

**Arrays are indexed sets.**



# Pairs and Maps

- Let  $A$  and  $B$  be sets. The Cartesian product of  $A$  and  $B$ , denoted by  $A \times B$ , is the set of all ordered pairs  $(a, b)$  where  $a \in A$  and  $b \in B$ :

$$A \times B = \{ (a,b) \mid a \in A \text{ and } b \in B \}$$

- A map is an associative container, whose elements are key-value pairs. The key serves as an index into the map, and the value represents the data being stored and retrieved.



# Associative Array (Dictionaries)

- An associate array is a map in which elements are indexed by a key rather than by their position.

$$a[i] = \begin{cases} v, & \text{if } i \mapsto v \text{ in } a \\ \perp, & \text{otherwise} \end{cases}$$

- Example:

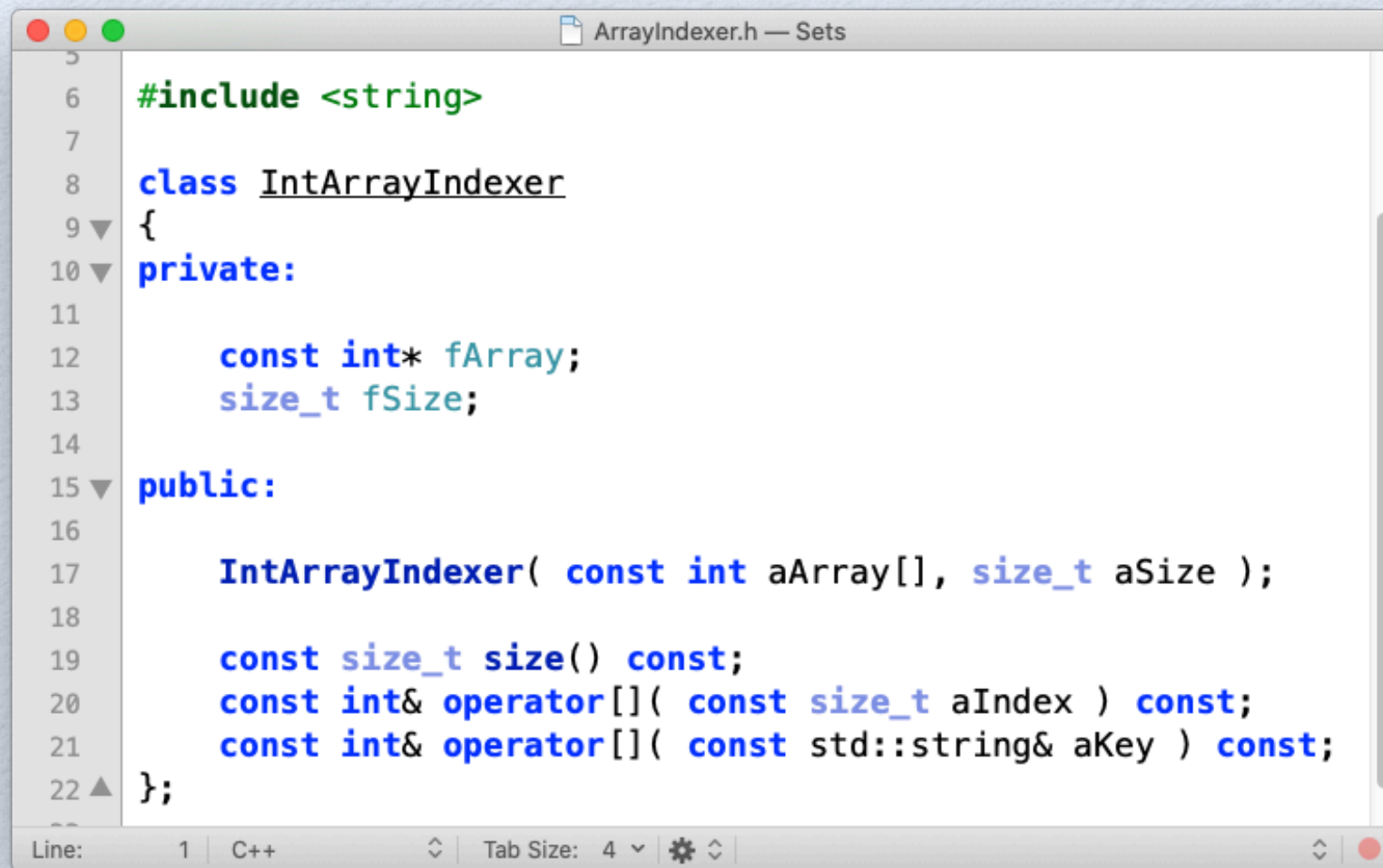
$$a = \{ ("u" \mapsto 345), ("v" \mapsto 2), ("w" \mapsto 39), ("x" \mapsto 5) \}$$

$$a["w"] = 39$$

$$a["z"] = \perp$$

# From Indices to Keys

- We can define an adapter class that defines an indexer:



```
5
6 #include <string>
7
8 class IntArrayIndexer
9 {
10 private:
11
12     const int* fArray;
13     size_t fSize;
14
15 public:
16
17     IntArrayIndexer( const int aArray[], size_t aSize );
18
19     const size_t size() const;
20     const int& operator[]( const size_t aIndex ) const;
21     const int& operator[]( const std::string& aKey ) const;
22 };
```

The screenshot shows a code editor window titled "ArrayIndexer.h — Sets". The code defines a class `IntArrayIndexer` with private attributes `fArray` and `fSize`, and public methods for initialization and indexing. The editor includes a line number margin on the left and a status bar at the bottom showing "Line: 1 C++" and "Tab Size: 4".



