Arrays and Dynamic Memory Allocation
Figure 2.1

(a) `shared_ptr<double> p;`
The variable `p = make_shared<double>();` declares the variable `p` as being a pointer attached to a dashed triangle, which represents the special `nullptr` value.

It is now considered good practice to use an alternative representation for `nullptr` called `0`. In the C and C++ languages, the pointer value `0` is reserved as an unsigned integer. In the C++ standard library, `nullptr` is a type that contains `0`.

Figure 2.1 shows the effect of the memory allocation. You can combine the declaration and allocation as follows:

```cpp
shared_ptr<double> p;
```

The statement `*p = 5.7;` assigns a value to the object pointed to by `p`.

The `make_shared<>()` function allocates storage from the heap for a double precision real value and returns a pointer to it.

The `shared_ptr` type is used in C++ to declare a shared pointer variable. The `shared_ptr` type is a class that stores a pointer to an object owned by another object. It also keeps track of the number of references to the object it points to.

When the number of references to an object decreases to zero, the object is automatically deleted. The `shared_ptr` class provides an easy way to manage memory without using raw pointers. It automatically initializes to `nullptr` as a pointer attached to a dashed triangle. Figure 2.1 shows the state of the variable `p` before and after its declaration.
The type is used in C++ to declare a shared pointer variable. The `shared_ptr` declaration allocates storage from the heap for a double precision value and returns a pointer to it. Smart pointers are automatically initialized to `nullptr` as being a pointer attached to a dashed triangle, which represents the special value. As a cenitve because each can never be allocated from the heap.

Before you can store a double precision value in the variable, you must allocate storage for a pointer to a double precision real, and assigning this storage to a variable.

```
shared_ptr<double> p;
```

You can access dynamically allocated storage with shared pointers just like you can with raw pointers. Here is an example of how to declare a shared pointer with the equivalent operation for raw pointers. There are operations for raw pointers. There are referenc-

```
*p = 5.7;
```

When the number of references to the object decreases to zero so that the shared object can no longer be accessed, the system automatically deletes the object from the heap.

Deletion of a shared object is done automatically by the execution system using a reference counting algorithm that keeps track of how many

```
nullptr
```

deletes the object from the heap.

```
p = make_shared<double>();
```

pointers point to a shared object. When the number of reference to the object decreases to zero so that the shared object can no longer be accessed, the system automatically deletes the object from the heap. Deletion of a shared object is done automatically by the execution system using a reference counting algorithm that keeps track of how many

```
*p = 5.7;
```

As a share pointer takes the place of the `new` operator in place of 0 called `nullptr`, it is now considered good practice to use an alternative representation for 0 called `nullptr` as a replacement for 0.

```
shared_ptr<double> p;
```

As a share pointer takes the place of the `new` operator in place of 0 called `nullptr`, it is now considered good practice to use an alternative representation for 0 called `nullptr` as a replacement for 0.

```
shared_ptr<double> p = make_shared<double>();
```

As a share pointer takes the place of the `new` operator in place of 0 called `nullptr`, it is now considered good practice to use an alternative representation for 0 called `nullptr` as a replacement for 0.

```
shared_ptr<double> p;
```

As a share pointer takes the place of the `new` operator in place of 0 called `nullptr`, it is now considered good practice to use an alternative representation for 0 called `nullptr` as a replacement for 0.

```
*p = 5.7;
```

As a share pointer takes the place of the `new` operator in place of 0 called `nullptr`, it is now considered good practice to use an alternative representation for 0 called `nullptr` as a replacement for 0.

```
(p) shared_ptr<double> p;
```

As a share pointer takes the place of the `new` operator in place of 0 called `nullptr`, it is now considered good practice to use an alternative representation for 0 called `nullptr` as a replacement for 0.

```
p = make_shared<double>();
```

As a share pointer takes the place of the `new` operator in place of 0 called `nullptr`, it is now considered good practice to use an alternative representation for 0 called `nullptr` as a replacement for 0.

```
*p = 5.7;
```

As a share pointer takes the place of the `new` operator in place of 0 called `nullptr`, it is now considered good practice to use an alternative representation for 0 called `nullptr` as a replacement for 0.
Design Patterns for Data Structures

2.1 Pointer Types

shared_ptr<int> pI = make_shared<int>();
shared_ptr<int> pJ = make_shared<int>();
shared_ptr<int> pK;
*pI = 5;
*pJ = 3;
pK = pI;
pI = pJ;
pI = 2 + *pK;
cout << *pI << " " << *pJ << " " << *pK << endl;
The `*` operator is used to dereference a pointer. When placed before a pointer variable, the notation indicates the cell to which the pointer points. For example, the statement

```
*pI = 5.7;
```

assigns the value 5.7 to the memory cell to which `pI` points, as in Figure 2.1(c). From the hardware point of view, `pI` is the memory address of the location where the value is stored, and `*pI` is the memory location itself.

