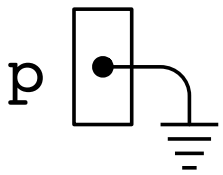
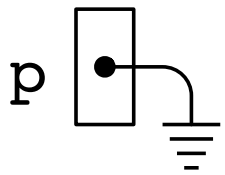
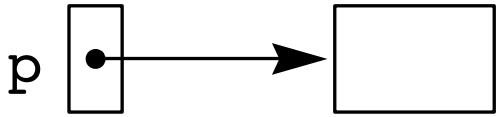
Arrays and Dynamic Memory Allocation



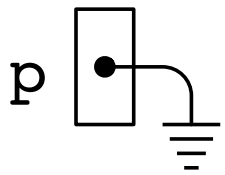
(a) `shared_ptr<double> p;`



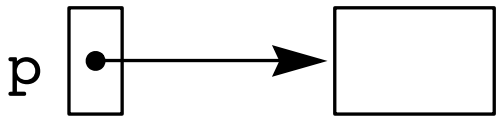
(a) `shared_ptr<double> p;`



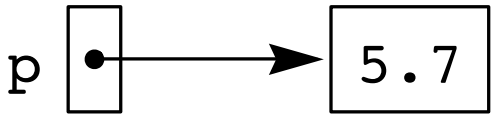
(b) `p = make_shared<double>();`



(a) `shared_ptr<double> p;`

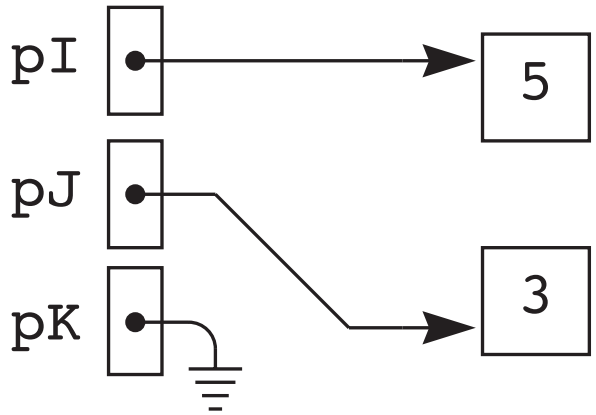


(b) `p = make_shared<double>();`

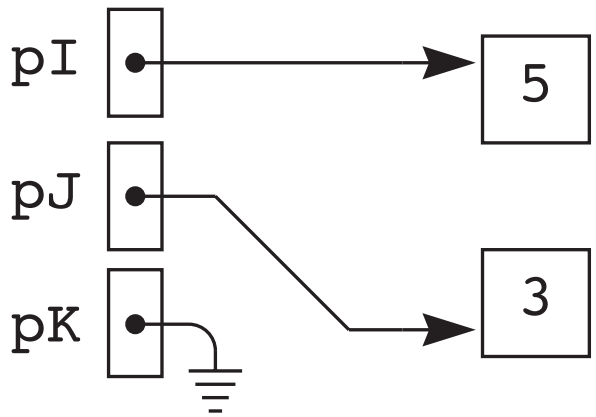


(c) `*p = 5.7;`

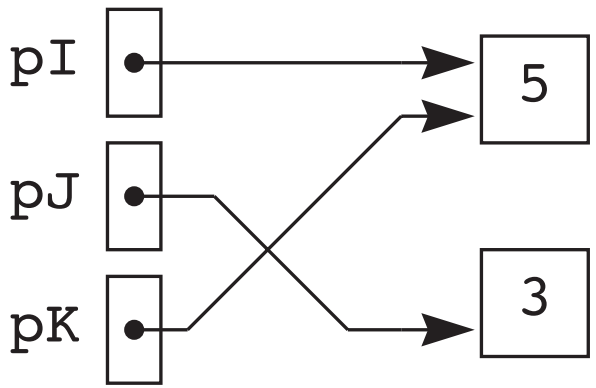
```
shared_ptr<int> pI = make_shared<int>();
shared_ptr<int> pJ = make_shared<int>();
shared_ptr<int> pK;
*pI = 5;
*pJ = 3;
pK = pI;
pI = pJ;
*pI = 2 + *pK;
cout << *pI << " " << *pJ << " " << *pK << endl;
```



