



**COS30008**

**Data Structures and Patterns**

Memory Management



# Memory Management and Copy Control

## Overview

- Types of memory
- Copy constructor, assignment operator, and destructor
- Reference counting with smart pointers

## References

- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)
- Bruno R. Preiss: Data Structures and Algorithms with Object-Oriented Design Patterns in C++. John Wiley & Sons, Inc. (1999)
- Andrew W. Appel with Jens Palsberg: Modern Compiler Implementation in Java. 2nd Edition, Cambridge University Press (2002).
- Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman: Compilers - Principles, Techniques, and Tools. Addison-Wesley (1988)



# Static Read-Write Memory

- C++ allows for two forms of global variables:
  - Static non-class variables,
  - Static class variables.
- Static variables are mapped to the global memory. Access to them depends on the visibility specifiers.
- We can find a program's global memory in the so-called read-write `.data` segment.

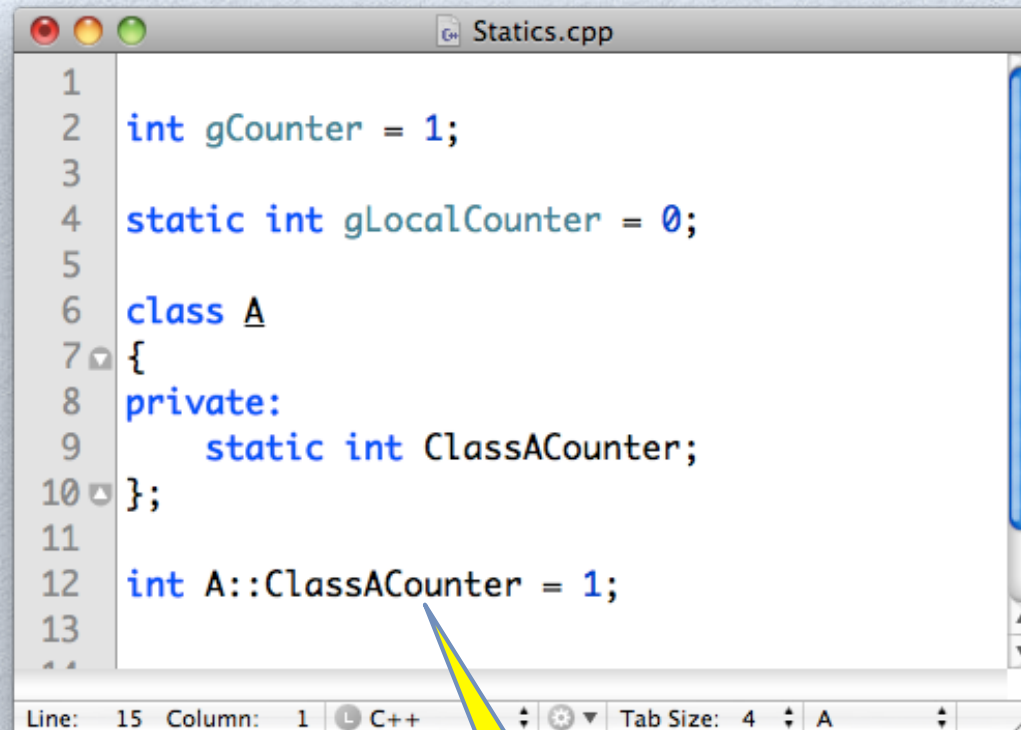


# The Keyword `static`

- The keyword `static` can be used to
  - mark the `linkage` of a variable or function `internal`,
  - `retain the value` of a local variable between function calls,
  - declare a `class instance variable`,
  - define a `class method`.



# Read-Write Static Variables



The screenshot shows a code editor window titled 'Statics.cpp'. The code is as follows:

```
1  
2 int gCounter = 1;  
3  
4 static int gLocalCounter = 0;  
5  
6 class A  
7 {  
8 private:  
9     static int ClassACounter;  
10 };  
11  
12 int A::ClassACounter = 1;  
13
```

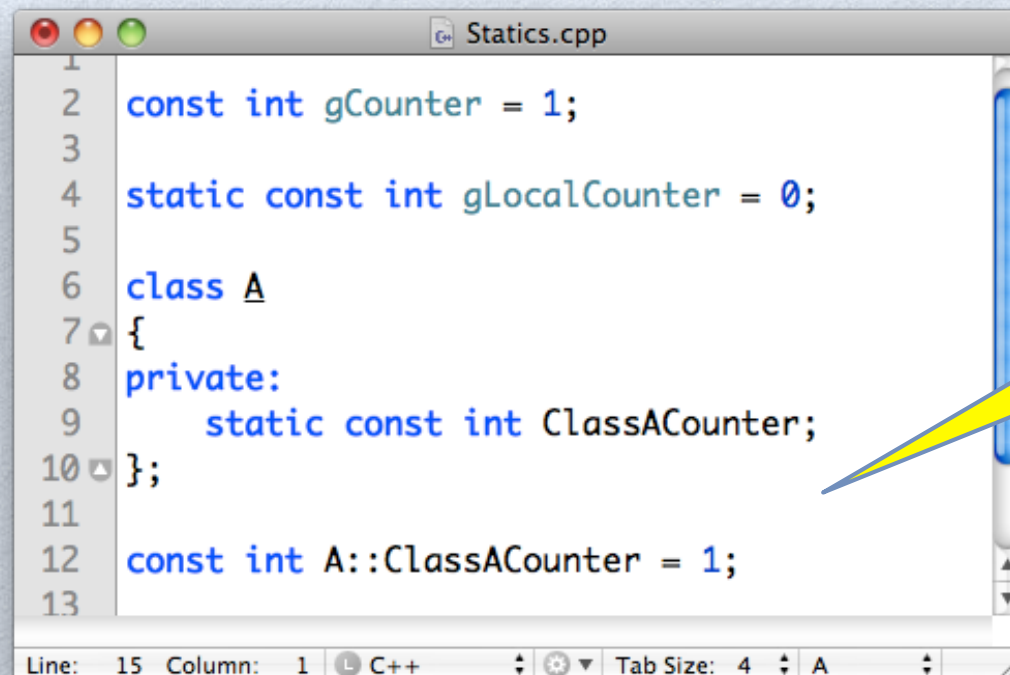
A yellow callout bubble points to line 12, containing the text: 'Static class variables must be initialized outside the class.'

The editor's status bar at the bottom indicates 'Line: 15 Column: 1 C++ Tab Size: 4 A'.



# Static Read-Only Memory

- In combination with the `const` specifier we can also define read-only global variables or class variables:



```
1
2  const int gCounter = 1;
3
4  static const int gLocalCounter = 0;
5
6  class A
7  {
8  private:
9      static const int ClassACounter;
10 };
11
12 const int A::ClassACounter = 1;
13
```

The screenshot shows a code editor window with the title 'Statics.cpp'. The code defines a global constant integer `gCounter` with value 1, a static global constant integer `gLocalCounter` with value 0, and a class `A` with a private static constant integer `ClassACounter`. The definition for `A::ClassACounter` is provided as `const int A::ClassACounter = 1;` at the bottom. The editor interface includes a line number margin on the left and a status bar at the bottom showing 'Line: 15 Column: 1 C++ Tab Size: 4'.

Const variables are often stored in the program's read-only .text segment.

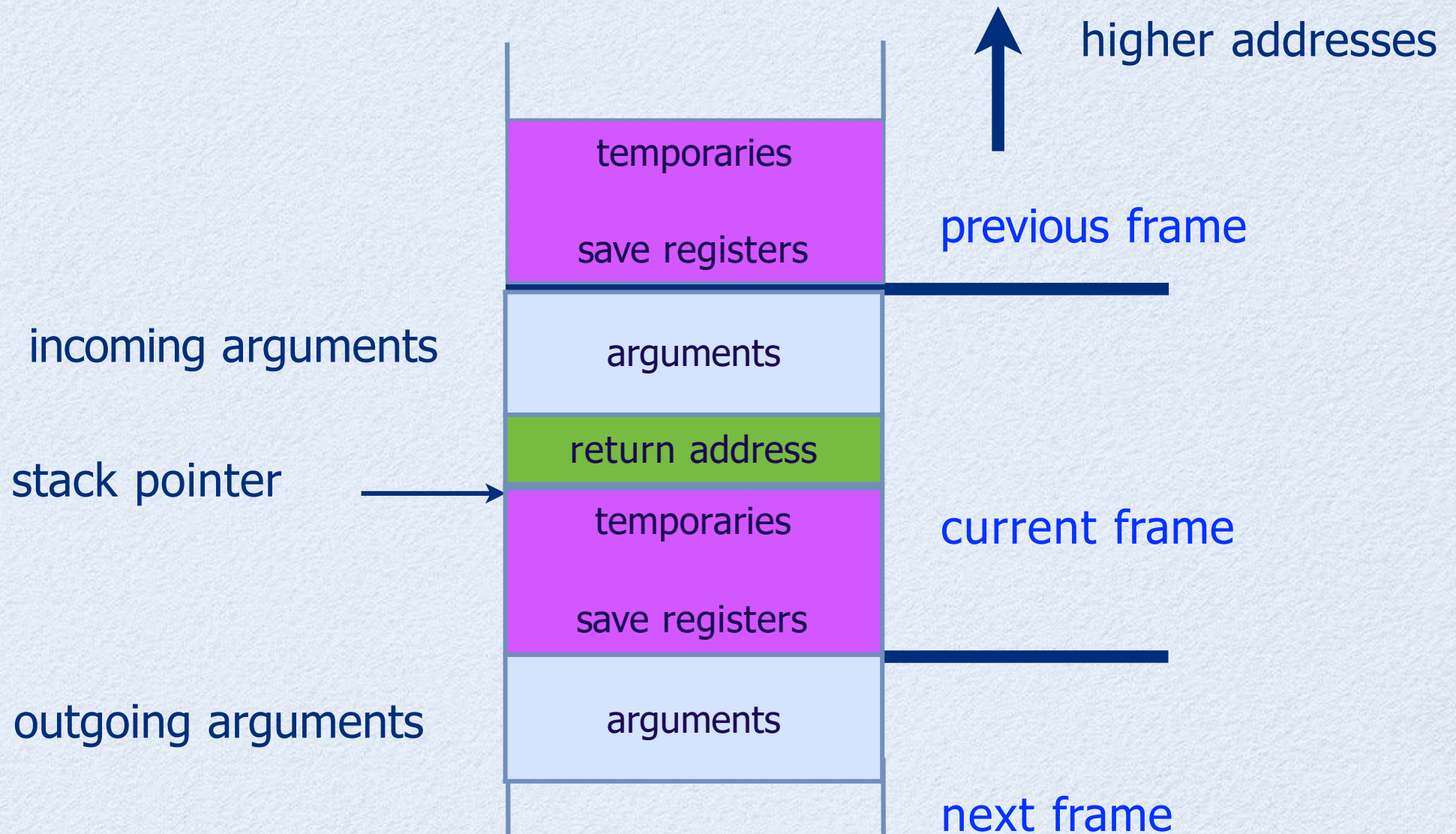


# Program Memory: Stack

- All **value-based** objects are stored in the program's stack.
- The program stack is automatically allocated and freed.
- References to stack locations are only valid when passed to a callee. References to stack locations cannot be returned from a function.



# Stack Frames (C)





# Program Memory: Heap

- Every program maintains a heap for dynamically allocated objects.
- Each heap object is accessed through a pointer.
- Heap objects are **not automatically freed** when pointer variables become inaccessible (i.e., go out of scope).
- **Memory management** becomes essential in C++ to reclaim memory and to prevent the occurrences of so-called **memory leaks**.



# List::~~List()

```
List.h

~List()                                     // destructor - frees all nodes
{
    while ( fRoot != nullptr )
    {
        if ( fRoot != &fRoot->getPrevious() )    // more than one element
        {
            Node* lTemp = const_cast<Node*>(&fRoot->getPrevious()); // select last

            lTemp->isolate();                        // remove from list
            delete lTemp;                            // free
        }
        else
        {
            delete fRoot;                            // free last
            break;                                    // stop loop
        }
    }
}
```

Line: 68 Column: 9 C++ Tab Size: 4 fCount

Release memory associated with list node object on the heap.