You can combine the declaration, allocation, and initialization all in one statement as follows.

```
shared_ptr<double> p = make_shared<double>(5.7);
```

The function `make_shared<>()` has an optional parameter for initializing the value to which the pointer points.

You can assign one pointer to another, but you must be careful to consider the effect of such an assignment. Because a pointer "points to" an item, if you give the pointer's value to a second pointer, the second pointer will point to the same item to which the first pointer points. Consider the following code fragment, illustrated in Figure 2.2.

```
shared_ptr<int> pI = make_shared<int>();
shared_ptr<int> pJ = make_shared<int>();
shared_ptr<int> pK;
*pI = 5;
*pJ = 3;
pK = pI;
pI = pJ;
*pI = 2 + *pK;
```

The assignment of `pI` to `pK` is a pointer assignment, not an integer assignment. It makes `pK` point to the same cell to which `pI` points as shown in Figure 2.2(b). There is no change of any cell content. Similarly, the assignment of `pJ` to `pI` makes `pI` point to the same cell to which `pJ` points as in Figure 2.2(c).

Execution of the output statement streams

```
cout << *pI << " " << *pJ << " " << *pK << endl;
```

Revised: July 20, 2021

Copyright ©: 1998, Dung X. Nguyen and J. Stanley Warford
The * operator is used to dereference a pointer. When placed before a pointer variable, the notation indicates the cell to which the pointer points. For example, the statement 

\[
*pI = 5.7;
\]

assigns the value 5.7 to the memory cell to which \( p \) points, as in Figure 2.1(c). From the hardware point of view, \( p \) is the memory address of the location where the value is stored, and \( *p \) is the memory location itself.

You can combine the declaration, allocation, and initialization all in one statement as follows.

\[
\text{shared\_ptr\_double } p = \text{make\_shared\_double}(5.7);
\]

The function \( \text{make\_shared\_()} \) has an optional parameter for initializing the value to which the pointer points.

You can assign one pointer to another, but you must be careful to consider the effect of such an assignment. Because a pointer “points to” an item, if you give the pointer’s value to a second pointer, the second pointer will point to the same item to which the first pointer points. Consider the following code fragment, illustrated in Figure 2.2.

\[
\text{shared\_ptr\_int } pI = \text{make\_shared\_int}();
\]
\[
\text{shared\_ptr\_int } pJ = \text{make\_shared\_int}();
\]
\[
\text{shared\_ptr\_int } pK;
\]
\[
*pI = 5;
\]
\[
*pJ = 3;
\]
\[
pK = pI;
\]
\[
pI = pJ;
\]
\[
*pI = 2 + *pK;
\]

\[
\text{cout } \ll *pI \ll \" \ ll *pJ \ll \" \ ll *pK \ll \n;
\]

The assignment of \( pI \) to \( pK \) is a pointer assignment, not an integer assignment. It makes \( pK \) point to the same cell to which \( pI \) points as shown in Figure 2.2(b). There is no change of any cell content. Similarly, the assignment of \( pJ \) to \( pI \) makes \( pI \) point to the same cell to which \( pJ \) points as in Figure 2.2(c). Execution of the output statement streams
The * operator is used to dereference a pointer. When placed before a pointer variable, the notation indicates the cell to which the pointer points. For example, the statement

\[
*pI = 5.7;
\]

assigns the value 5.7 to the memory cell to which \( p \) points, as in Figure 2.1(c). From the hardware point of view, \( p \) is the memory address of the location where the value is stored, and \( *p \) is the memory location itself.

You can combine the declaration, allocation, and initialization all in one statement as follows.

\[
\text{shared\_ptr<double> } p = \text{make\_shared<double>}(5.7);
\]

The function \( \text{make\_shared<>()} \) has an optional parameter for initializing the value to which the pointer points.

You can assign one pointer to another, but you must be careful to consider the effect of such an assignment. Because a pointer “points to” an item, if you give the pointer’s value to a second pointer, the second pointer will point to the same item to which the first pointer points. Consider the following code fragment, illustrated in Figure 2.2.

\[
\begin{align*}
\text{shared\_ptr<int> } pI & = \text{make\_shared<int>}(); \\
\text{shared\_ptr<int> } pJ & = \text{make\_shared<int>}(); \\
\text{shared\_ptr<int> } pK; \\
*pI & = 5; \\
*pJ & = 3; \\
pK & = pI; \\
pI & = pJ; \\
*pI & = 2 + *pK;
\end{align*}
\]

\[
\text{cout} \ll *pI \ll " \" \ll *pJ \ll " \" \ll *pK \ll \text{endl};
\]

The assignment of \( pI \) to \( pK \) is a pointer assignment, not an integer assignment. It makes \( pK \) point to the same cell to which \( pI \) points as shown in Figure 2.2(b). There is no change of any cell content. Similarly, the assignment of \( pJ \) to \( pI \) makes \( pI \) point to the same cell to which \( pJ \) points as in Figure 2.2(c). Execution of the output statement streams
The \* operator is used to dereference a pointer. When placed before a pointer variable, the notation indicates the cell to which the pointer points. For example, the statement \*pI = 5.7; assigns the value 5.7 to the memory cell to which pI points, as in Figure 2.1(c). From the hardware point of view, pI is the memory address of the location where the value is stored, and \*pI is the memory location itself.