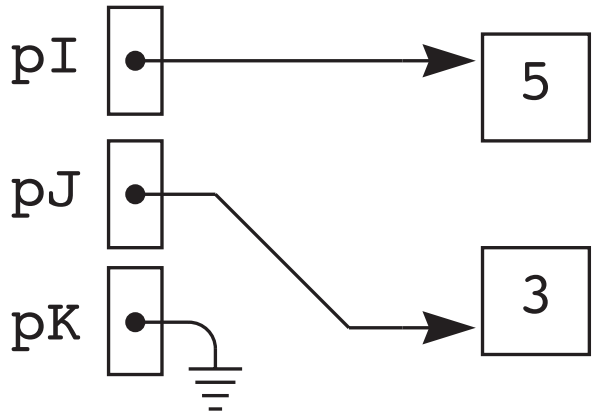
(a)
```
*pI = 5;
*pJ = 3;
```



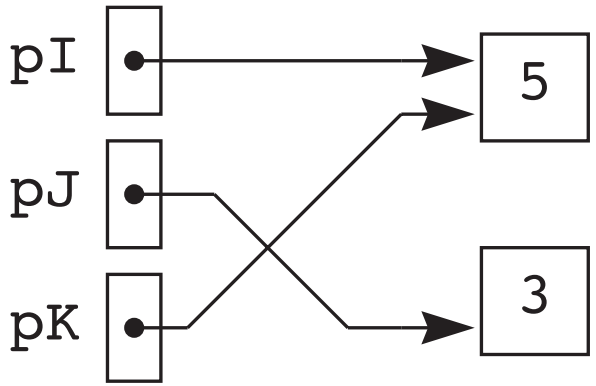
(a)
```
*pI = 5;
*pJ = 3;
```



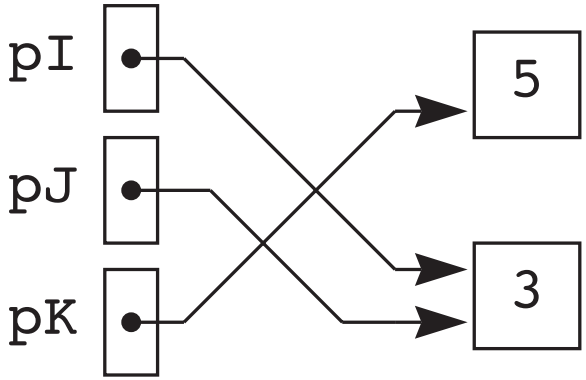
(b)
```
pK = pI;
```



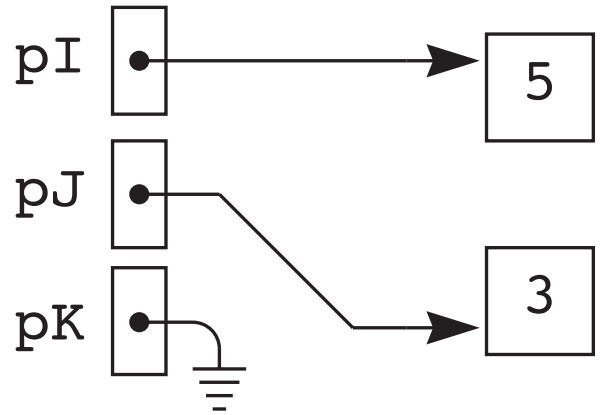
(a)
```
*pI = 5;
*pJ = 3;
```



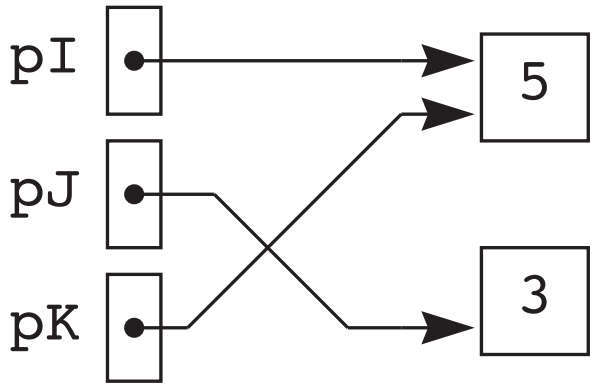
(b)
```
pK = pI;
```



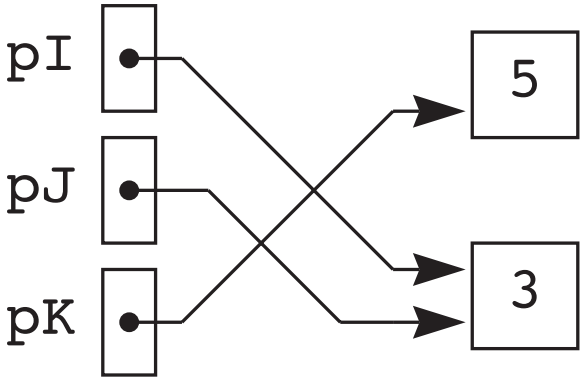
(c)
```
pI = pJ;
```



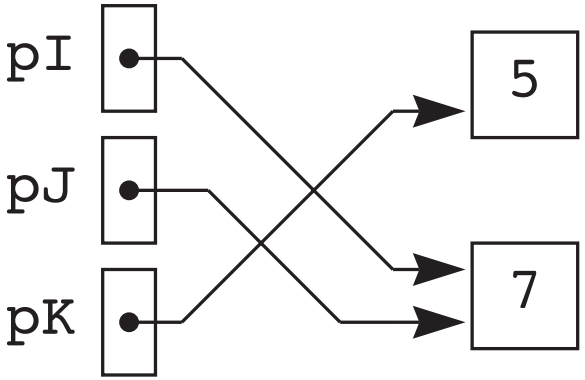
(a)
```
*pI = 5;
*pJ = 3;
```



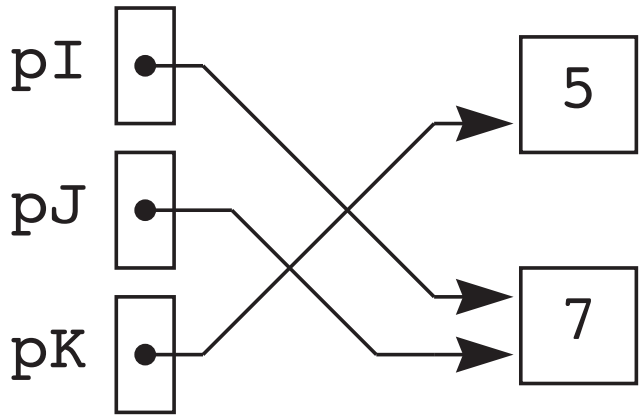
(b) `pK = pI;`



(c) `pI = pJ;`



(d) `*pI = 2 + *pK;`



(d) `*pI = 2 + *pK;`

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

7 7 5

Design Patterns for Data Structures