# The Dominion Over Objects

- Alias control is one of the most difficult problems to master in object-oriented programming.
- Aliases are the default in reference-based object models used, for example, in Java and C#.
- To guarantee uniqueness of value-based objects in C++, we are required to define copy constructors.



# The Copy Constructor

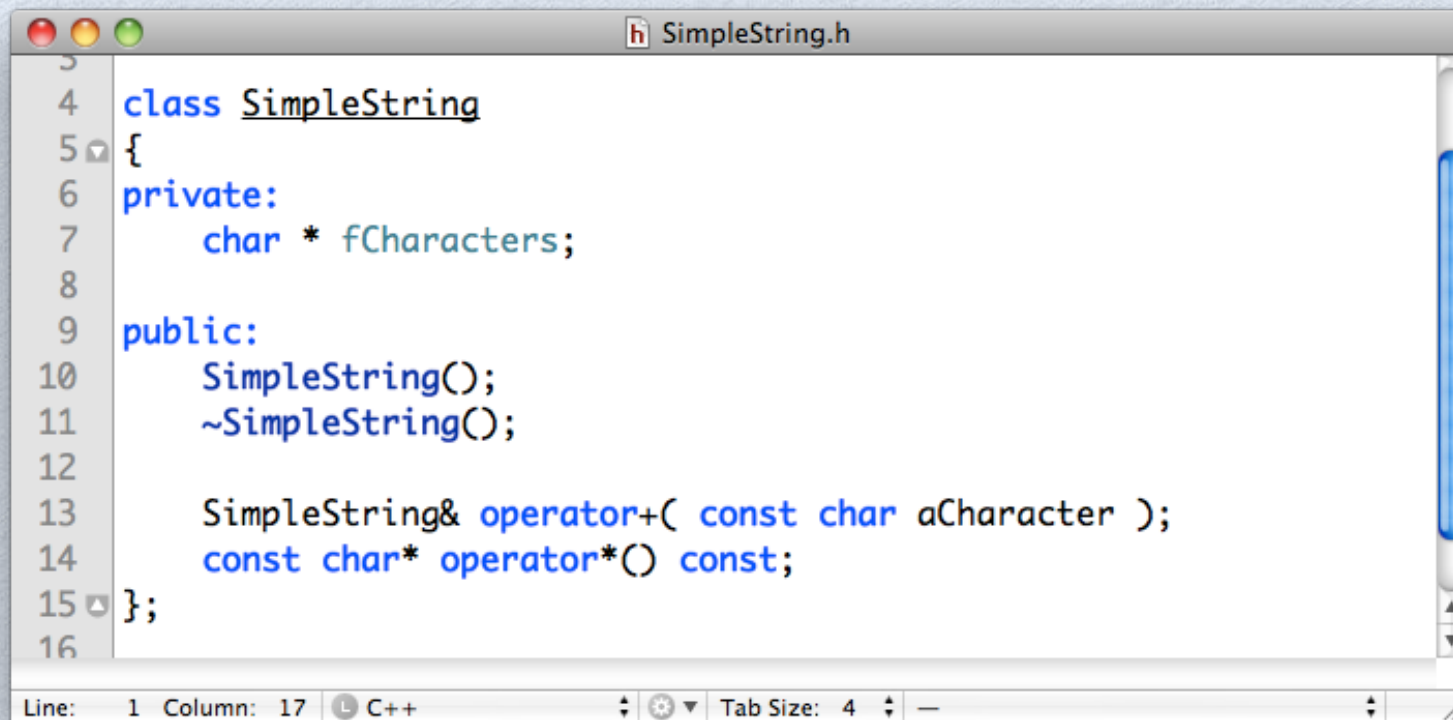
- Whenever one defines a new type, one needs to specify **implicitly** or **explicitly** what has to happen when objects of that type are **copied**, **assigned**, and **destroyed**.
- The **copy constructor** is a special member, taking just a single parameter that is a **const reference** to an object of the class itself.



# A simple String class



# SimpleString

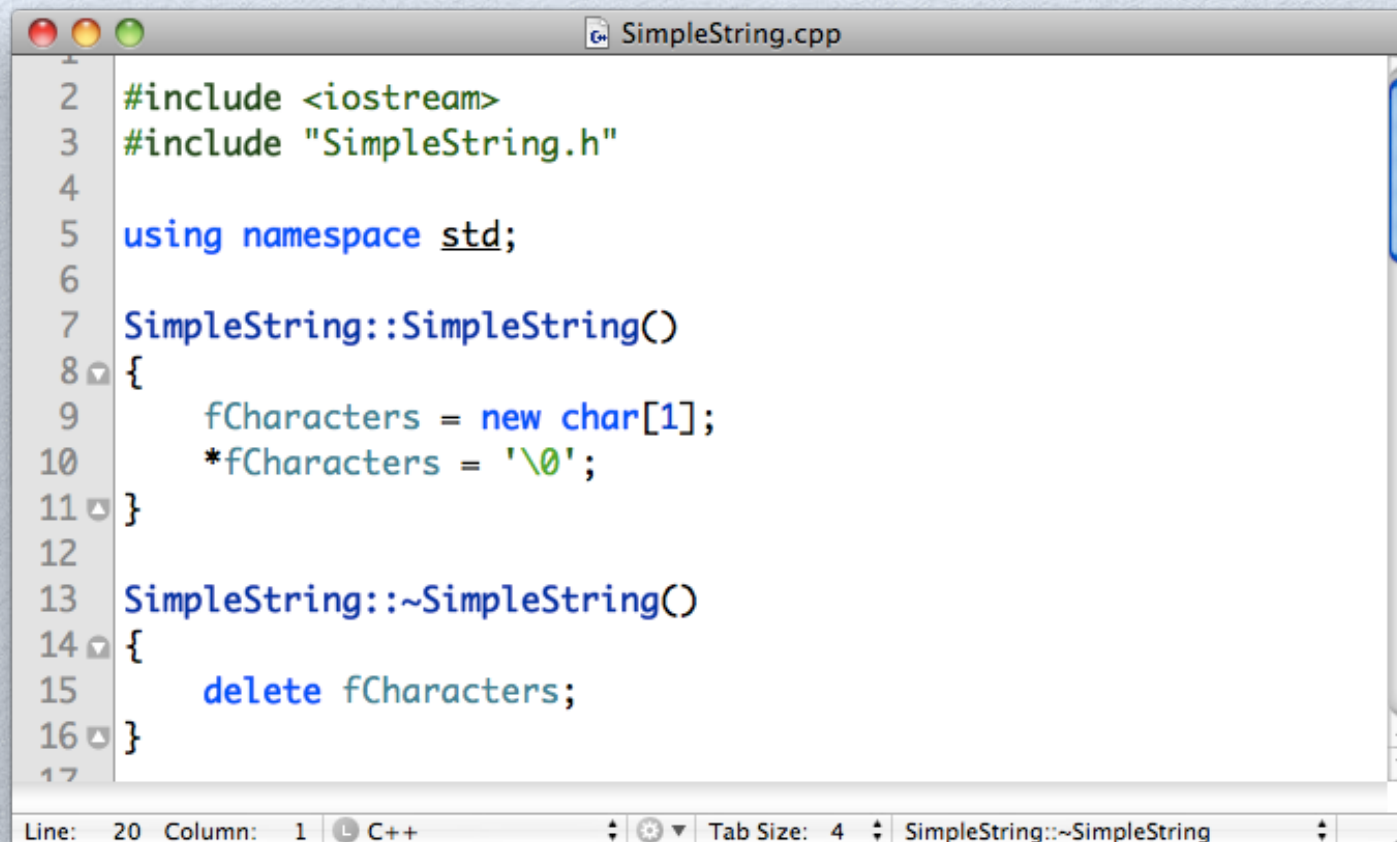


```
3
4 class SimpleString
5 {
6 private:
7     char * fCharacters;
8
9 public:
10     SimpleString();
11     ~SimpleString();
12
13     SimpleString& operator+( const char aCharacter );
14     const char* operator*() const;
15 };
16
```

Line: 1 Column: 17 C++ Tab Size: 4



# SimpleString: Constructor & Destructor

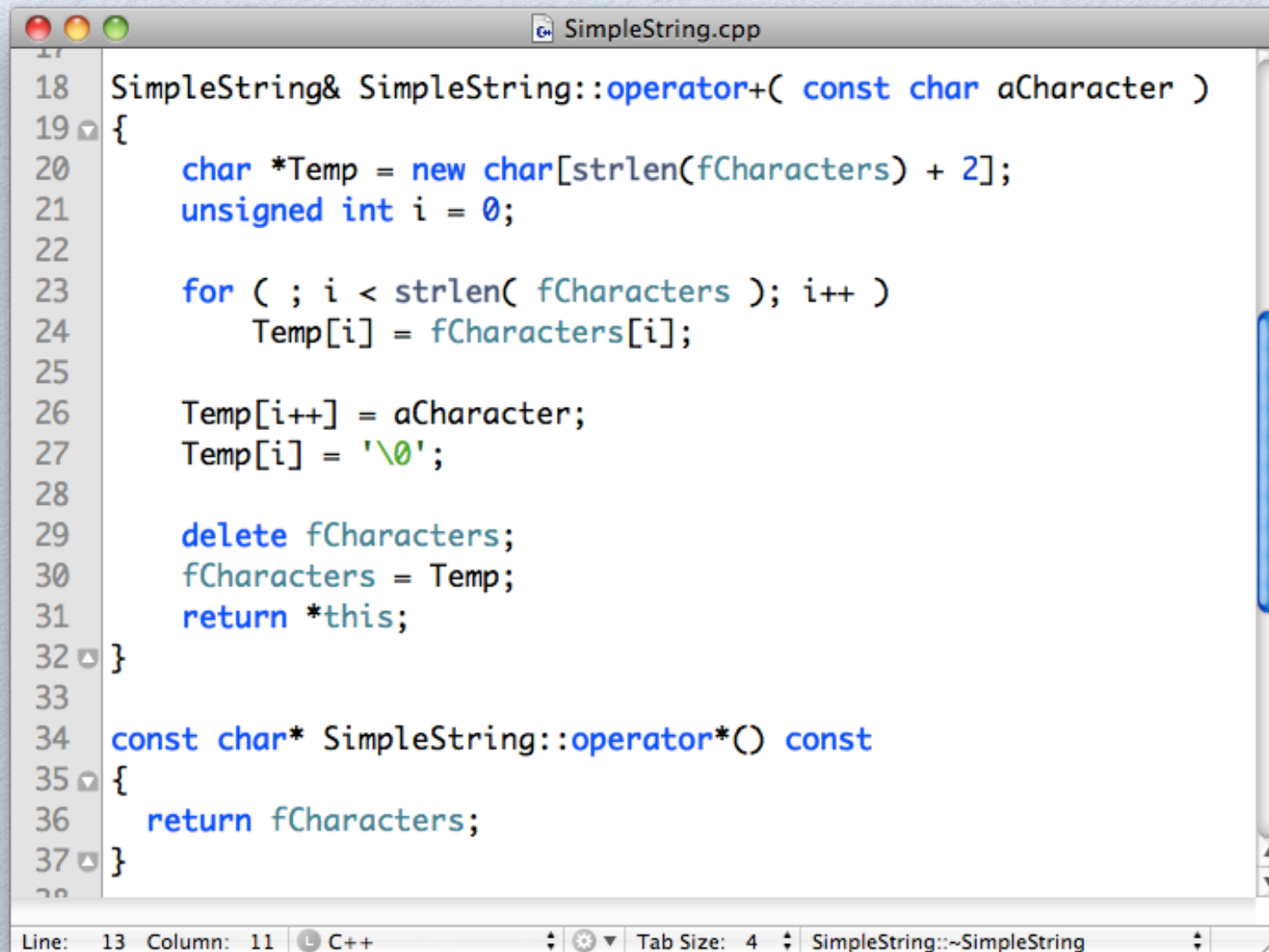


```
1
2 #include <iostream>
3 #include "SimpleString.h"
4
5 using namespace std;
6
7 SimpleString::SimpleString()
8 {
9     fCharacters = new char[1];
10    *fCharacters = '\0';
11 }
12
13 SimpleString::~SimpleString()
14 {
15     delete fCharacters;
16 }
17
```

