

# Systematic Traversal of Sets



# Iterators

- We can use a loop statement and a loop counter to traverse all elements of an array in sequence.
- However, not all data types are arrays and simple indexing may not suffice.
- Iterators offer programmers a suitable alternative to define traversal in a data type agnostic way.
- Conceptually, iterators are objects that implement the necessary infrastructure to iterate over elements of a sequence. They do this via a common interface.
- The C++ ecosystem has popularized and uses five types of iterators. Iterator objects have the look-and-feel of pointers, and advancing an iterator means to increment/decrement a pointer-like object.

# C++ Iterators

Input Iterator

Output Iterator

Forward Iterator

Bidirectional Iterator

Random Access Iterator



# Abilities of Iterators

Iterator Category	Ability	Provider
Input Iterator	Read forward	istream
Output Iterator	Write forward	ostream, inserter
Forward Iterator	Read and write forward	
Bidirectional Iterator	Read and write forward and backward	list, set, multiset, map, multimap, vector, deque, string, array
Random Access Iterator	Read and write with random access	

# Input Iterator

Expression	Effect
<code>*iter</code>	Provides read access to the actual element
<code>iter-&gt;member</code>	Provides read access to a member of the actual element
<code>++iter</code>	Steps forward (returns new position)
<code>iter++</code>	Steps forward (returns old position)
<code>iter1 == iter2</code>	Returns whether iter1 and iter2 are equal
<code>iter1 != iter2</code>	Returns whether iter1 and iter2 are not equal

# Output Iterator

Expression	Effect
<code>*iter = value</code>	Provides write access to the actual element
<code>++iter</code>	Steps forward (returns new position)
<code>iter++</code>	Steps forward (returns old position)

An output iterator is like a “black hole.”



# Forward Iterator

Expression	Effect
<code>*iter</code>	Provides read access to the actual element
<code>iter-&gt;member</code>	Provides read access to a member of the actual element
<code>++iter</code>	Steps forward (returns new position)
<code>iter++</code>	Steps forward (returns old position)
<code>iter1 == iter2</code>	Returns whether iter1 and iter2 are equal
<code>iter1 != iter2</code>	Returns whether iter1 and iter2 are not equal
<code>iter1 = iter2</code>	Assigns an iterator

# Bidirectional Iterator

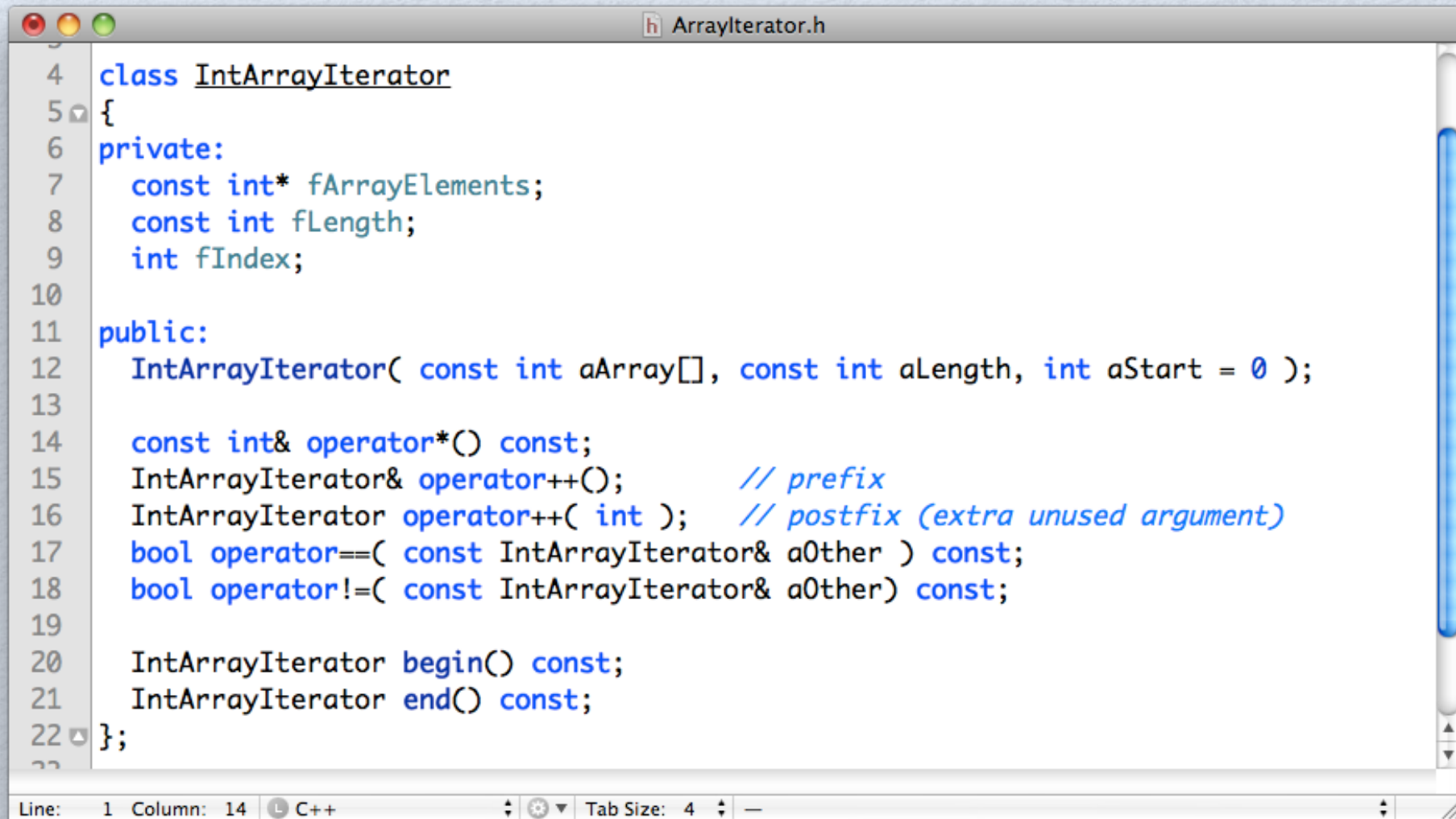
Expression	Effect
<code>*iter</code>	Provides read access to the actual element
<code>iter-&gt;member</code>	Provides read access to a member of the actual element
<code>++iter</code>	Steps forward (returns new position)
<code>iter++</code>	Steps forward (returns old position)
<code>--iter</code>	Steps backward (returns new position)
<code>iter--</code>	Steps backward (returns old position)
<code>iter1 == iter2</code>	Returns whether iter1 and iter2 are equal
<code>iter1 != iter2</code>	Returns whether iter1 and iter2 are not equal
<code>iter1 = iter2</code>	Assigns an iterator



# Random Access Iterator

Expression	Effect
<code>iter[n]</code>	Provides read access to the element at index <code>n</code>
<code>iter += n</code>	Steps <code>n</code> elements forward or backward
<code>iter -= n</code>	Steps <code>n</code> elements forward or backward
<code>n+iter</code>	Returns the iterator of the <code>n</code> th next element
<code>n-iter</code>	Returns the iterator of the <code>n</code> th previous element
<code>iter - iter2</code>	Returns disjoint distance between <code>iter1</code> and <code>iter2</code>
<code>iter1 &lt; iter2</code>	Returns whether <code>iter1</code> is before <code>iter2</code>
<code>iter1 &gt; iter2</code>	Returns whether <code>iter1</code> is after <code>iter2</code>
<code>iter1 &lt;= iter2</code>	Returns whether <code>iter1</code> is not after <code>iter2</code>
<code>iter1 &gt;= iter2</code>	Returns whether <code>iter1</code> is not before <code>iter2</code>