```
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

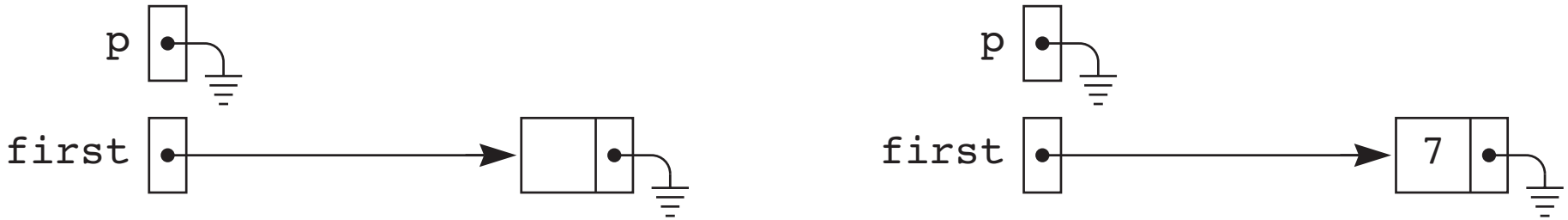
```
(*p).value
```

Figure A.1

Operators	Associativity
`::`	left to right
`[]` `()` `->` `.` *po* `++` *po* `--`	left to right
`!` `~` *pr* `++` *pr* `--` *un* `+` *un* `-` *un* `*` *un* `&` `new` `delete` `sizeof`	right to left
`(`*type*`)`	right to left
`*` `/` `%`	left to right
`+` `-`	left to right
`<<` `>>`	left to right
`<` `<=` `>` `>=`	left to right
`==` `!=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
tr `?:`	right to left
`=` `+=` `-=` `*=` `/=` `|=` `%=` `&=` `^=` `<<=` `>>=`	right to left
`,`	left to right

```
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 << " ";
}
```

