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.

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

- Example:

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

  $a[\text{“w”}] = 39$

  $a[\text{“z”}] = \bot$

From Indices to Keys

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

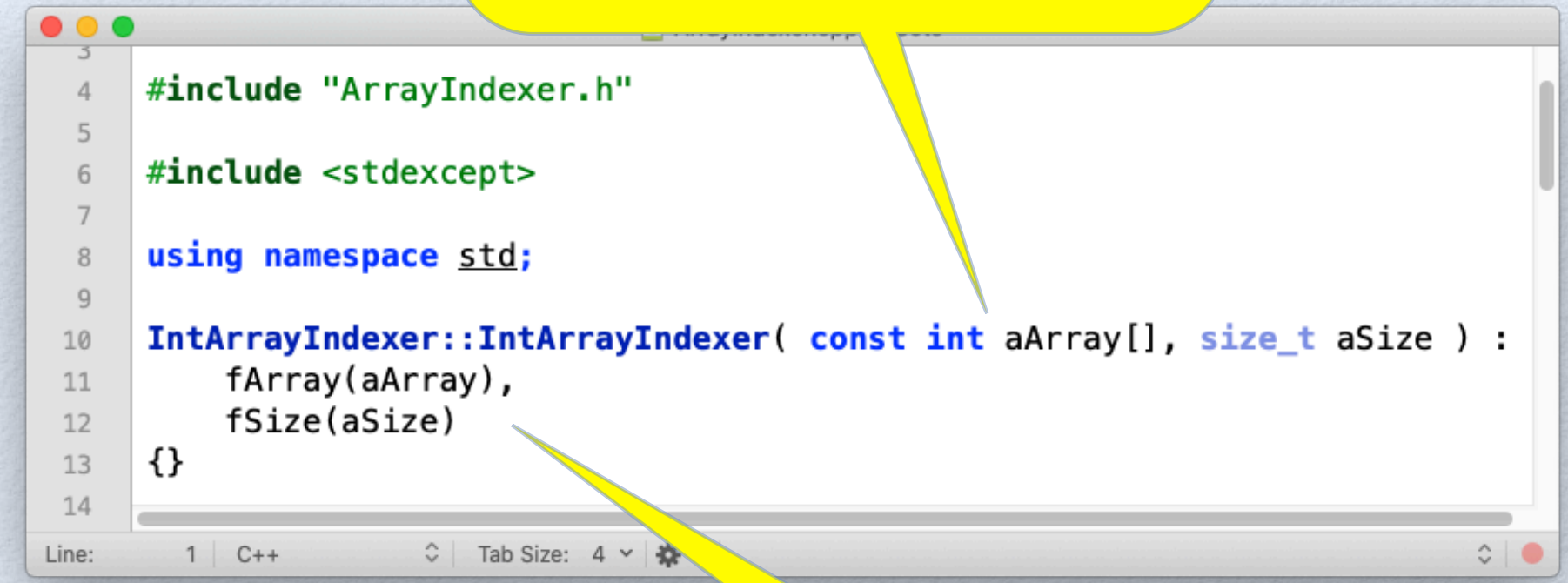
ArrayIndexer.h — Sets

```
#include <string>

class IntArrayIndexer
{
private:

    const int* fArray;
    size_t fSize;

public:

    IntArrayIndexer( const int aArray[], size_t aSize );

    const size_t size() const;
    const int& operator[]( const size_t aIndex ) const;
    const int& operator[]( const std::string& aKey ) const;
};
```