# A Read-Only Forward Iterator



```
1
2
3
4 class IntArrayIterator
5 {
6 private:
7     const int* fArrayElements;
8     const int fLength;
9     int fIndex;
10
11 public:
12     IntArrayIterator( const int aArray[], const int aLength, int aStart = 0 );
13
14     const int& operator*() const;
15     IntArrayIterator& operator++();           // prefix
16     IntArrayIterator operator++( int );       // postfix (extra unused argument)
17     bool operator==( const IntArrayIterator& aOther ) const;
18     bool operator!=( const IntArrayIterator& aOther) const;
19
20     IntArrayIterator begin() const;
21     IntArrayIterator end() const;
22 };
23
```

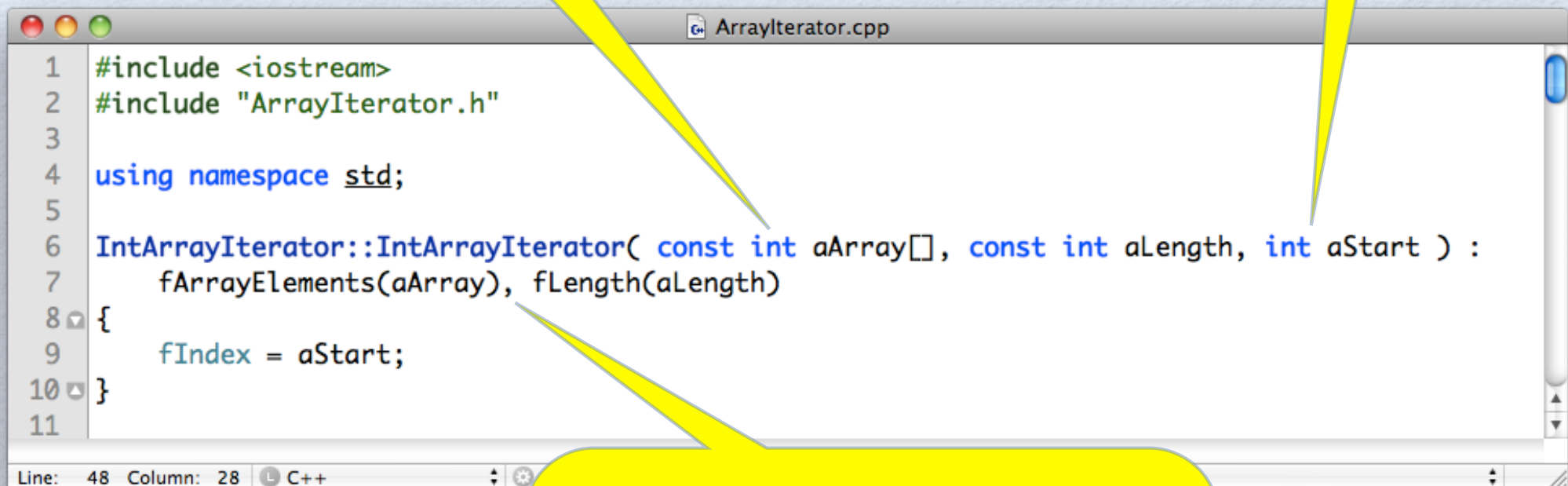
Line: 1 Column: 14 C++ Tab Size: 4



# Forward Iterator Constructor

Arrays are passed as pointers to the first element to functions in C++.

We must not repeat the default value.



```
1 #include <iostream>
2 #include "ArrayIterator.h"
3
4 using namespace std;
5
6 IntArrayIterator::IntArrayIterator( const int aArray[], const int aLength, int aStart ) :
7     fArrayElements(aArray), fLength(aLength)
8 {
9     fIndex = aStart;
10 }
11
```

Line: 48 Column: 28 C++

We must use member initializer to initialize const instance variables!

# The Dereference Operator

A screenshot of a C++ code editor window titled 'ArrayIterator.cpp'. The code defines the dereference operator for an 'IntArrayIterator' class. The code is as follows:

```
11  
12 const int& IntArrayIterator::operator*() const  
13 {  
14     return fArrayElements[fIndex];  
15 }  
16
```

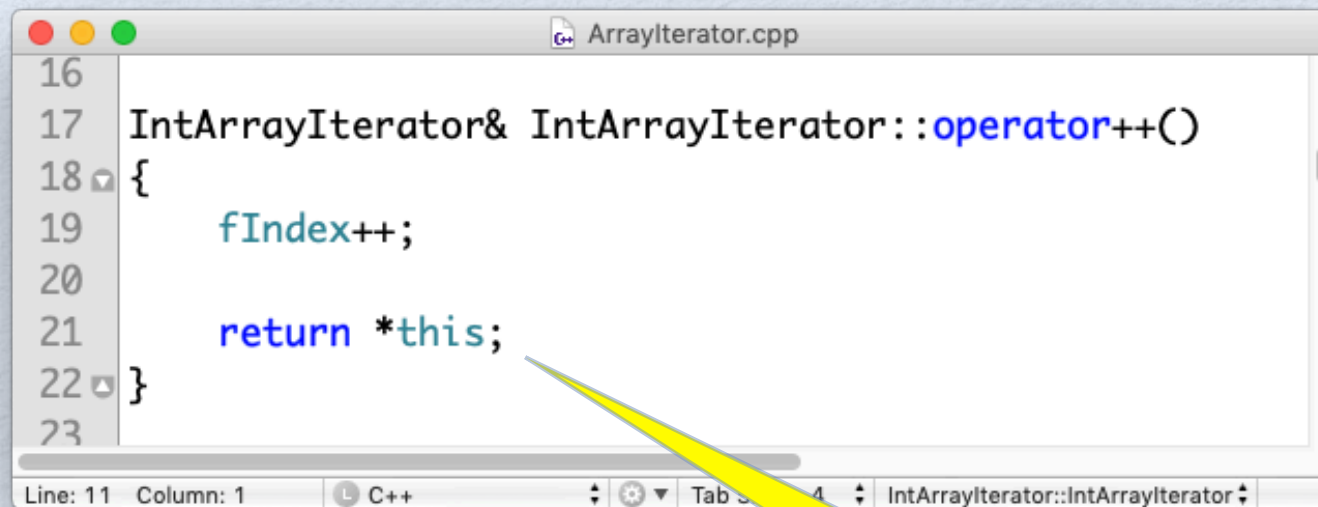
The editor shows line numbers 11 to 16 on the left. The status bar at the bottom indicates 'Line: 20 Column: 1', 'C++', 'Tab Size: 4', and the current file name 'IntArrayIterator::IntArrayIt...'.

- The dereference operator returns the element the iterator is currently positioned on.
- The dereference operator is a `const` operation, that is, it does not change any instance variables of the iterator.
- We use a `const reference` to avoid copying the original value stored in the underlying collection.
- `No range check is required.` The `operator*()` should only be called if the iterator has not yet reached the end of the underlying collection.