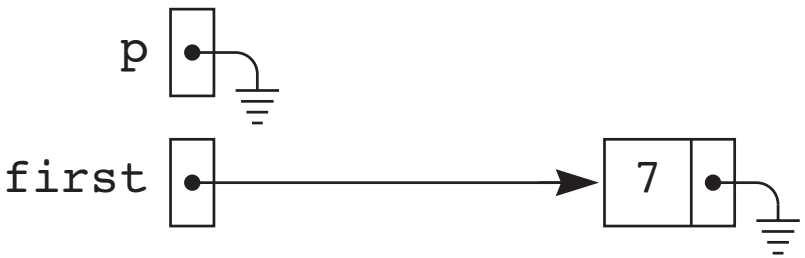
(a) `first = make_shared<Node>();`



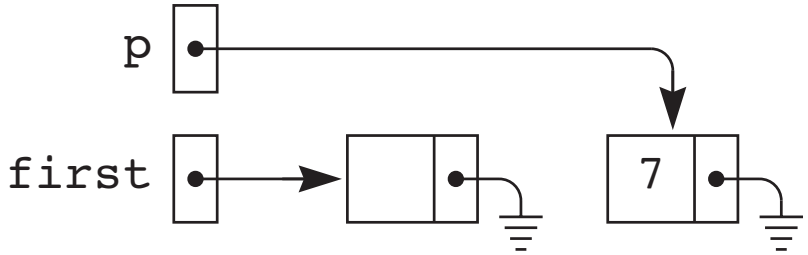
(a) `first = make_shared<Node>();`

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

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



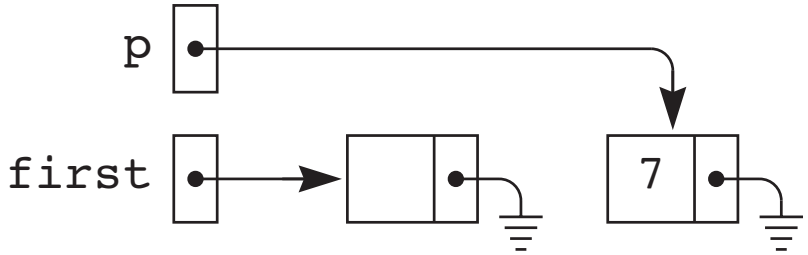
(b) `first->Value = 7;`



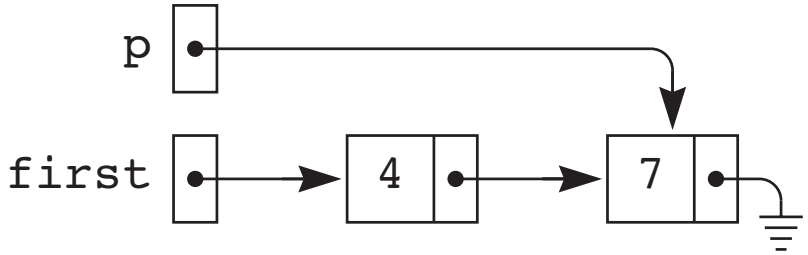
(c)
```
p = first;
first = make_shared<Node>();
```

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

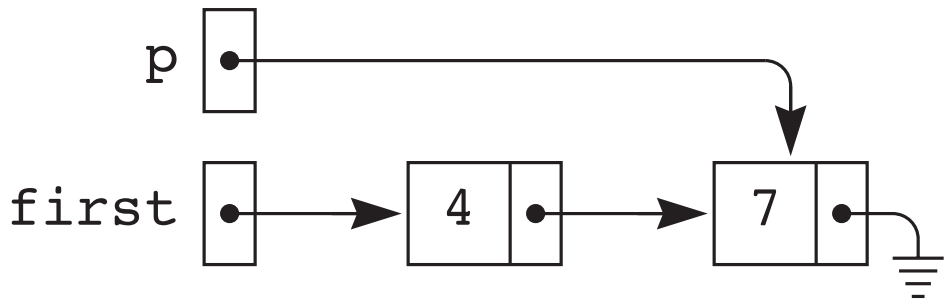
(b) `first->Value = 7;`



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



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



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

