SWIN
BUR
NE

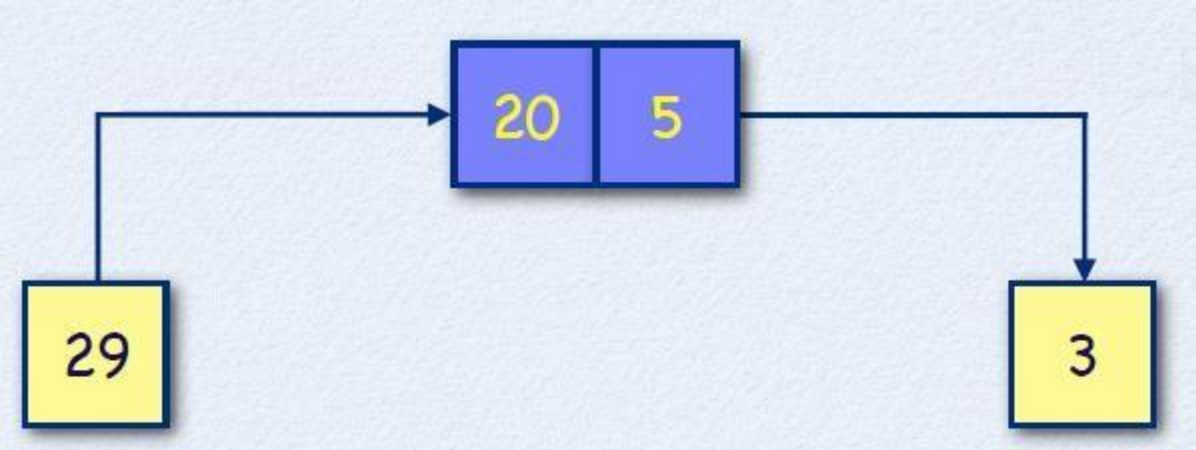
SWINBURNE
UNIVERSITY OF
TECHNOLOGY

# Queue,
# Copy Control and
# Memory Management

FACTUM PER LITTERAS

# Queue

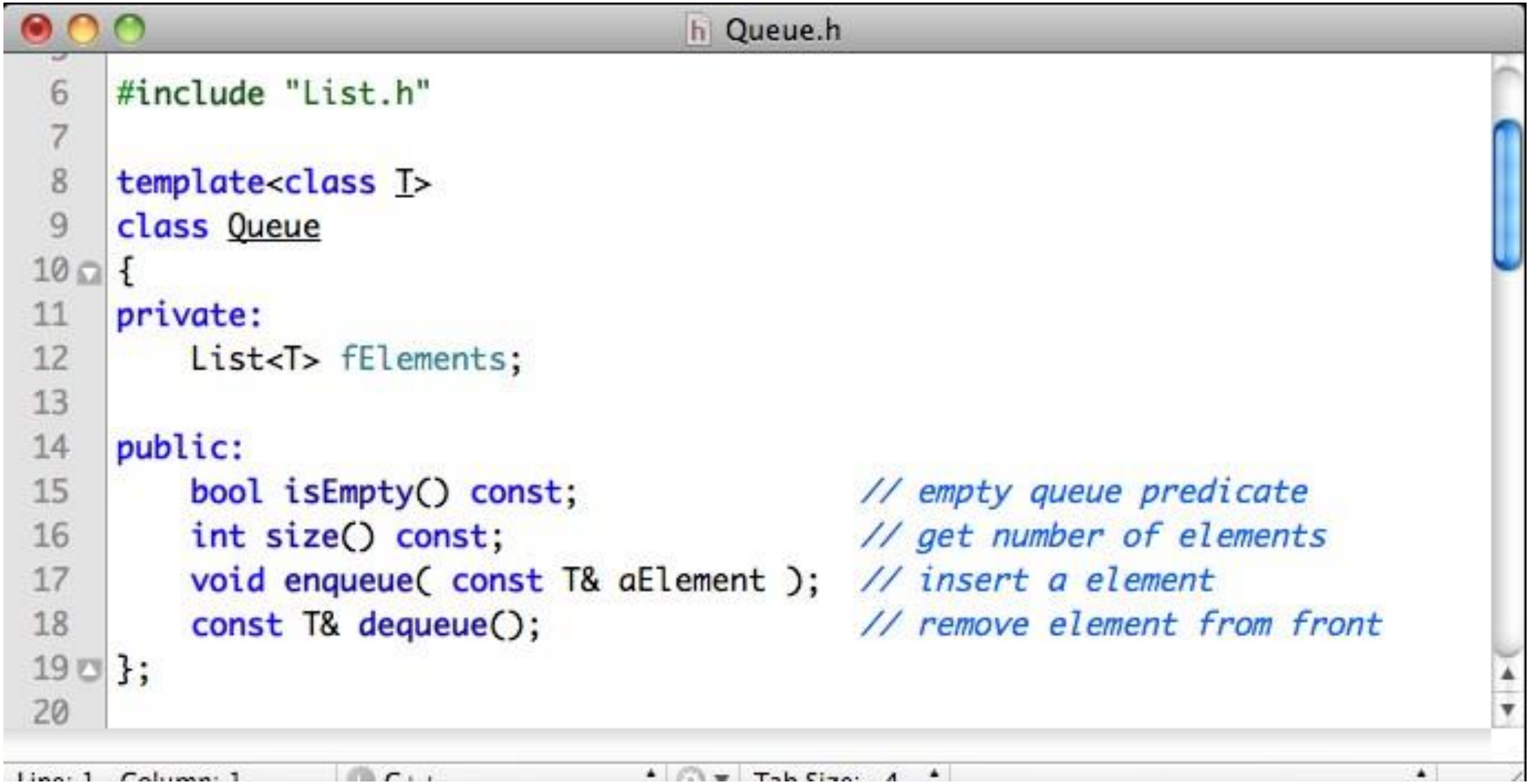
- Queue is another important basic data structure.
- A queue is a special version of a linear sequence where access to element is only possible at its front and end.



# Queue Behaviour

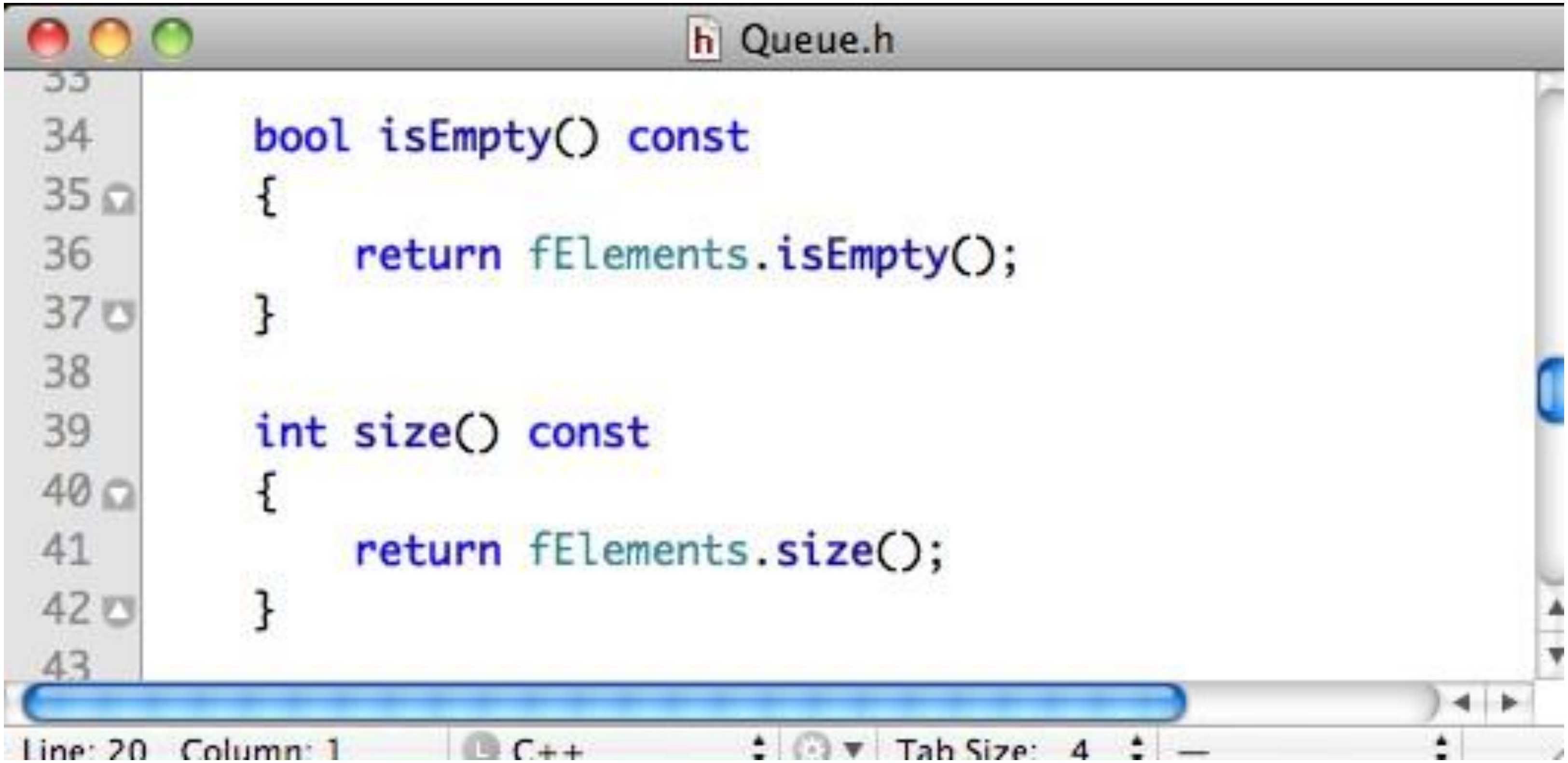
- Queues manage elements in first-in, first-out (FIFO) manner, just like a real queue.
- A queue underflow happens when one tries to dequeue on an empty queue.
- A queue overflow happens when one tries to enqueue on a full queue.

# A Queue Interface



```
#include "List.h"

template<class T>
class Queue
{
private:
    List<T> fElements;

public:
    bool isEmpty() const;                       // empty queue predicate
    int size() const;                           // get number of elements
    void enqueue( const T& aElement );          // insert a element
    const T& dequeue();                         // remove element from front
};

```

# Queue Member functions

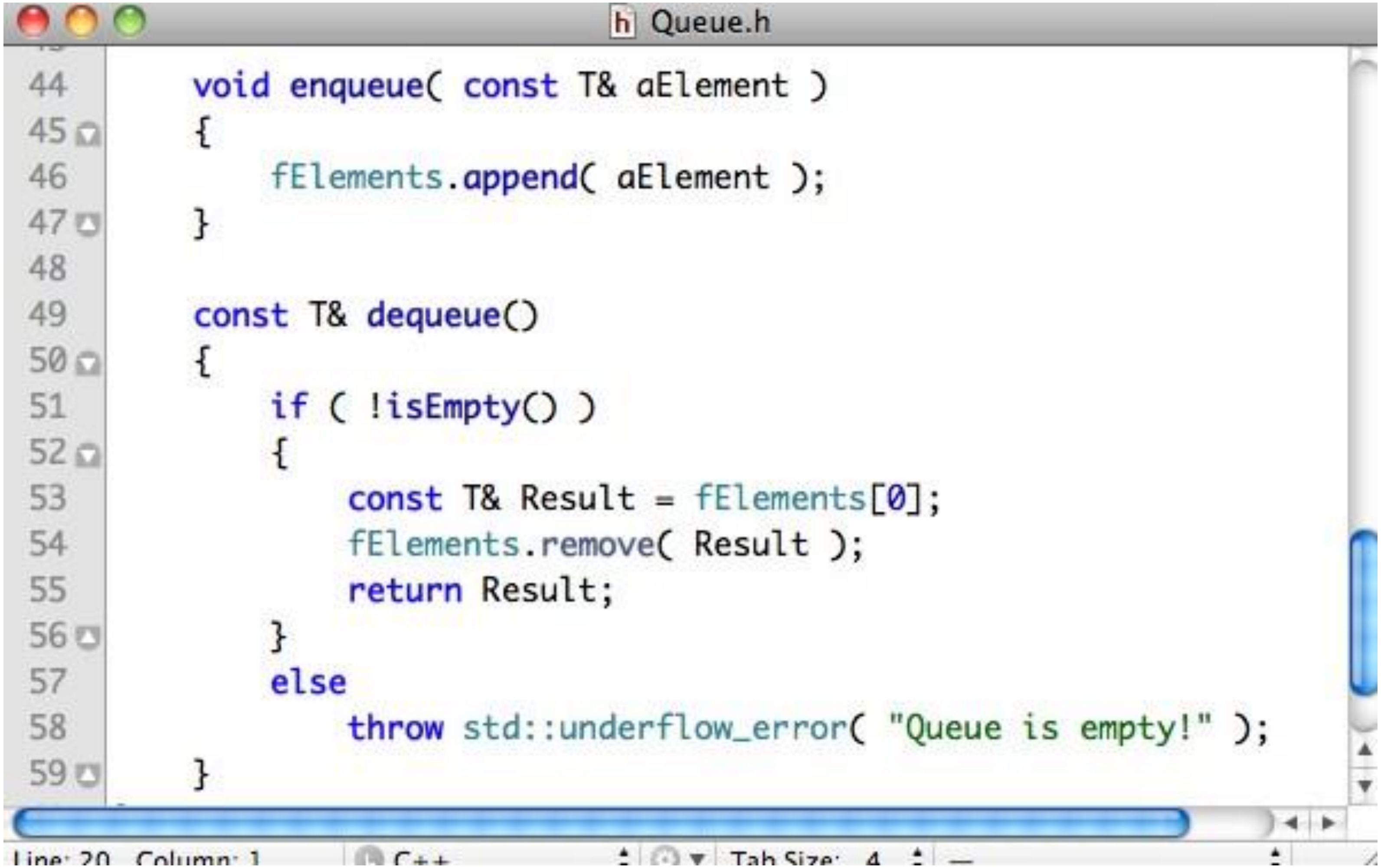


```
bool isEmpty() const
{
    return fElements.isEmpty();
}

int size() const
{
    return fElements.size();
}
```

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

SWINBURNE UNIVERSITY OF TECHNOLOGY

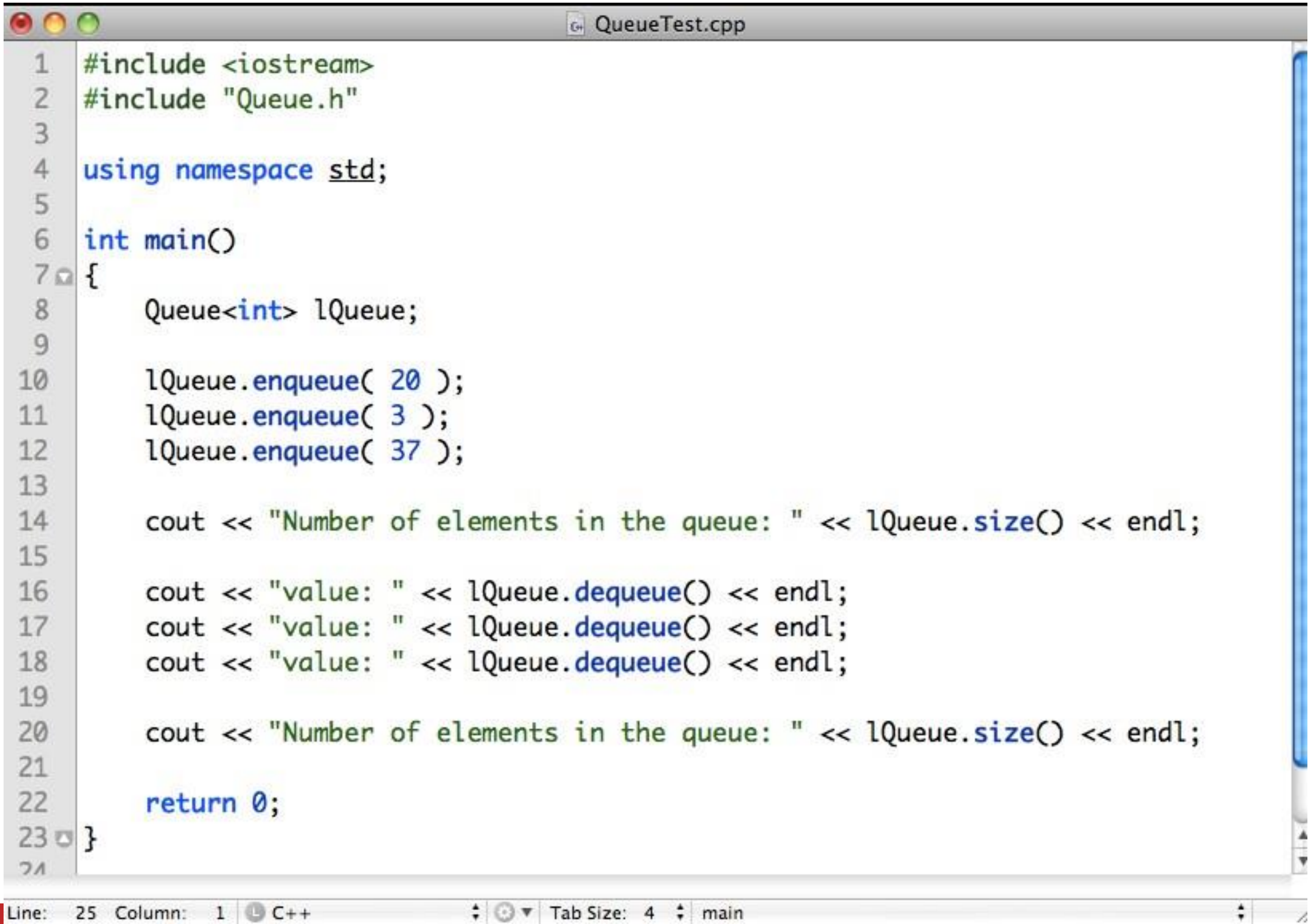
# Queue Member functions



```
    void enqueue( const T& aElement )
    {
        fElements.append( aElement );
    }

    const T& dequeue()
    {
        if ( !isEmpty() )
        {
            const T& Result = fElements[0];
            fElements.remove( Result );
            return Result;
        }
        else
            throw std::underflow_error( "Queue is empty!" );
    }
```