Line: 20 Column: 1 C++ Tab Size: 4 SimpleString::~SimpleString



# SimpleString: The Operators



The screenshot shows a code editor window titled "SimpleString.cpp". The code defines two operators for the SimpleString class. The first operator, `operator+`, takes a `const char` and returns a `SimpleString&`. It creates a new character array, copies the existing characters, appends the new character, and updates the internal pointer. The second operator, `operator*`, takes no arguments and returns a `const char*` pointing to the internal character array. The editor interface includes a line number margin on the left, a scrollbar on the right, and a status bar at the bottom showing "Line: 13 Column: 11 C++" and "Tab Size: 4".

```
17
18 SimpleString& SimpleString::operator+( const char aCharacter )
19 {
20     char *Temp = new char[strlen(fCharacters) + 2];
21     unsigned int i = 0;
22
23     for ( ; i < strlen( fCharacters ); i++ )
24         Temp[i] = fCharacters[i];
25
26     Temp[i++] = aCharacter;
27     Temp[i] = '\0';
28
29     delete fCharacters;
30     fCharacters = Temp;
31     return *this;
32 }
33
34 const char* SimpleString::operator*() const
35 {
36     return fCharacters;
37 }
```

Line: 13 Column: 11 C++ Tab Size: 4 SimpleString::~SimpleString



# Implicit Copy Constructor

```
SimpleString.cpp
39
40 int main()
41 {
42     SimpleString s1;
43     s1 + 'A';
44     SimpleString s2 = s1;
45     s2 + 'B';
46
47     cout << "S1: " << *s1 << endl;
48     cout << "S2: " << *s2 << endl;
49
50     return 0;
51 }
52
```

Line: 13 Column: 11 C++ Tab Size: 4 SimpleString::~Sim...

```
COS30008
Kamala: COS30008 Markus$ ./SimpleString
S1: A
S2: AB
SimpleString(17203,0x7fff7167d300) malloc: *** error
for object 0x7fef2a404be0: pointer being freed was no
t allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
Kamala: COS30008 Markus$
```

On Windows you may not see anything.



# What Has Happened?

```
SimpleString.cpp
39
40 int main()
41 {
42     SimpleString s1;
43     s1 + 'A';
44     SimpleString s2 = s1;
45     s2 + 'B';
46
47     cout << "S1: " << *s1 << endl;
48     cout << "S2: " << *s2 << endl;
49
50     <delete s1;>
51     <delete s2;>
52
53     return 0;
54 }
```

Line: 44 Column: 26 C++ Tab Size: 4 main

## Shallow copy:

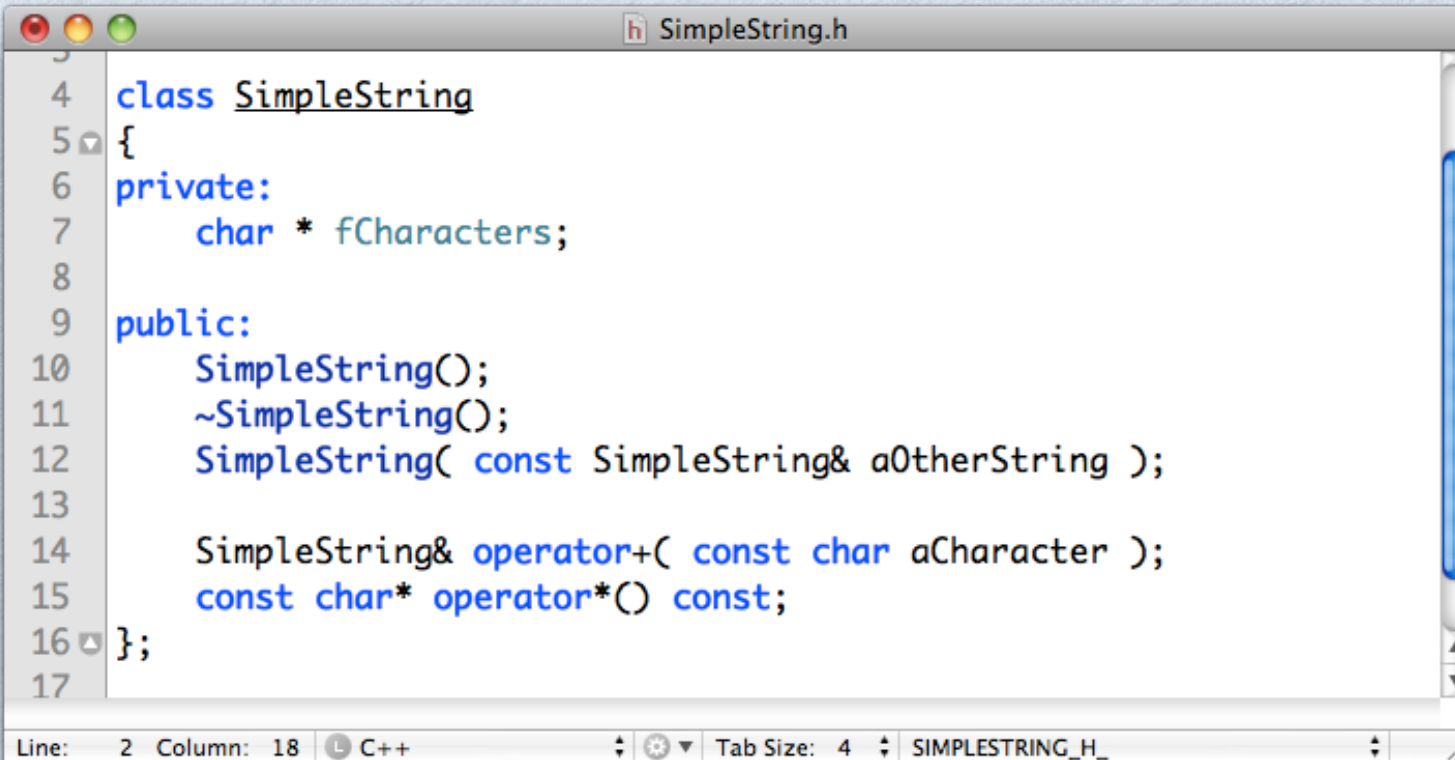
$s2.fCharacters = s1.fCharacters$

## Double free:

delete s2.fCharacters, which was called in s2 + 'B'.



# We need an explicit copy constructor!

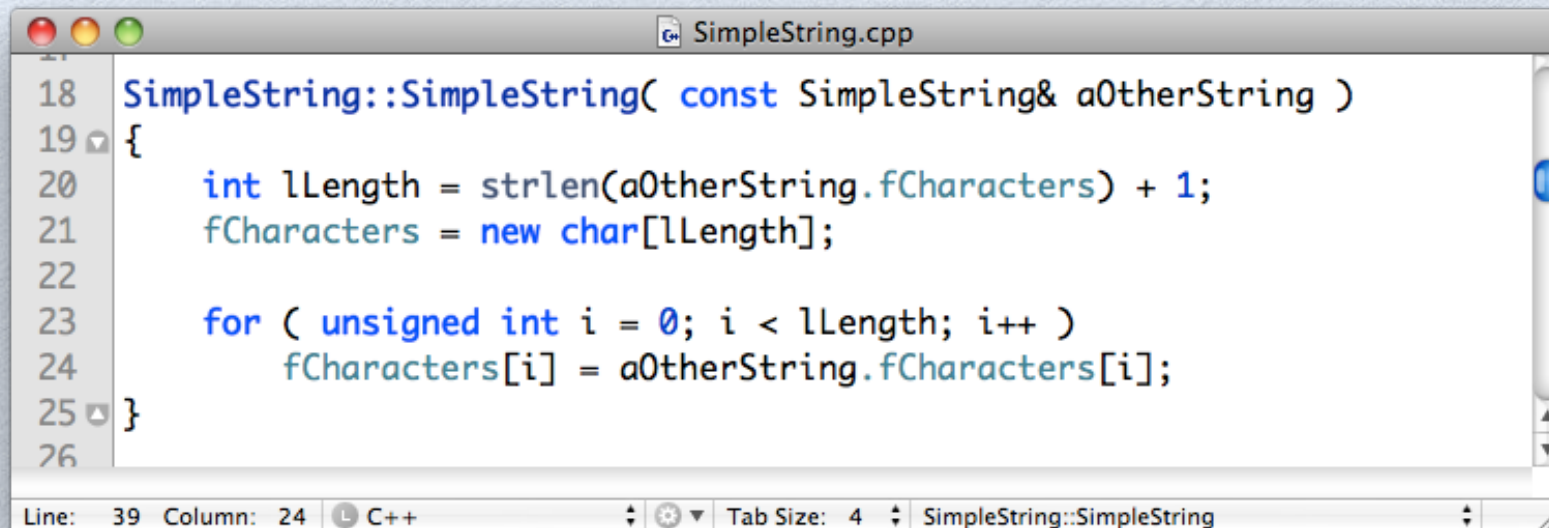


```
1
2
3
4 class SimpleString
5 {
6 private:
7     char * fCharacters;
8
9 public:
10     SimpleString();
11     ~SimpleString();
12     SimpleString( const SimpleString& aOtherString );
13
14     SimpleString& operator+( const char aCharacter );
15     const char* operator*() const;
16 };
17
```

Line: 2 Column: 18 C++ Tab Size: 4 SIMPLESTRING\_H\_



# The Explicit Copy Constructor



```
SimpleString.cpp
18 SimpleString::SimpleString( const SimpleString& aOtherString )
19 {
20     int lLength = strlen(aOtherString.fCharacters) + 1;
21     fCharacters = new char[lLength];
22
23     for ( unsigned int i = 0; i < lLength; i++ )
24         fCharacters[i] = aOtherString.fCharacters[i];
25 }
26
```

Line: 39 Column: 24 C++ Tab Size: 4 SimpleString::SimpleString


- When a copy constructor is called, then all instance variables are uninitialized in the beginning.