```
for (shared_ptr<Node> q = first; q != nullptr; q = q-> next) {
    cout << q -> value << " ";
}
```

4 7

Figure 2.3

```
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;                                  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>
```

C++ parameter passing mechanisms

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

```
#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

```
#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
```

```
#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
```

```
#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;
}
```

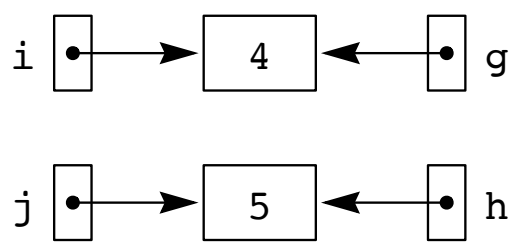
Output

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

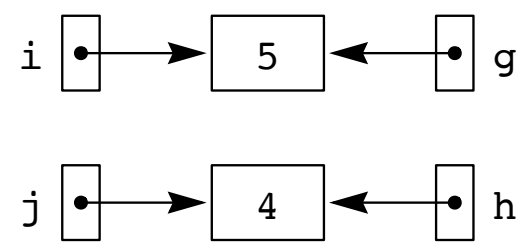
```
#include <iostream>
using namespace std;

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



(a) Passing pointers by value, before.



(b) Passing pointers by value, after.

```
#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

```
#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;
}
```

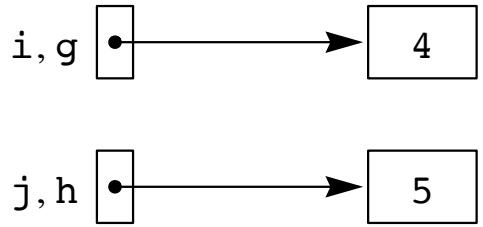
Output

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

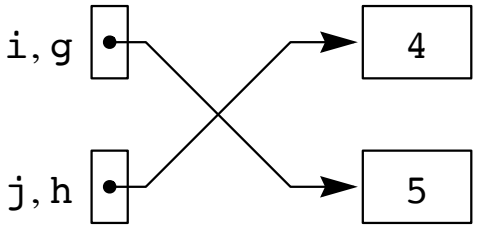
```
#include <iostream>
using namespace std;

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,