# Prefix Increment

- The prefix increment operator advances the iterator and returns a reference of this iterator.



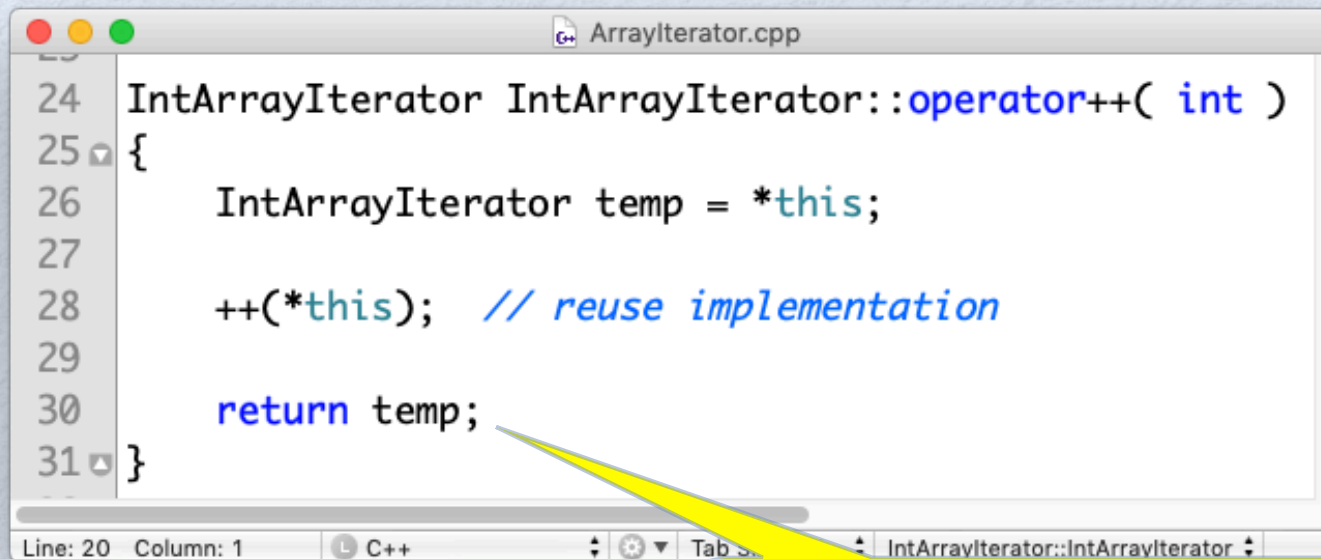
```
16  
17 IntArrayIterator& IntArrayIterator::operator++()  
18 {  
19     fIndex++;  
20  
21     return *this;  
22 }  
23
```

The screenshot shows a code editor window titled 'ArrayIterator.cpp'. The code defines the prefix increment operator for 'IntArrayIterator'. Line 17 shows the function signature 'IntArrayIterator& IntArrayIterator::operator++()'. Line 18 is an opening curly brace. Line 19 increments the 'fIndex' member. Line 21 returns a reference to the current object using 'return \*this;'. Line 22 is a closing curly brace. A yellow callout bubble points to the 'return \*this;' statement.

Return a reference to the current iterator (set forward).

# Postfix Increment

- The postfix increment operator advances the iterator and returns a copy of the old iterator.



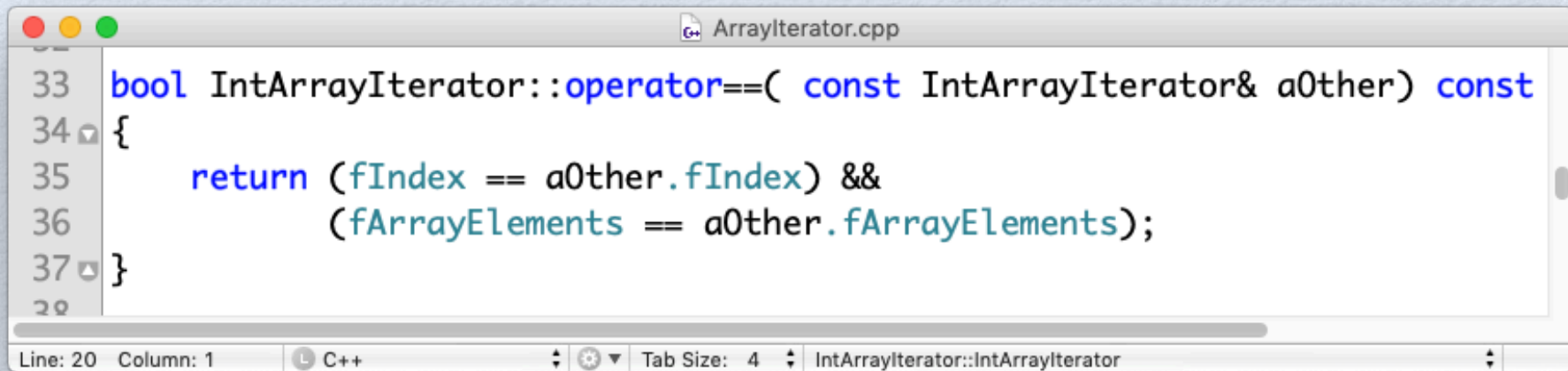
```
ArrayIterator.cpp
24 IntArrayIterator IntArrayIterator::operator++( int )
25 {
26     IntArrayIterator temp = *this;
27
28     ++(*this); // reuse implementation
29
30     return temp;
31 }
```

The screenshot shows a code editor window titled 'ArrayIterator.cpp'. The code defines the postfix increment operator for the 'IntArrayIterator' class. It creates a temporary object 'temp' by dereferencing 'this' and then increments the current object. A yellow callout bubble points to the 'return temp;' statement.

Return a copy of the old iterator  
(position unchanged).



# Iterator Equivalence



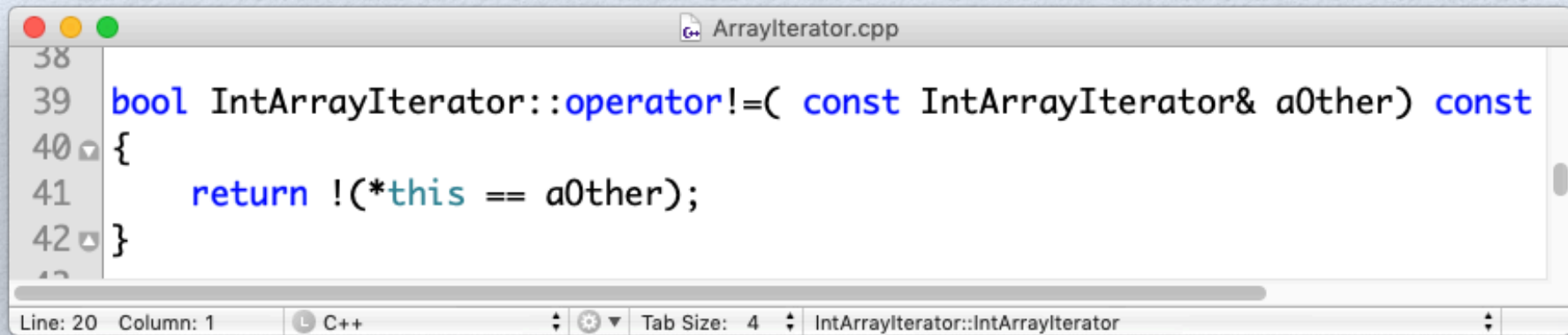
```
ArrayIterator.cpp
33 bool IntArrayIterator::operator==( const IntArrayIterator& aOther) const
34 {
35     return (fIndex == aOther.fIndex) &&
36           (fArrayElements == aOther.fArrayElements);
37 }
38
```

Line: 20 Column: 1 C++ Tab Size: 4 IntArrayIterator::IntArrayIterator

Two iterators are equal if and only if they refer to the same element (this may require considering the context of ==):

- `fIndex` is the current index into the array
- Arrays are passed as a pointer to the first element (arrays decay to pointers) that is constant throughout runtime.