# Indexer Constructor

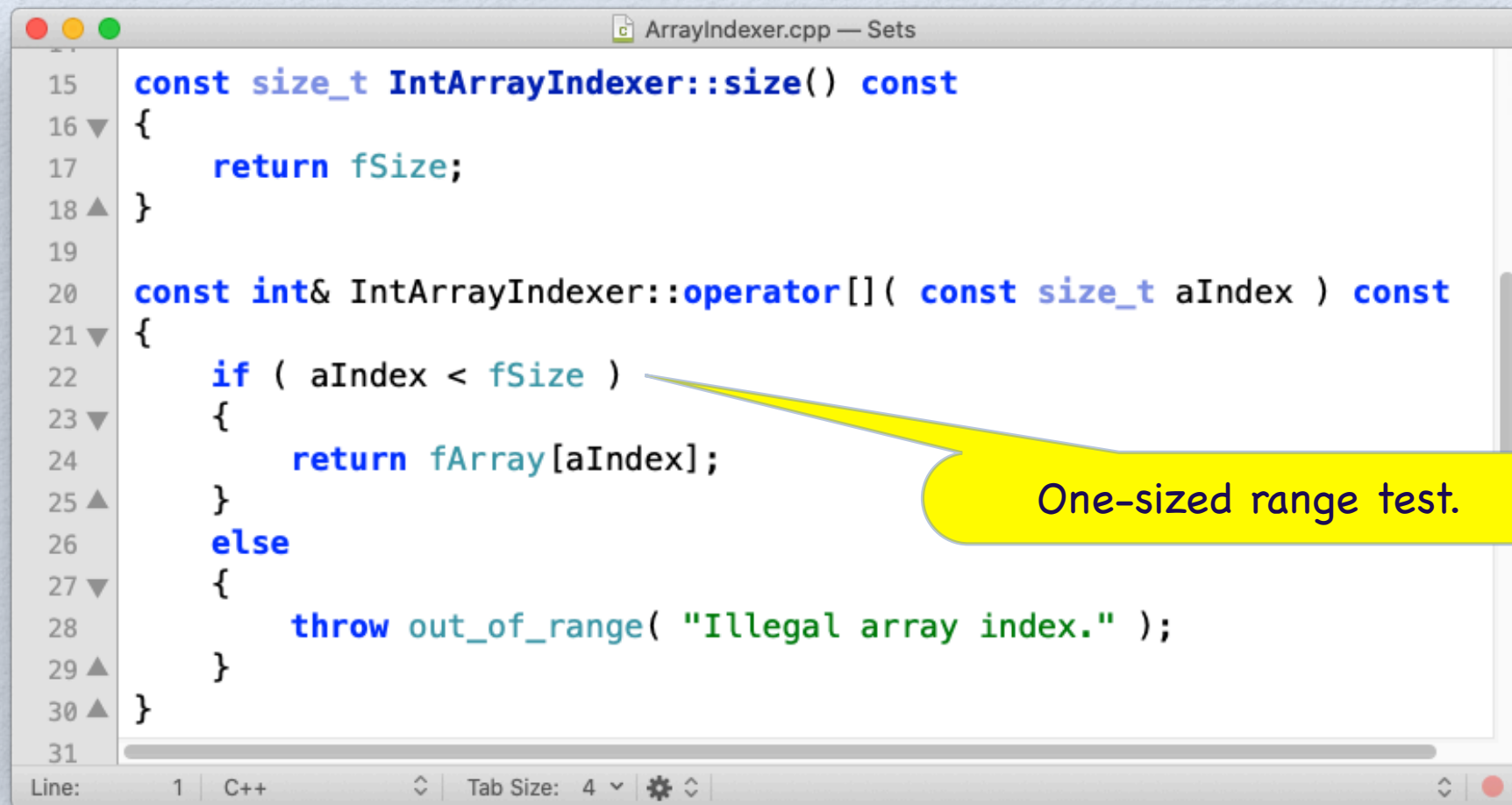
Arrays are passed as pointers to the first element to functions in C++.

```
3
4 #include "ArrayIndexer.h"
5
6 #include <stdexcept>
7
8 using namespace std;
9
10 IntArrayIndexer::IntArrayIndexer( const int aArray[], size_t aSize ) :
11     fArray(aArray),
12     fSize(aSize)
13 {}
14
```

Line: 1 C++ Tab Size: 4

We must use member initializer to initialize const instance variables!

