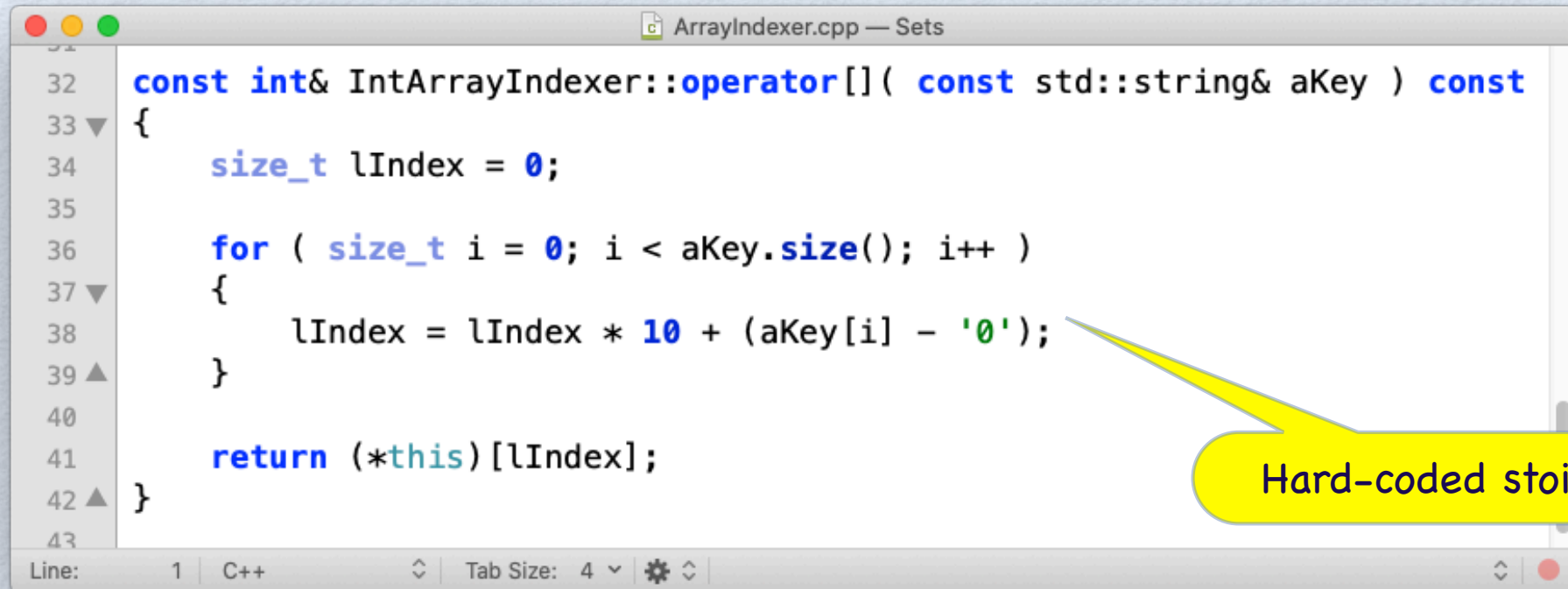


Additional Flexibility: Lambda Expressions

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
```