You can combine the declaration, allocation, and initialization all in one statement as follows.

```cpp
class Example {  
public:  
    Example() : value(0) {}  
    void setValue(int v) { value = v; }  
    int getValue() const { return value; }  
private:  
    int value;  
};
```
Figure 2.2

Allocating storage for pointers to integers and performing integer and pointer assignments.

so that \( b \) is a double, no tap o i n t e r .

You can assign one pointer to another, but you must be careful to consider the effect of such an assignment. Because a pointer "points to" an item, if you give the pointer's value to a second pointer, the second pointer will point to the same item to which the first pointer points. Consider the following code fragment, illustrated in Figure 2.2.

```cpp
int *pI = new int;
int *pJ = new int;
int *pK;
*pI = 5;
pJ = 3;
pK = pI;
pI = pJ;
*pI = 2 + *pK;
cout << *pI << " " << *pJ << " " << *pK << endl;
delete pI;
delete pK;
```

The assignment of \( pI \) to \( pK \) is a pointer assignment, not an integer assignment. It makes \( pK \) point to the same cell to which \( pI \) points as shown in Figure 2.2(b). There is no change of any cell content. Similarly, the assignment of \( pJ \) to \( pI \) makes \( pI \) point to the same cell to which \( pJ \) points as in Figure 2.2(c). Execution of the output statement streams \( 775 \) to \( cout \). Because \( pI \) and \( pJ \) now point to the same memory cell, the cell containing 7, its value is printed twice.
```
struct Node {
    int value;
    shared_ptr<Node> next;
};
shared_ptr<Node> first, p;
```
struct Node {
    int value;
    shared_ptr<Node> next;
};

shared_ptr<Node> first, p;

p->value

is equivalent to

(*p).value
### Design Patterns for Data Structures

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>::</code></td>
<td>left to right</td>
</tr>
<tr>
<td><code>[ ] () -&gt; . po ++ po --</code></td>
<td>left to right</td>
</tr>
<tr>
<td><code>! ~ pr ++ pr -- un + un - un * un &amp; new delete sizeof (type)</code></td>
<td>right to left</td>
</tr>
<tr>
<td><code>* / %</code></td>
<td>left to right</td>
</tr>
<tr>
<td><code>+ -</code></td>
<td>left to right</td>
</tr>
<tr>
<td><code>&lt;&lt; &gt;&gt;</code></td>
<td>left to right</td>
</tr>
<tr>
<td><code>&lt; &lt;= &gt; &gt;=</code></td>
<td>left to right</td>
</tr>
<tr>
<td><code>== !=</code></td>
<td>left to right</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>left to right</td>
</tr>
<tr>
<td><code>^</code></td>
<td>left to right</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>left to right</td>
</tr>
<tr>
<td>`</td>
<td></td>
</tr>
<tr>
<td><code>tr ?:</code></td>
<td>right to left</td>
</tr>
<tr>
<td>`= += -= *= /=</td>
<td>= %= &amp;= ^= &lt;&lt;= &gt;&gt;=`</td>
</tr>
<tr>
<td><code>,</code></td>
<td>left to right</td>
</tr>
</tbody>
</table>
first = make_shared<Node>();
first->value = 7;
p = first;
first = make_shared<Node>();
first->value = 4;
first->next = p;
for (shared_ptr<Node> q = first; q != nullptr; q = q->next) {
    cout << q->value << " ";
}

The for loop outputs 47
The loop can be simplified to
for (shared_ptr<Node> q = first; q; q = q->next) {
which is also a common coding pattern in C. The value
nullptr is interpreted as false, and so
q is interpreted as true when it has any value other than
nullptr.

Placeholder type specifiers
C++11 introduced placeholder type specifiers, also known as automatic typedeclaration. When you declare a variable with an initializer you can omit the type of the variable
and replace it with the keyword auto. The compiler will automatically deduce the
type of the variable from the initializer.
For example, instead of declaring
int i = 7;
replace
int with
auto as
auto i = 7;
The compiler will deduce that the type of
i is int from the constant
7 because it does not contain a decimal point. Here are some type deductions provided by
auto.
auto i = 7; int
auto x = 6.2; double
auto p = make_shared<double>(5.7); shared_ptr<double>
auto pI = make_shared<int>(); shared_ptr<int>
auto first = make_shared<Node>(); shared_ptr<Node>
for (auto q = first; q; q = q->next) shared_ptr<Node>
The benefit of using
auto may not seem like much in these simple examples. However, when dealing with complex data structures with involved types using
auto as an abbreviation can shorten and simplify your code.
(a) `first = make_shared<Node>();`

Because `p` and `p` now point to the same memory cell, the cell containing 7, its value is printed twice.