# Iterator Inequality



```
38
39 bool IntArrayIterator::operator!=( const IntArrayIterator& a0ther) const
40 {
41     return !(*this == a0ther);
42 }
```

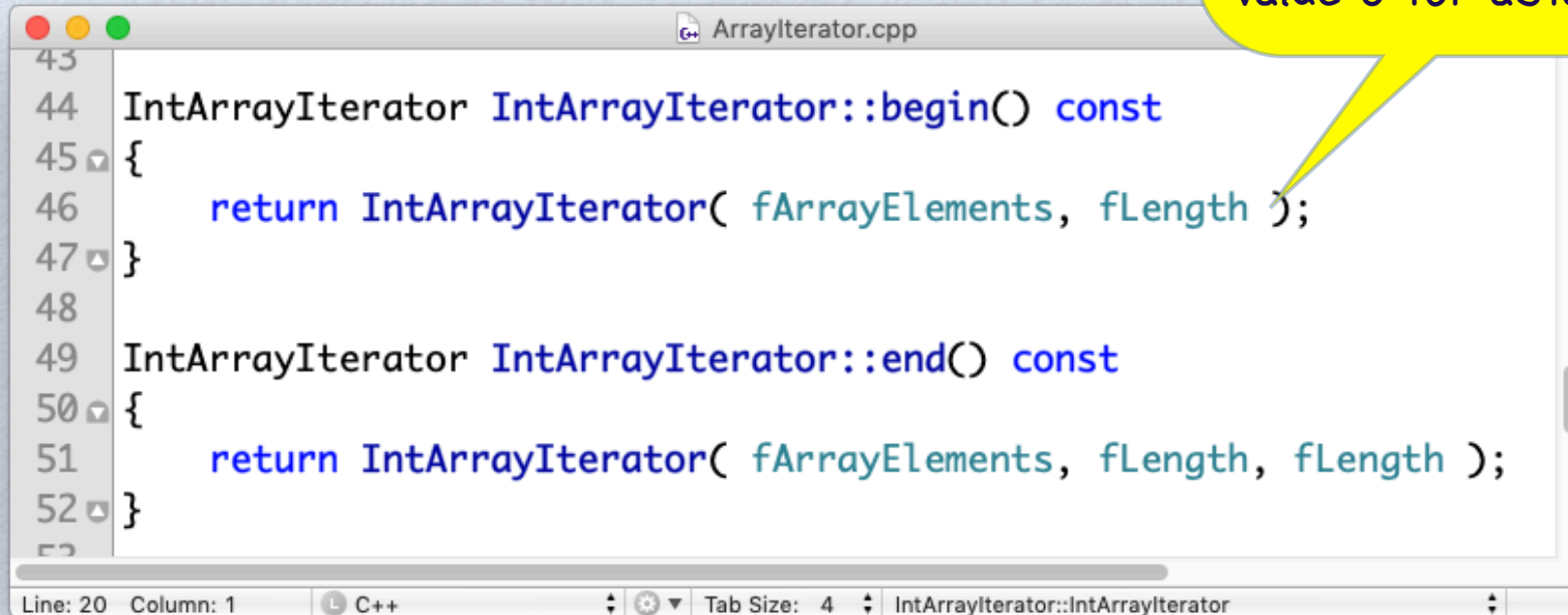
The screenshot shows a code editor window titled 'Arraylterator.cpp'. The code defines the '!=' operator for 'IntArrayIterator'. The implementation is: `bool IntArrayIterator::operator!=( const IntArrayIterator& a0ther) const { return !(*this == a0ther); }`. The editor's status bar at the bottom indicates 'Line: 20 Column: 1', 'C++', 'Tab Size: 4', and the current function 'IntArraylterator::IntArraylterator'.

We implement != in terms of ==.



# Auxiliary Methods

We use the default value 0 for aStart here.

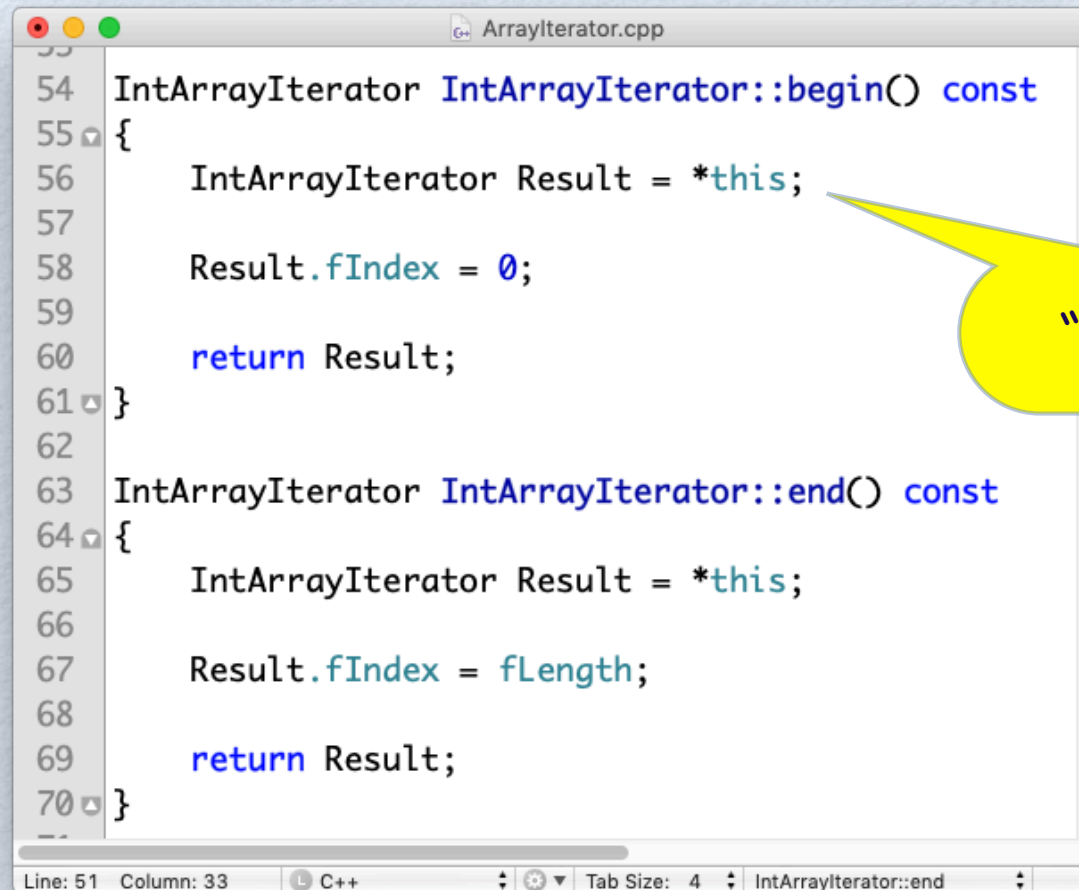


```
43  
44 IntArrayIterator IntArrayIterator::begin() const  
45 {  
46     return IntArrayIterator( fArrayElements, fLength );  
47 }  
48  
49 IntArrayIterator IntArrayIterator::end() const  
50 {  
51     return IntArrayIterator( fArrayElements, fLength, fLength );  
52 }  
53
```

Line: 20 Column: 1 C++ Tab Size: 4 IntArrayIterator::IntArrayIterator

- The methods `begin()` and `end()` return fresh iterators set to the first element and past-the-end element, respectively.
- The names and implementation of these auxiliary methods follow standard practices. The compiler may look for them when you use a for-each loop.