Line: 1 C++ Tab Size: 4

Indexer Constructor

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



```
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

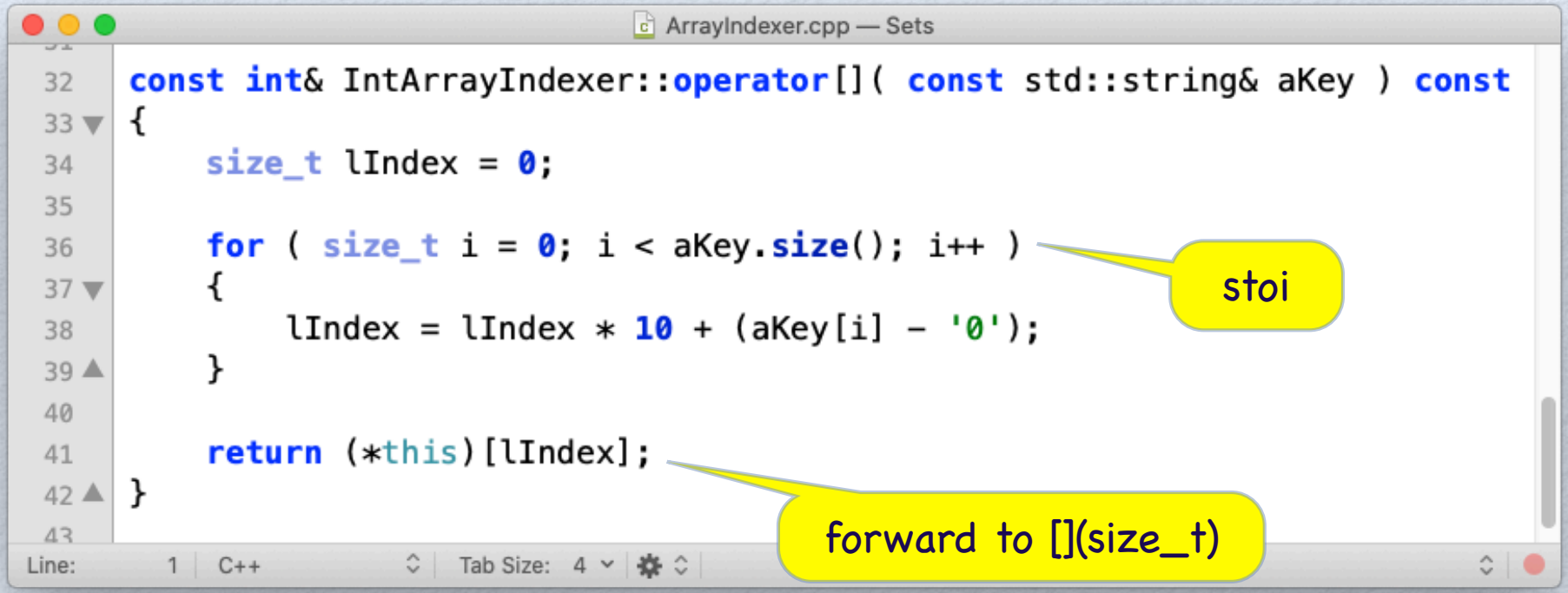
ArrayIndexer.cpp — Sets

```
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
```

Line: 1 C++ Tab Size: 4

One-sized range test.

The Indexer



- 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

COS3008

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

Main.cpp — Sets

```
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  }
```

Line: 1 C++ Tab Size: 4

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

Indexer.java

```
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

Indexer.java

```
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

ArrayIndexer.cpp — Sets