In practice, pointers rarely point to primitive types like `double` and `int`. Instead, they point to a `struct` or a `class`. An example is an `odestructure` for an `inked list` declared as:

```cpp
struct Node {
    int value;
    shared_ptr<Node> next;
};
```

`Variable p` has type `pointer to Node`. Therefore, `*p` has type `Node`, which is a `struct`.

You access a field of a `struct` or a `class` with the period operator `.` which is placed between the reference to the `struct` and its field. Thus, `(*p).value` is the value field of the `struct` to which `p` points, and `(*p).next` is the `next` field of the `struct`.

The parentheses are necessary because the period operator has higher precedence than the `*` operator as Figure A.1 of the Appendix shows. Fortunately, C++ provides the `->` operator, which allows the programmer to combine the `*` operator and the period operator without using parentheses. For example, `p->value` is equivalent to `(*p).value`.

Figure 2.3 is a trace of the execution of the following code fragment, which uses the `->` operator.
Figure 2.3 is a trace of the execution of the following code fragment, which uses the -> operator.

(a) `first = make_shared<Node>();`

(b) `first->Value = 7;`

In practice, pointers rarely point to primitive types like `double` and `int`. Instead, they point to a `struct` or a `class`. An example is an `typedef` for an `linked list` declared as:

```cpp
struct Node {
    int value;
    shared_ptr<Node> next;
};
```

Variable `p` has type `pointer` to `Node`. Therefore, `*p` has type `Node`, which is a `struct`. You access a field of a `struct` or a `class` with the period operator, which is placed between the reference to the `struct` and its field. Thus, `(*p).value` is the value field of the `struct` to which `p` points, and `(*p).next` is the `next` field of the `struct`.
Figure 2.3

(a) first = make_shared<Node>();

(b) first->Value = 7;

(c) p = first;
    first = make_shared<Node>();
Figure 2.3

(a) first = make_shared<Node>();

(b) first->Value = 7;

(c) p = first;
    first = make_shared<Node>();

(d) first->value = 4;
    first->next = p;

Because p and p now point to the same memory cell, the cell containing 7, its value is printed twice.

In practice, pointers rarely point to primitive types like double and int. Instead, they point to a struct or a class. An example is an destructor for an inked list declared as

```
struct Node {
    int value;
    shared_ptr<Node> next;
};
```

Variable p has type pointer to Node. Therefore, *p has type Node, which is a struct. Therefore, (*p).value is the value field of the struct to which p points, and (*p).next is the next field of the struct.

The parentheses are necessary because the period operator has higher precedence than the * operator as Figure A.1 of the Appendix shows. Fortunately, C++ provides the -> operator, which allows the programmer to combine the * operator and the period operator without using parentheses. For example, p->value is equivalent to (*p).value.
Figure 2.3 is a trace of the execution of the following code fragment, which uses the -> operator.

```cpp
first = new Node;
first->value = 7;
first->next = nullptr;
p = first;
first = new Node;
first->value = 4;
first->next = p;
for (Node *q = first; q != nullptr; q = q->next) {
    cout << q->value << " ";
}
delete p;
delete first;
```

The `for` loop outputs `4 7`. The loop can be simplified to

```cpp
for (Node *q = first; q; q = q->next) {
    cout << q->value << " ";
}
```

which is also a common coding pattern in C. The value `nullptr` is interpreted as `false`, and so `q` is interpreted as `true` when it has any value other than `nullptr`.

If a program fails to correctly delete all the unused storage from the heap, unused memory cells will accumulate in the heap and will not be available with later executions.

For example, instead of declaring `i` explicitly to be an integer as

```cpp
int i = 7;
```

replace `int` with `auto` as

```cpp
auto i = 7;
```

The compiler will deduce that the type of `i` is `int` from the constant `7` because it does not contain a decimal point. Here are some type deductions provided by `auto`.

- `auto i = 7;` gives `int`
- `auto x = 6.2;` gives `double`
- `auto p = make_shared<double>(5.7);` gives `shared_ptr<double>`
- `auto pI = make_shared<int>();` gives `shared_ptr<int>`
- `auto first = make_shared<Node>();` gives `shared_ptr<Node>`

for (auto q = first; q; q = q->next) shared_ptr<Node>
for (shared_ptr<Node> q = first; q != nullptr; q = q->next) {

for (shared_ptr<Node> q = first; q; q = q->next) {


Placeholder type specifiers
a.k.a. automatic type deduction

auto i = 7;
auto x = 6.2;
auto p = make_shared<double>(5.7);
auto pI = make_shared<int>();
auto first = make_shared<Node>();
for (auto q = first; q; q = q->next)
C++ parameter passing mechanisms

- Pass by value
- Pass by reference
- Pass by constant reference
```cpp
#include <iostream>
using namespace std;

void swapVal(int g, int h) {
    auto temp = g;
    g = h;
    h = temp;
    cout << "g == " << g << " , h == " << h << endl;
}

int main() {
    auto i = 4;
    auto j = 5;
    swapVal(i, j);
    cout << "i == " << i << " , j == " << j << endl;
    return EXIT_SUCCESS;
}

Output
```

The Appendix describes the three parameter passing mechanisms of C++:

- Pass by value
- Pass by reference
- Pass by constant reference

The general rule is to use pass by reference when you want the function to change the value of the actual parameter, and to use the others when you do not. Following are example programs that show the ramifications of using the different passing mechanisms in a function.