# Explicit Copy Constructor in Use

```
39
40 int main()
41 {
42     SimpleString s1;
43     s1 + 'A';
44     SimpleString s2 = s1;
45     s2 + 'B';
46
47     cout << "S1: " << *s1 << endl;
48     cout << "S2: " << *s2 << endl;
49
50     return 0;
51 }
52
```

S1: A  
S2: AB  
Kamala: COS3000



A terminal window titled 'COS30008' with standard macOS window controls (red, yellow, green buttons). The prompt is 'Kamala: COS30008 Markus\$'. The command './SimpleString' has been executed, resulting in two lines of output: 'S1: A' and 'S2: AB'. The prompt is now 'Kamala: COS30008 Markus\$' followed by a cursor.

```
Kamala: COS30008 Markus$ ./SimpleString
S1: A
S2: AB
Kamala: COS30008 Markus$ _
```



# What Has Happened?

```
SimpleString.cpp
39
40 int main()
41 {
42     SimpleString s1;
43     s1 + 'A';
44     SimpleString s2 = s1;
45     s2 + 'B';
46
47     cout << "S1: " << *s1 << endl;
48     cout << "S2: " << *s2 << endl;
49
50     <delete s1;>
51     <delete s2;>
52
53     return 0;
54 }
```

Line: 44 Column: 26 C++ Tab Size: 4 main

**Deep copy:**  
`s2.fCharacters != s1.fCharacters`

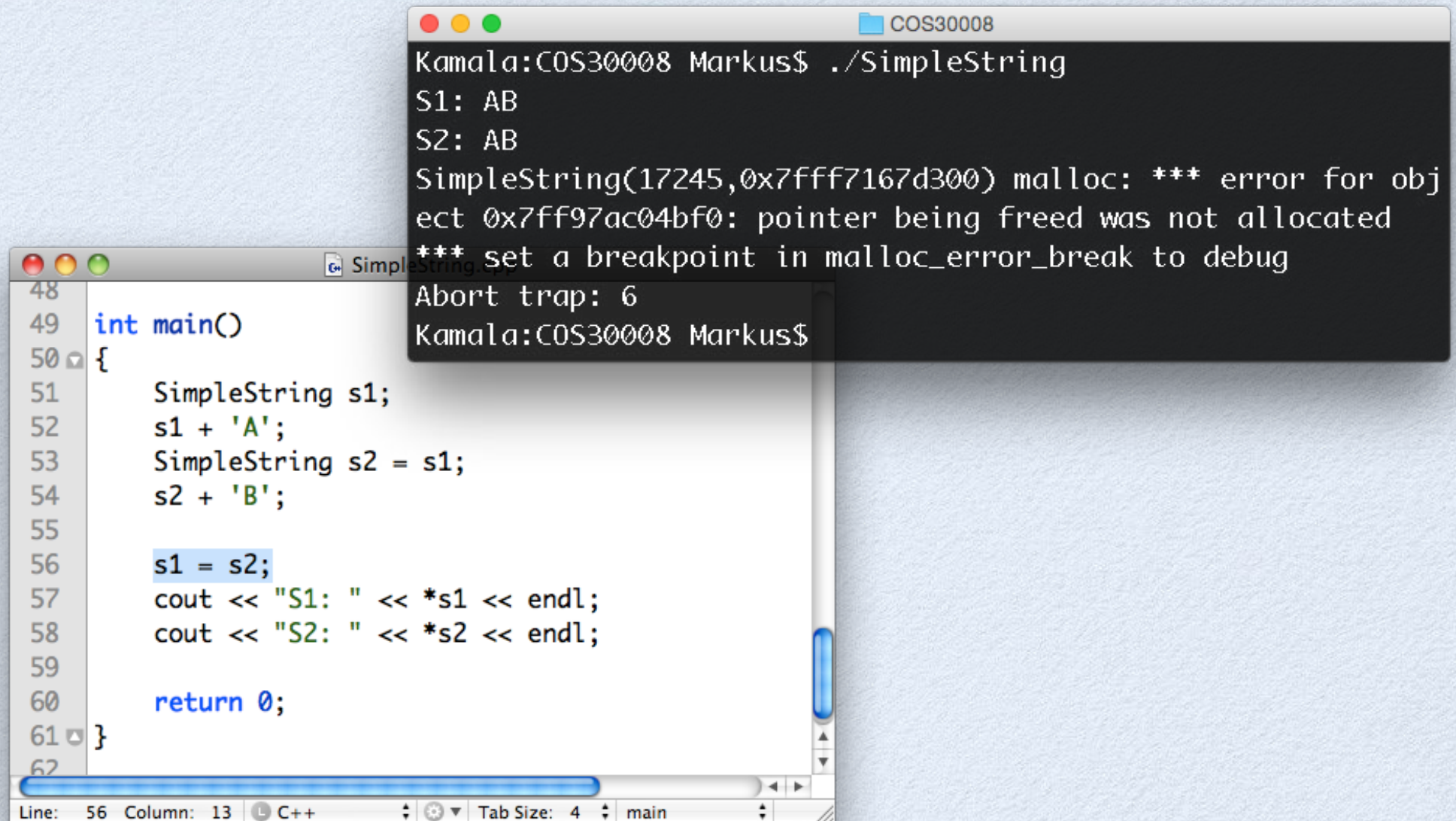




**That's it. No more problems, or?**



# A Simple Assignment



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

```
48  
49 int main()  
50 {  
51     SimpleString s1;  
52     s1 + 'A';  
53     SimpleString s2 = s1;  
54     s2 + 'B';  
55  
56     s1 = s2;  
57     cout << "S1: " << *s1 << endl;  
58     cout << "S2: " << *s2 << endl;  
59  
60     return 0;  
61 }  
62
```

The IDE status bar at the bottom shows 'Line: 56 Column: 13 C++ Tab Size: 4 main'.

Overlaid on the IDE is a terminal window titled 'COS30008' showing the execution of the program:

```
Kamala: COS30008 Markus$ ./SimpleString  
S1: AB  
S2: AB  
SimpleString(17245, 0x7fff7167d300) malloc: *** error for object 0x7fff97ac04bf0: pointer being freed was not allocated  
*** set a breakpoint in malloc_error_break to debug  
Abort trap: 6  
Kamala: COS30008 Markus$
```



# What Has Happened?

```
SimpleString.cpp
48
49 int main()
50 {
51     SimpleString s1;
52     s1 + 'A';
53     SimpleString s2 = s1;
54     s2 + 'B';
55
56     s1 = s2;
57     cout << "S1: " << *s1 << endl;
58     cout << "S2: " << *s2 << endl;
59
60     <delete s1;>
61     <delete s2;>
62
63     return 0;
64 }
```

Line: 56 Column: 13 C++ Tab Size: 4 main

**Shallow copy:**

$s2.fCharacters = s1.fCharacters$

**Double free:**

delete s2.fCharacters, which is the same as s1.fCharacters.

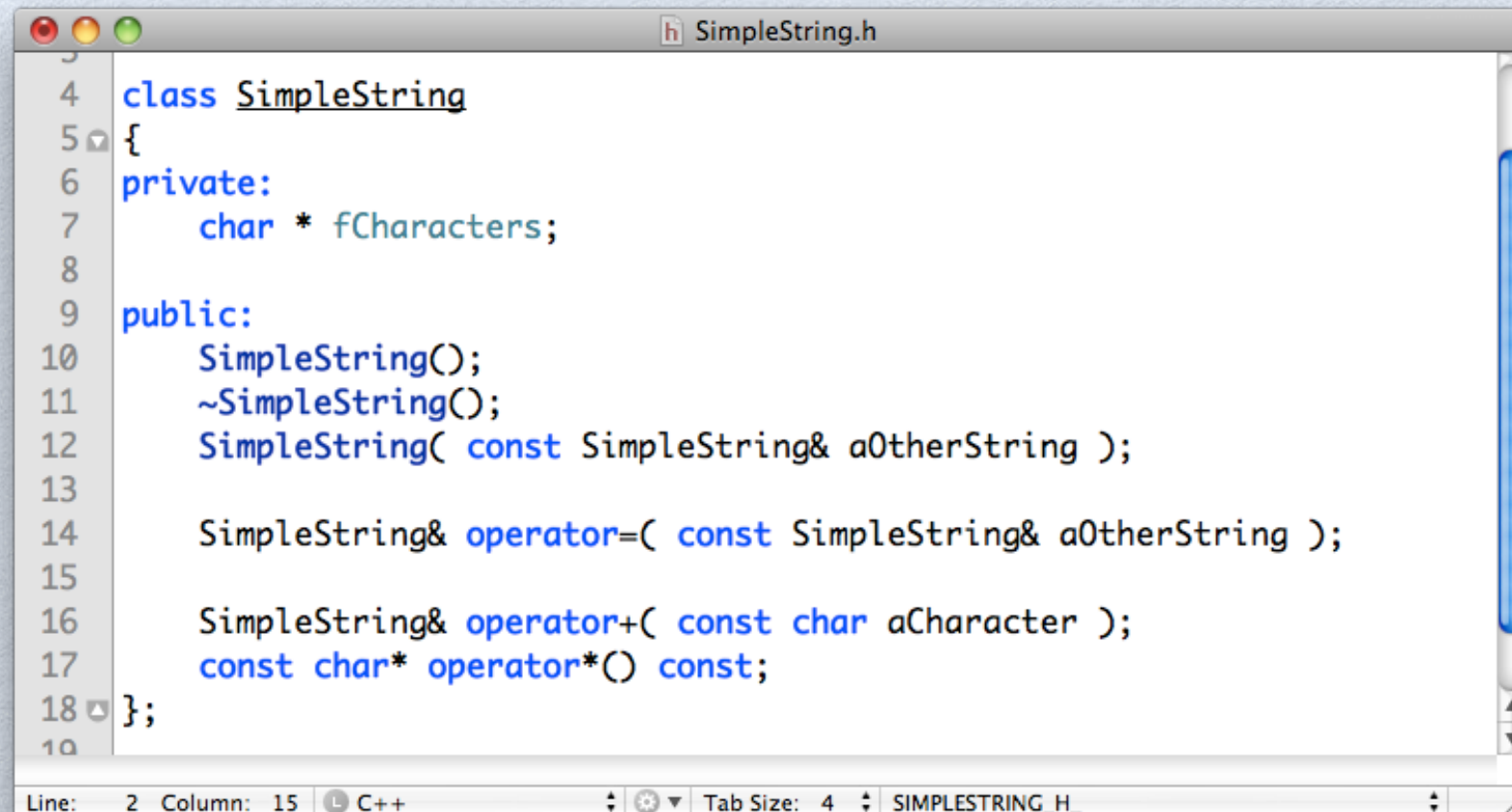


# Rule Of Thumb

- Copy control in C++ requires three elements:
  - a copy constructor
  - an assignment operator
  - a destructor
- Whenever one defines a copy constructor, one must also define an assignment operator and a destructor.
- C++ also supports move constructor and move assignment operator. They work similarly, but steal the memory for its r-value source. Moreover, while the compiler still synthesizes missing l-value copy constructors and assignment operators, their r-value counterparts are not synthesized if the programmer does specify either but not both.



# We need an explicit assignment operator!



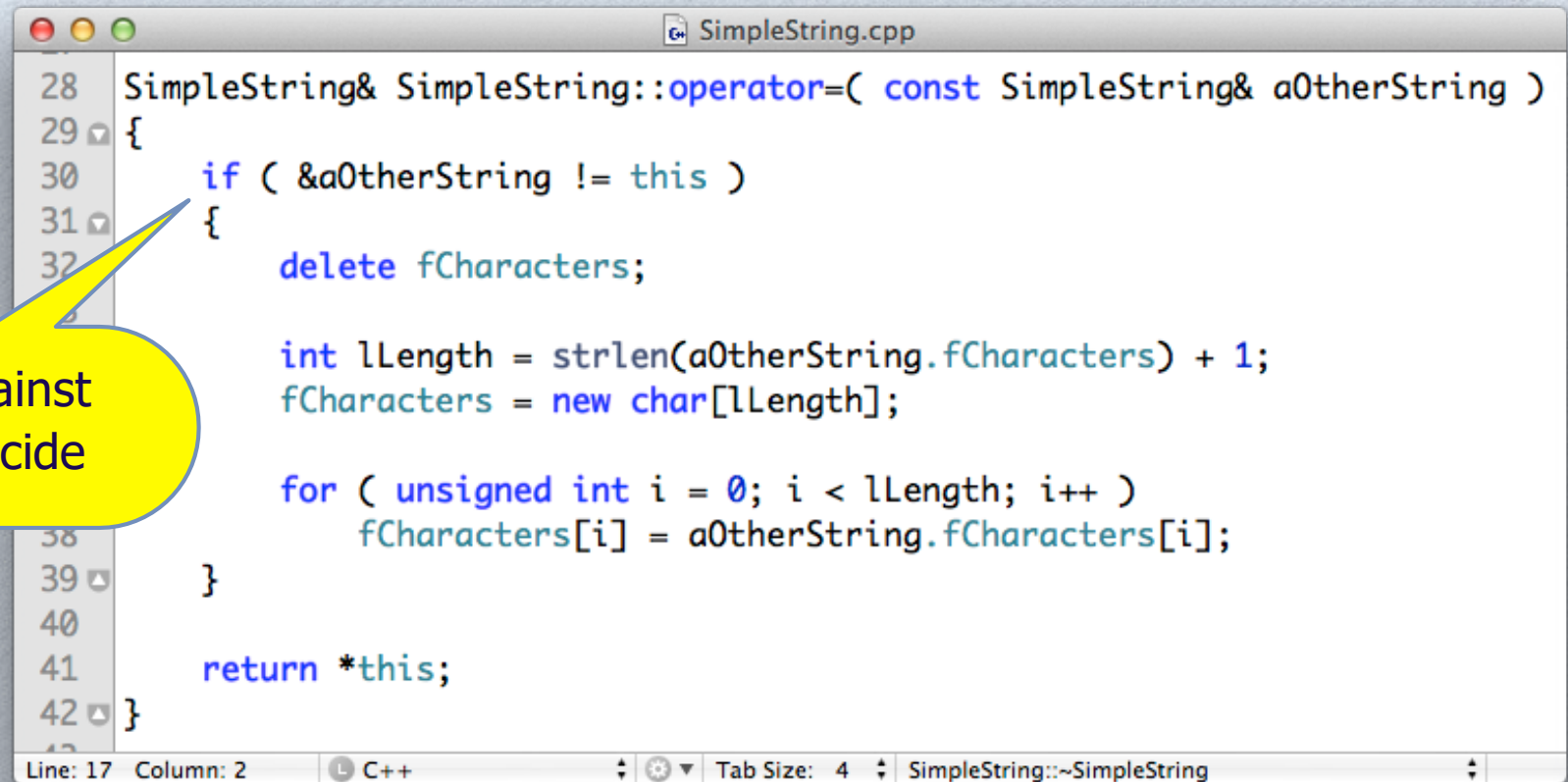
```
SimpleString.h
1
2
3
4 class SimpleString
5 {
6 private:
7     char * fCharacters;
8
9 public:
10     SimpleString();
11     ~SimpleString();
12     SimpleString( const SimpleString& aOtherString );
13
14     SimpleString& operator=( const SimpleString& aOtherString );
15
16     SimpleString& operator+=( const char aCharacter );
17     const char* operator*() const;
18 };
19
```