```
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  }
43
```

Hard-coded stoi

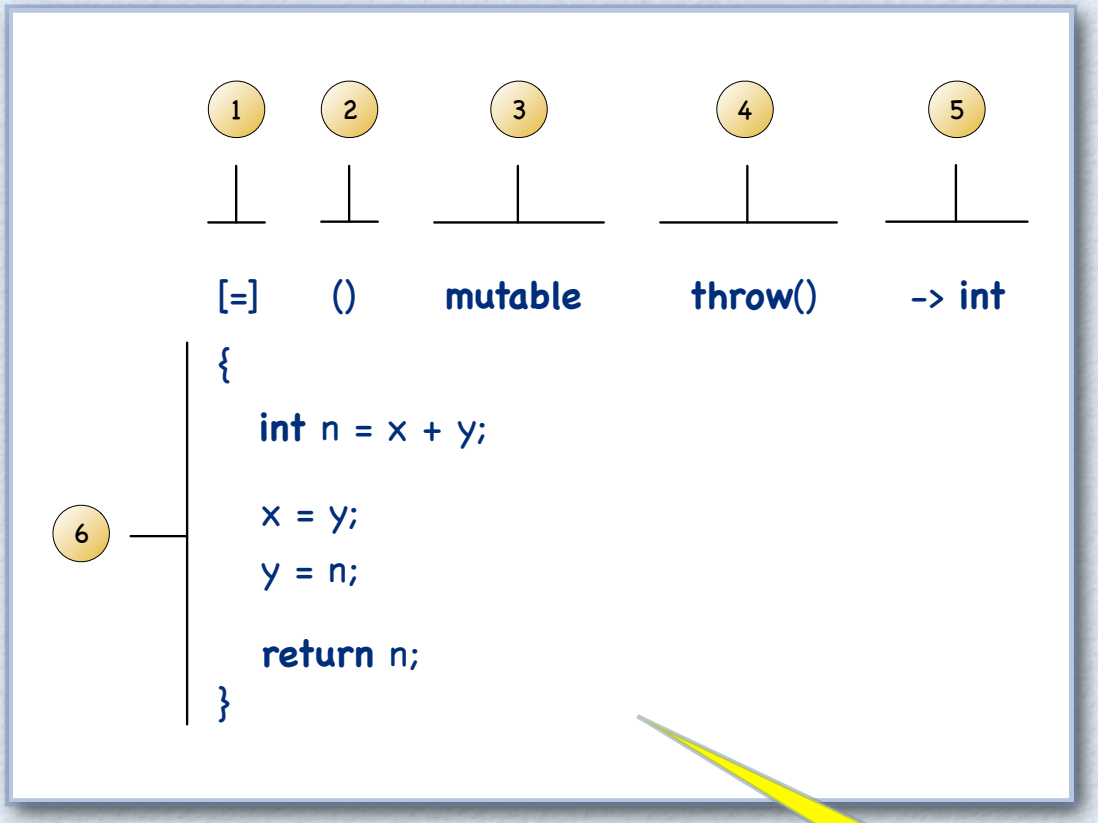
Line: 1 C++ Tab Size: 4

- 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() };
```

capture lSize

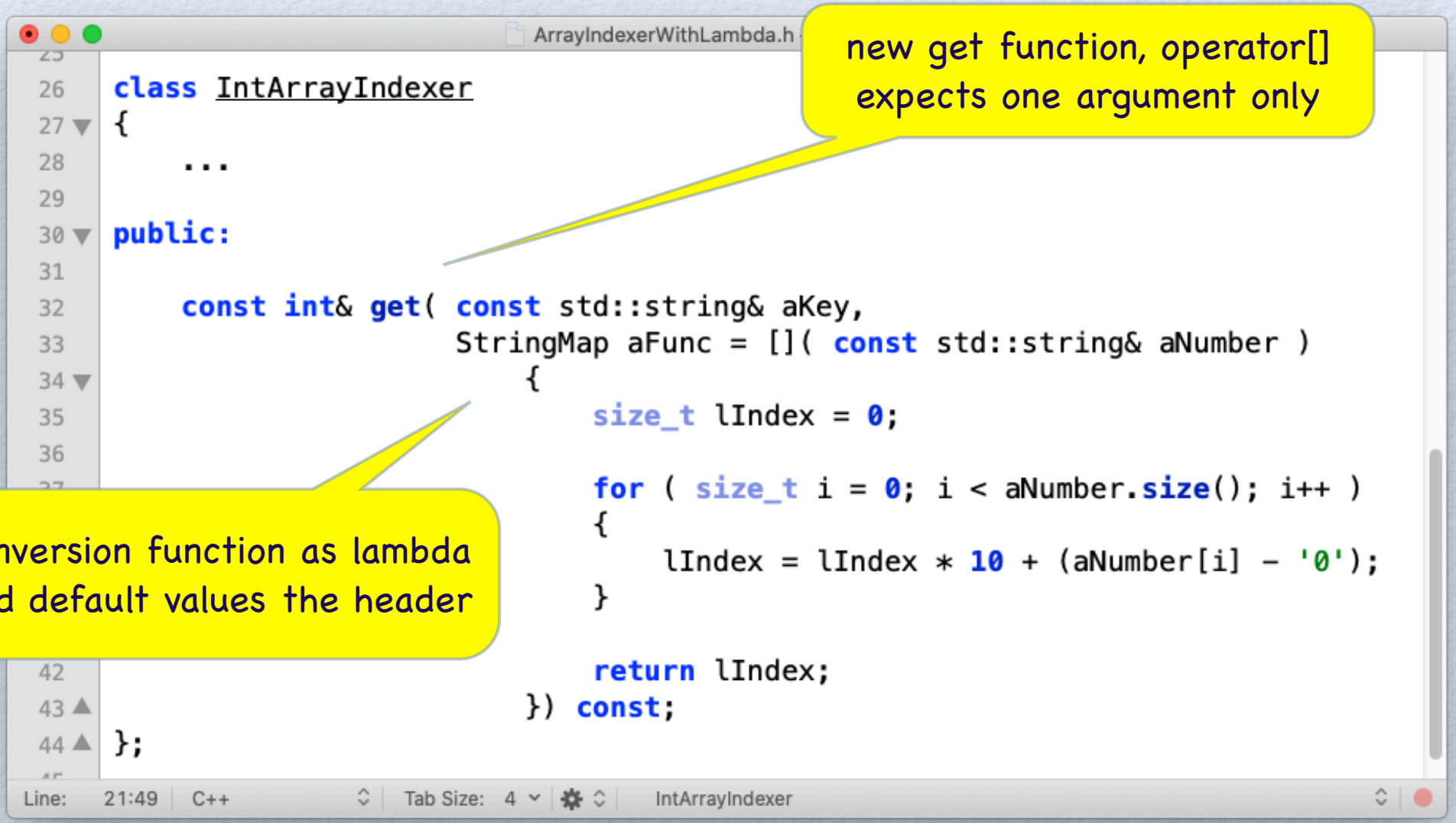
```
[lSize] (const string& aString)
   { return aString.size() < lSize; };
```