Figure 2.4 shows the effect of passing parameters by value. When the function call executes, the processor copies the value of the actual parameter onto the run-time stack. During execution of the called function, any changes that it makes to the formal parameters it makes to the copies. The actual parameters in the calling function are not affected by the changes, because the changes are made to the copies, not to the original actual parameters. The output in the figure shows that `swapVal()` changes the values of formal parameters `g` and `h` but not the actual parameters `i` and `j`. 
#include <iostream>
using namespace std;

void swapVal(int g, int h) {
    auto temp = g;
    g = h;
    h = temp;
    cout << "g == " << g <<", h == " << h << endl;
}

int main() {
    auto i = 4;
    auto j = 5;
    swapVal(i, j);
    cout << "i == " << i <<", j == " << j << endl;
    return EXIT_SUCCESS;
}

Output

g == 5, h == 4
i == 4, j == 5


```cpp
#include <iostream>
using namespace std;

void swapRef(int &g, int &h) {
    auto temp = g;
    g = h;
    h = temp;
    cout << "g == " << g << ", h == " << h << endl;
}

int main() {
    auto i = 4;
    auto j = 5;
    swapRef(i, j);
    cout << "i == " << i << ", j == " << j << endl;
    return EXIT_SUCCESS;
}

Output
```
#include <iostream>
using namespace std;

void swapRef(int &g, int &h) {
    auto temp = g;
    g = h;
    h = temp;
    cout << "g == " << g << " h == " << h << endl;
}

int main() {
    auto i = 4;
    auto j = 5;
    swapRef(i, j);
    cout << "i == " << i << " j == " << j << endl;
    return EXIT_SUCCESS;
}

Output

g == 5, h == 4
i == 5, j == 4
```cpp
#include <iostream>
using namespace std;

void swapPtrVal(shared_ptr<int> g, shared_ptr<int> h) {
    auto temp = *g; // before
    *g = *h;
    *h = temp;
    cout << "*g == " << *g <<", *h == " << *h << endl; // after
}

int main() {
    auto i = make_shared<int>(4);
    auto j = make_shared<int>(5);
    swapPtrVal(i, j);
    cout << "*i == " << *i <<", *j == " << *j << endl;
    return EXIT_SUCCESS;
}

Output
```
#include <iostream>
using namespace std;

void swapPtrVal(shared_ptr<int> g, shared_ptr<int> h) {
    auto temp = *g; // before
    *g = *h;
    *h = temp;
    cout << "*g == " << *g <<", *h == " << *h << endl; // after
}

int main() {
    auto i = make_shared<int>(4);
    auto j = make_shared<int>(5);
    swapPtrVal(i, j);
    cout << "*i == " << *i <<", *j == " << *j << endl;
    return EXIT_SUCCESS;
}

Output
*g == 5, *h == 4
*i == 5, *j == 4
#include <iostream>
using namespace std;

void swapPtrVal(shared_ptr<int> g, shared_ptr<int> h) {
    auto temp = *g; // before
    *g = *h;
    *h = temp;
    cout << "*g == " << *g << ', ' << *h << endl; // after
}

int main() {
    auto i = make_shared<int>(4);
    auto j = make_shared<int>(5);
    swapPtrVal(i, j);
    cout << "*i == " << *i << ', ' << *j << endl;
    return EXIT_SUCCESS;
}

Output

*g == 5, *h == 4
*i == 5, *j == 4

(a) Passing pointers by value, before.
(b) Passing pointers by value, after.
```cpp
#include <iostream>
using namespace std;

void swapPtrRef(shared_ptr<int> &g, shared_ptr<int> &h) {
    auto temp = g; // before
    g = h;
    h = temp;
    cout << "*g == " << *g << ", *h == " << *h << endl; // after
}

int main() {
    auto i = make_shared<int>(4);
    auto j = make_shared<int>(5);
    swapPtrRef(i, j);
    cout << "*i == " << *i << ", *j == " << *j << endl;
    return EXIT_SUCCESS;
}
```