Line: 2 Column: 15 C++ Tab Size: 4 SIMPLESTRING\_H\_



# The Explicit Assignment Operator

protection against  
accidental suicide



```
SimpleString.cpp
28 SimpleString& SimpleString::operator=( const SimpleString& a0therString )
29 {
30     if ( &a0therString != this )
31     {
32         delete fCharacters;

        int lLength = strlen(a0therString.fCharacters) + 1;
        fCharacters = new char[lLength];

        for ( unsigned int i = 0; i < lLength; i++ )
            fCharacters[i] = a0therString.fCharacters[i];
    }

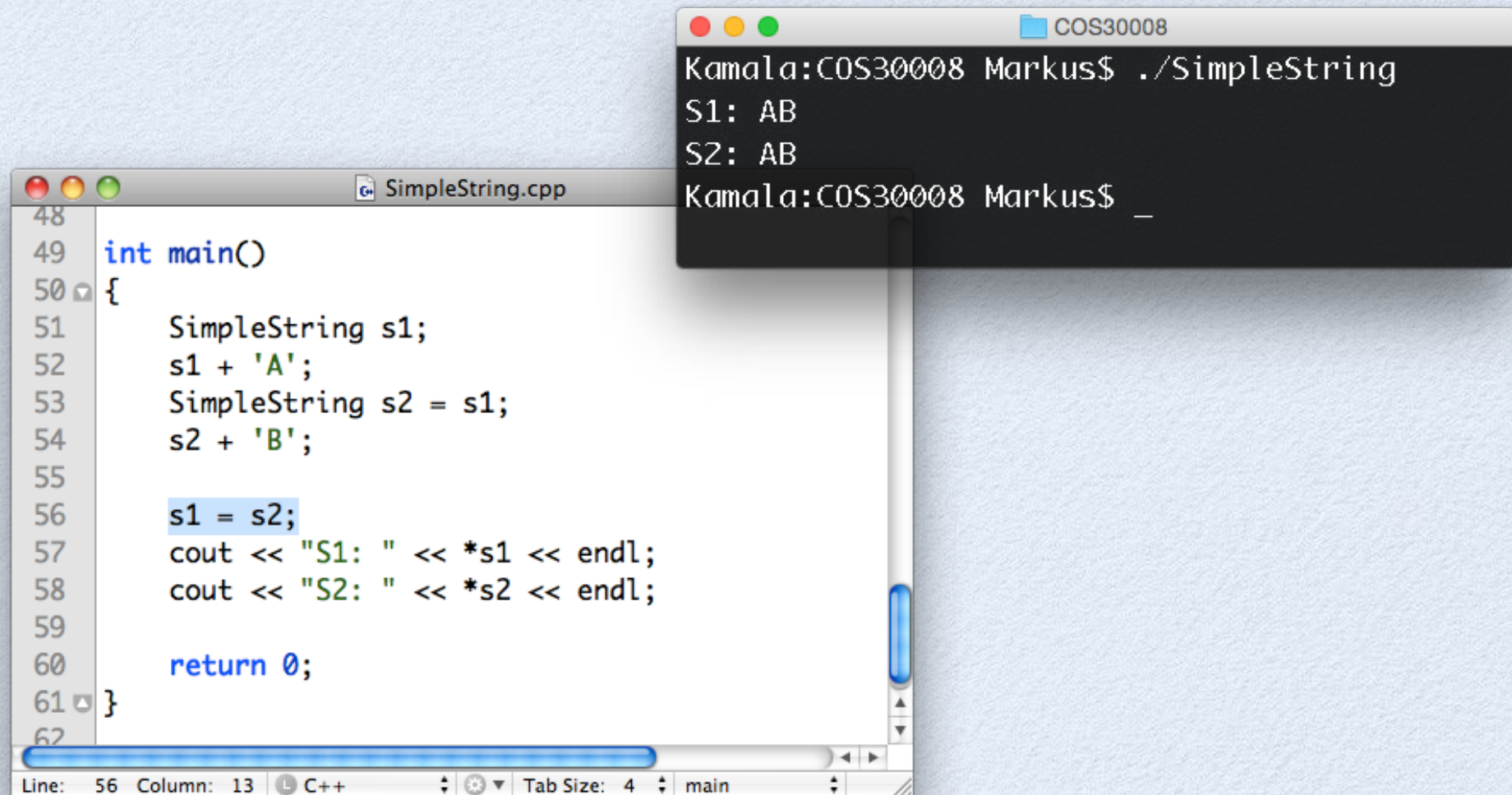
    return *this;
}

Line: 17 Column: 2 C++ Tab Size: 4 SimpleString::~SimpleString
```

- When the assignment operator is invoked, then all instance variables are initialized in the beginning. We need to release the memory first!



# Explicit Assignment Operator in Use



The image shows a C++ IDE window titled 'SimpleString.cpp' and a terminal window titled 'COS30008'. The IDE window displays the following code:

```
48
49 int main()
50 {
51     SimpleString s1;
52     s1 + 'A';
53     SimpleString s2 = s1;
54     s2 + 'B';
55
56     s1 = s2;
57     cout << "S1: " << *s1 << endl;
58     cout << "S2: " << *s2 << endl;
59
60     return 0;
61 }
62
```

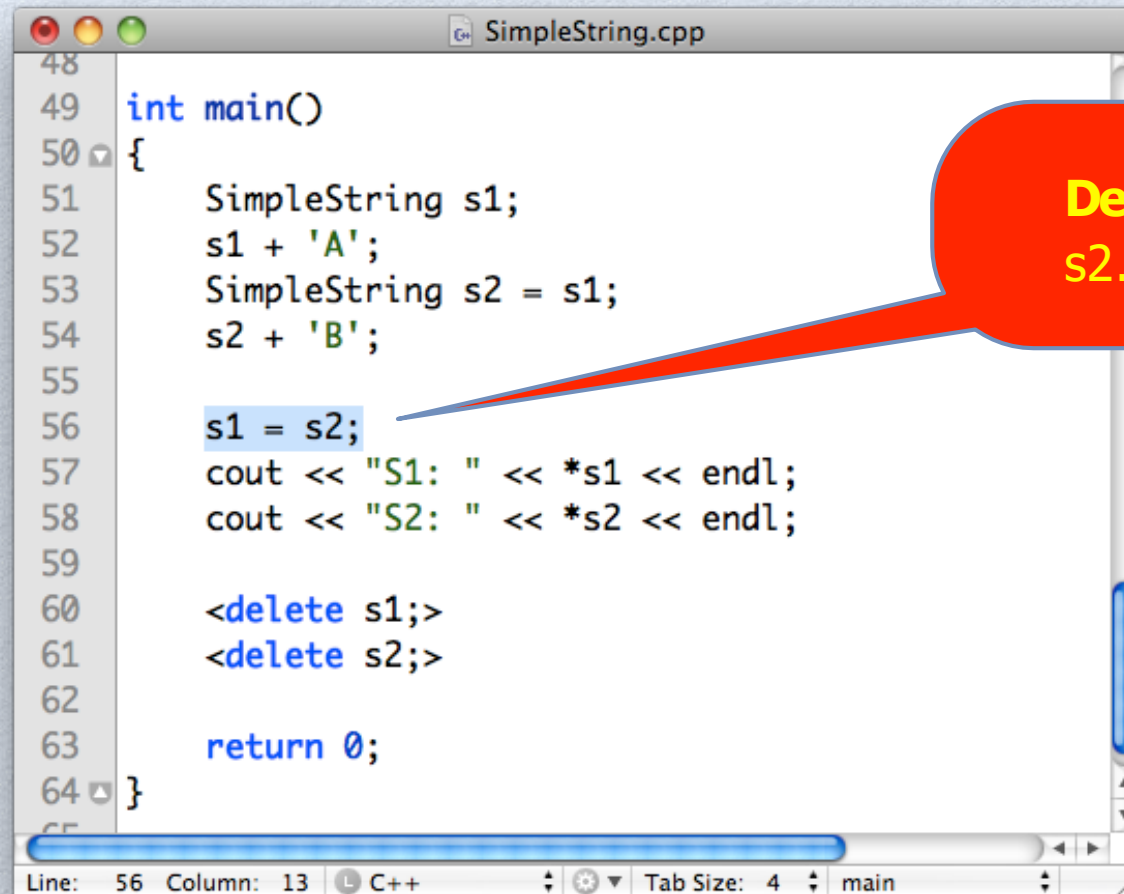
The terminal window shows the output of the program:

```
Kamala: COS30008 Markus$ ./SimpleString
S1: AB
S2: AB
Kamala: COS30008 Markus$ _
```

The IDE status bar at the bottom indicates 'Line: 56 Column: 13 C++ Tab Size: 4 main'.