# Auxiliary Methods: Copy This Object



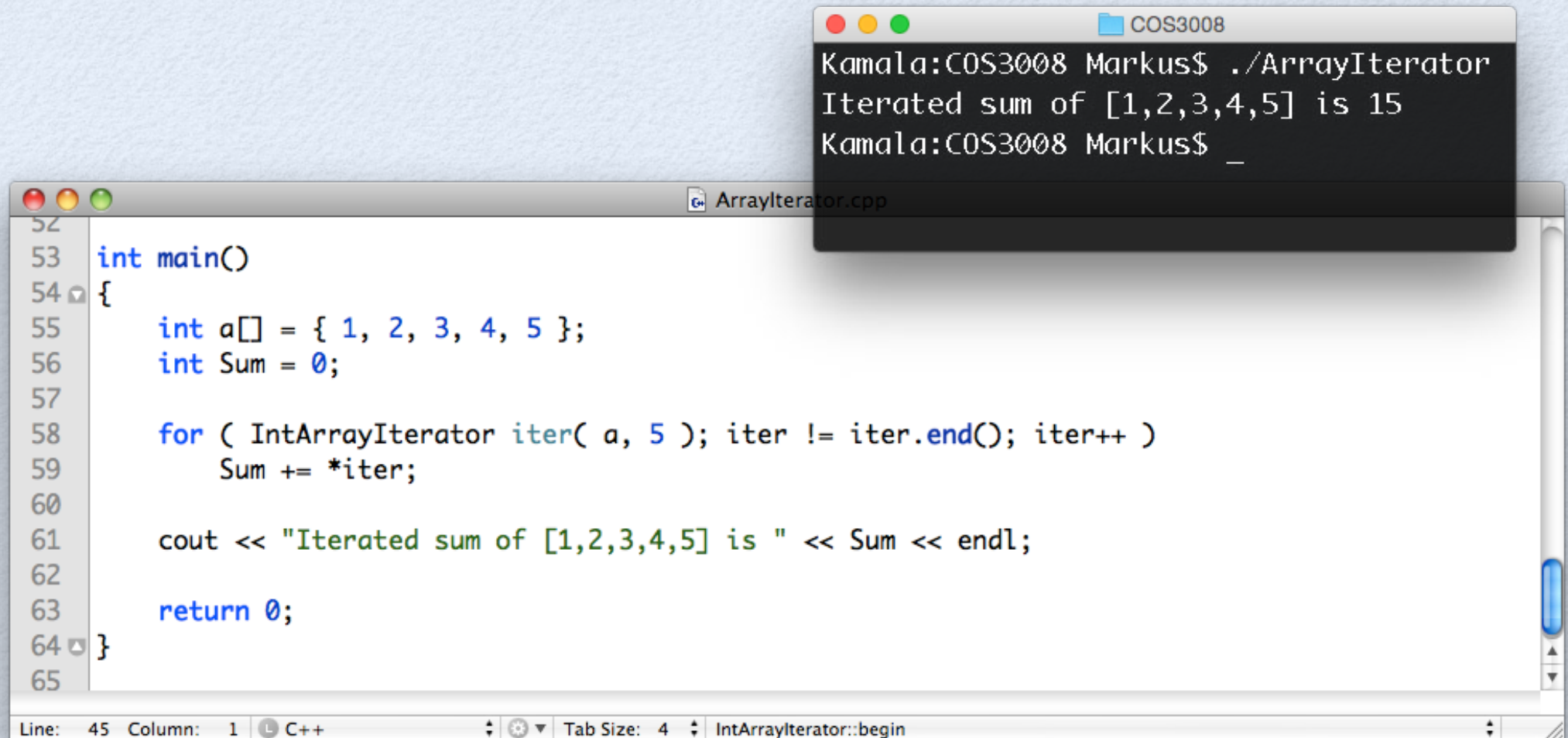
```
54 IntArrayIterator IntArrayIterator::begin() const
55 {
56     IntArrayIterator Result = *this;
57
58     Result.fIndex = 0;
59
60     return Result;
61 }
62
63 IntArrayIterator IntArrayIterator::end() const
64 {
65     IntArrayIterator Result = *this;
66
67     Result.fIndex = fLength;
68
69     return Result;
70 }
```

"clone" this object

- The methods "clone" `this` iterator object and set the position accordingly.



# Putting Everything Together



The image shows a C++ IDE window titled 'ArrayIterator.cpp' with the following code:

```
52  
53 int main()  
54 {  
55     int a[] = { 1, 2, 3, 4, 5 };  
56     int Sum = 0;  
57  
58     for ( IntArrayIterator iter( a, 5 ); iter != iter.end(); iter++ )  
59         Sum += *iter;  
60  
61     cout << "Iterated sum of [1,2,3,4,5] is " << Sum << endl;  
62  
63     return 0;  
64 }  
65
```

Below the code editor, the status bar shows 'Line: 45 Column: 1 C++', 'Tab Size: 4', and 'IntArrayIterator::begin'.

Overlaid on the IDE is a terminal window titled 'COS3008' showing the execution of the program:

```
Kamala:COS3008 Markus$ ./ArrayIterator  
Iterated sum of [1,2,3,4,5] is 15  
Kamala:COS3008 Markus$ _
```



**Can we do better?**



# C++11: For-Each-Loop

- The traditional for statement in C++ reads:

```
for ( init-statement; condition; expression )  
    statement
```

This form uses explicit loop variables, conditions, and increments over loop variables.

- C++11 introduces a simpler form, called **range for statement**, to iterate through the elements of a container or other sequence:

```
for ( declaration : expression )  
    statement
```

This form is called for-each, and expression must denote a sequence and declaration defines a variable, set in each step of the iteration.