Line: 20 Column: 1 C++ Tab Size: 4 —

# Queue Test

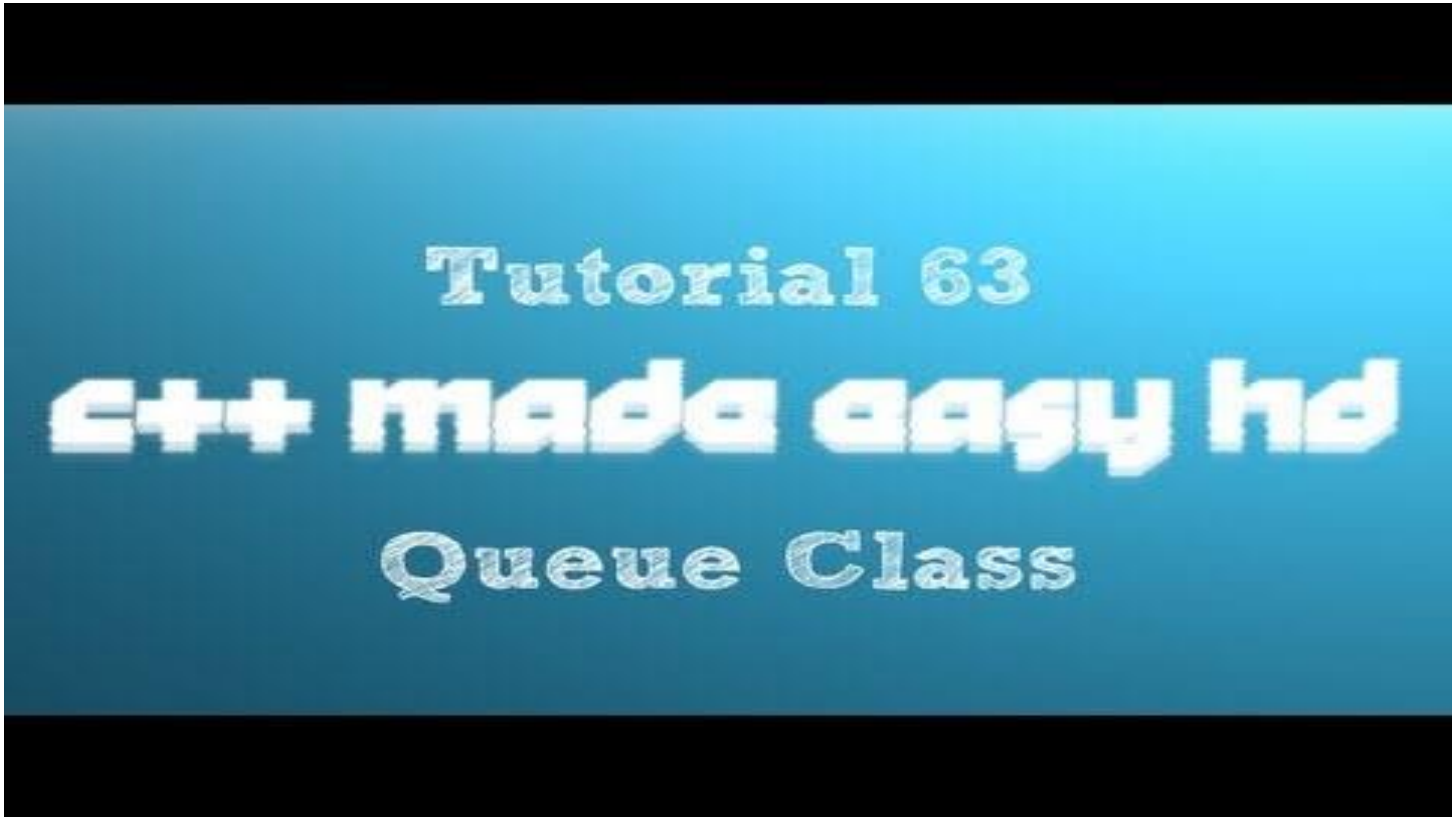


QueueTest.cpp

```
#include <iostream>
#include "Queue.h"

using namespace std;

int main()
{
    Queue<int> lQueue;

    lQueue.enqueue( 20 );
    lQueue.enqueue( 3 );
    lQueue.enqueue( 37 );

    cout << "Number of elements in the queue: " << lQueue.size() << endl;

    cout << "value: " << lQueue.dequeue() << endl;
    cout << "value: " << lQueue.dequeue() << endl;
    cout << "value: " << lQueue.dequeue() << endl;

    cout << "Number of elements in the queue: " << lQueue.size() << endl;

    return 0;
}
```

Line: 25 Column: 1 C++ Tab Size: 4 main

# Queue Class

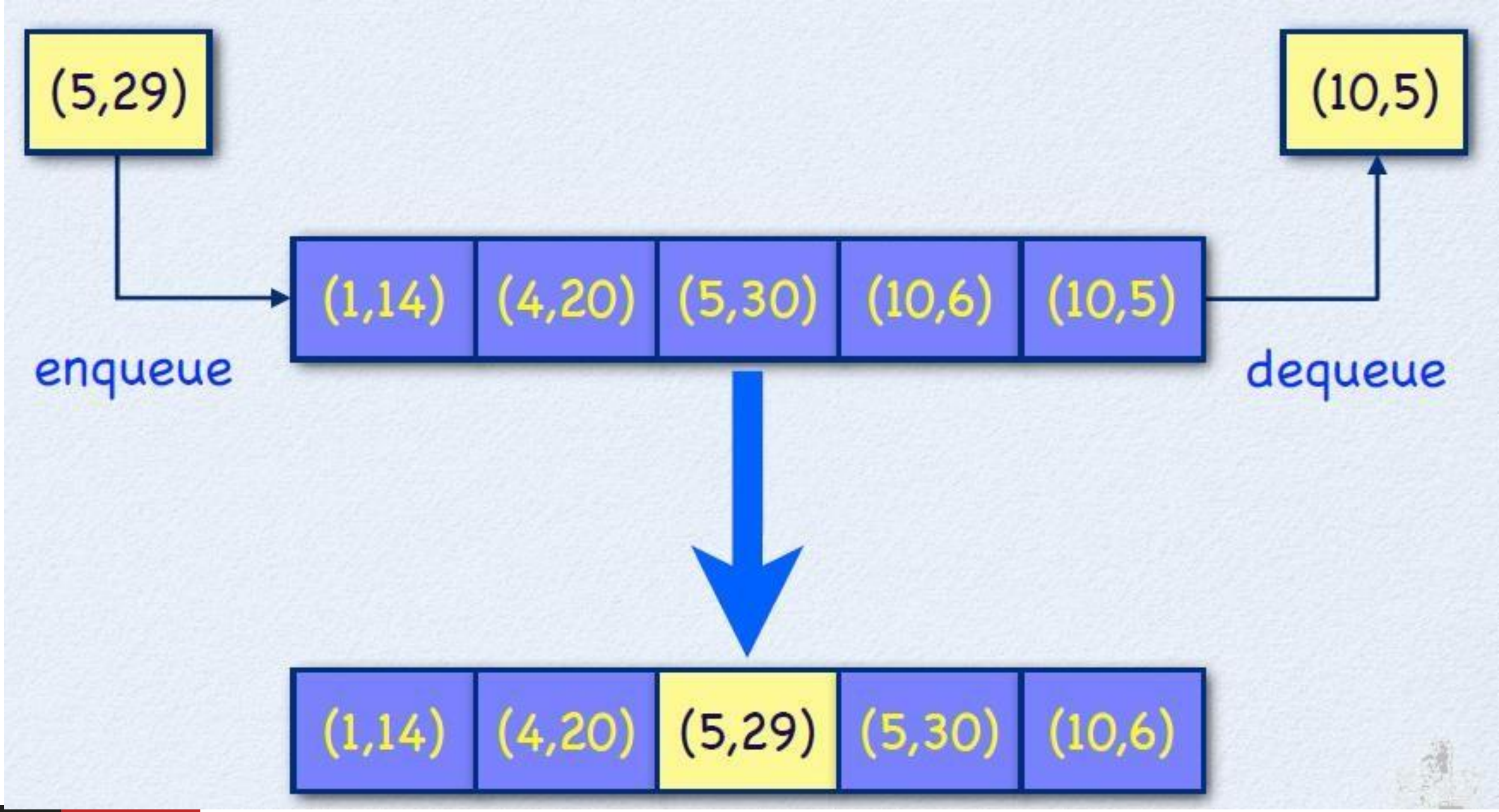


URL: https://www.youtube.com/watch?v=O2OsveGGtRU

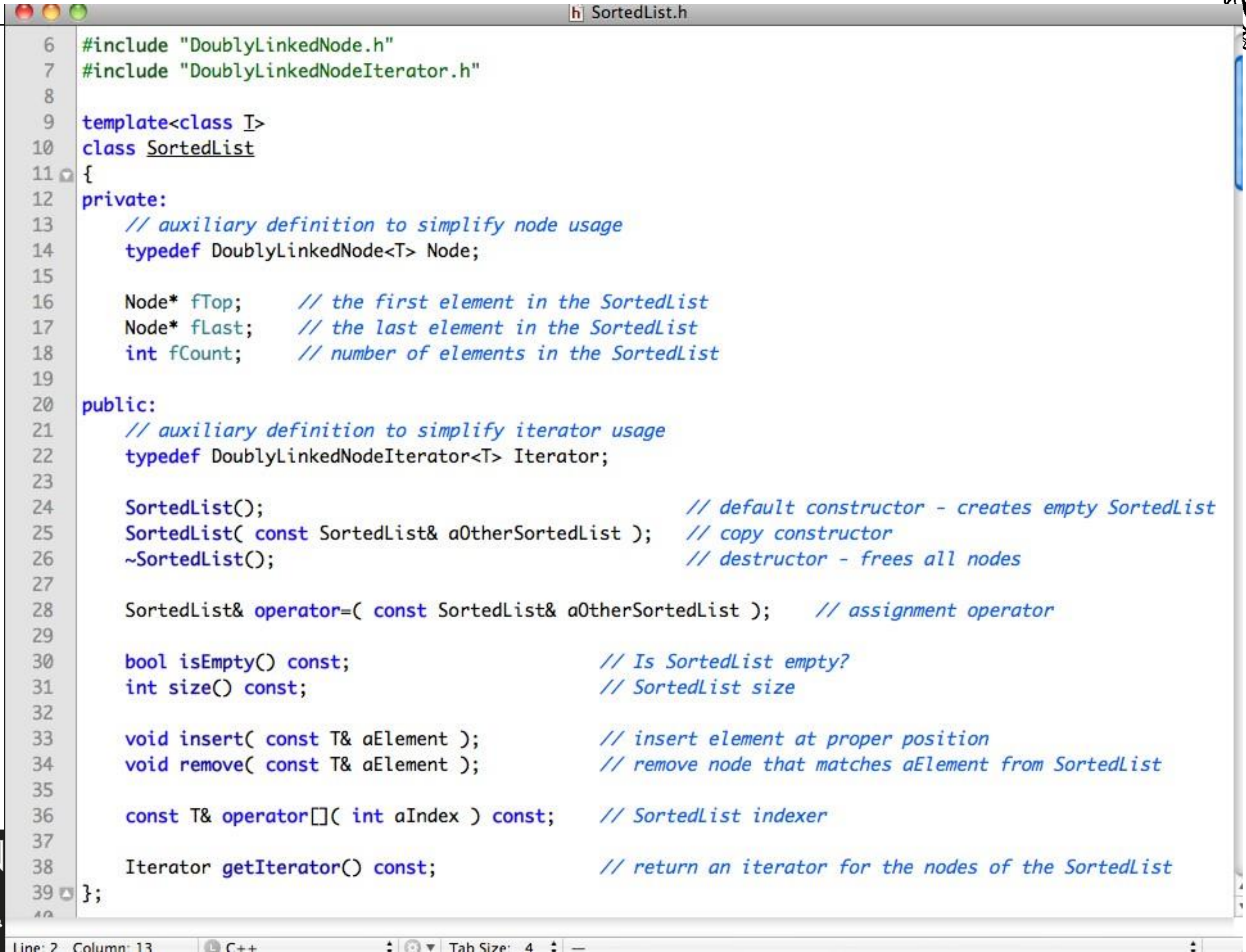
# Requirements for a Priority Queue

- The underlying data structure for a priority queue must be sorted.
- Elements are queued using an integer to specify priority. We can use a pair<Key, T> to store elements with their associated priority.
- We need to provide a matching operator < on key values to sort elements in the priority queue.

# Priority Queue



# Sorted List

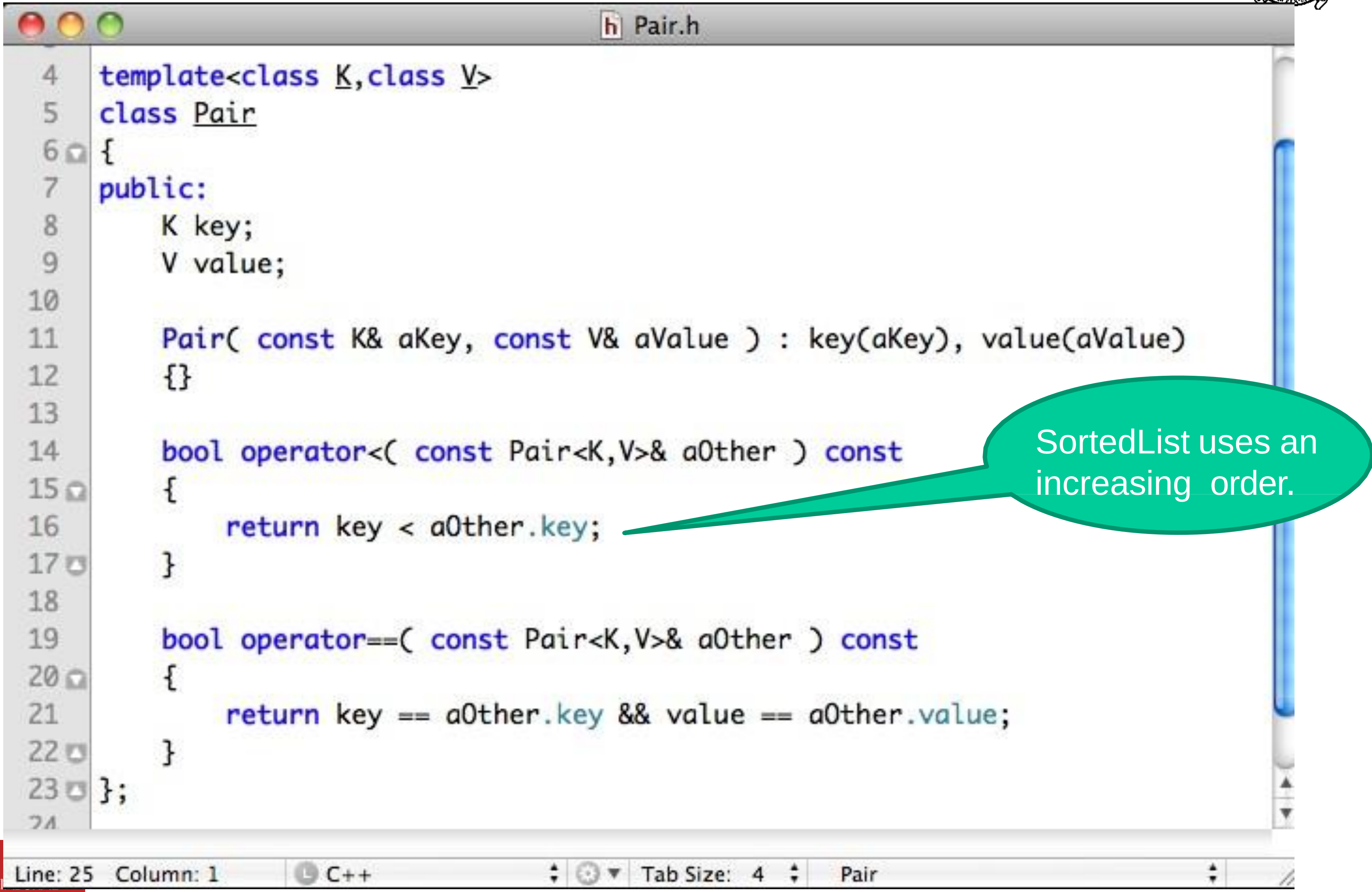


SortedList.h

```
#include "DoublyLinkedNode.h"
#include "DoublyLinkedNodeIterator.h"

template<class T>
class SortedList
{
private:
    // auxiliary definition to simplify node usage
    typedef DoublyLinkedNode<T> Node;

    Node* fTop;      // the first element in the SortedList
    Node* fLast;     // the last element in the SortedList
    int fCount;      // number of elements in the SortedList

public:
    // auxiliary definition to simplify iterator usage
    typedef DoublyLinkedNodeIterator<T> Iterator;

    SortedList();                                        // default constructor - creates empty SortedList
    SortedList( const SortedList& aOtherSortedList );    // copy constructor
    ~SortedList();                                       // destructor - frees all nodes

    SortedList& operator=( const SortedList& aOtherSortedList );    // assignment operator

    bool isEmpty() const;                        // Is SortedList empty?
    int size() const;                            // SortedList size

    void insert( const T& aElement );            // insert element at proper position
    void remove( const T& aElement );            // remove node that matches aElement from SortedList

    const T& operator[]( int aIndex ) const;     // SortedList indexer

    Iterator getIterator() const;                // return an iterator for the nodes of the SortedList
};
```