```
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

```
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).
```

```
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:" << endl;
        writeStream(cout, &arr, cap, length);
        // arr[2 * cap] = 123.4;
        // cout << arr[2 * cap] << endl;
    }
    return EXIT_SUCCESS;
}
```

```
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.

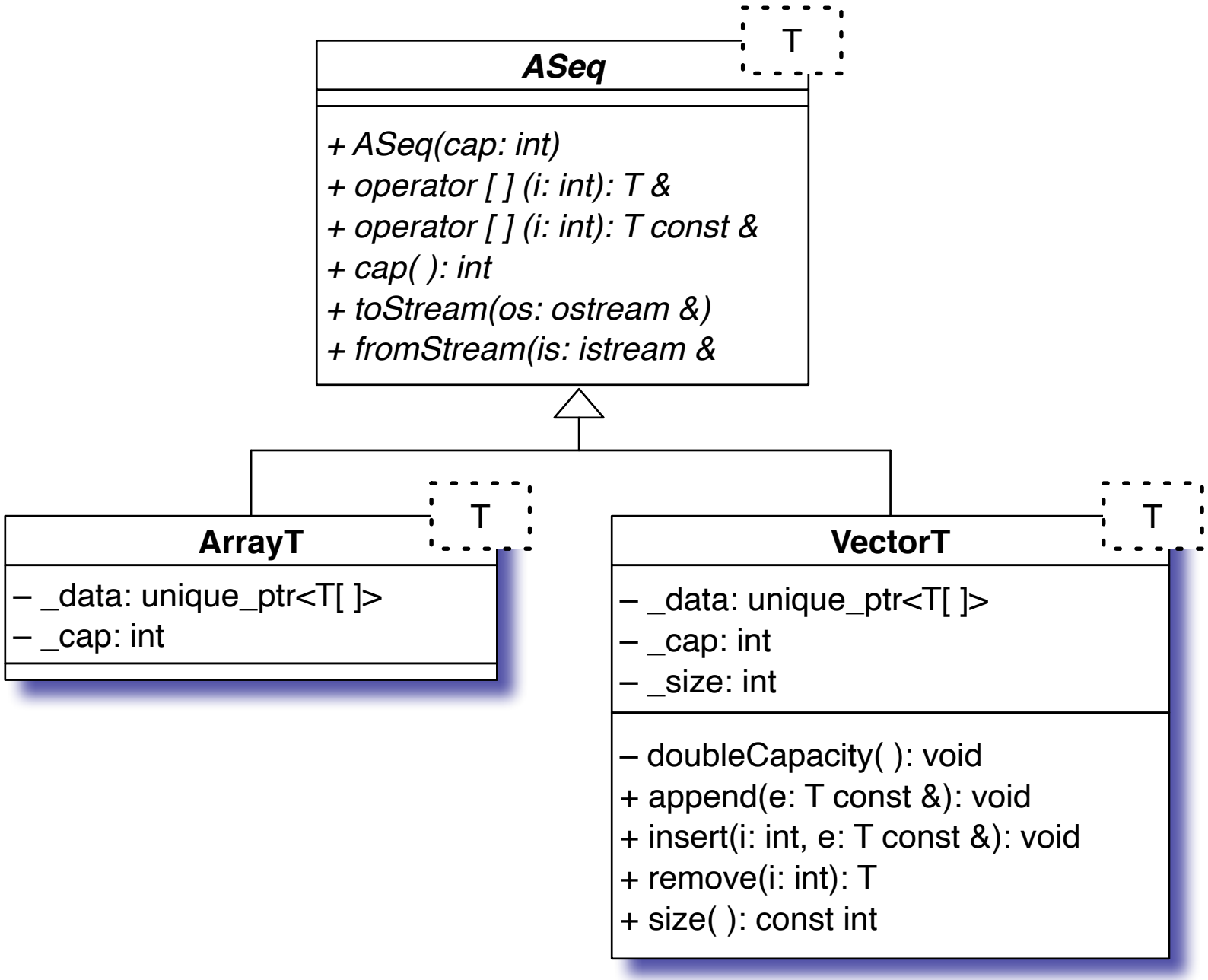
An abstract sequence class