# Basic Indexer Operations

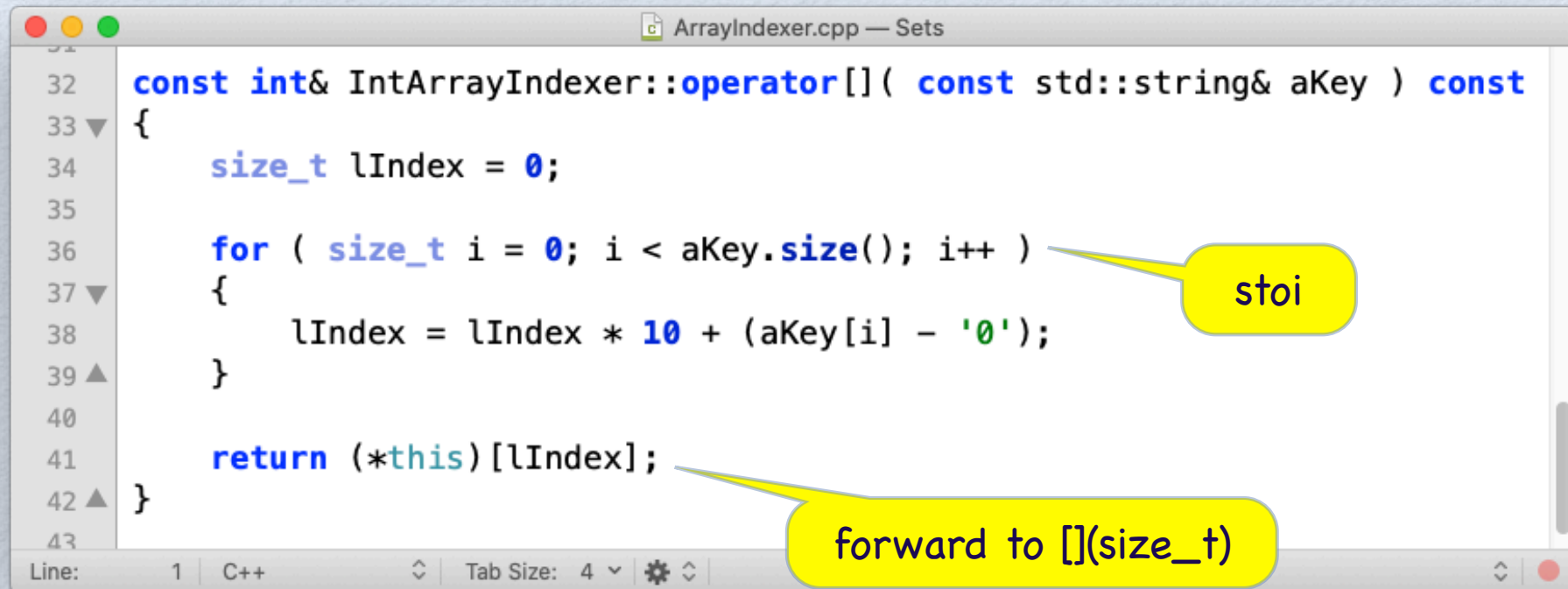


```
15  const size_t IntArrayIndexer::size() const
16  {
17      return fSize;
18  }
19
20  const int& IntArrayIndexer::operator[]( const size_t aIndex ) const
21  {
22      if ( aIndex < fSize )
23      {
24          return fArray[aIndex];
25      }
26      else
27      {
28          throw out_of_range( "Illegal array index." );
29      }
30  }
31
```

One-sized range test.



# The Indexer



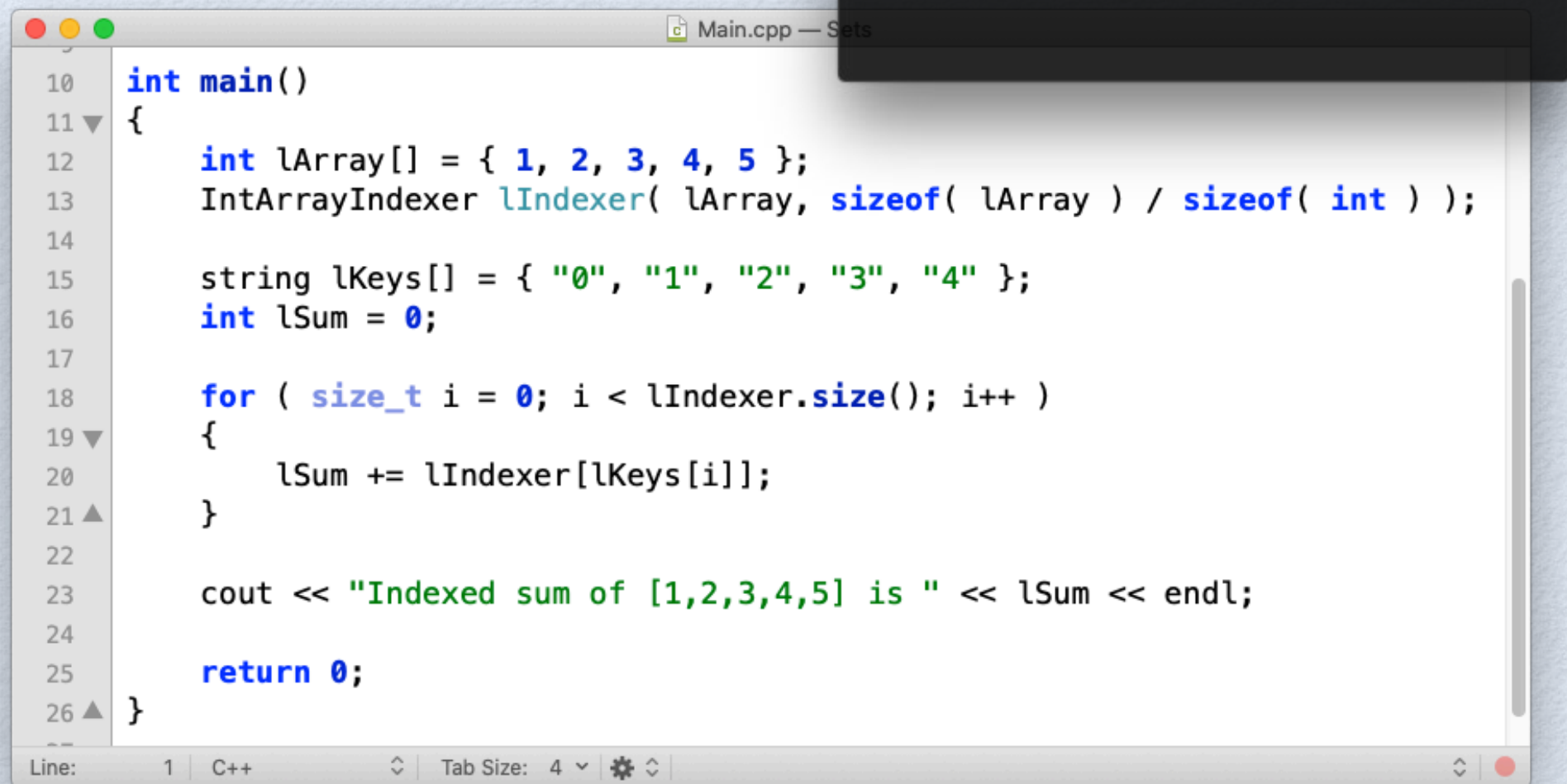
```
32 const int& IntArrayIndexer::operator[]( const std::string& aKey ) const
33 {
34     size_t lIndex = 0;
35
36     for ( size_t i = 0; i < aKey.size(); i++ )
37     {
38         lIndex = lIndex * 10 + (aKey[i] - '0');
39     }
40
41     return (*this)[lIndex];
42 }
```

stoi

forward to [](size\_t)

- We use the **const specifier** to indicate that the operator[]:
  - is a read-only getter
  - does not alter the elements of the underlying collection
- We use a **const reference** to avoid copying the original value stored in the underlying collection.