# Understanding C++11's range loop

**for ( declaration : expression ) statement**

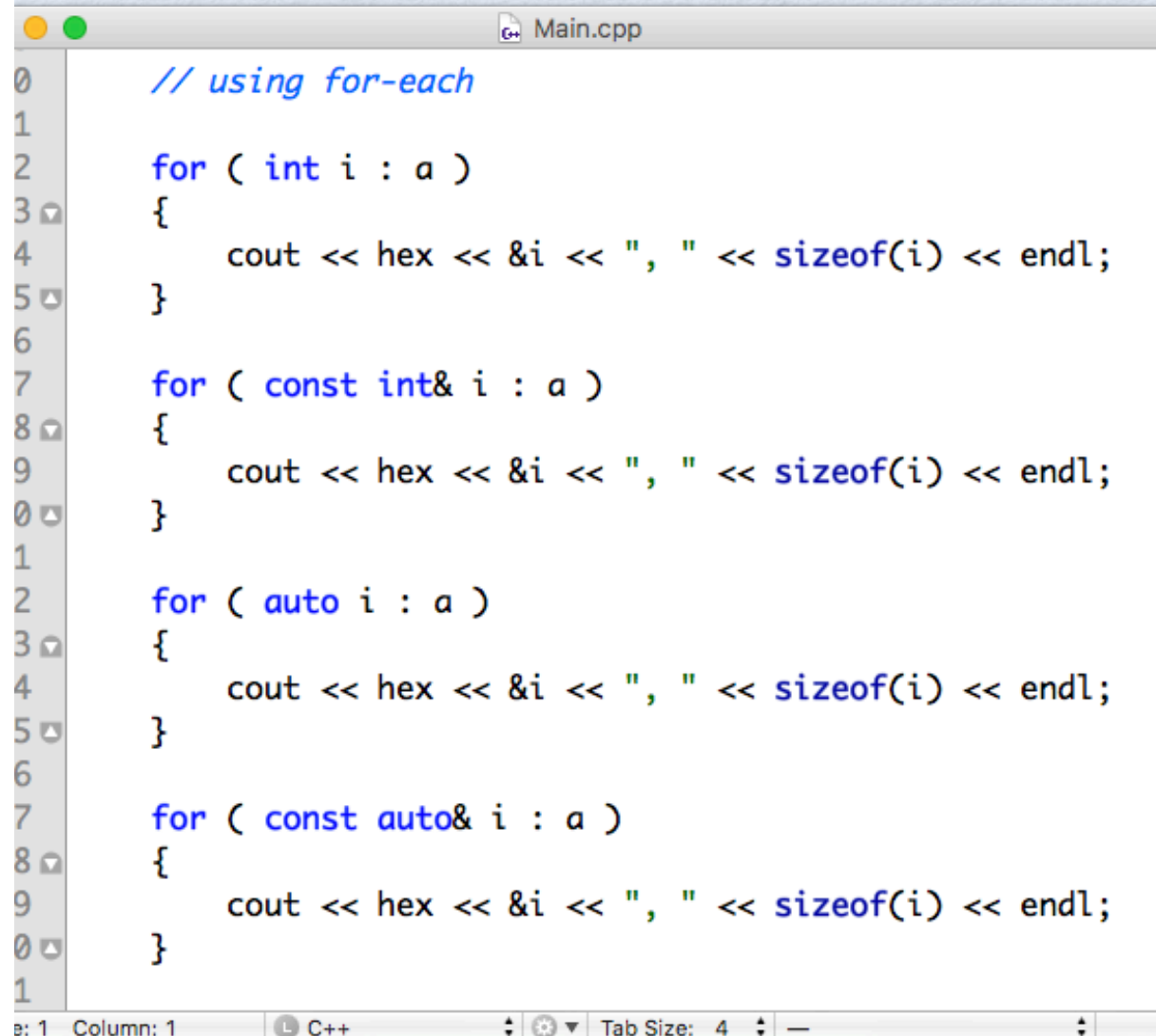
- According to the standard, this is equivalent to the following plain for loop:

```
auto&& __range = expression;           // C++11 forwarding (move)
for ( auto __begin = begin-expression,  // begin()
      __end = end-expression;           // end()
      __begin != __end;
      ++__begin )
{
    declaration = *__begin;
    statement;
}
```

Compare with page 188



# Using C++11's For-Each-Loop



```
0 // using for-each
1
2 for ( int i : a )
3 {
4     cout << hex << &i << ", " << sizeof(i) << endl;
5 }
6
7 for ( const int& i : a )
8 {
9     cout << hex << &i << ", " << sizeof(i) << endl;
10 }
11
12 for ( auto i : a )
13 {
14     cout << hex << &i << ", " << sizeof(i) << endl;
15 }
16
17 for ( const auto& i : a )
18 {
19     cout << hex << &i << ", " << sizeof(i) << endl;
20 }
21
```

The screenshot shows a code editor window titled 'Main.cpp'. It contains four examples of C++11 for-each loops. Each example prints the memory address of the element (in hexadecimal) and its size. The examples use different variable types: plain int, const int reference, auto, and const auto reference. The IDE interface includes a line number margin on the left and a status bar at the bottom showing 'Column: 1', 'C++', and 'Tab Size: 4'.

- Case 1: read-write variable
- Case 2: constant reference
- Case 3: auto variable
- Case 4: constant auto reference

# For-Each-Loop Behavior

```
Main.cpp
30 // using for-each
31
32 for ( int i : a )
33 {
34     cout << hex << &i << ", " << sizeof(i) << endl;
35 }
36
37 for ( const int& i : a )
38 {
39     cout << hex << &i << ", " << sizeof(i) << endl;
40 }
41
42 for ( auto i : a )
43 {
44     cout << hex << &i << ", " << sizeof(i) << endl;
45 }
46
47 for ( const auto& i : a )
48 {
49     cout << hex << &i << ", " << sizeof(i) << endl;
50 }
51
```