Line: 2   Column: 13   C++   Tab Size: 4

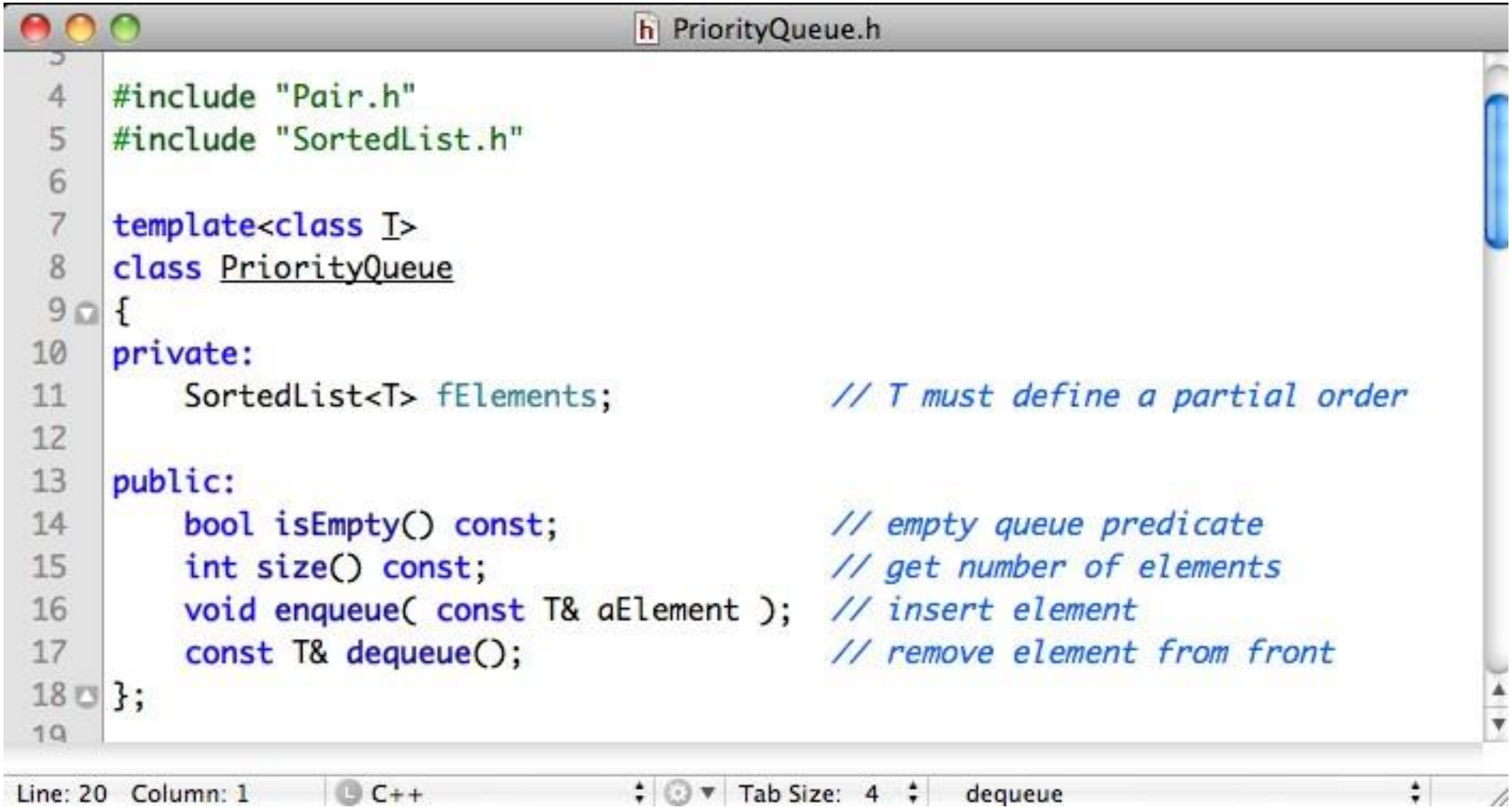
# Pair Class (Operator <)



```
template<class K,class V>
class Pair
{
public:
    K key;
    V value;

    Pair( const K& aKey, const V& aValue ) : key(aKey), value(aValue)
    {}

    bool operator<( const Pair<K,V>& aOther ) const
    {
        return key < aOther.key;
    }

    bool operator==( const Pair<K,V>& aOther ) const
    {
        return key == aOther.key && value == aOther.value;
    }
};
```

SortedList uses an increasing order.

# A Priority Queue

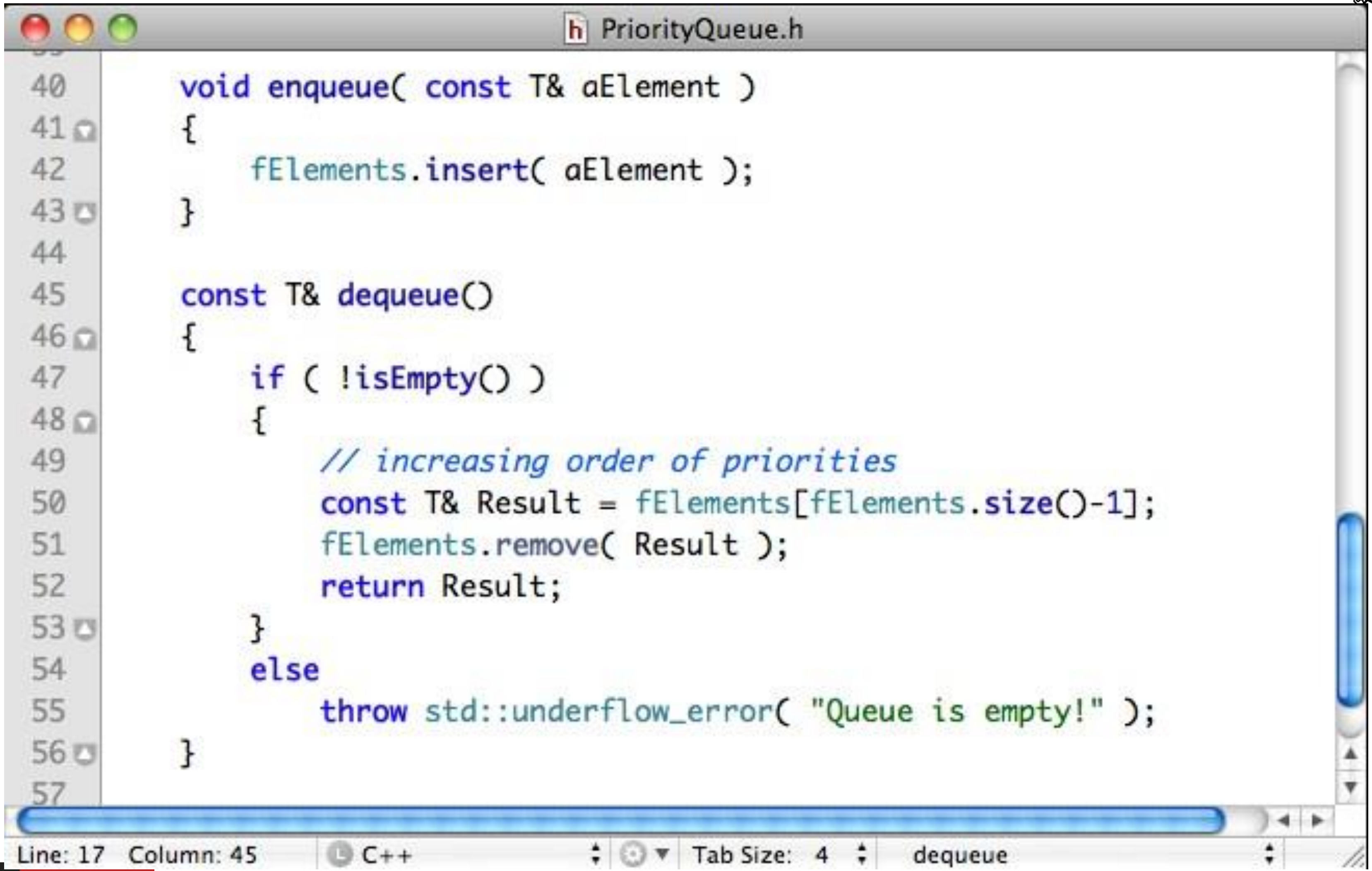


PriorityQueue.h

```cpp
#include "Pair.h"
#include "SortedList.h"

template<class T>
class PriorityQueue
{
private:
    SortedList<T> fElements;            // T must define a partial order

public:
    bool isEmpty() const;               // empty queue predicate
    int size() const;                   // get number of elements
    void enqueue( const T& aElement );  // insert element
    const T& dequeue();                 // remove element from front
};
```

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

# Priority Queue member function

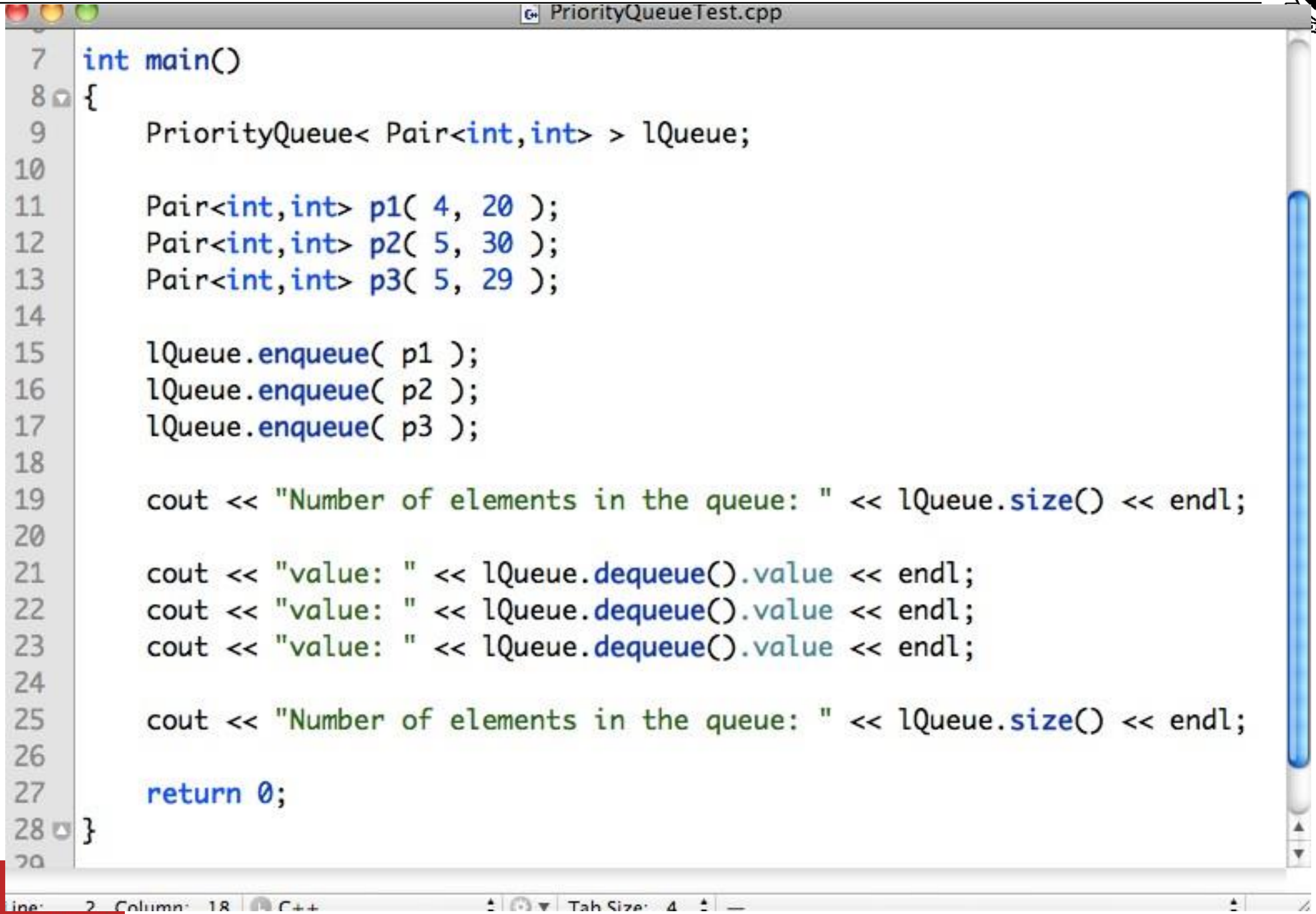


PriorityQueue.h

```
    void enqueue( const T& aElement )
    {
        fElements.insert( aElement );
    }

    const T& dequeue()
    {
        if ( !isEmpty() )
        {
            // increasing order of priorities
            const T& Result = fElements[fElements.size()-1];
            fElements.remove( Result );
            return Result;
        }
        else
            throw std::underflow_error( "Queue is empty!" );
    }
```

Line: 17 Column: 45 C++ Tab Size: 4 dequeue

# A PriorityQueue example



```cpp
int main()
{
    PriorityQueue< Pair<int,int> > lQueue;

    Pair<int,int> p1( 4, 20 );
    Pair<int,int> p2( 5, 30 );
    Pair<int,int> p3( 5, 29 );

    lQueue.enqueue( p1 );
    lQueue.enqueue( p2 );
    lQueue.enqueue( p3 );

    cout << "Number of elements in the queue: " << lQueue.size() << endl;

    cout << "value: " << lQueue.dequeue().value << endl;
    cout << "value: " << lQueue.dequeue().value << endl;
    cout << "value: " << lQueue.dequeue().value << endl;

    cout << "Number of elements in the queue: " << lQueue.size() << endl;

    return 0;
}
```