# Testing the Indexer



```
10 int main()
11 {
12     int lArray[] = { 1, 2, 3, 4, 5 };
13     IntArrayIndexer lIndexer( lArray, sizeof( lArray ) / sizeof( int ) );
14
15     string lKeys[] = { "0", "1", "2", "3", "4" };
16     int lSum = 0;
17
18     for ( size_t i = 0; i < lIndexer.size(); i++ )
19     {
20         lSum += lIndexer[lKeys[i]];
21     }
22
23     cout << "Indexed sum of [1,2,3,4,5] is " << lSum << endl;
24
25     return 0;
26 }
```

Kamala: COS3008 Markus\$ ./ArrayIndexer  
Indexed sum of [1,2,3,4,5] is 15  
Kamala: COS3008 Markus\$ \_





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

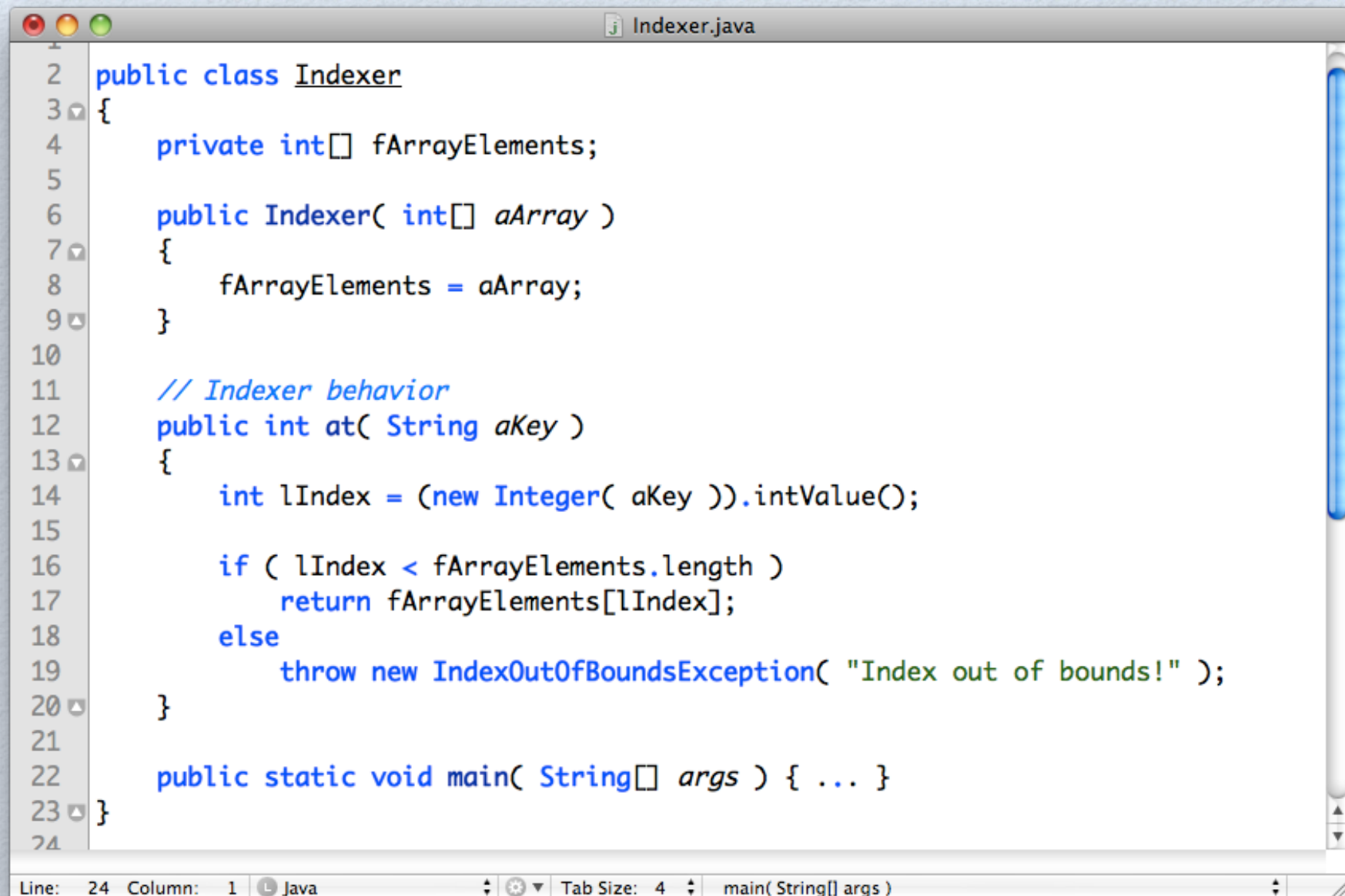


# The Transition to Java

- We need to define an Indexer class.
- Java does not support operator overloading. So, we need to map [] to a member function.
- The built-in type `Integer` provides the required conversion operations.
- We use `IndexOutOfBoundsException` to signal an index error.