# What Has Happened?



```
48
49 int main()
50 {
51     SimpleString s1;
52     s1 + 'A';
53     SimpleString s2 = s1;
54     s2 + 'B';
55
56     s1 = s2;
57     cout << "S1: " << *s1 << endl;
58     cout << "S2: " << *s2 << endl;
59
60     <delete s1;>
61     <delete s2;>
62
63     return 0;
64 }
```

Line: 56 Column: 13 C++ Tab Size: 4 main

**Deep copy:**  
`s2.fCharacters != s1.fCharacters`



# Cloning: Alias Control for References



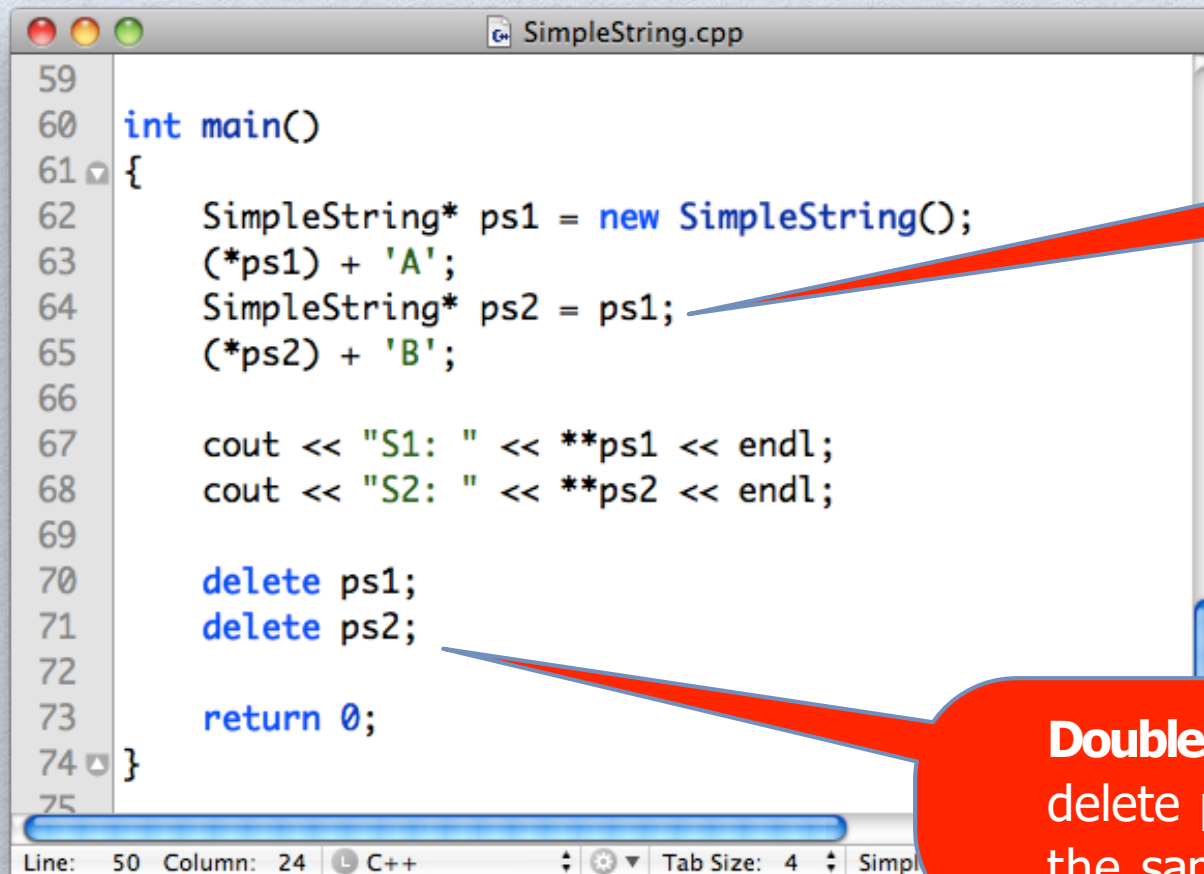
# Copying Pointers

```
SimpleString.cpp
59
60 int main()
61 {
62     SimpleString* ps1 = new SimpleString();
63     (*ps1) + 'A';
64     SimpleString* ps2 = ps1;
65     (*ps2) + 'B';
66
67     cout << "S1: " << **ps1 << endl;
68     cout << "S2: " << **ps2 << endl;
69
70     delete ps1;
71     delete ps2;
72
73     return 0;
74 }
75
Line: 50 Column: 24 C++
```

```
COS30008
Kamala: COS30008 Markus$ ./SimpleString
S1: AB
S2: AB
SimpleString(17284,0x7fff7167d300) malloc: *** error for object
0x7fc7cac04be0: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
Kamala: COS30008 Markus$ _
```



# What Has Happened?



```
59
60 int main()
61 {
62     SimpleString* ps1 = new SimpleString();
63     (*ps1) + 'A';
64     SimpleString* ps2 = ps1;
65     (*ps2) + 'B';
66
67     cout << "S1: " << **ps1 << endl;
68     cout << "S2: " << **ps2 << endl;
69
70     delete ps1;
71     delete ps2;
72
73     return 0;
74 }
75
```

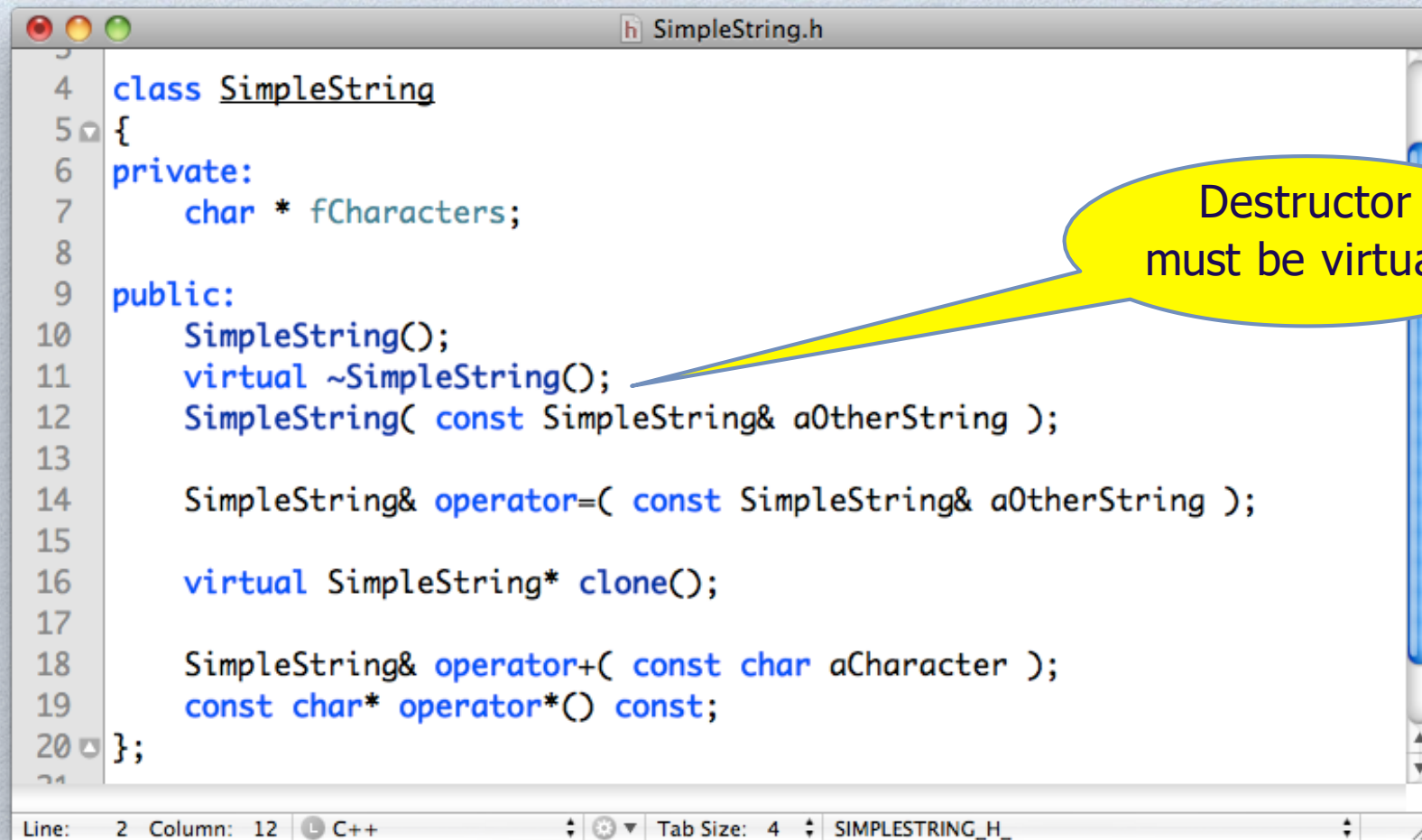
Line: 50 Column: 24 C++ Tab Size: 4

**Shallow copy:**  
 $ps2. = ps1$

**Double free:**  
delete ps2, which is  
the same as ps1.



# Solution: A clone() Method



```
SimpleString.h
1
2
3
4 class SimpleString
5 {
6 private:
7     char * fCharacters;
8
9 public:
10    SimpleString();
11    virtual ~SimpleString();
12    SimpleString( const SimpleString& aOtherString );
13
14    SimpleString& operator=( const SimpleString& aOtherString );
15
16    virtual SimpleString* clone();
17
18    SimpleString& operator+( const char aCharacter );
19    const char* operator*() const;
20 };
```

Destructor must be virtual!

Line: 2 Column: 12 C++ Tab Size: 4 SIMPLESTRING\_H\_

- It is best to define the destructor of a class virtual always in order to avoid problems later.



# The Use of clone()

```
SimpleString.cpp
37
38 SimpleString* SimpleString::clone()
39 {
40     return new SimpleString( *this );
41 }
42
```

Line: 69 Column: 30 C++ Tab Size: 4 m...

```
COS30008
Kamala: COS30008 Markus$ ./SimpleString
S1: A
S2: AB
Kamala: COS30008 Markus$
```

```
SimpleString.cpp
64
65 int main()
66 {
67     SimpleString* ps1 = new SimpleString();
68     (*ps1) + 'A';
69     SimpleString* ps2 = ps1->clone();
70     (*ps2) + 'B';
71
72     cout << "S1: " << **ps1 << endl;
73     cout << "S2: " << **ps2 << endl;
74
75     delete ps1;
76     delete ps2;
77
78     return 0;
79 }
80
```

Line: 63 Column: 1 C++ Tab Size: 4 SimpleString::...



# Problems With Cloning

- The member function `clone()` must be defined `virtual` to allow for proper redefinition in subtypes.
- Whenever a class contains a virtual function, then its `destructor` is required to be defined `virtual` as well.
- The member function `clone()` can only return one `type`. When a subtype redefines `clone()`, only the super type can be returned.



# Non-virtual Cloning Does Not Work!

- One could define clone() non-virtual and use overloading. But this does not work as method selection starts at the **static type of the pointer**.

```
SimpleString* pToString = new SubtypeOfSimpleString();
```

```
SimpleString* c1 = pToString->clone(); // SimpleString::clone()
```