**Output**

```
*g == 5, *h == 4
*i == 5, *j == 4
```
#include <iostream>
using namespace std;

void swapPtrRef(shared_ptr<int> &g, shared_ptr<int> &h) {
    auto temp = g; // before
    g = h;
    h = temp;
    cout << "*g == " << *g << " , *h == " << *h << endl; // after
}

int main() {
    auto i = make_shared<int>(4);
    auto j = make_shared<int>(5);
    swapPtrRef(i, j);
    cout << "*i == " << *i << " , *j == " << *j << endl;
    return EXIT_SUCCESS;
}

Output
*g == 5, *h == 4
*i == 5, *j == 4
#include <iostream>
using namespace std;

void swapPtrRef(shared_ptr<int> &g, shared_ptr<int> &h) {
    auto temp = g; // before
    g = h; 
    h = temp;
    cout << "*g == " << *g << ", *h == " << *h << endl; // after
}

int main() {
    auto i = make_shared<int>(4);
    auto j = make_shared<int>(5);
    swapPtrRef(i, j);
    cout << "*i == " << *i << ", *j == " << *j << endl;
    return EXIT_SUCCESS;
}

Output

*g == 5, *h == 4
*i == 5, *j == 4

(a) Passing pointers by reference, before.

(b) Passing pointers by reference, after.
Reference types

In the function of Figure 2.5,

```c
void swapRef(int &g, int &h) {
    which is called as follows
    auto i = 4;
    auto j = 5;
    swapRef(i, j);
    int & is known as a reference type.
```
Reference types

auto i = 4;
int &g = i;  // Must be initialized here.
cout << "g == " << g << ", i == " << i << endl;
g = 5;
cout << "g == " << g << ", i == " << i << endl;

is legal and produces the output

g == 4, i == 4
g == 5, i == 5
A classic C++ array

#include <iostream> // istream, ostream.
using namespace std;

void readStream(istream &is, unique_ptr<double[]> *d, int cap, int &len);
// Pre: d is allocated with capacity cap.
// Post: num values are input from is to d[0..num - 1], where
// num == min(number of elements in is, cap).
// len == num.

void writeStream(ostream &os, unique_ptr<double[]> const *d, int cap, int len);
// Pre: d is allocated with capacity cap.
// Post: num values are output from d[0..num - 1] to os, where
// num == min(len, cap).
```cpp
int main() {
    int cap = promptIntGE("Enter array capacity", 1);
    auto arr = make_unique<double[]>(cap);
    ifstream ifs;
promptFileOpen(ifs);
    if (ifs) {
        int length = 0;
        readStream(ifs, &arr, cap, length);
        ifs.close();
        cout << "Read count == " << length << endl;
        cout << "Array data:"
writeStream(cout, &arr, cap, length);
        // arr[2 * cap] = 123.4;
        // cout << arr[2 * cap] << endl;
    }
    return EXIT_SUCCESS;
}
```
```cpp
void readStream(istream &is, unique_ptr<double[]> *d, int cap, int &len) {
    len = 0;
    for (int i = 0; i < cap && is >> (*d)[i]; i++) {
        len++;
    }
}

void writeStream(ostream &os, unique_ptr<double[]> const *d, int cap, int len) {
    for (int i = 0; i < len && i < cap; i++) {
        os.width(12);
        os << (*d)[i];
        if (i % 6 == 5) {
            os << endl;
        }
    }
    os << endl;
}
```
Demo ArrayClassic
Buffer overflow

• C++ allows access to data out of range.

• Malicious software can exploit the failure to check the bounds.

• This is the basis of most software viruses.
Figure 2.10

An abstract sequence class

```cpp
// ========= ASeq =========
template<class T>
class ASeq {
public:
    explicit ASeq(int cap = 0) {}; // Avoid implicit conversion.
    virtual ~ASeq() = default;
    virtual T &operator[](int i) = 0; // For read/write.
    virtual T const &operator[](int i) const = 0; // For read-only.
    virtual int cap() const = 0;
    virtual void toStream(ostream &os) const = 0;
    // Pre: operator<< is defined for T.
    // Post: A string representation of this sequence is streamed to os.

    virtual int fromStream(istream &is) = 0;
    // Pre: operator>> is defined for T.
    // Post: The items of input stream is are appended to this sequence.

Figure 2.10
The template class for an abstract sequence. Partial contents of ASeq.hpp.

An abstract sequence class

Figure 2.10 shows the template class for the ASeq abstract class. An array is a specific example of a more general sequence. The mathematical notion of a sequence is a function $f$ from the integers to a set of data of type $T$ indexed by integers. For example, the sequence 40 20 70 50 10 is a set of data of type int indexed by integers 0..4 with the function $f$ defined as

- $f(0) = 40$,
- $f(1) = 20$,
- $f(2) = 70$,
- $f(3) = 50$,
- $f(4) = 10$.

A specific implementation of an abstract sequence encloses the index in square brackets. For example, if myArr inherits from ASeq and its elements are the integers in the above sequence, then myArr[2] has the value 70.

The code template <class T> class ASeq declares ASeq to be a template class. $T$ is like a formal parameter for the type. The actual parameter could be int, double, or some other type or class. Parameter cap in
```
An abstract sequence class