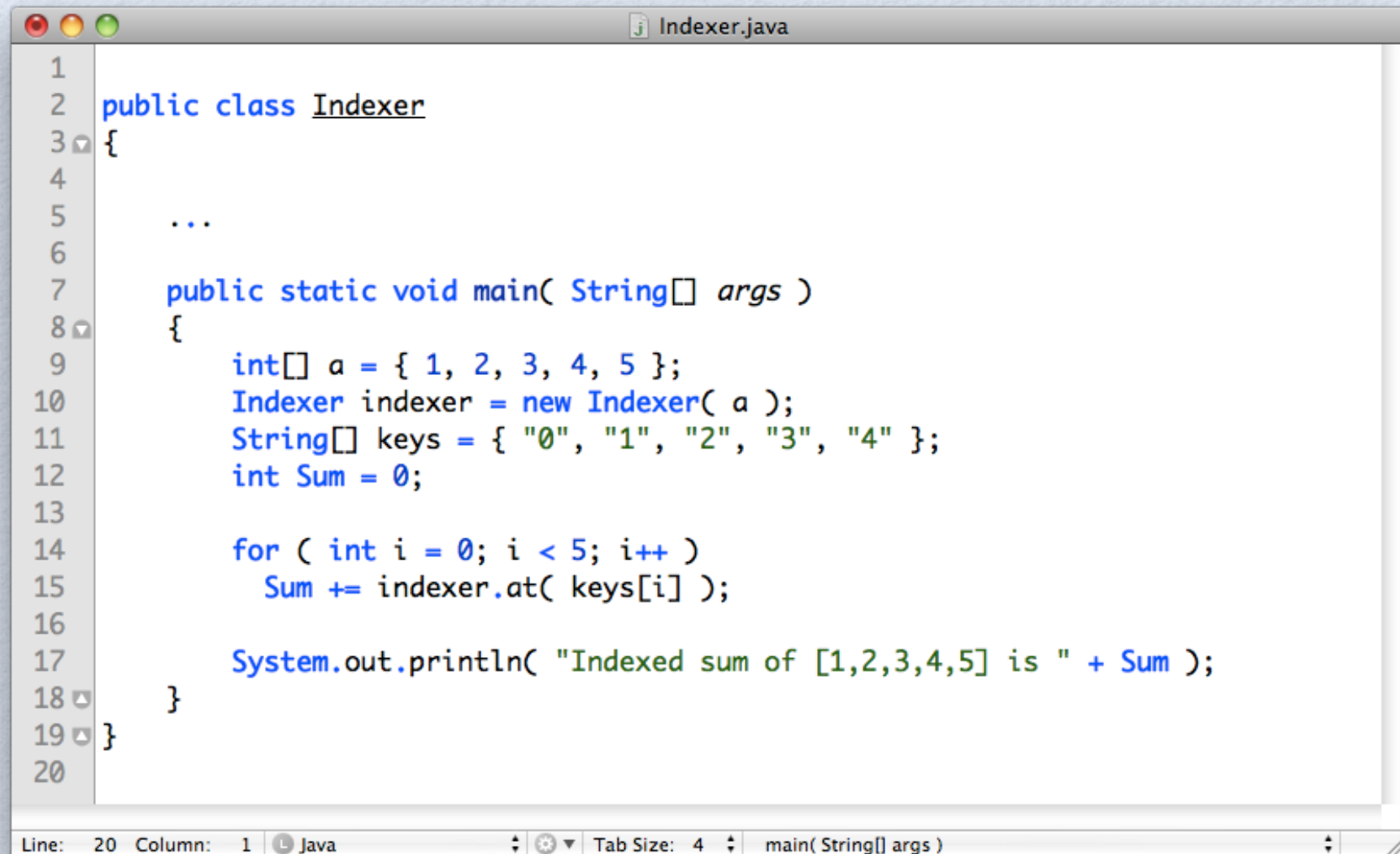
# Indexer's at( String aKey ) Method



```
1
2 public class Indexer
3 {
4     private int[] fArrayElements;
5
6     public Indexer( int[] aArray )
7     {
8         fArrayElements = aArray;
9     }
10
11     // Indexer behavior
12     public int at( String aKey )
13     {
14         int lIndex = (new Integer( aKey )).intValue();
15
16         if ( lIndex < fArrayElements.length )
17             return fArrayElements[lIndex];
18         else
19             throw new IndexOutOfBoundsException( "Index out of bounds!" );
20     }
21
22     public static void main( String[] args ) { ... }
23 }
24
```