Line: 2 Column: 18 C++ Tab Size: 4

# Priority Queues

Tutorial 64

c++ made easy hd

Prority Queues

URL: https://www.youtube.com/watch?v=DHuJ-n-L8Ko

SWINBURNE
UNIVERSITY OF
TECHNOLOGY

# Copy Control and Memory Management

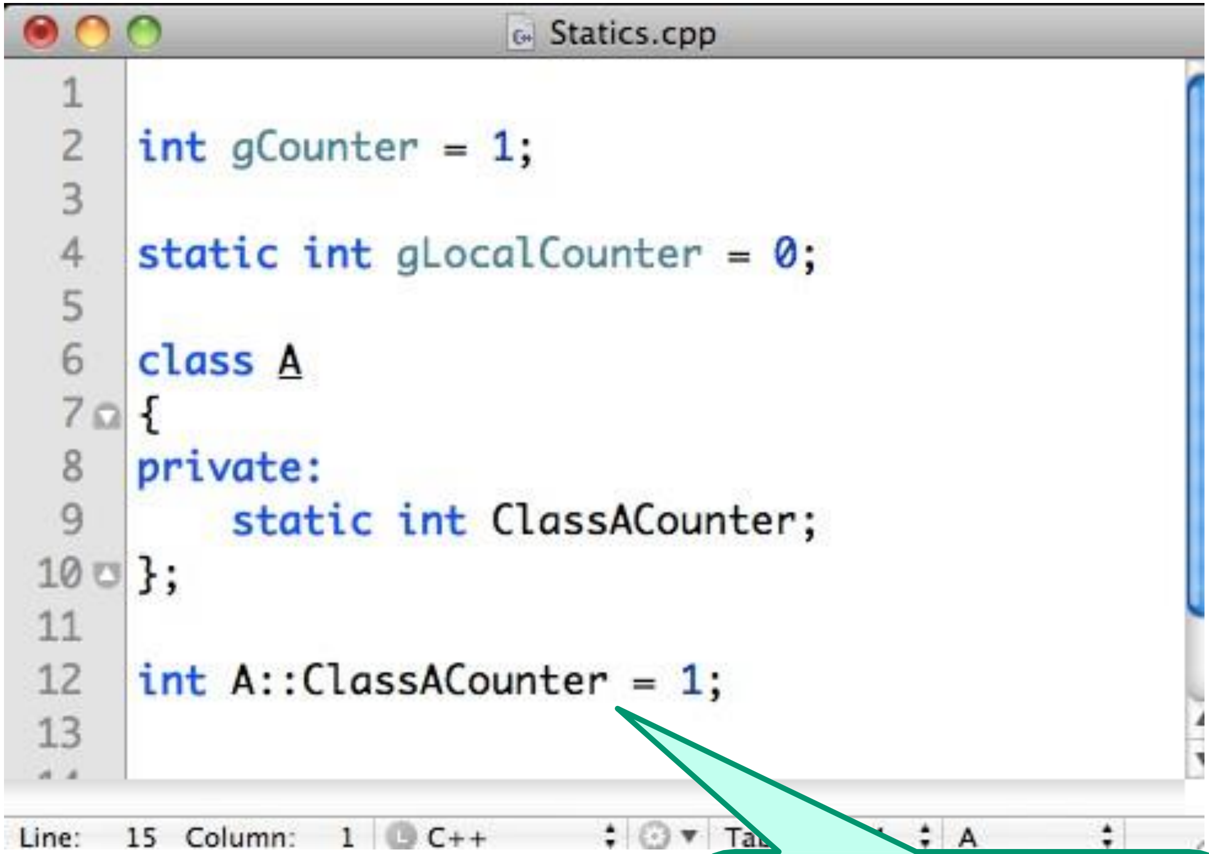
# Static variables

- 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 specified.
- Generally, local variables and function arguments are stored on the stack, while global and static variables are stored on the heap.
  - Static variables get created only once no matter how many times the function is called or how many class instances are created.

# 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.

# Operate on Static Variables

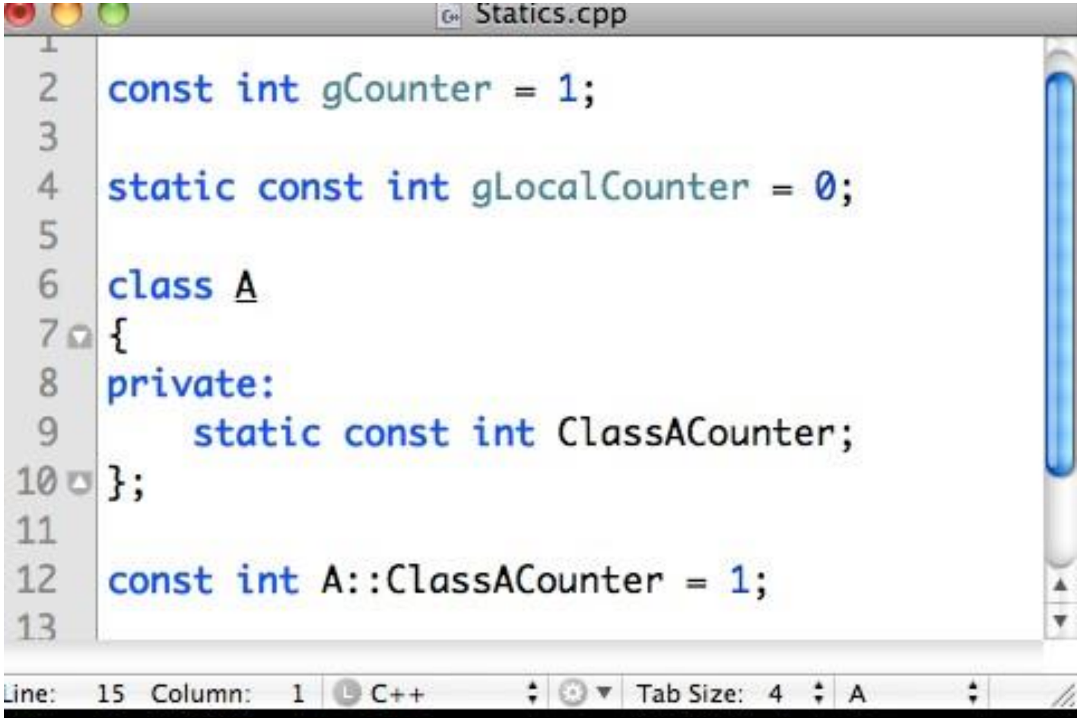


```
int gCounter = 1;

static int gLocalCounter = 0;

class A
{
private:
    static int ClassACounter;
};

int A::ClassACounter = 1;
```

**Static class variables must be initialized outside the class.**

# Read-Only Static

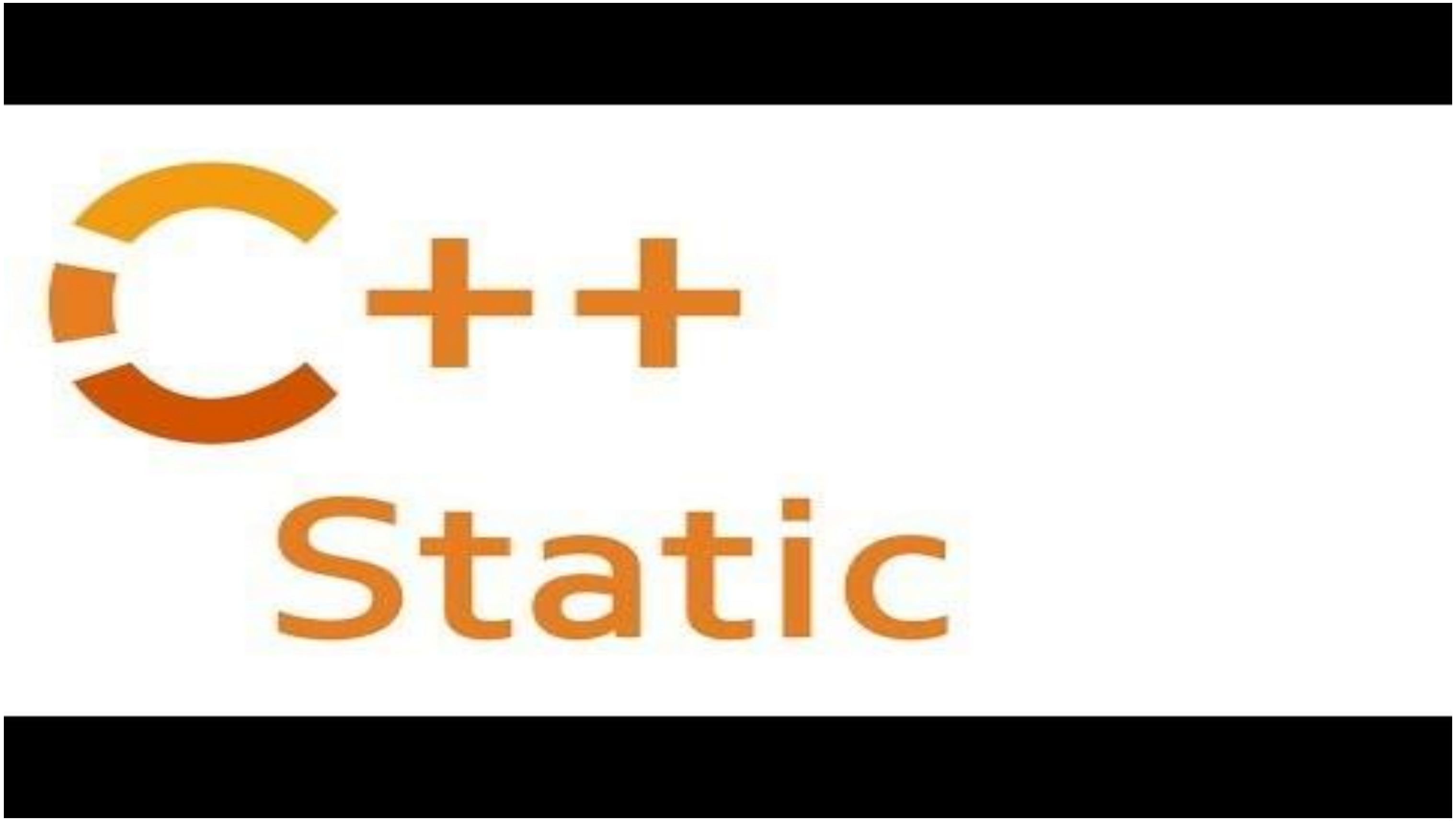
- In combination with the const specifier we can also define read-only global variables or class variables:



```

const int gCounter = 1;

static const int gLocalCounter = 0;

class A
{
private:
    static const int ClassACounter;
};

const int A::ClassACounter = 1;

```

Line: 15 Column: 1 C++ Tab Size: 4 A

# Static in C++

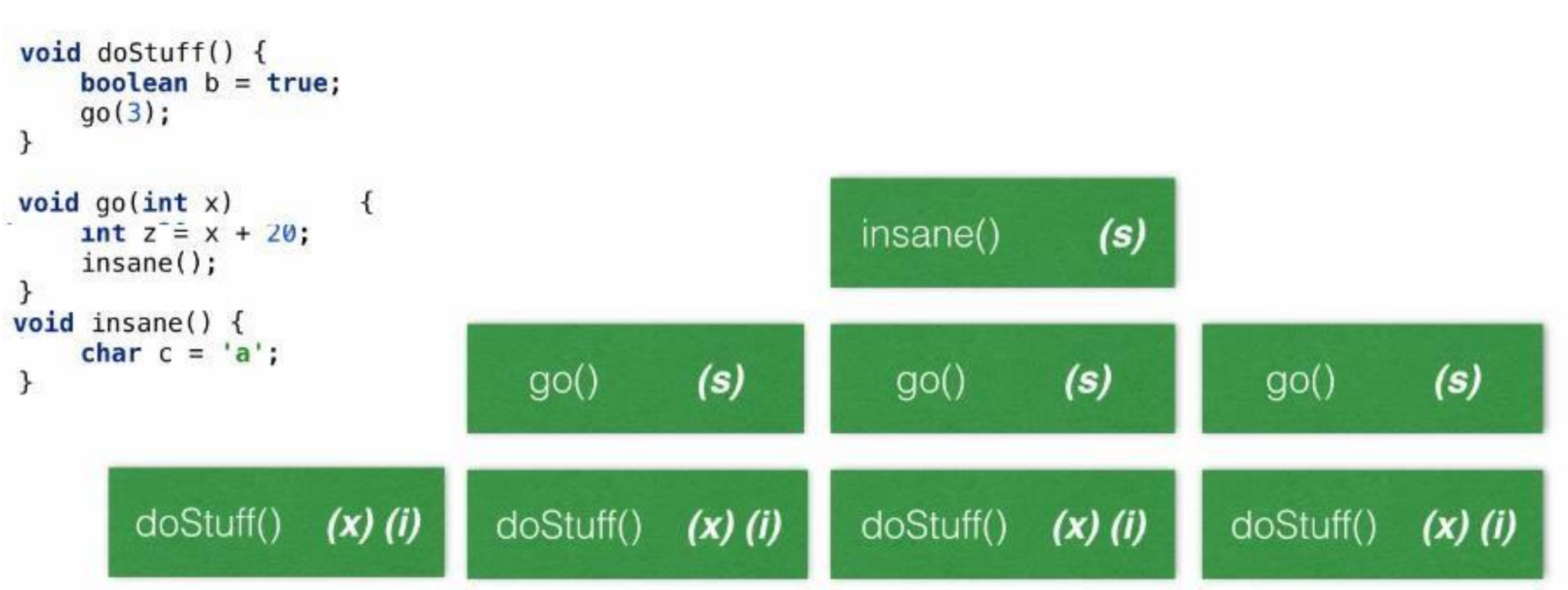


URL: https://www.youtube.com/watch?v=f3FVU-iwNuA

# 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 (a function called by another).
- References to stack locations cannot be returned from a function.

# Example of Stack



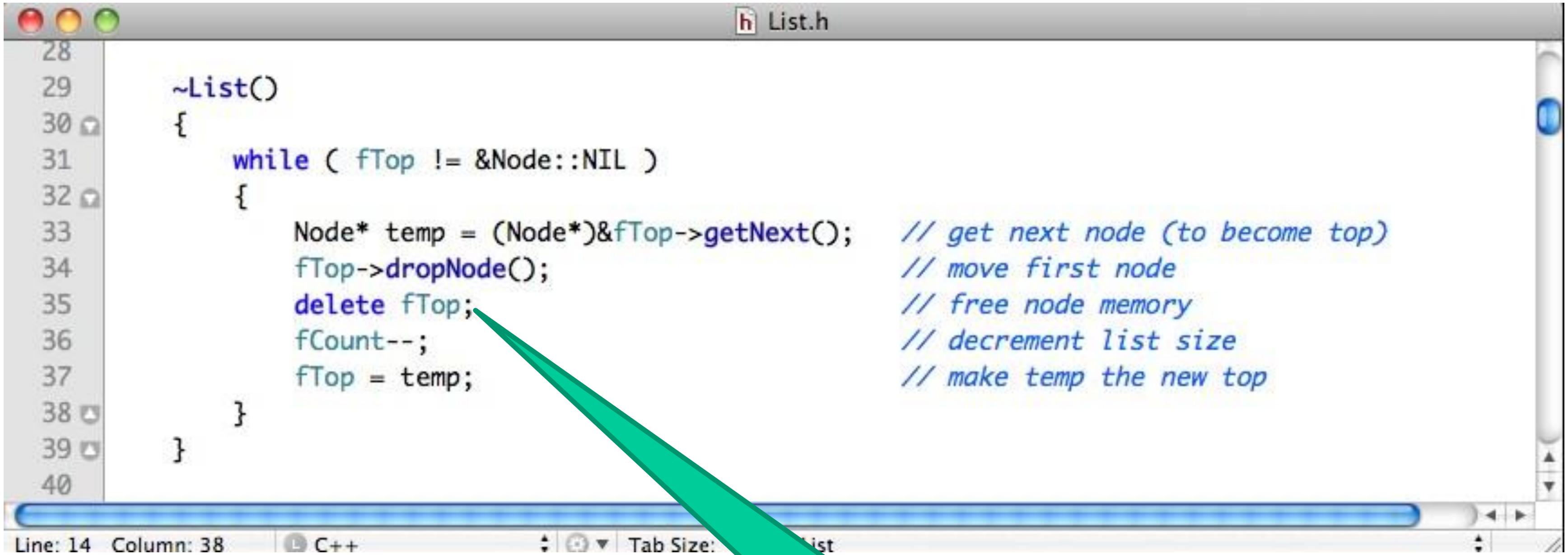
```
void doStuff() {
    boolean b = true;
    go(3);
}

void go(int x)        {
    int z = x + 20;
    insane();
}
void insane() {
    char c = 'a';
}
```

More readings on stack:
[http://cryptroix.com/2016/10/16/journey-to-the-stack/](http://cryptroix.com/2016/10/16/journey-to-the-stack/)

# 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.
  - □ a memory leak is a type of resource leak that occurs when a  computer program incorrectly manages memory allocations in such  a way that memory which is no longer needed is not released

# Example: list destructor



```
~List()
{
    while ( fTop != &Node::NIL )
    {
        Node* temp = (Node*)&fTop->getNext();   // get next node (to become top)
        fTop->dropNode();                       // move first node
        delete fTop;                            // free node memory
        fCount--;                               // decrement list size
        fTop = temp;                            // make temp the new top
    }
}
```

Release memory associated with ListNode object on the heap.