Lambda Capture List

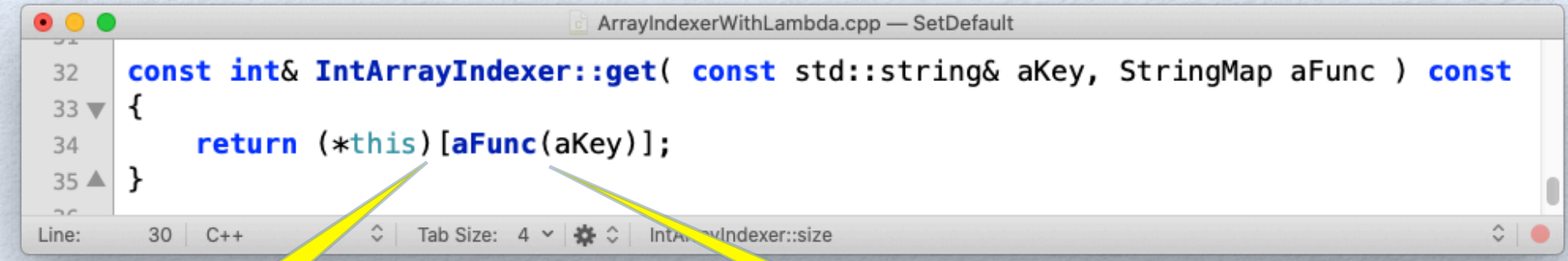
[]	Lambda does not use variables from the enclosing environment. Variables from the environment cannot be accessed.
[identifier list]	The variables listed in the comma-separated identifier list are captured by value and copied into the body of lambda. The lambda sees only stored values. Updates in the environment have not effect on lambda.
[&]	All variables in the environment are implicitly captured by reference. Updates in the environment affect lambda.
[=]	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.
[&, identifier list]	Implicit capture by reference of all variables in the environment, except those that occur in identifier list. Identifier list must not contain &.
[=, reference list]	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 **this** and all names must be preceded by &.

Keep Lambda Captures Simple

Indexer with Lambda



Implementation of get()



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

MainWithLambdaDefault.cpp — SetsLambda

```
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  }
```

Line: 20:45 C++ Tab Size: 4 main

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 specifies cannot be used in function parameters. For parameters we need to specify the actual type.

Indexer with Binary Keys

MainWithLambda.cpp — SetsLambda

```
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  }
```

Line: 1 C++ Tab Size: 4

lambda

application

Lambda Expression allow for highly flexible data types.