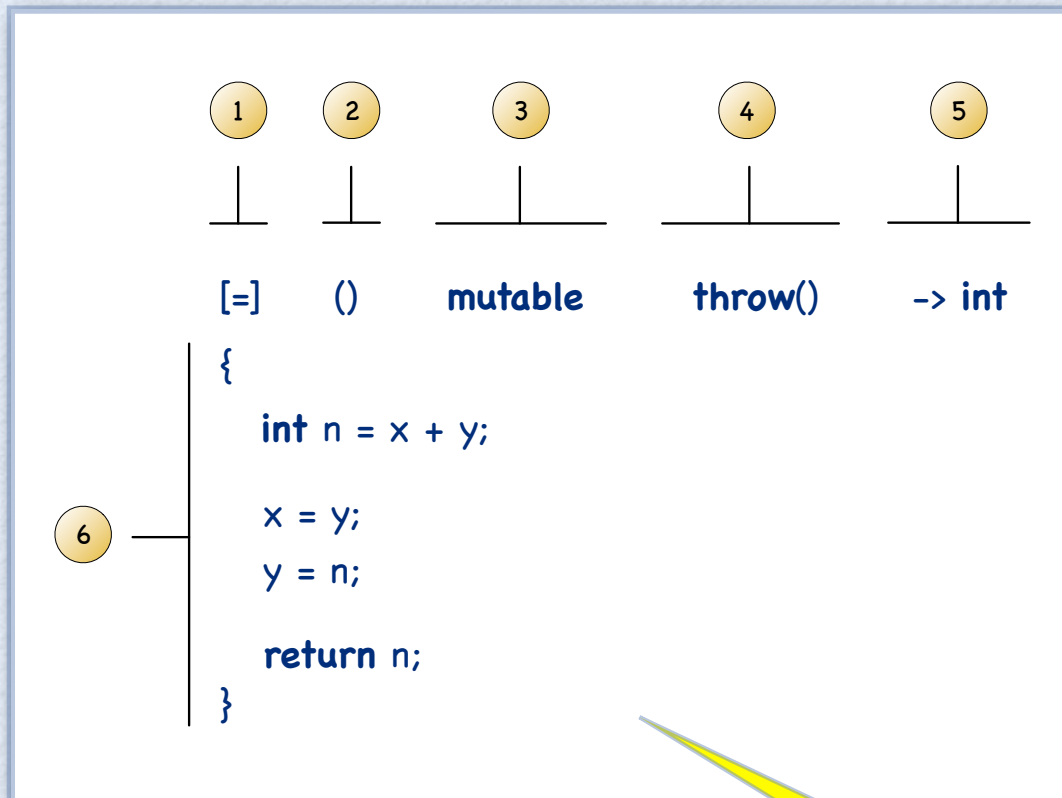
Hard-coded stoi

- 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; };  
std::cout << f() << std::endl; // prints 42
```