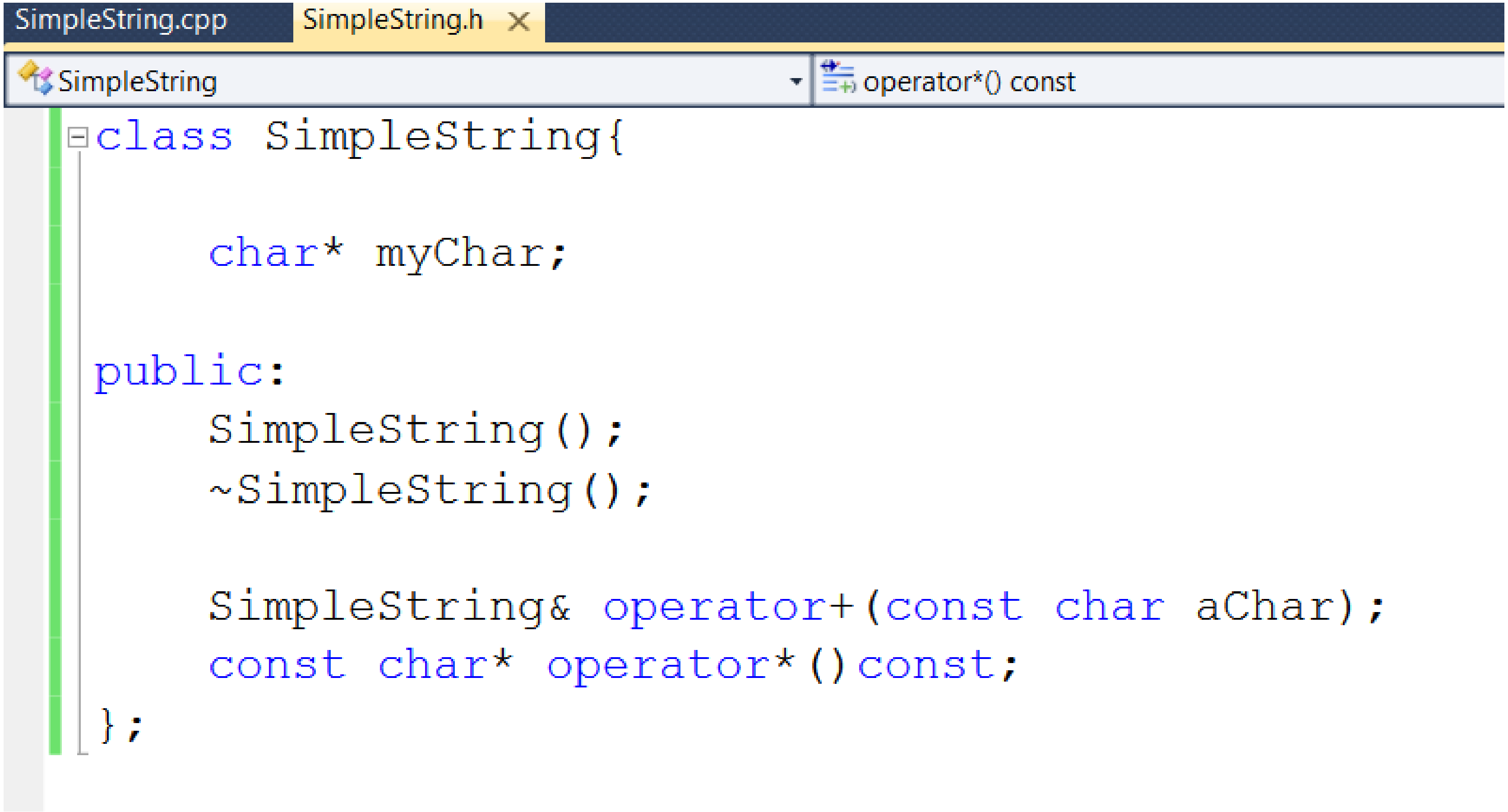
# Alias control

- 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#.
  - □ A reference variable is an **alias**, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer  to the variable.
    - To guarantee uniqueness of value-based objects in C++, we are required to define copy constructors.

# The Copy Constructor

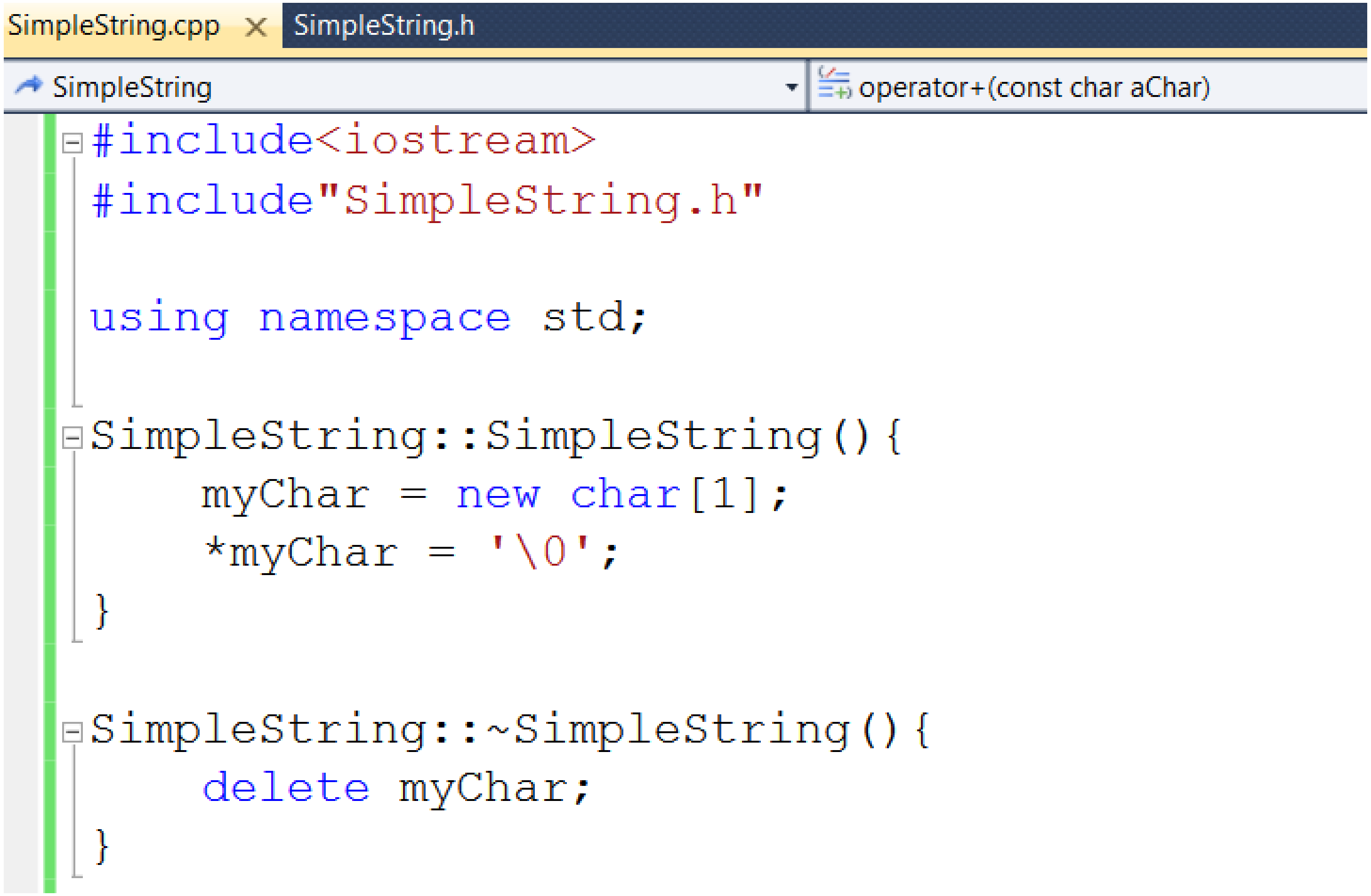
- The **copy constructor** is a **constructor** which creates an object by initializing it with an object of the same class, which has been created previously. The **copy constructor** is used to: Initialize one object from another of the same type.
- Whenever one defines a new type, one needs to specify implicitly or explicitly what has to happen when objects of that  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.

# Example: SimpleString



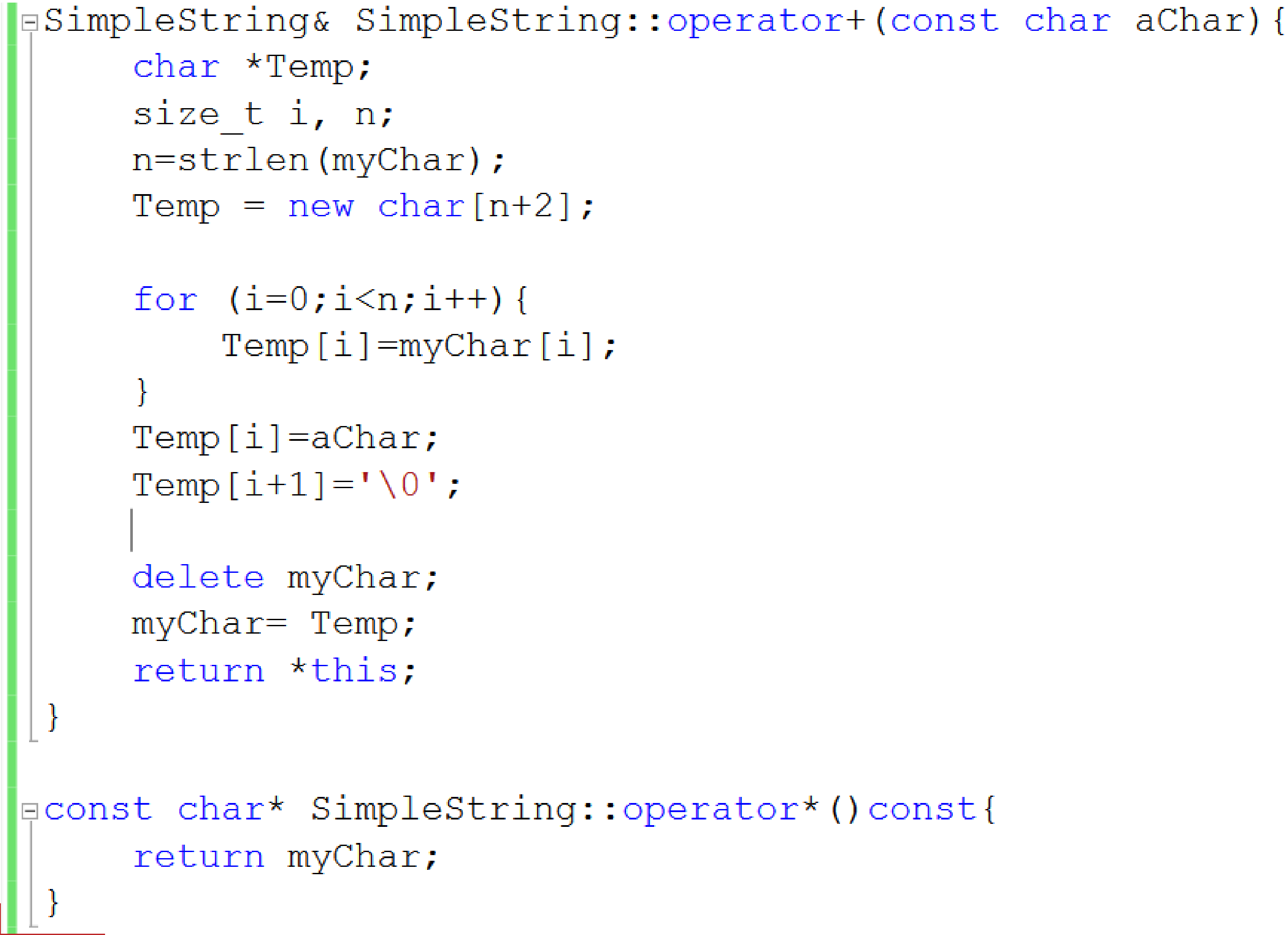
```
class SimpleString{

    char* myChar;

public:
    SimpleString();
    ~SimpleString();

    SimpleString& operator+(const char aChar);
    const char* operator*()const;
};
```

# SimpleString: Constructor & Destructor



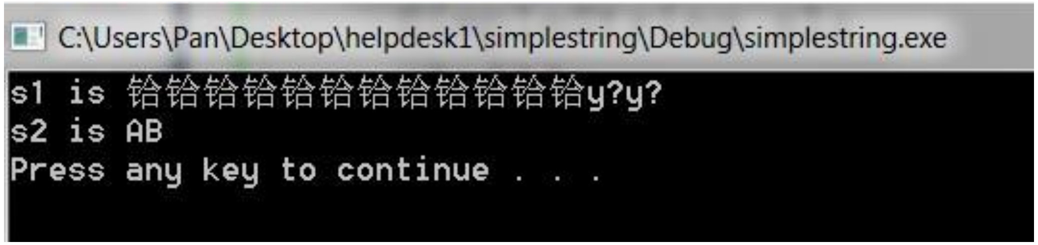
```cpp
#include<iostream>
#include"SimpleString.h"

using namespace std;

SimpleString::SimpleString(){
    myChar = new char[1];
    *myChar = '\0';
}

SimpleString::~SimpleString(){
    delete myChar;
}
```

# SimpleString: The Operators



```
SimpleString& SimpleString::operator+(const char aChar){
    char *Temp;
    size_t i, n;
    n=strlen(myChar);
    Temp = new char[n+2];

    for (i=0;i<n;i++){
        Temp[i]=myChar[i];
    }
    Temp[i]=aChar;
    Temp[i+1]='\0';

    delete myChar;
    myChar= Temp;
    return *this;
}

const char* SimpleString::operator*()const{
    return myChar;
}
```

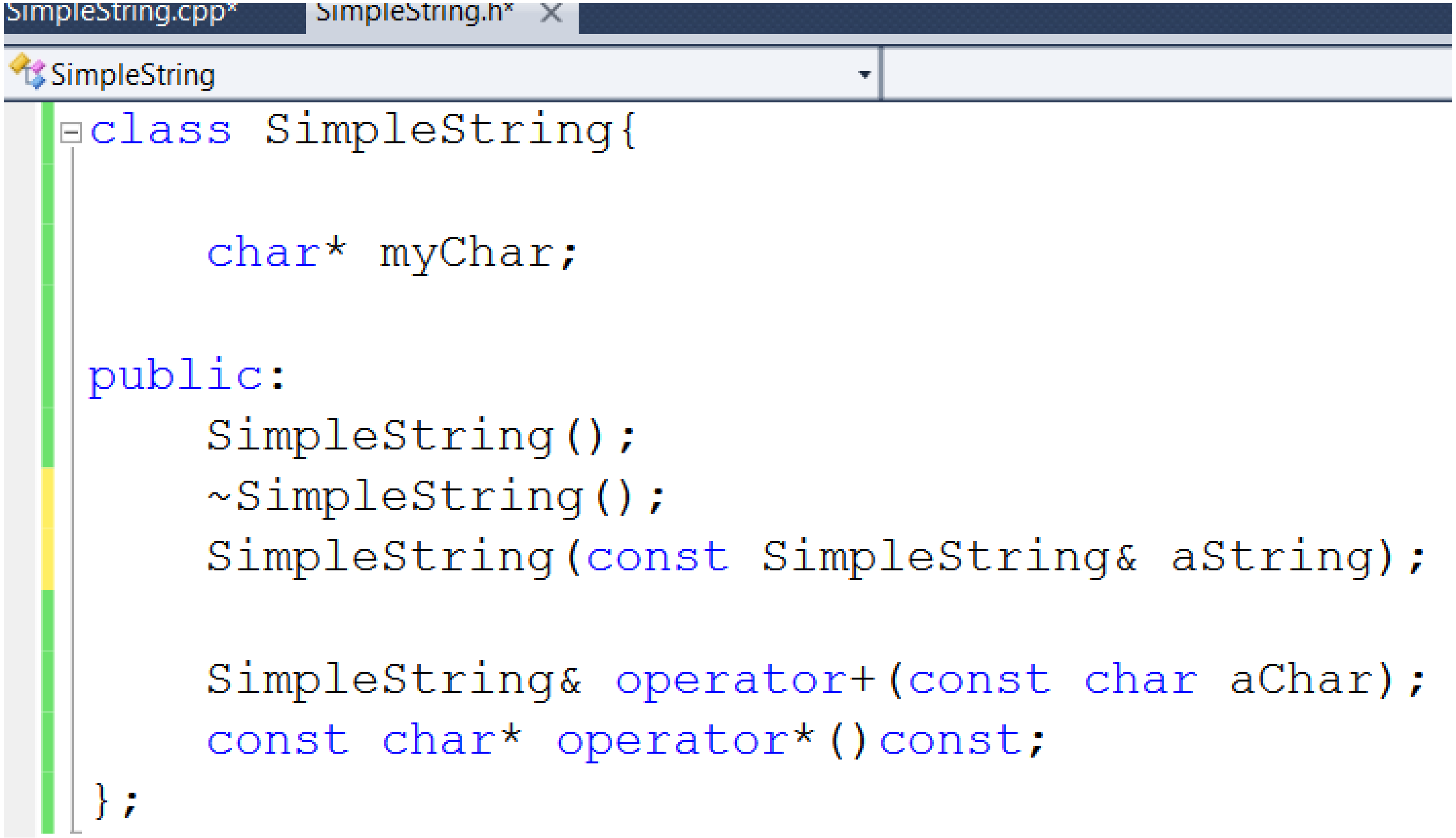
```
int main(){

    SimpleString s1;
    s1+'A';

    SimpleString s2=s1;
    s2+'B';

    cout<<"s1 is "<<*s1<<endl;
    cout<<"s2 is "<<*s2<<endl;

    system("pause");
    return 0;
}
```