# **Reference-based Semantics: When Do We Destroy Objects?**

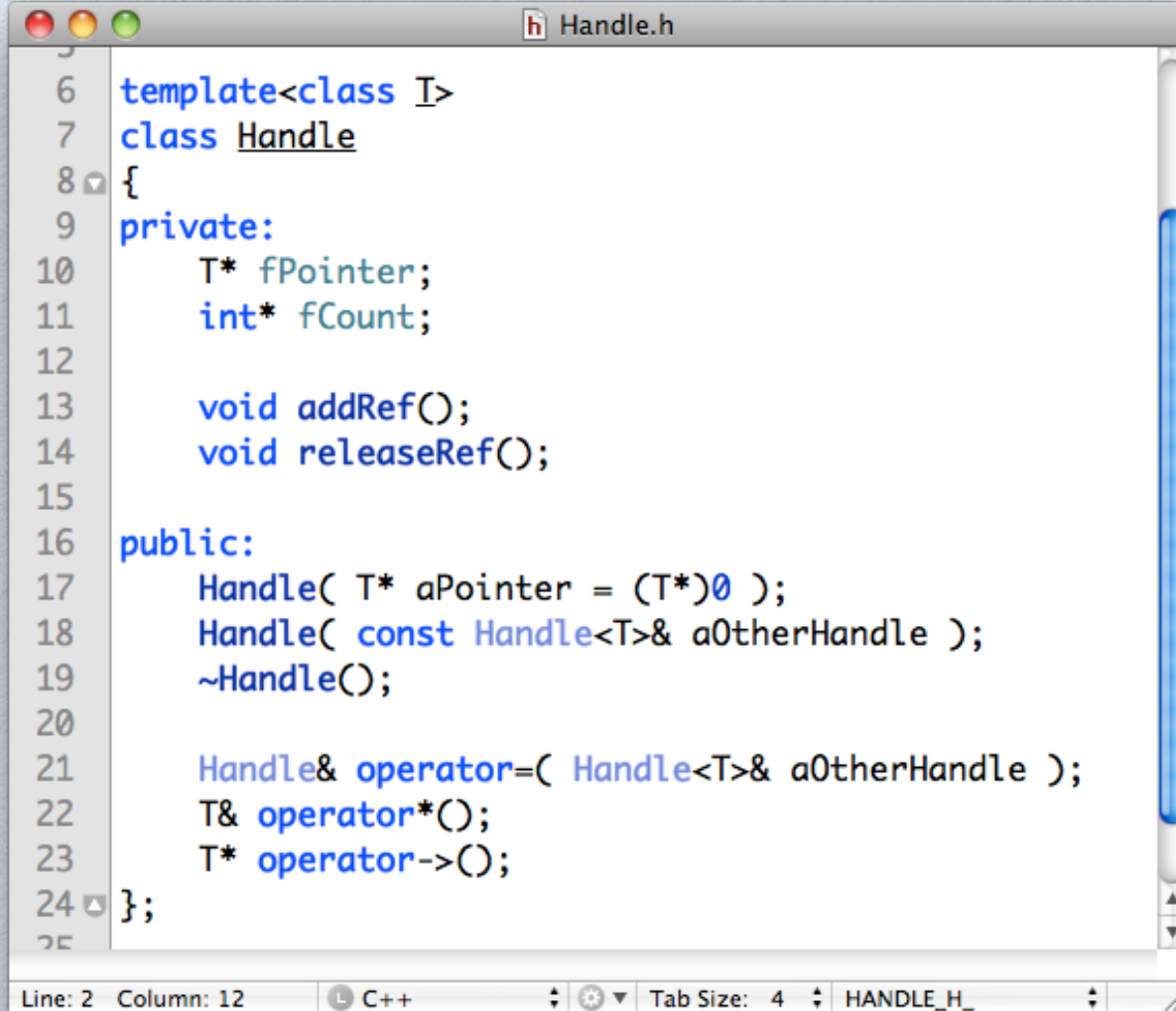


# Reference Counting

- A simple technique to record the number of active uses of an object is **reference counting**.
- Each time a heap-based object is assigned to a variable the object's reference count is incremented and the reference count of what the variable previously pointed to is decremented.
- Some compilers emit the necessary code, but in case of C++ reference counting must be defined (semi-)manually.



# Smart Pointers: Handle



```
1
2
3
4
5
6 template<class T>
7 class Handle
8 {
9 private:
10     T* fPointer;
11     int* fCount;
12
13     void addRef();
14     void releaseRef();
15
16 public:
17     Handle( T* aPointer = (T*)0 );
18     Handle( const Handle<T>& aOtherHandle );
19     ~Handle();
20
21     Handle& operator=( Handle<T>& aOtherHandle );
22     T& operator*();
23     T* operator->();
24 };
25
```

Line: 2 Column: 12 C++ Tab Size: 4 HANDLE\_H\_

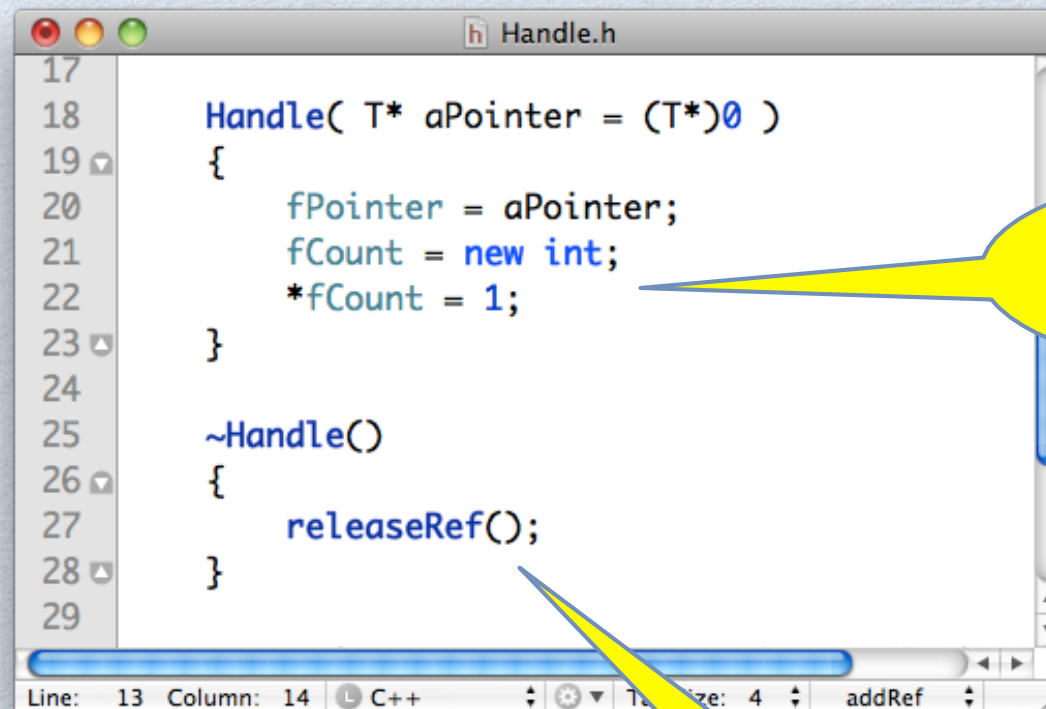


# The Use of Handle

- The template class Handle provides a **pointer-like behavior**:
  - Copying a Handle will create a **shared alias** of the underlying object.
  - To create a Handle, the user will be expected to pass a **fresh, dynamically allocated object** of the type managed by the Handle.
  - The Handle will own the underlying object. In particular, the Handle **assumes responsibility for deleting** the owned object once there are no longer any Handles attached to it.



# Handle: Constructor & Destructor



```
17
18  Handle( T* aPointer = (T*)0 )
19  {
20      fPointer = aPointer;
21      fCount = new int;
22      *fCount = 1;
23  }
24
25  ~Handle()
26  {
27      releaseRef();
28  }
29
```

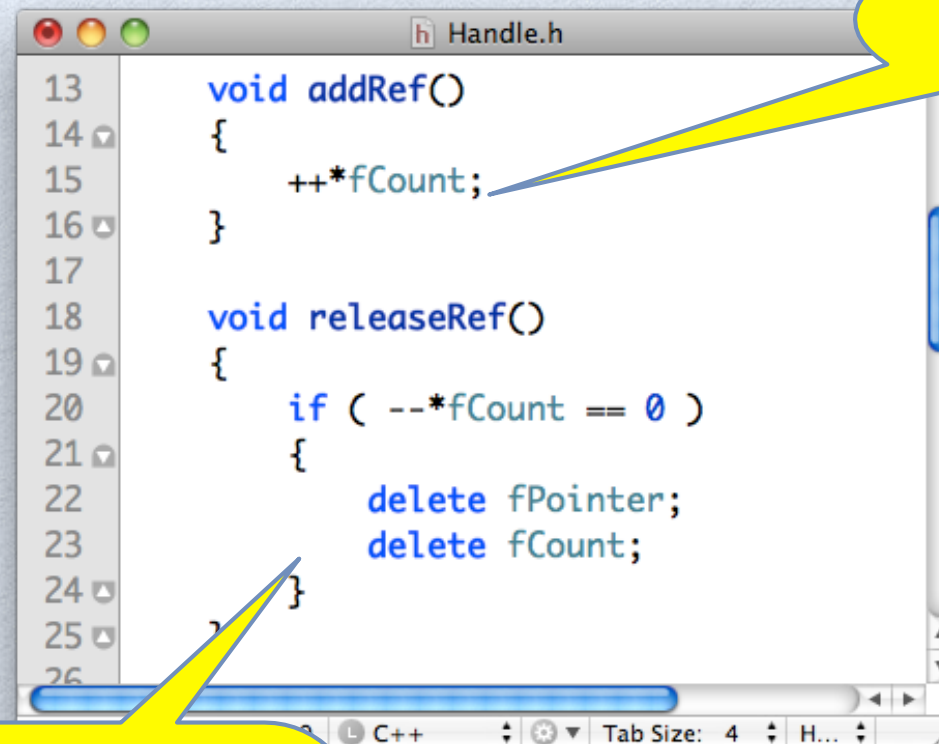
Line: 13 Column: 14 C++

Create a shared counter

Decrement reference count



# Handle: addRef & releaseRef



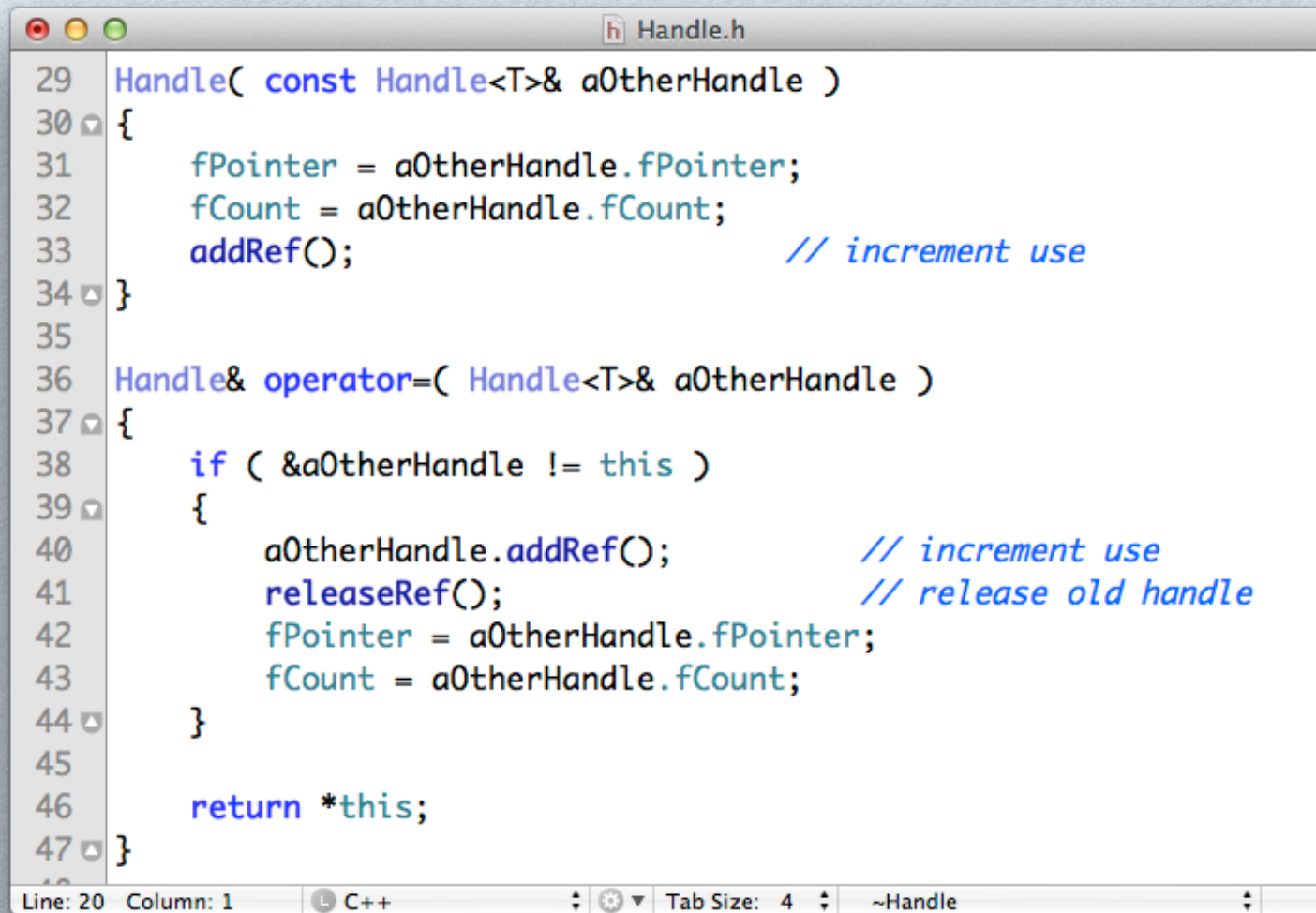
```
13 void addRef()
14 {
15     ++*fCount;
16 }
17
18 void releaseRef()
19 {
20     if ( --*fCount == 0 )
21     {
22         delete fPointer;
23         delete fCount;
24     }
25 }
26
```

Increment  
reference count

Decrement reference count  
and delete object if it is no  
longer referenced anywhere.