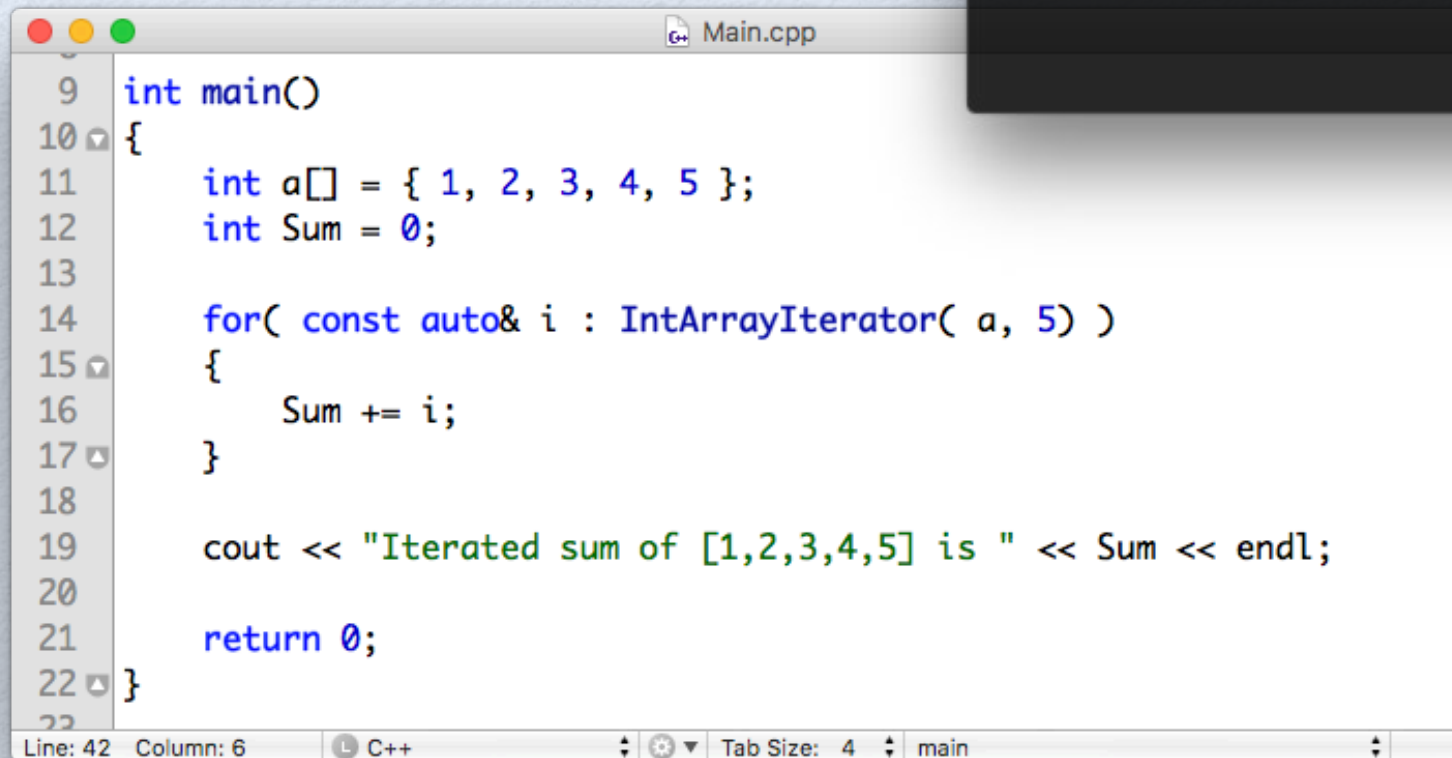
```
Debug
./ArrayIterator
0x7fff5a95e5dc, 4
0x7fff5a95e5dc, 4
0x7fff5a95e5dc, 4
0x7fff5a95e5dc, 4
0x7fff5a95e5dc, 4
0x7fff5a95e6f0, 4
0x7fff5a95e6f4, 4
0x7fff5a95e6f8, 4
0x7fff5a95e6fc, 4
0x7fff5a95e700, 4
0x7fff5a95e59c, 4
0x7fff5a95e59c, 4
0x7fff5a95e59c, 4
0x7fff5a95e59c, 4
0x7fff5a95e59c, 4
0x7fff5a95e6f0, 4
0x7fff5a95e6f4, 4
0x7fff5a95e6f8, 4
0x7fff5a95e6fc, 4
0x7fff5a95e700, 4
```



# Which declaration to use?

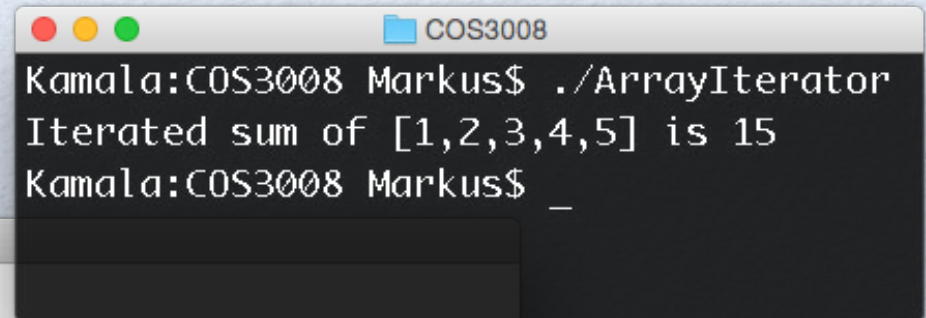
- In a for-each loop always use a reference variable. This avoids unnecessary copies.
- Prefer auto to explicit type declarations. Iterator types can be quite complex and hard to express. Using auto – automatic type deduction – simplifies things dramatically.
- You still need to understand what type deduction means and what the results are. Code becomes less readable, as fewer explicit detail is available.

# Iterator Idiom at Work



```
9 int main()
10 {
11     int a[] = { 1, 2, 3, 4, 5 };
12     int Sum = 0;
13
14     for( const auto& i : IntArrayIterator( a, 5) )
15     {
16         Sum += i;
17     }
18
19     cout << "Iterated sum of [1,2,3,4,5] is " << Sum << endl;
20
21     return 0;
22 }
```

Line: 42 Column: 6 C++ Tab Size: 4 main



```
Kamala:COS3008 Markus$ ./ArrayIterator
Iterated sum of [1,2,3,4,5] is 15
Kamala:COS3008 Markus$ _
```



# A Note on `auto`

- Using `auto` saves typing and prevents correctness and performance issues when dealing with complex types.
- Automatic type deduction via `auto` is no free lunch. The programmer has to guide the compiler to produce the right answer. Failing to do so, can result in a wrong type altogether.
- The use of `auto` can hamper program comprehension. We may have to perform an in-depth study of the code base to understand what type we are dealing with and which methods can be safely invoked on a given object.
- **Using `auto` can produce undefined behavior.** The code still compiles fine, but there is a chance the result will be unpredictable. In this case an explicit type specification is required.





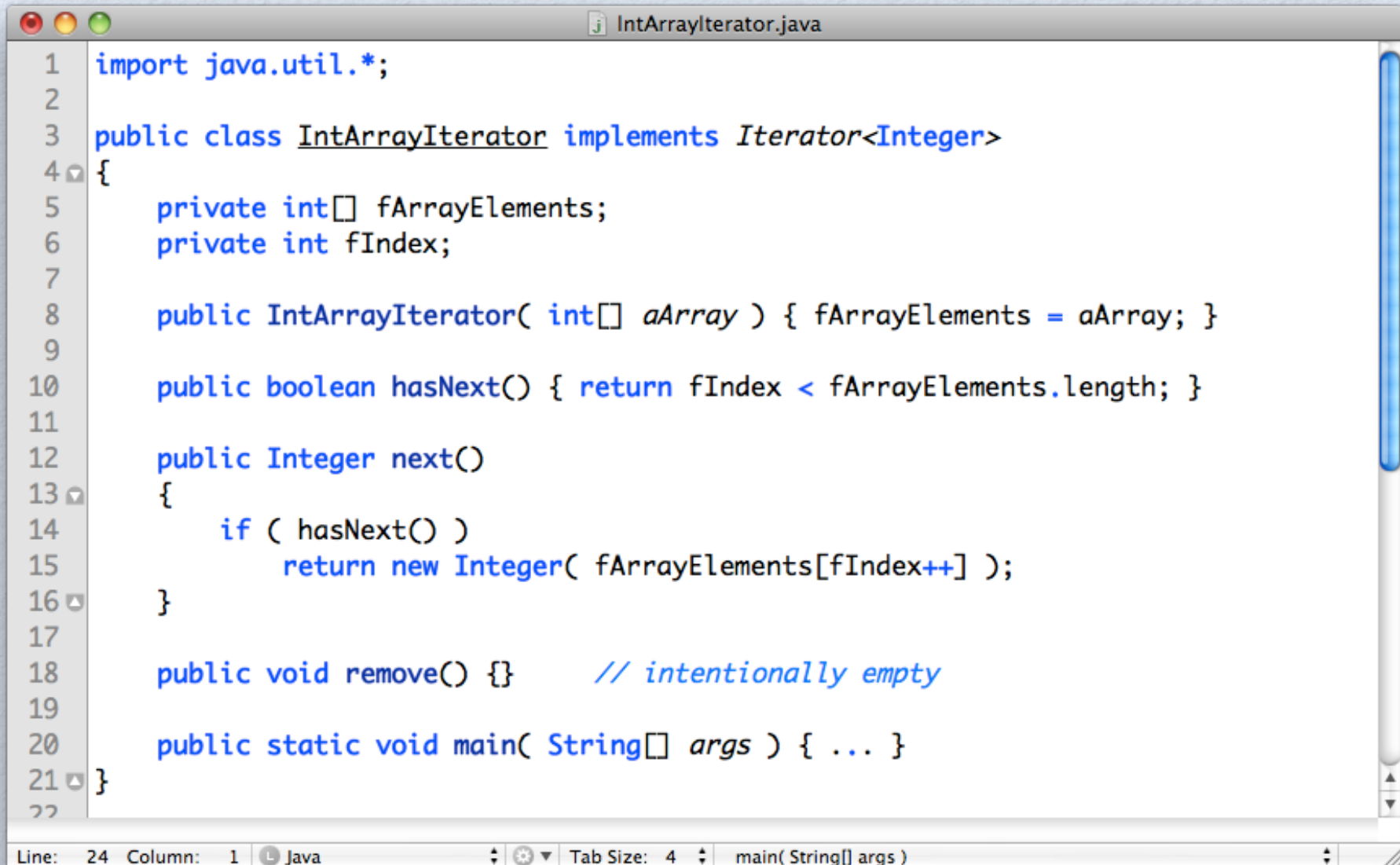
**How can we define an  
iterator in Java?**