C:\Users\Pan\Desktop\helpdesk1\simplestring\Debug\simplestring.exe

```
s1 is 铪铪铪铪铪铪铪铪铪铪铪铪y?y?
s2 is AB
Press any key to continue . . .
```

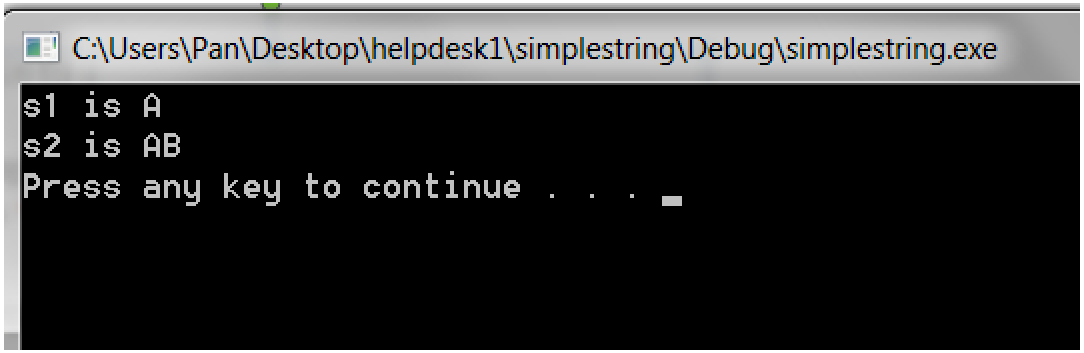
# Explicit copy constructor



```
class SimpleString{

    char* myChar;

public:
    SimpleString();
    ~SimpleString();
    SimpleString(const SimpleString& aString);

    SimpleString& operator+(const char aChar);
    const char* operator*()const;
};
```

# Explicit Copy Constructor in Use

```
int main(){

    SimpleString s1;
    s1+'A';

    SimpleString s2(s1);

    //s2=s1;
    s2+'B';

    cout<<"s1 is "<<*s1<<endl;
    cout<<"s2 is "<<*s2<<endl;
```



C:\Users\Pan\Desktop\helpdesk1\simplestring\Debug\simplestring.exe

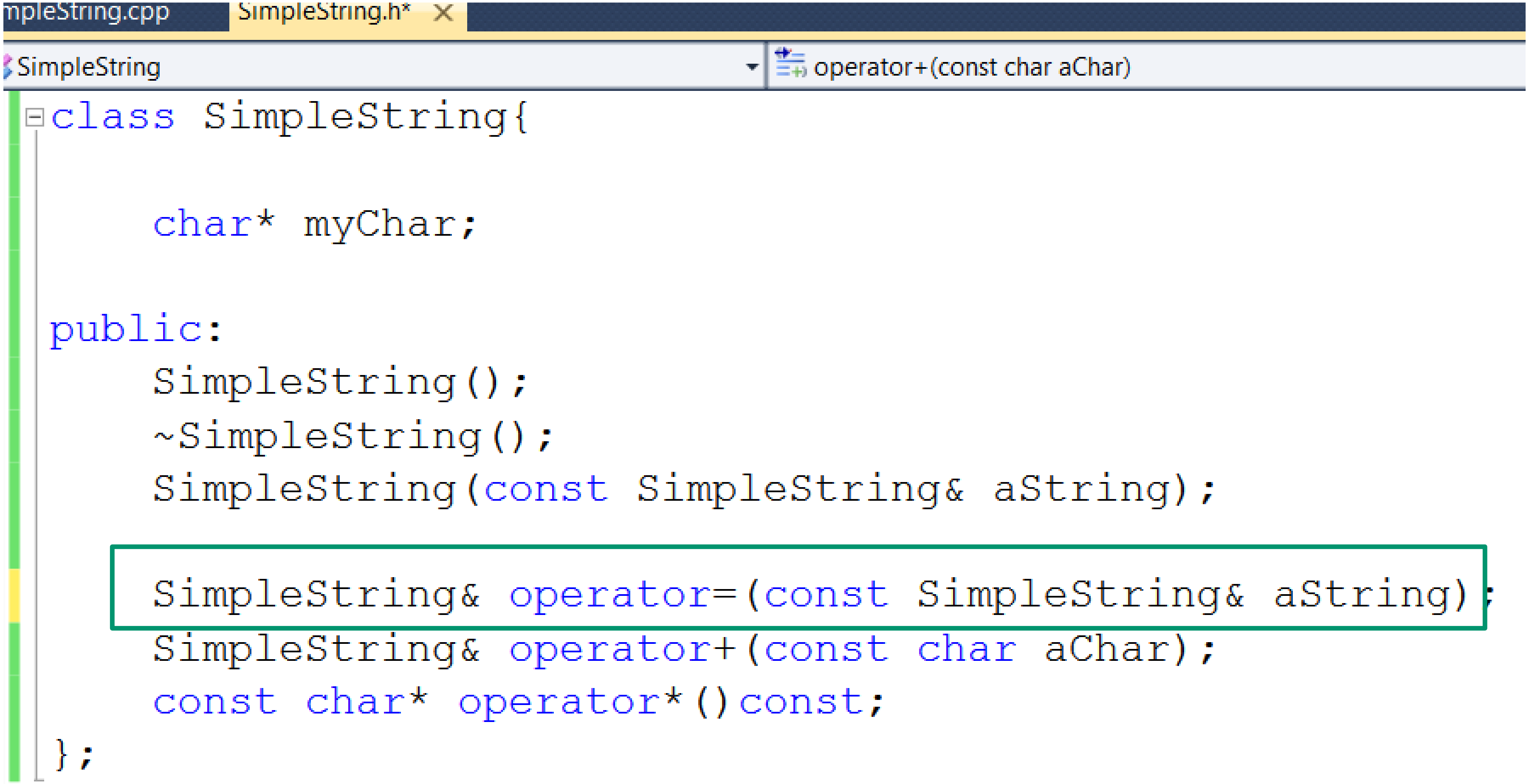
```
s1 is A
s2 is AB
Press any key to continue . . .
```

# 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.

# What if I want to use “=”

- Overload the operator “=”



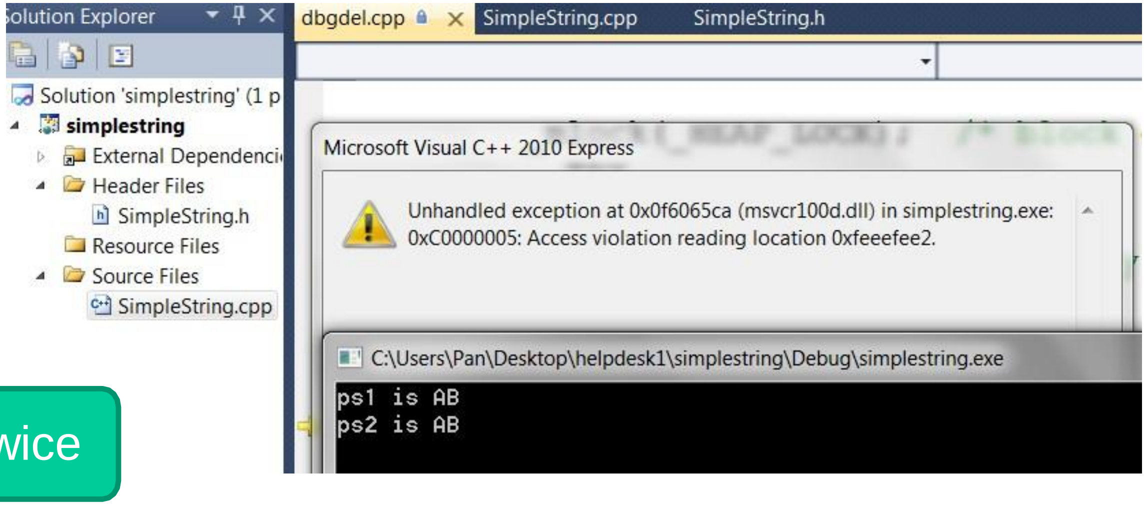
```
class SimpleString{

    char* myChar;

public:
    SimpleString();
    ~SimpleString();
    SimpleString(const SimpleString& aString);

    SimpleString& operator=(const SimpleString& aString);
    SimpleString& operator+(const char aChar);
    const char* operator*()const;
};
```

# Copying Pointers

```
int main(){


    SimpleString* ps1= new SimpleString();
    *ps1+'A';

    SimpleString* ps2=ps1;
    *ps2+'B';


    cout<<"ps1 is "<<**ps1<<endl;
    cout<<"ps2 is "<<**ps2<<endl;

    delete ps1;
    delete ps2;
```

Shallow Copy

Free twice

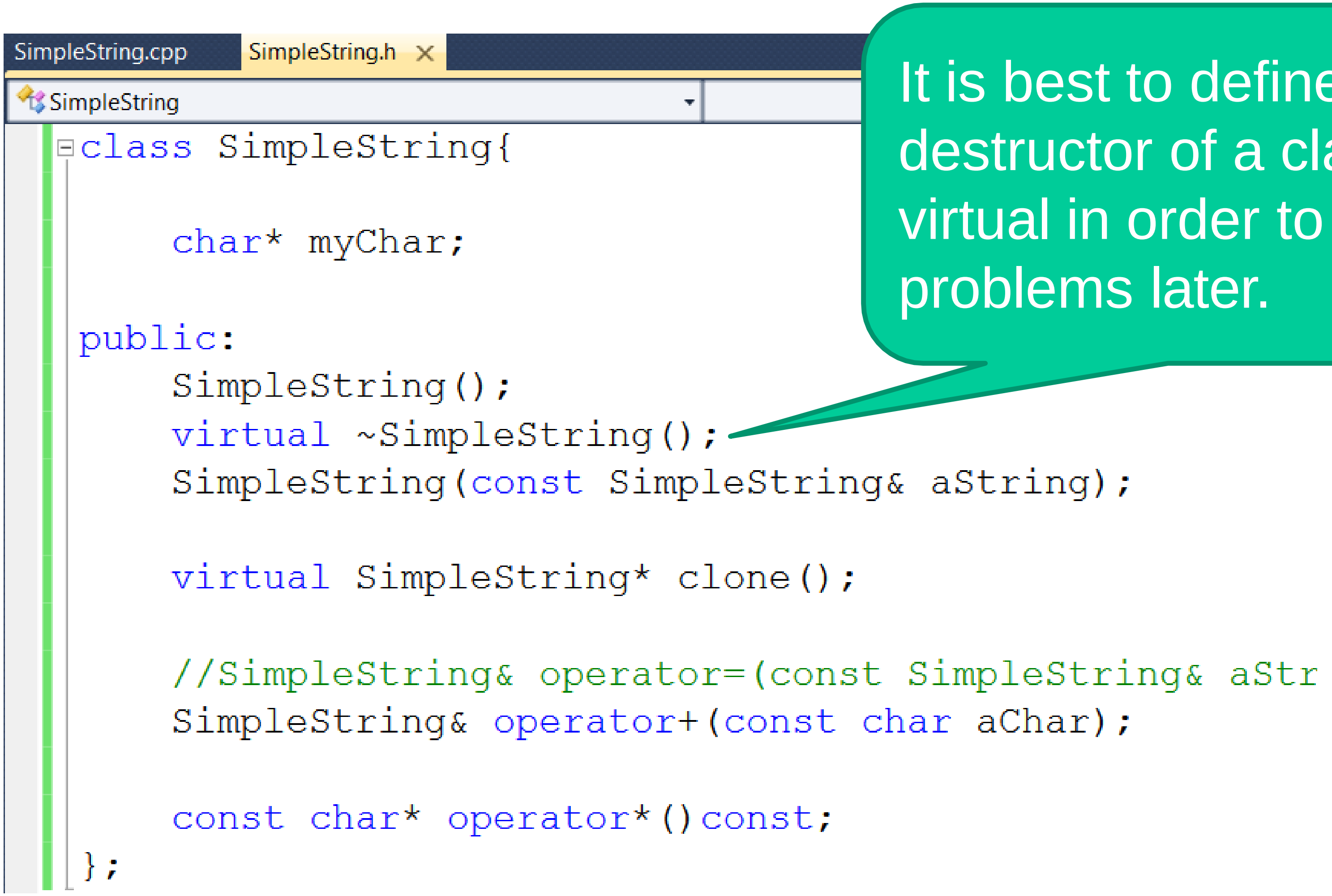


Unhandled exception at 0x0f6065ca (msvcr100d.dll) in simplestring.exe: 0xC0000005: Access violation reading location 0xfeeefee2.

C:\Users\Pan\Desktop\helpdesk1\simplestring\Debug\simplestring.exe

```
ps1 is AB
ps2 is AB
```

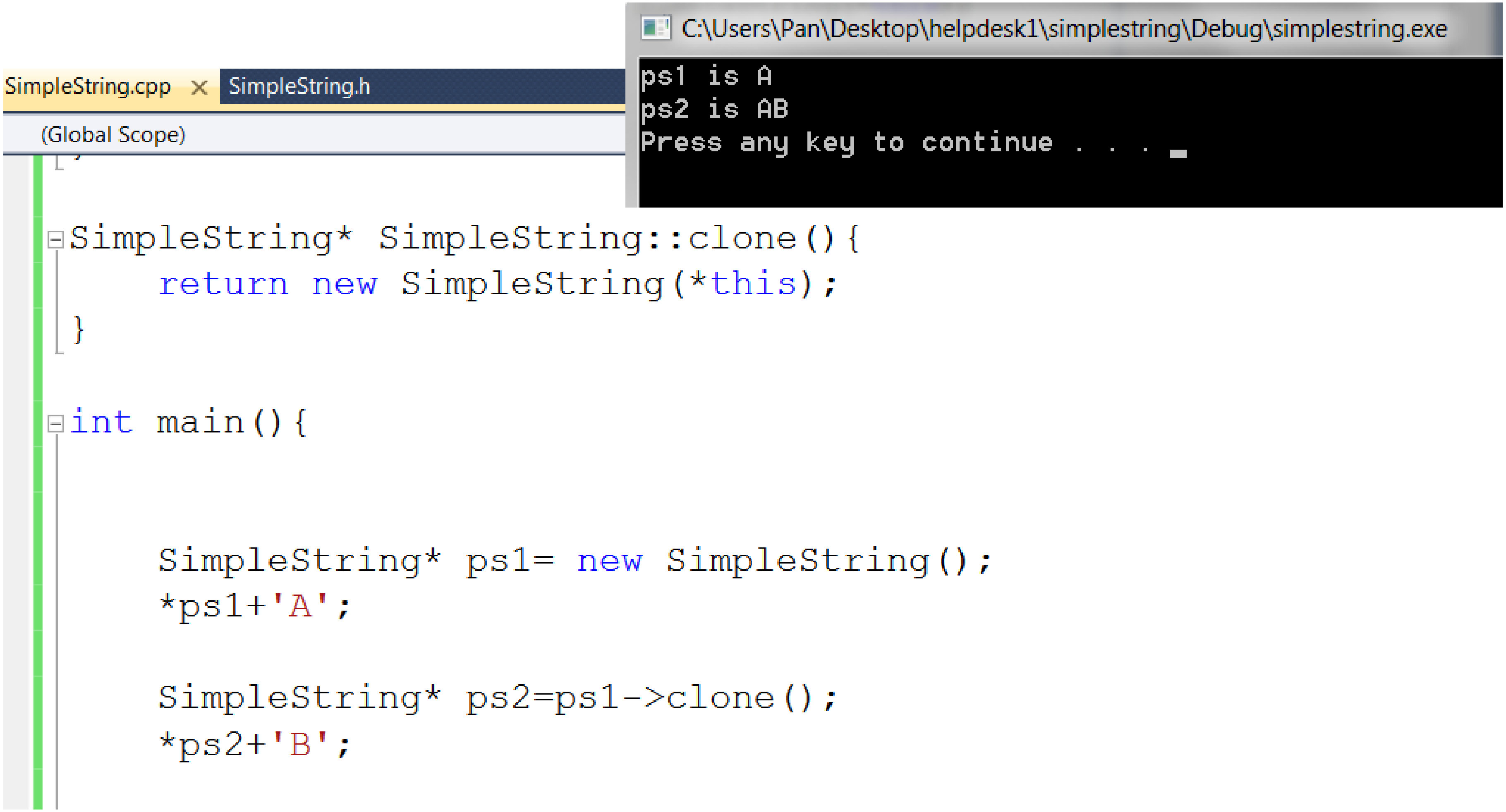
# Solution: A clone() function



```
class SimpleString{

    char* myChar;

public:
    SimpleString();
    virtual ~SimpleString();
    SimpleString(const SimpleString& aString);

    virtual SimpleString* clone();

    //SimpleString& operator=(const SimpleString& aStr
    SimpleString& operator+(const char aChar);

    const char* operator*()const;
};
```