// ========= operator<< =========
template<class T>
ostream &operator<<(ostream &os, ASeq<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

// ========= operator>> =========
template<class T>
istream &operator>>(istream &is, ASeq<T> &rhs) {
    rhs.fromStream(is);
    return is;
}
Safe arrays and vectors

### ArrayT
- `_data`: unique_ptr<T[]>
- `_cap`: int

### VectorT
- `_data`: unique_ptr<T[]>
- `_cap`: int
- `_size`: int
- `doubleCapacity()`: void
- `append(e: T const &)`: void
- `insert(i: int, e: T const &)`: void
- `remove(i: int)`: T
- `size()`: const int

### ASeq
+ `ASeq(cap: int)`
+ `operator [ ] (i: int): T &`
+ `operator [ ] (i: int): T const &`
+ `cap()`: int
+ `toStream(os: ostream &)`
+ `fromStream(is: istream &)`
A safe array of doubles

typedef ArrayT<double> ArrayDouble;

int main() {
    ArrayDouble arr(promptIntGE("Enter array capacity", 1));
    ifstream ifs;
promptFileOpen(ifs);
    if (ifs) {
        int length = arr.fromStream(ifs);
        ifs.close();
        cout << "Read count == " << length << endl;
        cout << "Array data:" << endl;
        writeFormatted(cout, arr, length, 16, 1, 6);
        // arr[2 * arr.cap()] = 123.4;
        // cout << arr[2 * arr.cap()] << endl;
        return EXIT_SUCCESS;
    }
}
The safe array class

// ======= ArrayT =======
template<class T>
class ArrayT : public ASeq<T> {
private:
    unique_ptr<T[]> _data;
    int _cap;
public:
    explicit ArrayT(int cap = 1);
    int cap() const override;
    T &operator[](int i) override; // For read/write.
    T const &operator[](int i) const override; // For read-only.

    void toStream(ostream &os) const override;
    // Pre: operator<< is defined for T.
    // Post: A string representation of this array is returned
    // to output stream os.

    int fromStream(istream &is) override;
    // Pre: operator>> is defined for T.
    // Post: The items of input stream is are appended to this array.
};
The safe array class

// ========= Constructor =========
template<class T>
ArrayT<T>::ArrayT(int cap) {
    if (cap < 1) {
        cerr << "ArrayT constructor precondition 0 < cap violated."
             << endl;
        cerr << "cap == " << cap << endl;
        throw -1;
    }
    _data = make_unique<T[]>(cap);
    _cap = cap;
}

// ========= cap =========
template<class T>
int ArrayT<T>::cap() const {
    return _cap;
}
Figure 2.15

The safe array class

```cpp
// ========= toStream =========
template<class T>
void ArrayT<T>::toStream(ostream &os) const {
    os << "(";
    for (int i = 0; i < _cap - 1; i++) {
        os << _data[i] << ", ";
    }
    os << _data[_cap - 1] << ")";
}

// ========= fromStream =========
template<class T>
int ArrayT<T>::fromStream(istream &is) {
    int len = 0;
    if constexpr(!is_shared_ptr<T>::value)
        for (int i = 0; i < _cap && is >> _data[i]; i++) {
            len++;
        }
    return len;
}
```
The safe array class

// ========= operator[] =========
template<class T>
T &ArrayT<T>::operator[](int i) {
    if (i < 0 || _cap <= i) {
        cerr << "ArrayT index out of bounds: index == " << i << endl;
        throw -1;
    }
    return _data[i];
}

template<class T>
T const &ArrayT<T>::operator[](int i) const {
    if (i < 0 || _cap <= i) {
        cerr << "ArrayT index out of bounds: index == " << i << endl;
        throw -1;
    }
    return _data[i];
}
Demo ArrayT
A vector class

• Similar to an array, can subscript.

• Capacity increases automatically.

• Insert operation shifts current values to the right.

• Remove operation shifts current values to the left.
Design Patterns for Data Structures

```
  v  10
  [0]
(a) v.insert(0, 10);
```
(a) \texttt{v.insert(0, 10);} \\
\texttt{v} \begin{array}{c}
\texttt{0} \\
\texttt{10}
\end{array}
\begin{array}{c}
\texttt{0} \\
\end{array}

(b) \texttt{v.insert(0, 20);} \\
\texttt{v} \begin{array}{c}
\texttt{0} \\
\texttt{20} \\
\texttt{10}
\end{array}
\begin{array}{c}
\texttt{0} \\
\texttt{1}
\end{array}

Capacity is doubled.
(a) `v.insert(0, 10);`

(b) `v.insert(0, 20);`  
Capacity is doubled.

(c) `v.insert(0, 30);`  
Capacity is doubled.
(a) `v.insert(0, 10);`

(a) `v. insert(0, 10);`

(b) `v.insert(0, 20);`
Capacity is doubled.