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

# The Indexer's main Method



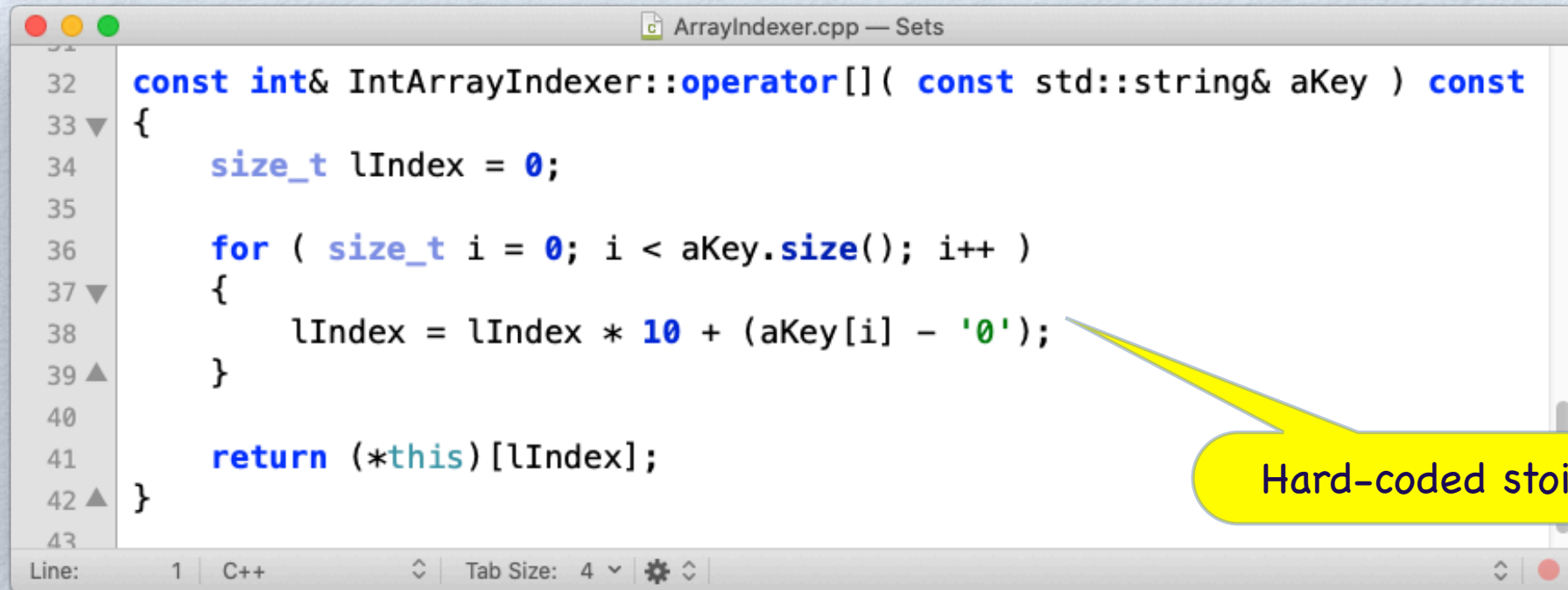
```
1
2 public class Indexer
3 {
4
5     ...
6
7     public static void main( String[] args )
8     {
9         int[] a = { 1, 2, 3, 4, 5 };
10        Indexer indexer = new Indexer( a );
11        String[] keys = { "0", "1", "2", "3", "4" };
12        int Sum = 0;
13
14        for ( int i = 0; i < 5; i++ )
15            Sum += indexer.at( keys[i] );
16
17        System.out.println( "Indexed sum of [1,2,3,4,5] is " + Sum );
18    }
19 }
20
```

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



# Additional Flexibility: Lambda Expressions

# Hard-coded Conversion



The screenshot shows a code editor window titled "ArrayIndexer.cpp — Sets". The code defines a function `operator[]` for `IntArrayIndexer`. It takes a `const std::string& aKey` and returns a `const int&`. The function initializes `lIndex` to 0 and iterates over each character in `aKey`. In the loop, it calculates the index using a hard-coded conversion: `lIndex = lIndex * 10 + (aKey[i] - '0');`. A yellow callout bubble points to the `'0'` character, with the text "Hard-coded stoi".

```
32  const int& IntArrayIndexer::operator[]( const std::string& aKey ) const
33  {
34      size_t lIndex = 0;
35
36      for ( size_t i = 0; i < aKey.size(); i++ )
37      {
38          lIndex = lIndex * 10 + (aKey[i] - '0');
39      }
40
41      return (*this)[lIndex];
42  }
```

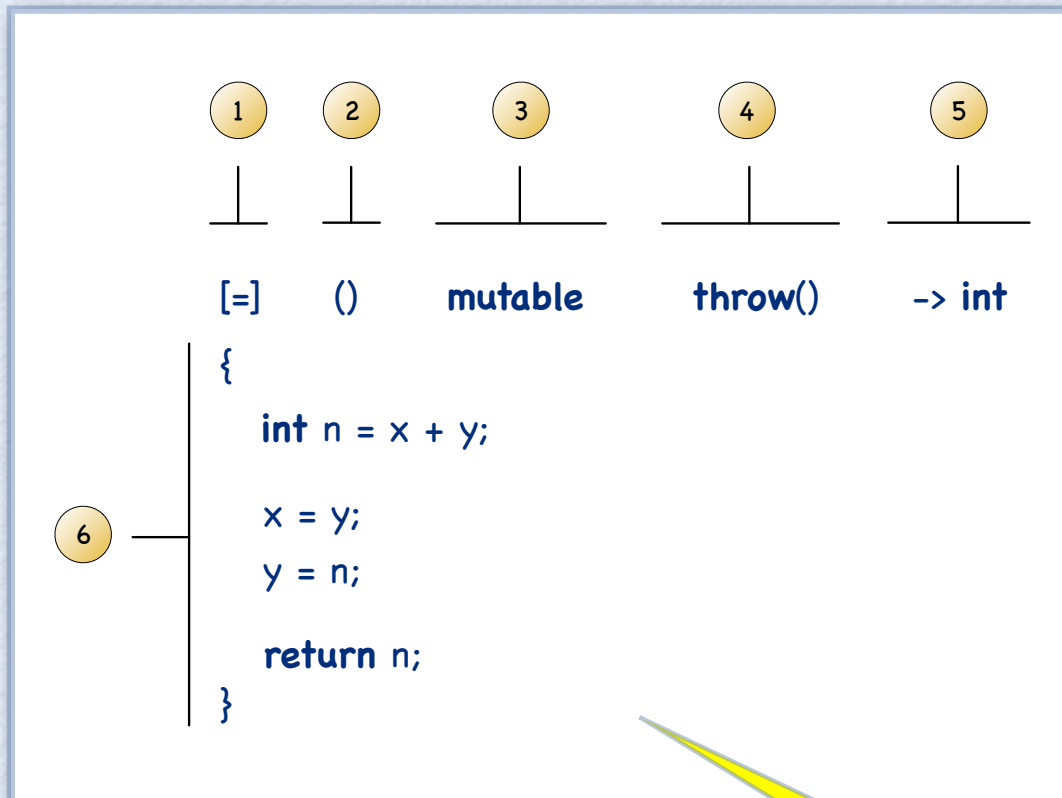
- The indexer uses a hard-coded conversion from string to `size_t`.
- This limits the application of the indexer.
- A better option would be to pass a conversion function explicitly to the indexer, so that we can change the conversion procedure at runtime.