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

# The Use of clone()



C:\Users\Pan\Desktop\helpdesk1\simplestring\Debug\simplestring.exe
SimpleString.cpp
SimpleString.h
(Global Scope)

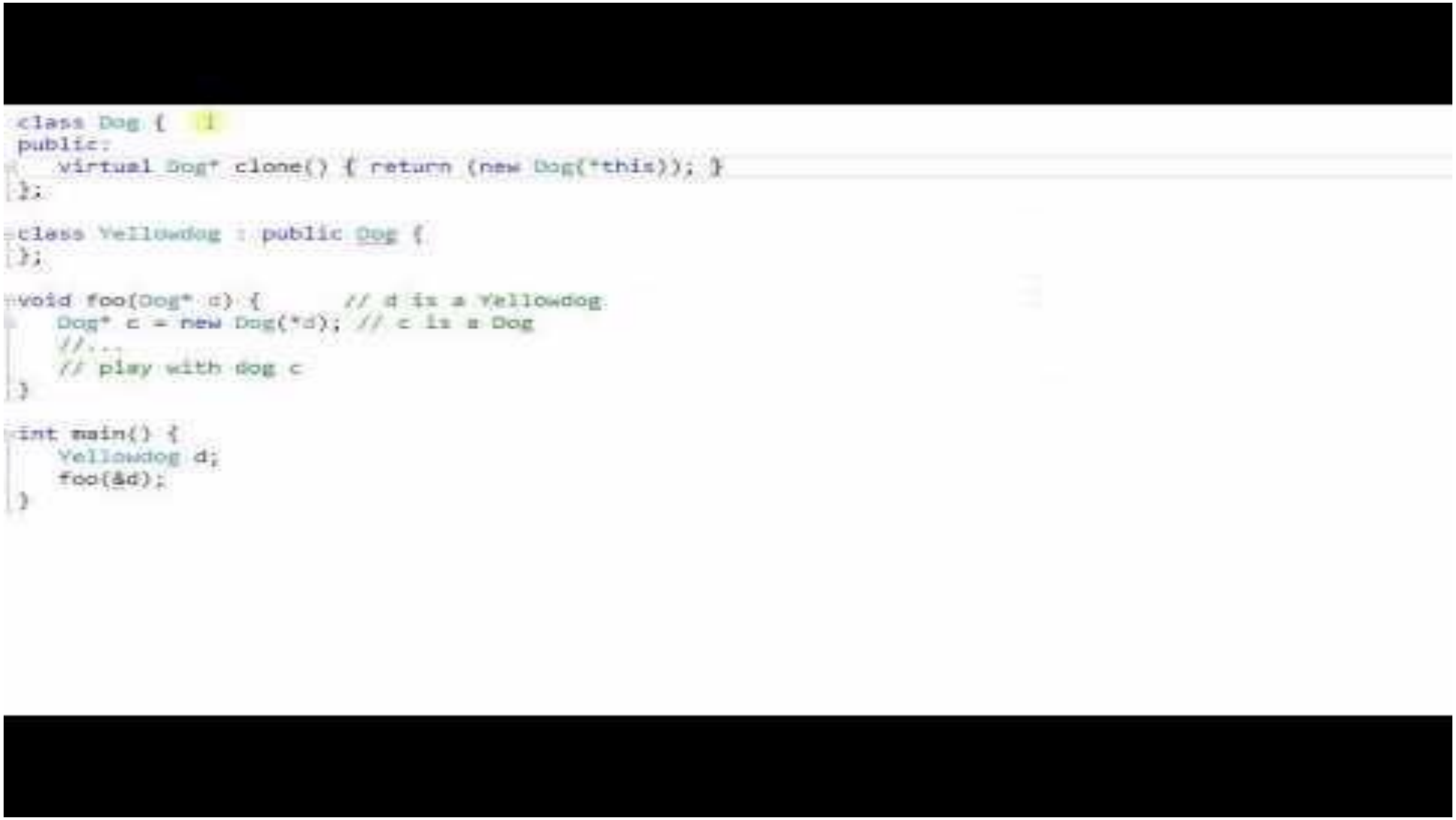
```
ps1 is A
ps2 is AB
Press any key to continue . . .
```

```
SimpleString* SimpleString::clone(){
    return new SimpleString(*this);
}

int main(){


    SimpleString* ps1= new SimpleString();
    *ps1+'A';

    SimpleString* ps2=ps1->clone();
    *ps2+'B';
```

# Note with clone()

- if a class has *any* virtual function, it should have a virtual destructor,
- The member function clone() must be defined virtual to allow for proper redefinition in subtypes.
- The member function clone() can only return one type. When a subtype redefines clone(), only the super type can be returned

# Virtual Constructor – Clone() Function



```
class Dog {
public:
    virtual Dog* clone() { return (new Dog(*this)); }
};

class Yellowdog : public Dog {
};

void foo(Dog* d) {        // d is a Yellowdog
    Dog* c = new Dog(*d); // c is a Dog
    //...
    // play with dog c
}

int main() {
    Yellowdog d;
    foo(&d);
}
```

URL: https://www.youtube.com/watch?v=UHP-DKrxgBs

# 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.

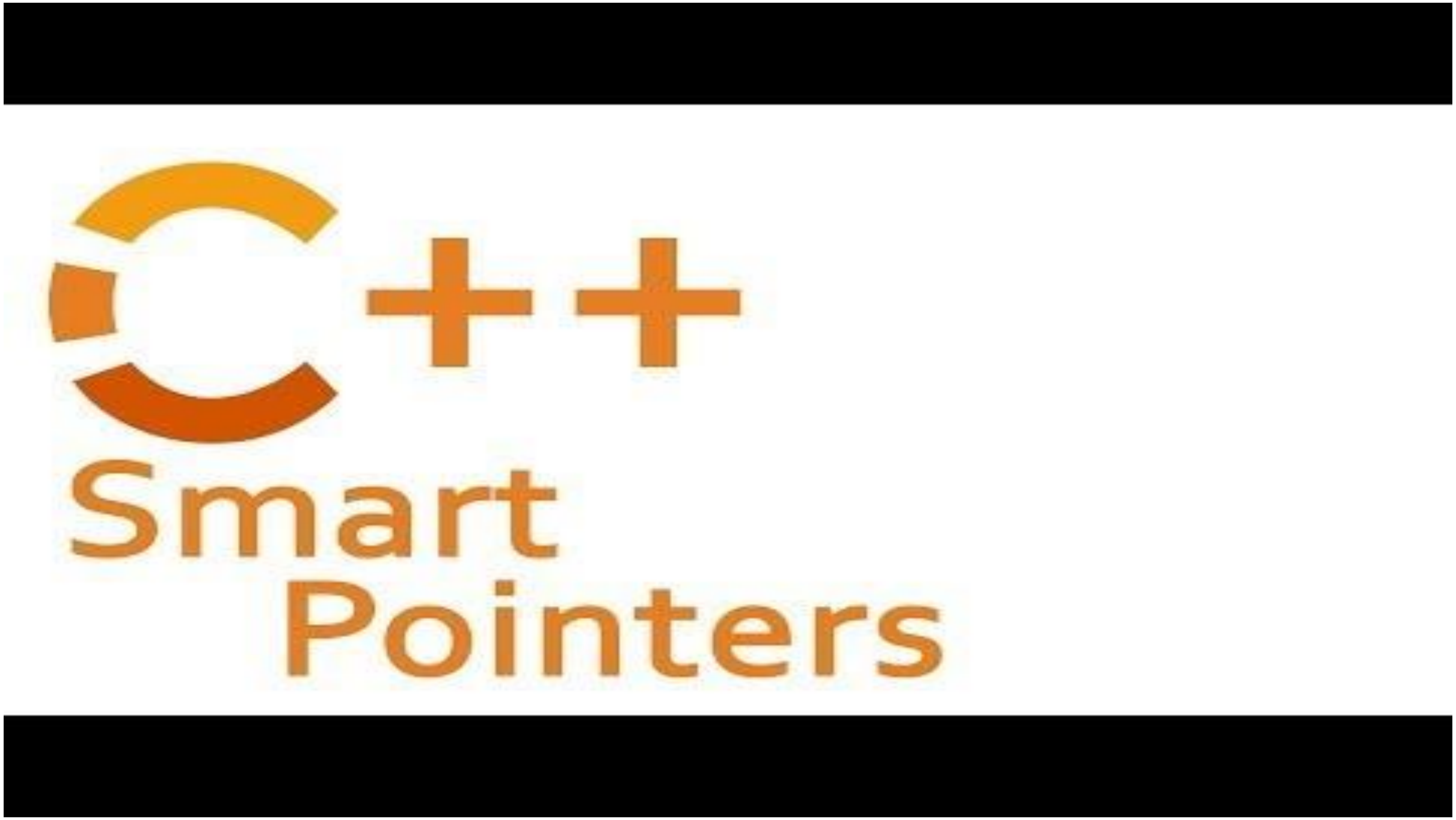
# Smart Pointers in C++

- There are several smart pointers available:
  - std::unique_ptr
  - std::shared_ptr
  - std::weak_ptr
- Smart pointers are used to guarantee automatic deletion of heap-allocated class instances.
- With the unique and shared smart pointers, you do not have to call delete yourself for an object, as it will be done automatically.

# Smart Pointers in C++

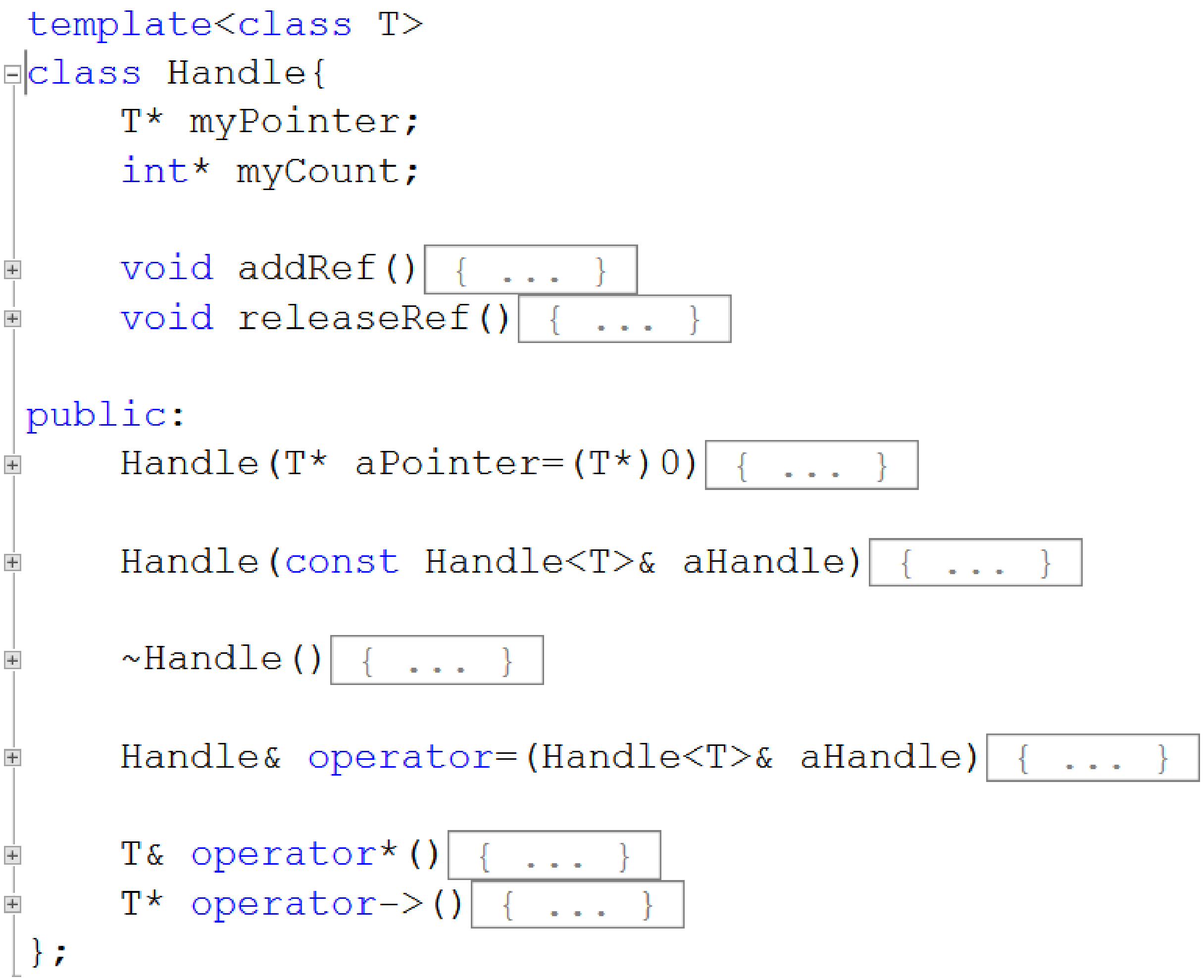
- std::unique_ptr
  - The unique pointer is a scoped pointer. When this pointer goes out of scope, it will get destroyed and call delete on the object. It cannot be copied.
- std::shared_ptr
  - The shared pointer keeps track of how many references you have to a pointer. Reference count will increase each time the pointer is shared. The object will get deleted when the reference count gets to zero.
- std::weak_ptr
  - The weak pointer will not increase the reference count. It is used only for storing a reference to an object to check if it is still valid or has expired.