(b) `v.insert(0, 20);`
Capacity is doubled.

(c) `v.insert(0, 30);`
Capacity is doubled.

(c) `v.insert(0, 30);`
Capacity is doubled.

(d) `v.insert(1, 40);`
Figure 2.17

```java
v.insert(4, 50);
Capacity is doubled.
```
(e) \( v \text{.insert}(4, 50); \)
Capacity is doubled.

(f) \( v \text{.insert}(2, 60); \)
Demo VectorT
A vector implementation

template<class T>
// ========== VectorT ==========
template<class T>
class VectorT : public ASeq<T> {
private:
    unique_ptr<T[]> _data;
    int _cap; // Invariant: 0 < _cap, and _cap is a power of 2.
    int _size; // Invariant: 0 <= _size <= _cap.

    void doubleCapacity();
A vector implementation

```cpp
// ======= doubleCapacity =======
template<class T>
void VectorT<T>::doubleCapacity() {
    _cap *= 2;
    T *newDat = new T[_cap];
    for (int k = 0; k < _size; k++) {
        newDat[k] = _data[k];
    }
    _data.reset(newDat);
}
```
A vector implementation

public:
    VectorT();
    // Post: This vector is initialized with capacity of 1 and size of 0.

    int cap() const override { return _cap; }
    // Post: The capacity of this vector is returned.

    int size() const { return _size; }
    // Post: The size of this vector is returned.

(a) Specification of the constructor. Specification and implementation of cap() and size().

    // ========= Constructor =========
    template<class T>
    VectorT<T>::VectorT() {
        _data = make_unique<T[]>(1);
        _cap = 1;
        _size = 0;
    }

(b) Implementation of the constructor.

Figure 2.20 shows that method append() has no precondition. You can append a value to an empty vector. And even if you append a value to a vector that is full, it will automatically double its capacity to accommodate the appended value.

Compare the preconditions of insert() and remove(). The precondition for insert() is

// Pre: 0 <= i && i <= size().

and the precondition for remove() is
A vector implementation

```cpp
public:
    void append(T const &e);
    // Post: Element e is appended to this vector, possibly increasing cap().

    void insert(int i, T const &e);
    // Pre: 0 <= i && i <= size().
    // Post: Items [i..size()-1] are shifted right and element e is
    // inserted at position i.
    // size() is increased by 1, possibly increasing cap().

    T remove(int i);
    // Pre: 0 <= i && i < size(). T has a copy constructor.
    // Post: Element e is removed from position i and returned.
    // Items [i+1..size()-1] are shifted left.
    // size() is decreased by 1 (and cap() is unchanged).
```

(a) Specification of append(), insert(), and remove().
A vector implementation

```c++
// ========= append =========
template<class T>
void VectorT<T>::append(T const &e) {
    if (_size == _cap) {
        doubleCapacity();
    }
    _data[_size++] = e;
}
```

(b) Implementation of append().
A vector implementation

T &operator[](int i) override; // For read/write.
T const &operator[](int i) const override; // For read-only.

(a) Specification of operator[].
A vector implementation

// ========= operator[] =========
template<class T>
T &VectorT<T>::operator[](int i) {
    if (i < 0 || _size <= i) {
        cerr << "VectorT index out of bounds: index == " << i << endl;
        throw -1;
    }
    return _data[i];
}

template<class T>
T const &VectorT<T>::operator[](int i) const {
    if (i < 0 || _size <= i) {
        cerr << "VectorT index out of bounds: index == " << i << endl;
        throw -1;
    }
    return _data[i];
}

(b) Implementation of operator[].
A vector implementation

// ======== operator<< ========
template<class T>
ostream &operator<<(ostream &os, VectorT<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

// ======== toStream ========
template<class T>
void VectorT<T>::toStream(ostream &os) const {
    os << "(
    for (int i = 0; i < _size - 1; i++) {
        os << _data[i] << ", ";
    }
    if (_size > 0) {
        os << _data[_size-1];
    }
    os << ")";
}
A vector implementation

```cpp
// ======== operator >> ========
template<class T>
istream &operator>>(istream &is, VectorT<T> &rhs) {
    rhs.fromStream(is);
    return is;
}

// ======== fromStream ========
template<class T>
void VectorT<T>::fromStream(istream &is) {
    T temp;
    while (is >> temp) {
        append(temp);
    }
}
```
Unit tests

```
i 10 0 w c s  (10)
The capacity is 1
The size is 1
```
```
i 20 0 w c s  (20, 10)
The capacity is 2
The size is 2
```
```
i 30 0 w c s
```
```
i 40 1 w c s
```
```
i 50 4 w c s
```
```
i 60 2 w c s q
VectorT unit-insert
```
```
(30, 30, 20, 10)
The capacity is 4
The size is 3
```
```
(30, 40, 20, 10)
The capacity is 4
...