129   double price;
1130 };
1131
1132 class Person {
1133 public:
1134   Person(string& n, double p)
1135   : name(n), favBook(p) { }
1136 private:
1137   string name;
1138   Book favBook;
1139 };
1140
1141 class Book {
1142 public:
1143   Book(double price_in)
1144   : price(price_in) { }
1145 private:
1146   double price;
1147 };
1148
1149 class Person {
1150 public:
1151   Person(string& n, double p)
1152   : name(n), favBook(p) { }
1153 private:
1154   string name;
1155   Book favBook;
1156 };
1157
1158 class Book {
1159 public:
1160   Book(double price_in)
1161   : price(price_in) { }
1162 private:
1163   double price;
1164 };
1165
1166 class Person {
1167 public:
1168   Person(string& n, double p)
1169   : name(n), favBook(p) { }
1170 private:
1171   string name;
1172   Book favBook;
1173 };
1174
1175 class Book {
1176 public:
1177   Book(double price_in)
1178   : price(price_in) { }
1179 private:
1180   double price;
1181 };
1182
1183 class Person {
1184 public:
1185   Person(string& n, double p)
1186   : name(n), favBook(p) { }
1187 private:
1188   string name;
1189   Book favBook;
1190 };
1191
1192 class Book {
1193 public:
1194   Book(double price_in)
1195   : price(price_in) { }
1196 private:
1197   double price;
1198 };
1199
1200 class Person {
1201 public:
1202   Person(string& n, double p)
1203   : name(n), favBook(p) { }
1204 private:
1205   string name;
1206   Book favBook;
1207 };
1208
1209 class Book {
1210 public:
1211   Book(double price_in)
1212   : price(price_in) { }
1213 private:
1214   double price;
1215 };
1216
1217 class Person {
1218 public:
1219   Person(string& n, double p)
1220   : name(n), favBook(p) { }
1221 private:
1222   string name;
1223   Book favBook;
1224 };
1225
1226 class Book {
1227 public:
1228   Book(double price_in)
1229   : price(price_in) { }
1230 private:
1231   double price;
1232 };
1233
1234 class Person {
1235 public:
1236   Person(string& n, double p)
1237   : name(n), favBook(p) { }
1238 private:
1239   string name;
1240   Book favBook;
1241 };
1242
1243 class Book {
1244 public:
1245   Book(double price_in)
1246   : price(price_in) { }
1247 private:
1248   double price;
1249 };
1250
1251 class Person {
1252 public:
1253   Person(string& n, double p)
1254   : name(n), favBook(p) { }
1255 private:
1256   string name;
1257   Book favBook;
1258 };
1259
1260 class Book {
1261 public:
1262   Book(double price_in)
1263   : price(price_in) { }
1264 private:
1265   double price;
1266 };
1267
1268 class Person {
1269 public:
1270   Person(string& n, double p)
1271   : name(n), favBook(p) { }
1272 private:
1273   string name;
1274   Book favBook;
1275 };
1276
1277 class Book {
1278 public:
1279   Book(double price_in)
1280   : price(price_in) { }
1281 private:
1282   double price;
1283 };
1284
1285 class Person {
1286 public:
1287   Person(string& n, double p)
1288   : name(n), favBook(p) { }
1289 private:
1290   string name;
1291   Book favBook;
1292 };
1293
1294 class Book {
1295 public:
1296   Book(double price_in)
1297   : price(price_in) { }
1298 private:
1299   double price;
1300 };
1301
1302 class Person {
1303 public:
1304   Person(string& n, double p)
1305   : name(n), favBook(p) { }
1306 private:
1307   string name;
1308   Book favBook;
1309 };
1310
1311 class Book {
1312 public:
1313   Book(double price_in)
1314   : price(price_in) { }
1315 private:
1316   double price;
1317 };
1318
1319 class Person {
1320 public:
1321   Person(string& n, double p)
1322   : name(n), favBook(p) { }
1323 private:
1324   string name;
1325   Book favBook;
1326 };
1327
1328 class Book {
1329 public:
1330   Book(double price_in)
1331   : price(price_in) { }
1332 private:
1333   double price;
1334 };
1335
1336 class Person {
1337 public:
1338   Person(string& n, double p)
1339   : name(n), favBook(p) { }
1340 private:
1341   string name;
1342   Book favBook;
1343 };
1344
1345 class Book {
1346 public:
1347   Book(double price_in)
1348   : price(price_in) { }
1349 private:
1350   double price;
1351 };
1352
1353 class Person {
1354 public:
1355   Person(string& n, double p)
1356   : name(n), favBook(p) { }
1357 private:
1358   string name;
1359   Book favBook;
1360 };
1361
1362 class Book {
1363 public:
1364   Book(double price_in)
1365   : price(price_in) { }
1366 private:
1367   double price;
1368 };
1369
1370 class Person {
1371 public:
1372   Person(string& n, double p)
1373   : name(n), favBook(p) { }
1374 private:
1375   string name;
1376   Book favBook;
1377 };
1378
1379 class Book {
1380 public:
1381   Book(double price_in)
1382   : price(price_in) { }
1383 private:
1384   double price;
1385 };
1386
1387 class Person {
1388 public:
1389   Person(string& n, double p)
1390   : name(n), favBook(p) { }
1391 private:
1392   string name;
1393   Book favBook;
1394 };
1395
1396 class Book {
1397 public:
1398   Book(double price_in)
1399   : price(price_in) { }
1400 private:
1401   double price;
1402 };
1403
1404 class Person {
1405 public:
1406   Person(string& n, double p)
1407   : name(n), favBook(p) { }
1408 private:
1409   string name;
1410   Book favBook;
1411 };
1412
1413 class Book {
1414 public:
1415   Book(double price_in)
1416   : price(price_in) { }
1417 private:
1418   double price;
1419 };
1420
1421 class Person {
1422 public:
1423   Person(string& n, double p)
1424   : name(n), favBook(p) { }
1425 private:
1426   string name;
1427   Book favBook;
1428 };
1429
1430 class Book {
1431 public:
1432   Book(double price_in)
1433   : price(price_in) { }
1434 private:
1435   double price;
1436 };
1437
1438 class Person {
1439 public:
1440   Person(string& n, double p)
1441   : name(n), favBook(p) { }
1442 private:
1443   string name;
1444   Book favBook;
1445 };
1446
1447 class Book {
1448 public:
1449   Book(double price_in)
1450   : price(price_in) { }
1451 private:
1452   double price;
1453 };
1454
1455 class Person {
1456 public:
```