# Lambda Expressions in C++

- C++11 adds support for lambda expressions. A lambda expression, or just `lambda`, is an anonymous function object (closure) that represents a `callable unit of code`.
- Like any function, a lambda has a return type, a parameter list, and a function body.
- Unlike a function, lambdas may be defined inside a function.
- Note, C++ supports two kinds of callables: classes that override the call operator (i.e., `operator()`) and lambda expressions.

# C++ Lambda



1. Capture clause
2. Parameter list, optional
3. Mutable specification, optional
4. Exception specification, optional
5. Trailing return type, optional
6. Lambda body

Variables `x` and `y` are captured by value, but can be altered within the body of lambda.



# C++ Lambda Examples

Function declaration with lambda

```
auto f = [] { return 42; };  
cout << f() << endl; // prints 42
```

```
[] (const string& aLHS, const string& aRHS)  
{ return aLHS.size < aRHS.size(); }
```

```
[lSize] (const string& aString)  
{ return aString.size < lSize; }
```

capture lSize

# Lambda Capture List

<code>[]</code>	Lambda does not use variables from the enclosing environment. Variables from the environment cannot be accessed.
<code>[identifier list]</code>	The variables listed in the comma-separated identifier list are captured by value and copied into the body of lambda. The lambda sees only stored values. Updates in the environment have not effect on lambda.
<code>[&amp;]</code>	All variables in the environment are implicitly captured by reference. Updates in the environment affect lambda.
<code>[=]</code>	All variables in the environment are implicitly captured by value. Values are copied into the body of lambda. Updates in the environment have no effect on lambda.
<code>[&amp;, identifier list]</code>	Implicit capture by reference of all variables in the environment, except those that occur in identifier list. Identifier list must not contain &.
<code>[=, reference list]</code>	Implicit capture by value (copied into the body of lambda) of all variables in the environment, except those that occur in identifier list. Reference list may not contain <b>this</b> and all names must be preceded by &.



# Keep Lambda Captures Simple

# Indexer with Lambda

```
25  
26 class IntArrayIndexer  
27 {  
28     ...  
29  
30 public:  
31  
32     const int& get( const std::string& aKey,  
33                     StringMap aFunc = [](const std::string& aNumber )  
34     {  
35         size_t lIndex = 0;  
36  
37         for ( size_t i = 0; i < aNumber.size(); i++ )  
38         {  
39             lIndex = lIndex * 10 + (aNumber[i] - '0');  
40         }  
41  
42         return lIndex;  
43     }) const;  
44 };
```

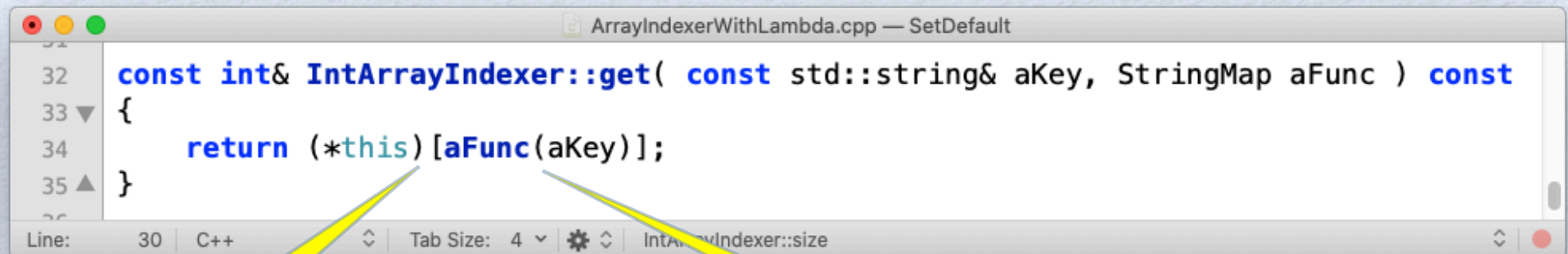
new get function, operator[] expects one argument only

conversion function as lambda and default values the header

Line: 21:49 C++ Tab Size: 4 IntArrayIndexer



# Implementation of get()



```
ArrayIndexerWithLambda.cpp — SetDefault
32 const int& IntArrayIndexer::get( const std::string& aKey, StringMap aFunc ) const
33 {
34     return (*this)[aFunc(aKey)];
35 }
```