# Handle: Copy Control

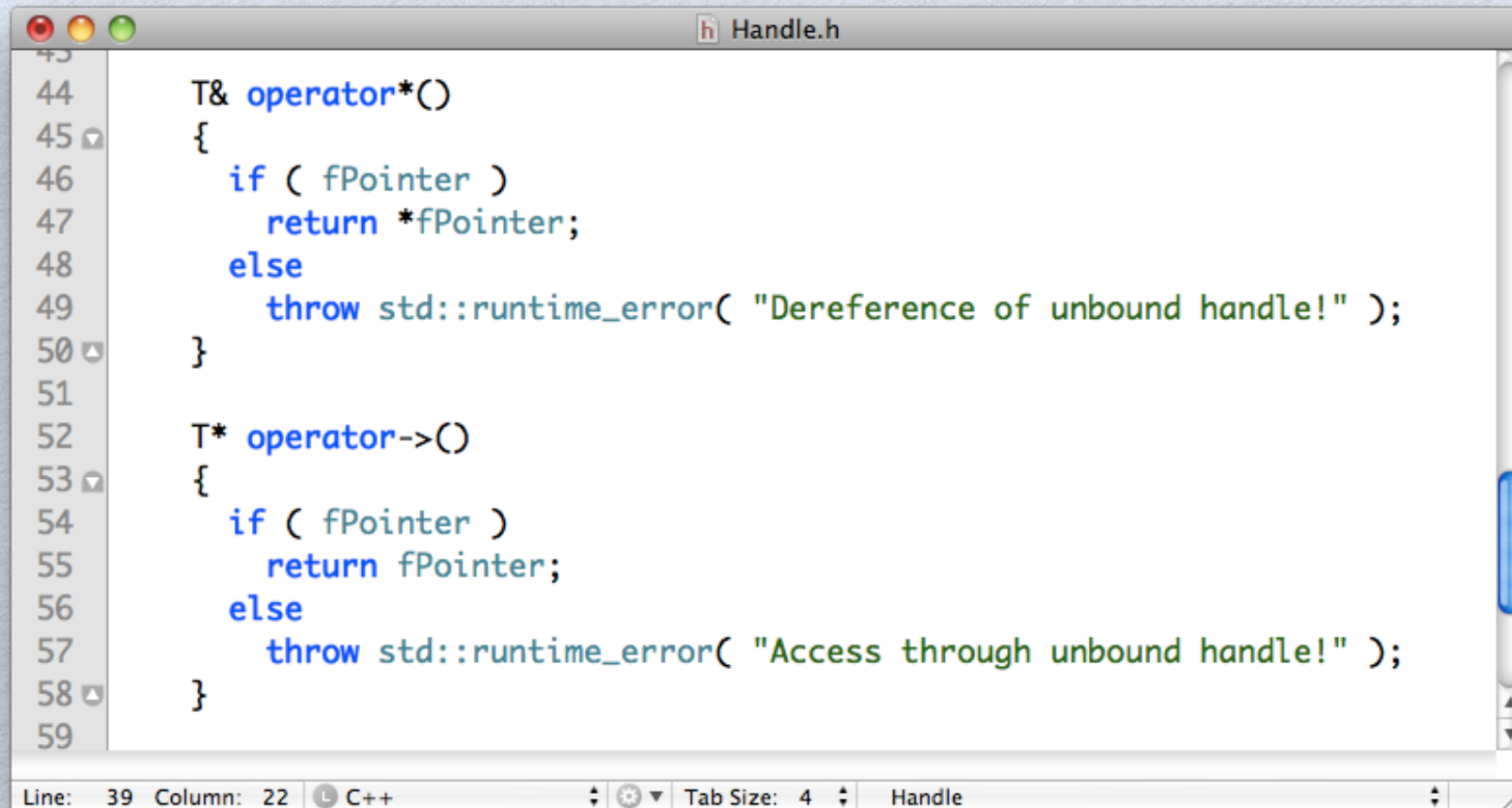


```
29 Handle( const Handle<T>& aOtherHandle )
30 {
31     fPointer = aOtherHandle.fPointer;
32     fCount = aOtherHandle.fCount;
33     addRef();                // increment use
34 }
35
36 Handle& operator=( Handle<T>& aOtherHandle )
37 {
38     if ( &aOtherHandle != this )
39     {
40         aOtherHandle.addRef();    // increment use
41         releaseRef();            // release old handle
42         fPointer = aOtherHandle.fPointer;
43         fCount = aOtherHandle.fCount;
44     }
45
46     return *this;
47 }
```

Line: 20 Column: 1 C++ Tab Size: 4 ~Handle



# Handle: Pointer Behavior

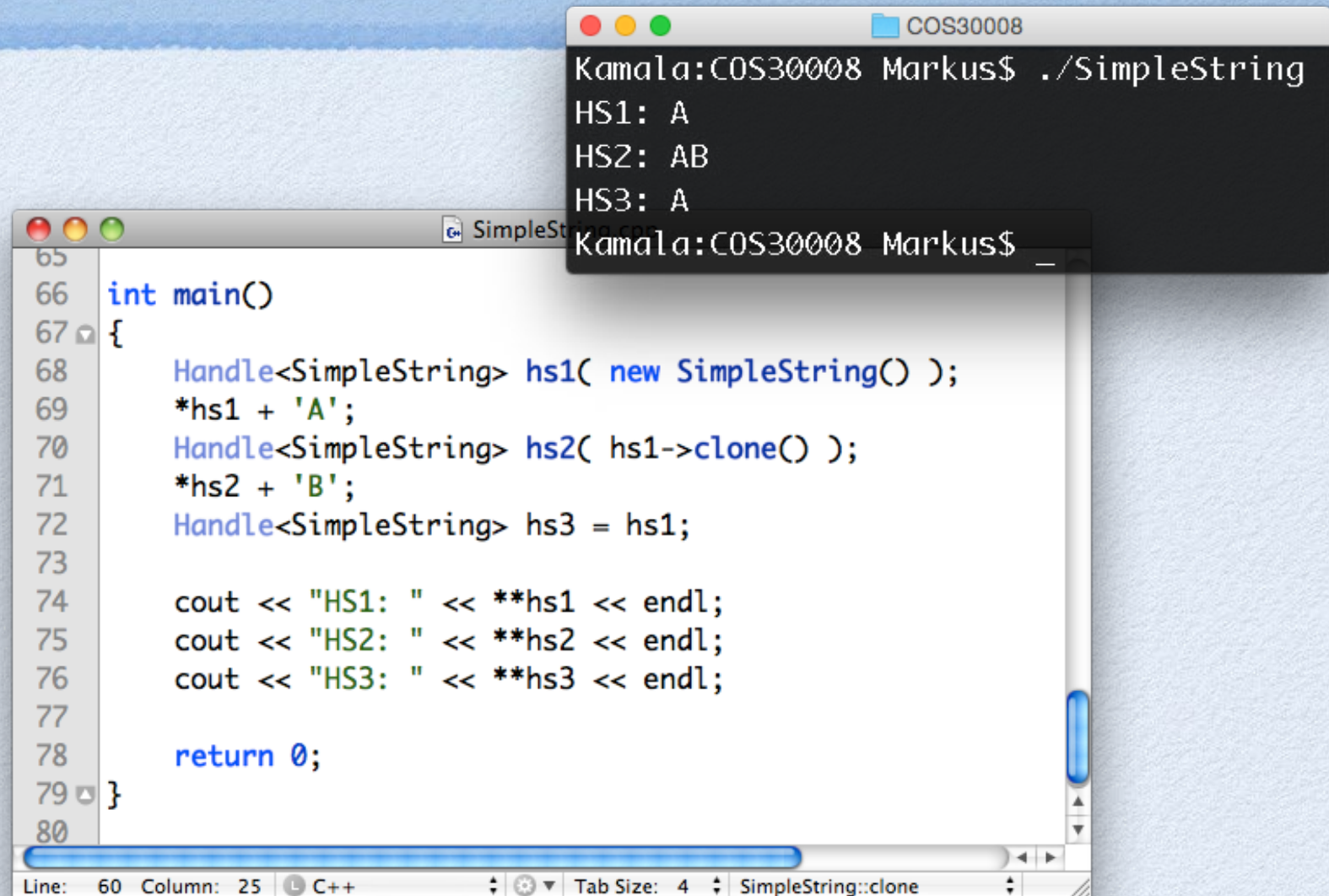


```
43
44     T& operator*()
45     {
46         if ( fPointer )
47             return *fPointer;
48         else
49             throw std::runtime_error( "Dereference of unbound handle!" );
50     }
51
52     T* operator->()
53     {
54         if ( fPointer )
55             return fPointer;
56         else
57             throw std::runtime_error( "Access through unbound handle!" );
58     }
59
```

Line: 39 Column: 22 C++ Tab Size: 4 Handle



# Using Handle

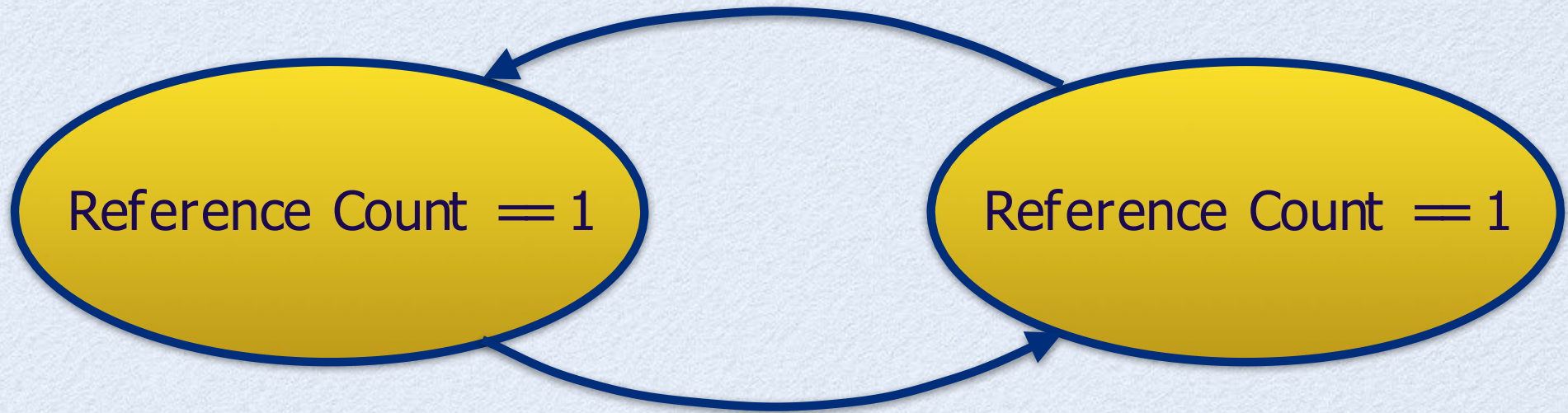


```
65
66 int main()
67 {
68     Handle<SimpleString> hs1( new SimpleString() );
69     *hs1 + 'A';
70     Handle<SimpleString> hs2( hs1->clone() );
71     *hs2 + 'B';
72     Handle<SimpleString> hs3 = hs1;
73
74     cout << "HS1: " << **hs1 << endl;
75     cout << "HS2: " << **hs2 << endl;
76     cout << "HS3: " << **hs3 << endl;
77
78     return 0;
79 }
80
```

Kamala: COS30008 Markus\$ ./SimpleString  
HS1: A  
HS2: AB  
HS3: A  
Kamala: COS30008 Markus\$



# Reference Counting Limits



- Reference counting **fails** on **circular** data structures like double-linked lists.
- Circular data structures require extra effort to reclaim allocated memory. **Know solution: Mark-and-Sweep**