# Smart Pointers in C++



URL: https://www.youtube.com/watch?v=UOB7-B2MfwA

# Smart Pointers: Handle Class

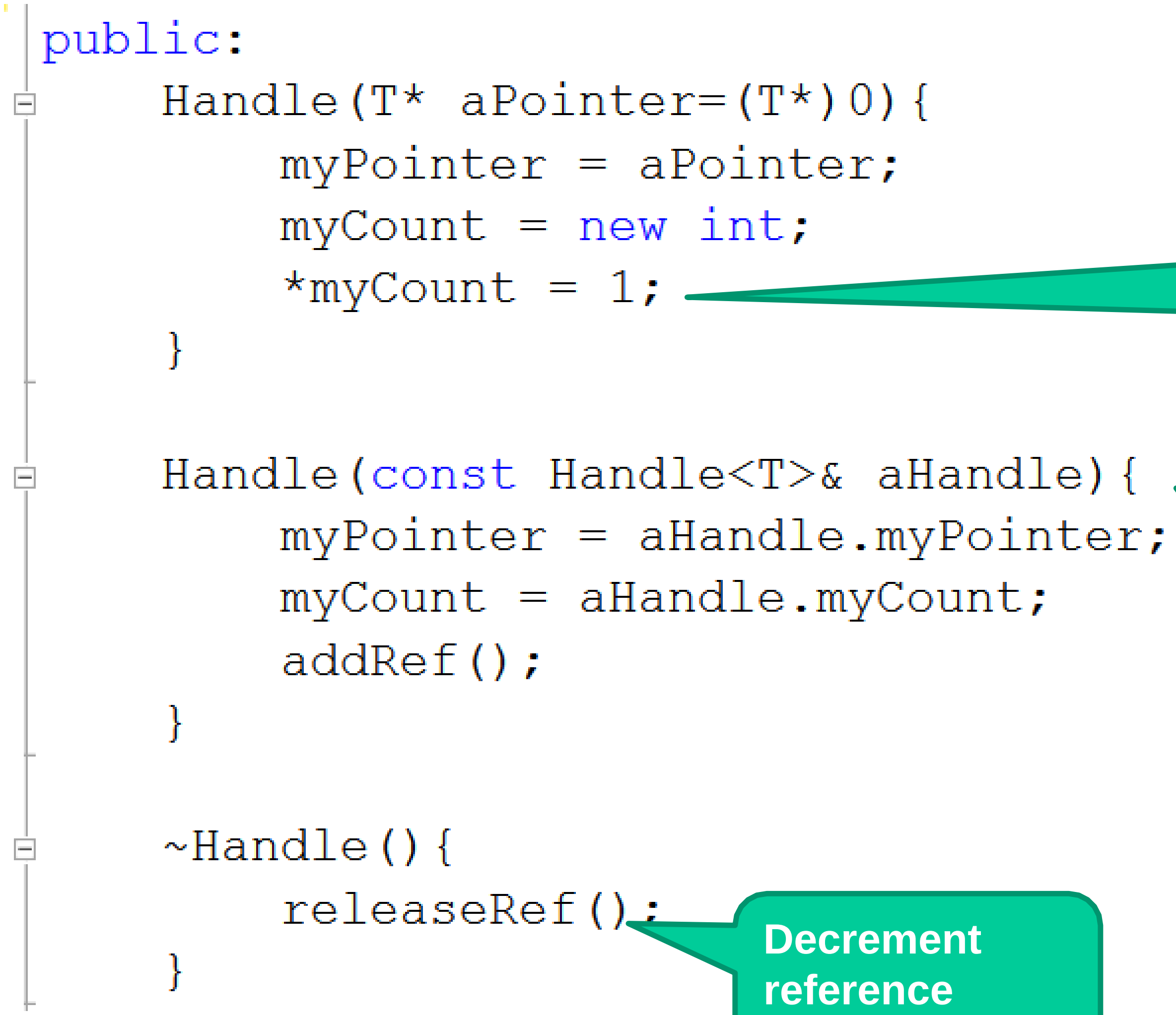


```
template<class T>
class Handle{
    T* myPointer;
    int* myCount;

    void addRef() { ... }
    void releaseRef() { ... }

public:
    Handle(T* aPointer=(T*)0) { ... }

    Handle(const Handle<T>& aHandle) { ... }

    ~Handle() { ... }

    Handle& operator=(Handle<T>& aHandle) { ... }

    T& operator*() { ... }
    T* operator->() { ... }
};
```

# 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



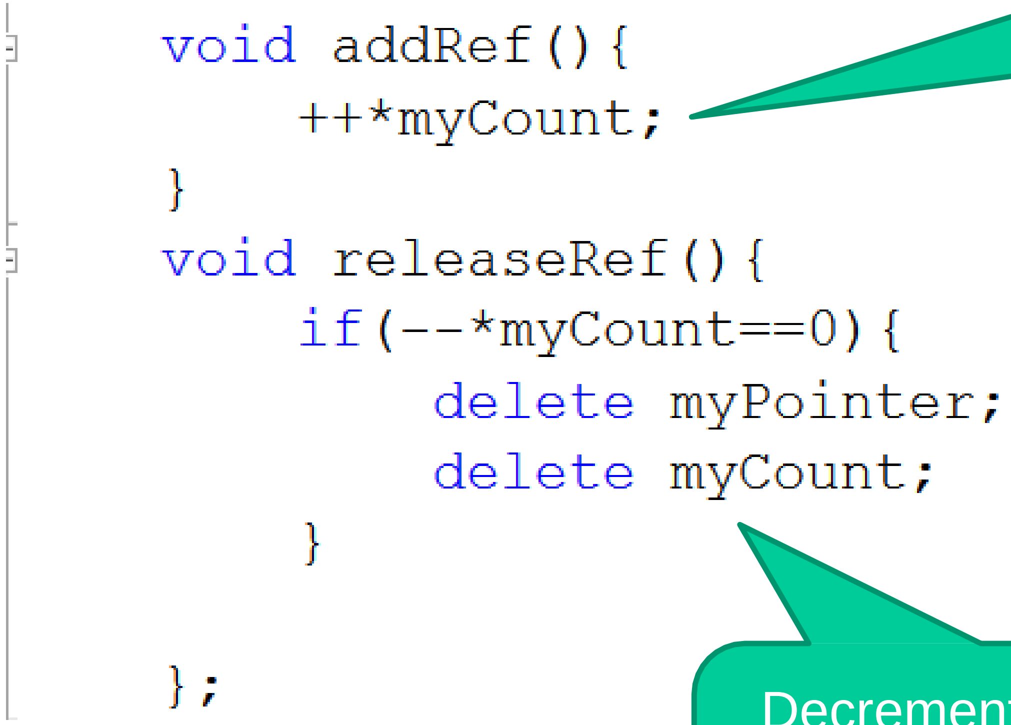
```
public:
    Handle(T* aPointer=(T*)0){
        myPointer = aPointer;
        myCount = new int;
        *myCount = 1;
    }

    Handle(const Handle<T>& aHandle){
        myPointer = aHandle.myPointer;
        myCount = aHandle.myCount;
        addRef();
    }

    ~Handle(){
        releaseRef();
    }
```

Create a shared counter

Copy constructor

Decrement reference count

# Handle: addRef & releaseRef



```
void addRef(){
    ++*myCount;
}
void releaseRef(){
    if(--*myCount==0){
        delete myPointer;
        delete myCount;
    }


};
```

Increment reference count

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

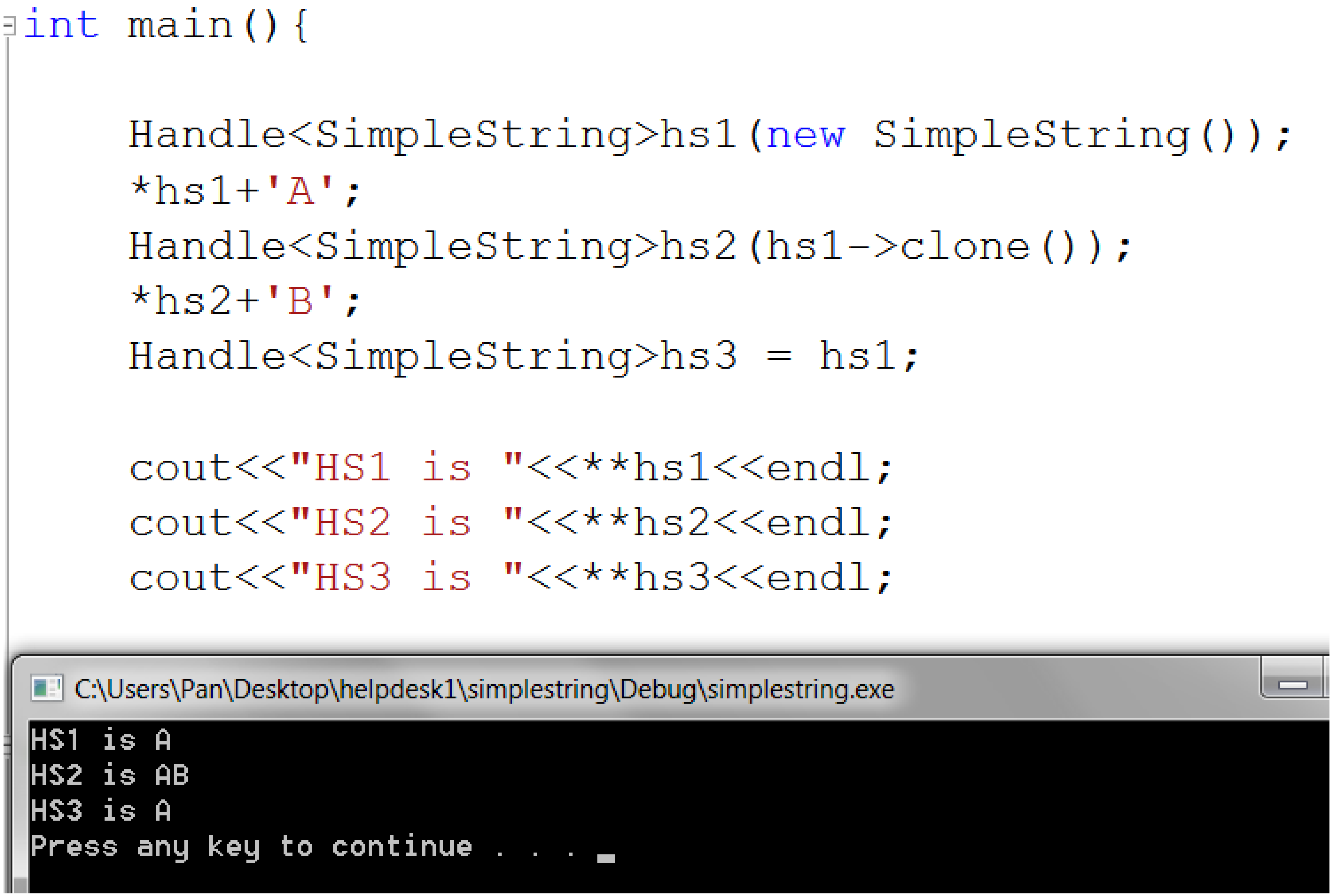
# Handle: Operators

```
Handle& operator=(Handle<T>& aHandle){
    if(&aHandle !=this){
        aHandle.addRef();
        releaseRef();
        myPointer=aHandle.myPointer;
        myCount=aHandle.myCount;
    }
    return *this;
}

T& operator*(){
    if(myPointer)return *myPointer;
    else throw std::runtime_error("Dereference of unbound");}
T* operator->(){
    if(myPointer)return myPointer;
    else throw std::runtime_error("Access through unbound");}
```

Assignment: copy control

Pointer Behaviour

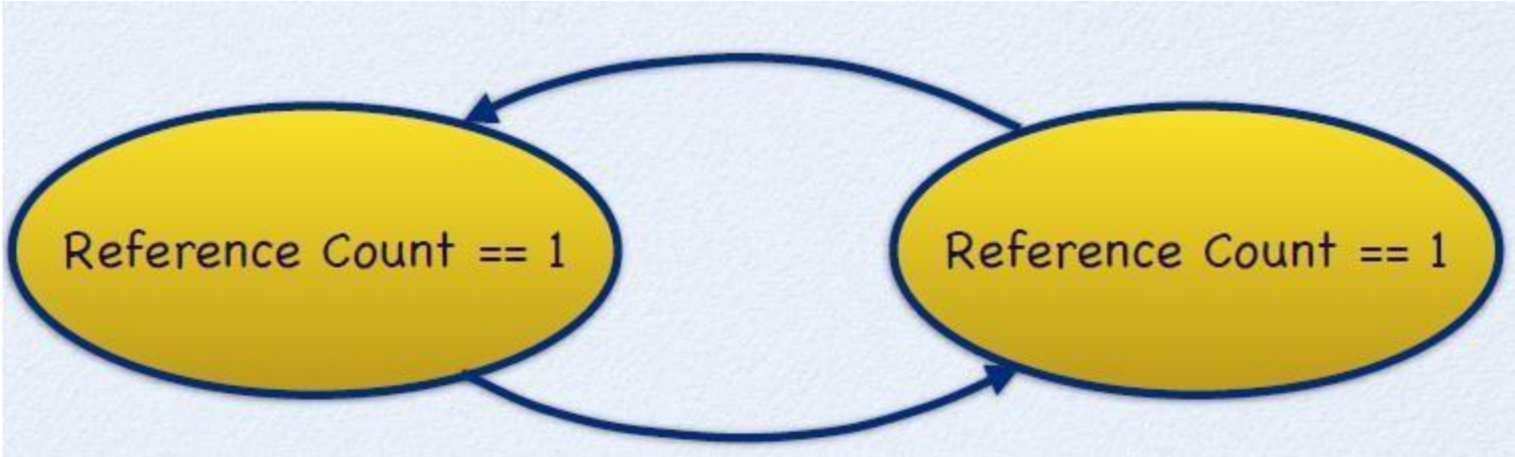
# Using Handle


C:\Users\Pan\Desktop\helpdesk1\simplestring\Debug\simplestring.exe

```
int main(){

    Handle<SimpleString>hs1(new SimpleString());
    *hs1+'A';
    Handle<SimpleString>hs2(hs1->clone());
    *hs2+'B';
    Handle<SimpleString>hs3 = hs1;

    cout<<"HS1 is "<<**hs1<<endl;
    cout<<"HS2 is "<<**hs2<<endl;
    cout<<"HS3 is "<<**hs3<<endl;
```

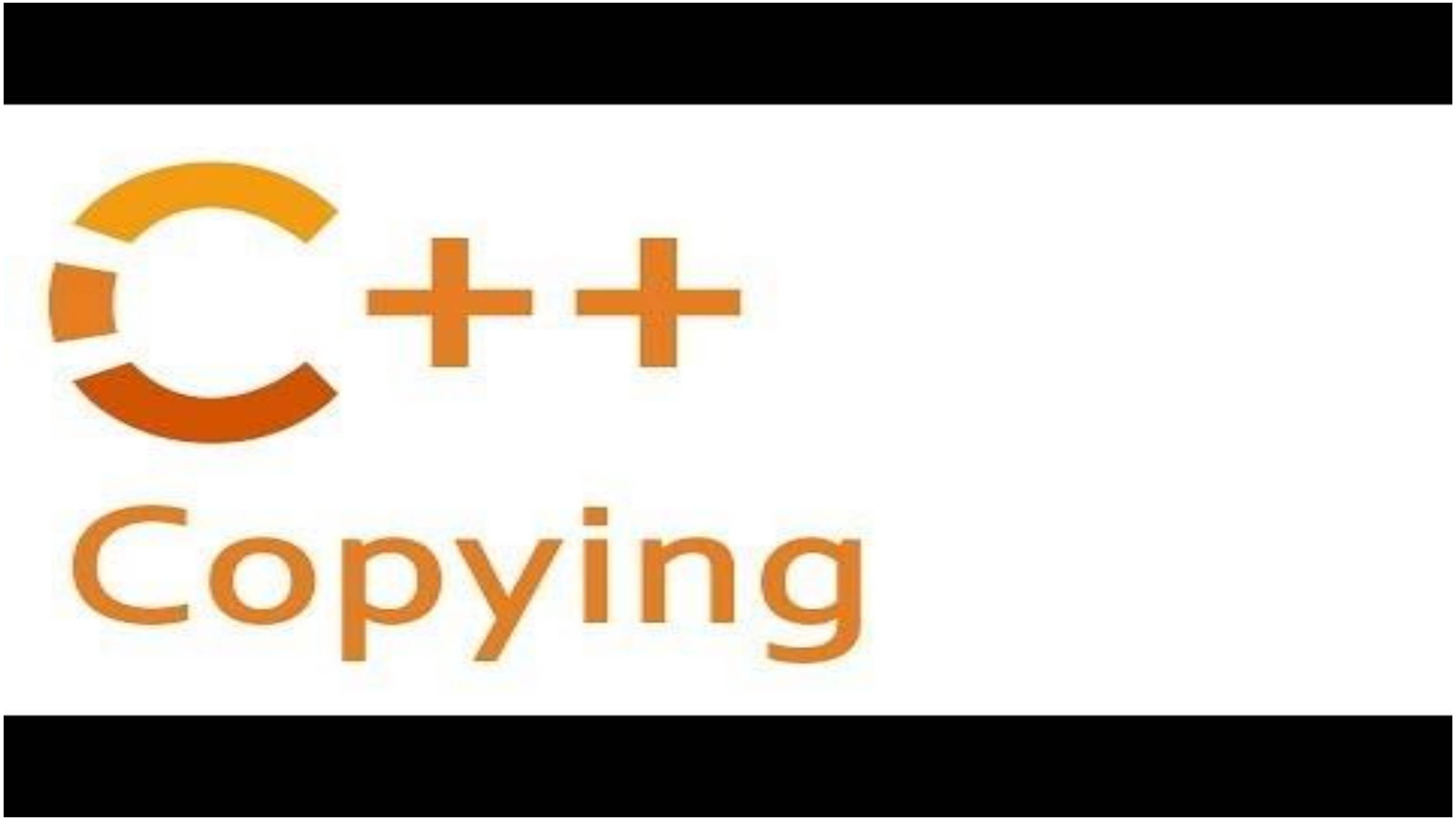
```
HS1 is A
HS2 is AB
HS3 is A
Press any key to continue . . .
```

# Reference Counting Limits



- Reference counting fails on circular data structures like double-linked lists.
- Circular data structures require extra effort to reclaim allocated memory.

# Copying and Copy Constructors



URL: https://www.youtube.com/watch?v=BvR1Pgzzr38

# End of Queue, Copy Control and Memory Management