Line: 30 C++ Tab Size: 4 IntArrayIndexer::size

forward to indexer

call lambda for conversion from string to size\_t

# `std::function<class Ret, class... Args>`

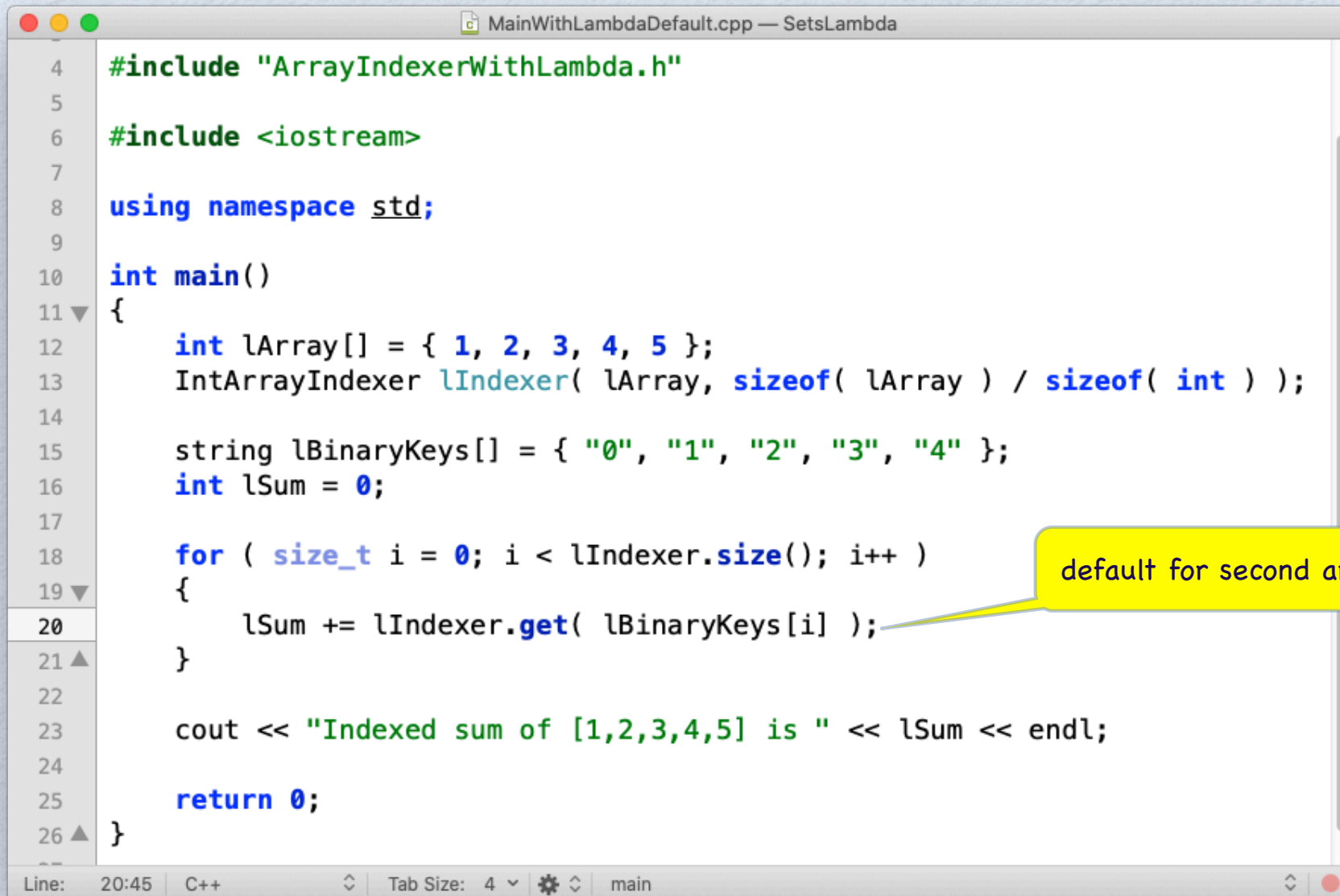
- Technically, a variable that stores a lambda is a function pointer. However, function pointers may lead to unreadable specifications or worse.
- C++11 offer a function wrapper `std::function<class Ret, class... Args>` for this purpose.
- Technically, `std::function` is a varadic template (it takes a variable number of arguments) that allows us to capture any function signature.
- For example:

```
using StringMap = std::function<size_t(const std::string&)>;
```

defines type `StringMap` as a function from `const string&` to `size_t` using C++11's typedef declaration (i.e, `using TypeName = aType;`).



# Application of Default Implementation



```
4  #include "ArrayIndexerWithLambda.h"
5
6  #include <iostream>
7
8  using namespace std;
9
10 int main()
11 {
12     int lArray[] = { 1, 2, 3, 4, 5 };
13     IntArrayIndexer lIndexer( lArray, sizeof( lArray ) / sizeof( int ) );
14
15     string lBinaryKeys[] = { "0", "1", "2", "3", "4" };
16     int lSum = 0;
17
18     for ( size_t i = 0; i < lIndexer.size(); i++ )
19     {
20         lSum += lIndexer.get( lBinaryKeys[i] );
21     }
22
23     cout << "Indexed sum of [1,2,3,4,5] is " << lSum << endl;
24
25     return 0;
26 }
```

default for second argument

# C++11 auto

- C++11 also introduces auto typing, that is, we can declare variables using auto as type name:

```
auto f = [] (const string& aLHS, const string& aRHS)
        { return aLHS.size < aRHS.size() };
```

- 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.
- Unfortunately, auto type specifier cannot be used in function parameters. For parameters we need to specify the actual type.



# Indexer with Binary Keys

```
10  int main()
11  {
12      int lArray[] = { 1, 2, 3, 4, 5 };
13      IntArrayIndexer lIndexer( lArray, sizeof( lArray ) / sizeof( int ) );
14
15      string lBinaryKeys[] = { "000", "001", "010", "011", "100" };
16      int lSum = 0;
17
18      auto lMapBinary = [] ( const std::string& aNumber )
19      {
20          size_t lIndex = 0;
21
22          for ( size_t i = 0; i < aNumber.size(); i++ )
23          {
24              lIndex = (lIndex << 1) + (aNumber[i] - '0');
25          }
26
27          return lIndex;
28      };
29
30      for ( size_t i = 0; i < lIndexer.size(); i++ )
31      {
32          lSum += lIndexer.get( lBinaryKeys[i], lMapBinary );
33      }
34
35      cout << "Indexed sum of [1,2,3,4,5] is " << lSum << endl;
36
37      return 0;
38  }
```

lambda

application

**Lambda Expression allow for  
highly flexible data types.**