# Iterator Interface – `java.util.Iterator`

- `boolean hasNext()`:
  - Returns true if the iteration has more elements. In other words, returns true if `next()` would return an element rather than throwing an exception.
- `E next()`:
  - Returns the next element in the iteration. Calling this method repeatedly until the `hasNext()` method returns false will return each element in the underlying collection exactly once.
- `void remove()`:
  - Removes the last element returned from the underlying collection. This is an optional operation.

# IntArrayIterator in Java

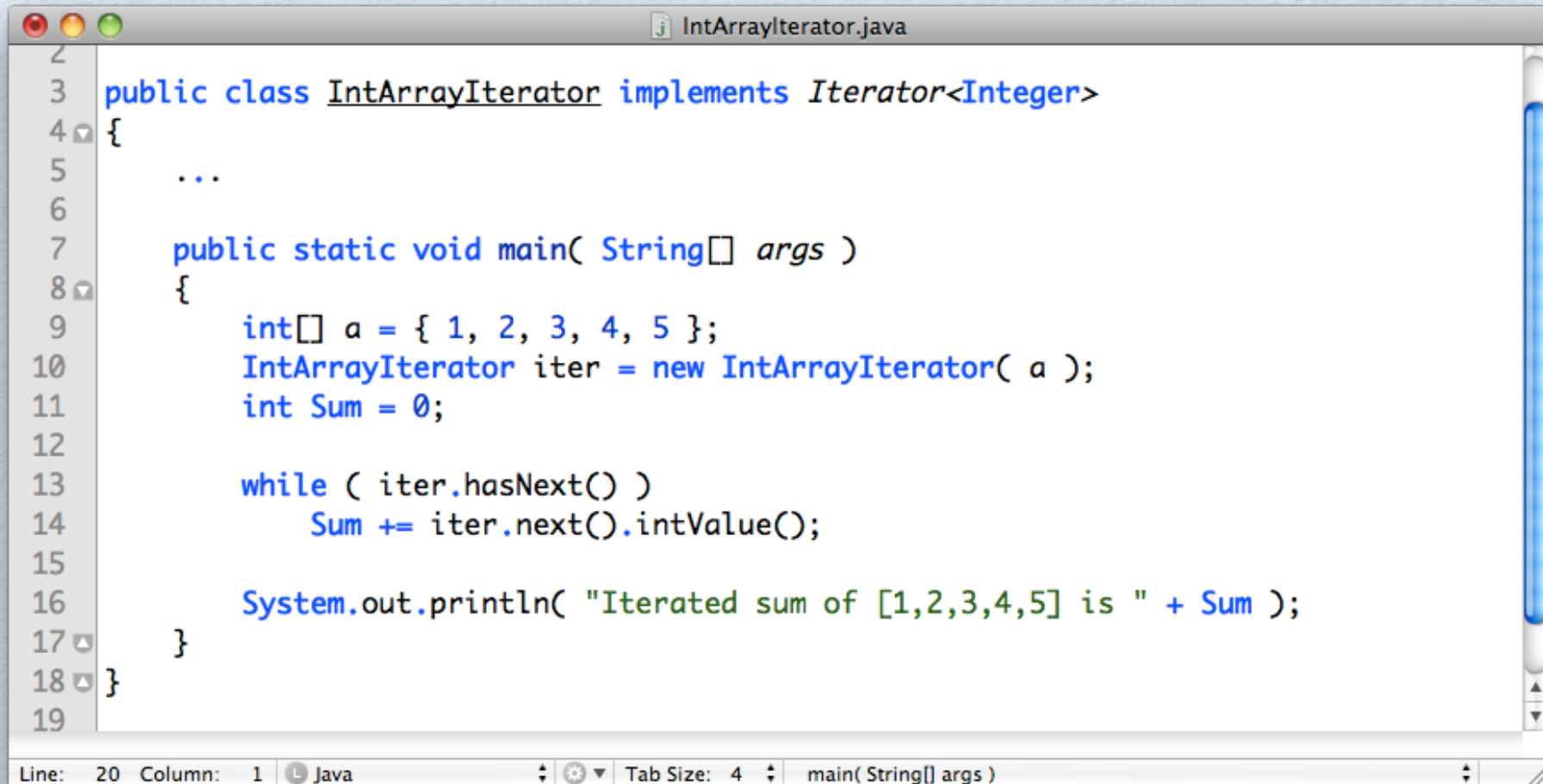


```
1 import java.util.*;
2
3 public class IntArrayIterator implements Iterator<Integer>
4 {
5     private int[] fArrayElements;
6     private int fIndex;
7
8     public IntArrayIterator( int[] aArray ) { fArrayElements = aArray; }
9
10    public boolean hasNext() { return fIndex < fArrayElements.length; }
11
12    public Integer next()
13    {
14        if ( hasNext() )
15            return new Integer( fArrayElements[fIndex++] );
16    }
17
18    public void remove() {}    // intentionally empty
19
20    public static void main( String[] args ) { ... }
21 }
22
```

Line: 24 Column: 1 Java Tab Size: 4 main( String[] args )



# The Iterator's main Method



```
2
3 public class IntArrayIterator implements Iterator<Integer>
4 {
5     ...
6
7     public static void main( String[] args )
8     {
9         int[] a = { 1, 2, 3, 4, 5 };
10        IntArrayIterator iter = new IntArrayIterator( a );
11        int Sum = 0;
12
13        while ( iter.hasNext() )
14            Sum += iter.next().intValue();
15
16        System.out.println( "Iterated sum of [1,2,3,4,5] is " + Sum );
17    }
18 }
19
```

Line: 20 Column: 1 Java Tab Size: 4 main( String[] args )