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

capture lSize

```
[lSize] (const std::string& aString)  
{ return aString.size() < 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>[&]</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>[&, 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 this and all names must be preceded by &.

Keep Lambda Captures Simple

Indexer with Lambda

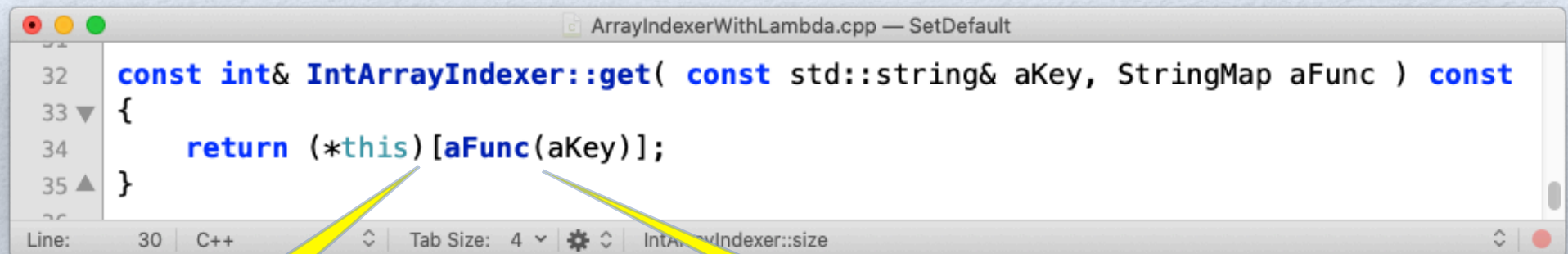
```
25  
26 class IntArrayIndexer  
27 {  
28     ...  
29  
30 public:  
31  
32     const int& get( const std::string& aKey,  
33                     StringMap aFunc = [](const std::string& aNumber )  
34                     {  
35                         size_t lIndex = 0;  
36  
37                         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 };  
45
```

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

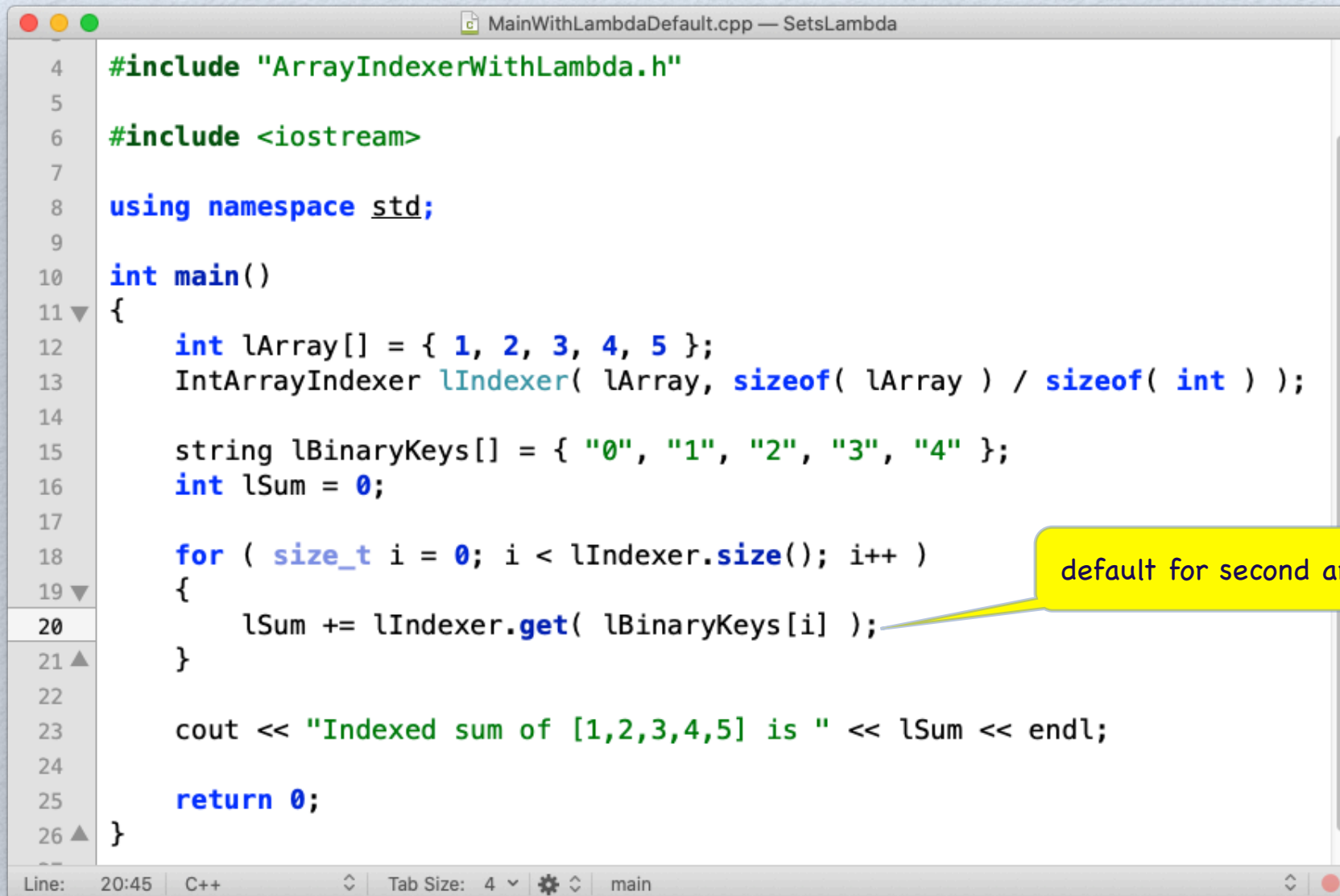
`std::function<class Ret, class... Args>`

- 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

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

C++11 auto

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

```
auto f = [] (const std::string& aLHS, const std::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.**