```
// ========= 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.
```

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



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;
}
```

The safe array class

```
// ========= 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;
}
```

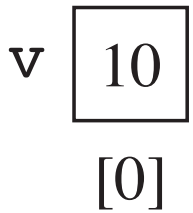
The safe array class

```
// ========= 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];
}
```

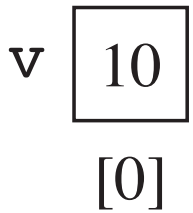
Demo ArrayT

A vector class

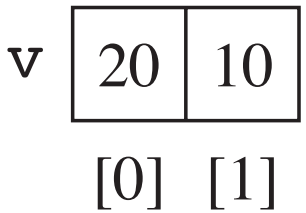
- Similar to an array, can subscript.
- Capacity increases automatically.
- Insert operation shifts current values to the right.
- Remove operation shifts current values to the left.



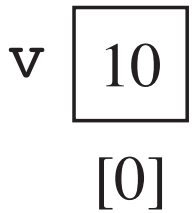
(a) `v.insert(0, 10);`



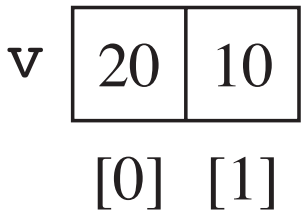
(a) `v.insert(0, 10);`



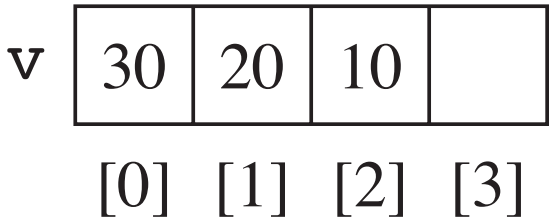
(b) `v.insert(0, 20);`
Capacity is doubled.



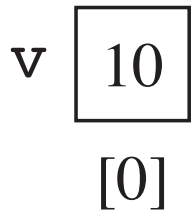
(a) `v.insert(0, 10);`



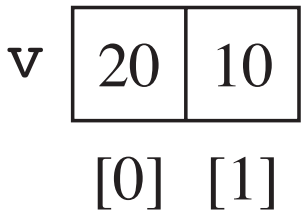
(b) `v.insert(0, 20);`
Capacity is doubled.



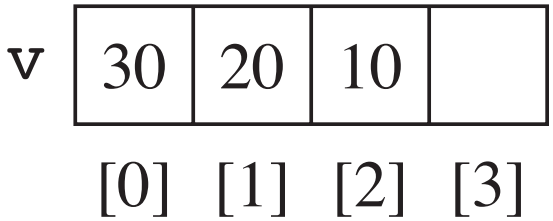
(c) `v.insert(0, 30);`
Capacity is doubled.



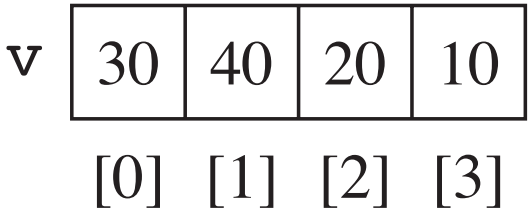
(a) `v.insert(0, 10);`



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



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



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

v	30	40	20	10	50			
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

(e) `v.insert(4, 50);`
Capacity is doubled.

v	30	40	20	10	50			
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

(e) `v.insert(4, 50);`
Capacity is doubled.

v	30	40	60	20	10	50		
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

(f) `v.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

```
// ========= 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.

A vector implementation

```
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

```
// ========= 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

```
// ========= 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
i 20 0 w c s
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
```

```
(10)
The capacity is 1
The size is 1

(20, 10)
The capacity is 2
The size is 2

(30, 20, 10)
The capacity is 4
The size is 3

(30, 40, 20, 10)
The capacity is 4
```

...