

# COS30008

Data Structures and Patterns  
Semester 1, 2022

# COS30008

**Convener:** Dr Markus Lumpe

**Lecture:** Monday 12:30-14:30 (online)

**Labs:** Monday 10:30 (ATC325)

Monday 14:30 (BA513)

Tuesday 08:30, 10:30, 12:30, 14:30, 16:30 (BA603)

Wednesday 08:30, 10:30, 12:30, 14:30 (BA603)

**Grading:** Problems sets (4), mid-term, final exam

**Assessments:** self-guided study projects with specific deadlines

# Subject Aims

- How can a given problem be effectively expressed?
- What are suitable data representations for specifying computational processes?
- What is the impact of data and its representation with respect to time and space consumption?
- What are the reoccurring structural artifacts in software and how can we identify them in order to facilitate problem solving?

# Learning Objectives

1. Apply object oriented design and implementation techniques.
2. Interpret the tradeoffs and issues involved in the design, implementation, and application of various data structures with respect to a given problem.
3. Design, implement, and evaluate software solutions using behavioral, creational, and structural software design patterns.
4. Explain the purpose and answer questions about data structures and design patterns that illustrate strengths and weaknesses with respect to resource consumption.
5. Assess the impact of data structures on algorithms.
6. Analyze algorithm designs and perform best-, average-, and worst-case analysis.

# Overview

The following gives a tentative list of topics not necessarily in the order in which they will be covered in the subject:

- Introduction
- Sets, Arrays, Indexers, and Iterators
- Basic Data Structures and Patterns
- Abstract Data Types and Data Representation
- One-Dimensional Data Structures
- Hierarchical Data Structures
- Algorithmic Patterns and Problem Solvers

# Why?

“Smart data structures and dumb code works a lot better than the other way around.”

Eric S. Raymond: The Cathedral and the Bazaar

# A Brief Introduction to C++

## Overview

- C++ Programming Model
- Coding Conventions (used in COS30008)
- A Simple Particle Simulation: A First Example
  - Object Construction
  - Operators and Member Functions
  - Class Composition
- Object-oriented Programming in C++

## References

- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)
- Scott Meyers: Effective Modern C++. O'Reilly (2014)
- Anthony Williams: C++ Concurrency in Action - Practical Multithreading. Manning Publications Co. (2012)

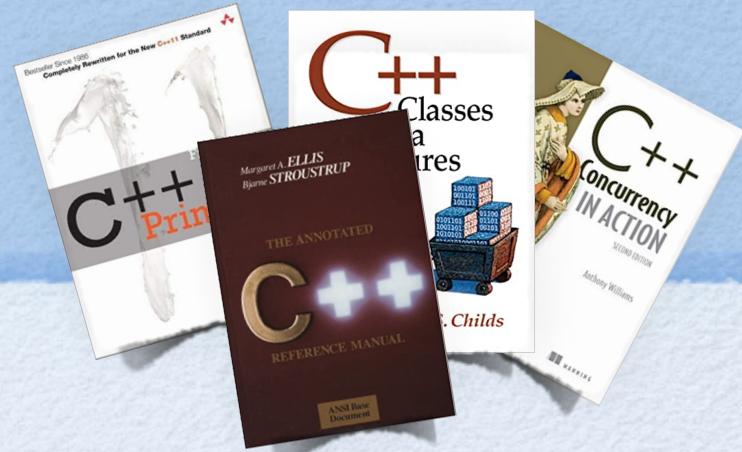
# Why C++

- We need to know more than just Java or C#.
- C++ is highly efficient and provides a better match to implement low-level software layers like device controllers or networking protocols.
- C++ is being widely used to develop commercial applications and is at the center of operating system and modern game development.
- Memory is tangible in C++ and we can, therefore, study the effects of design decisions on memory management more directly.

# Core Properties of Programming Languages

- Programming languages provide us with a framework to organize computational complexity in our own minds.
- Programming languages offer us the means by which we communicate our understanding about a computerized problem solution.

# What is C++



- C++ is a general-purpose, high-level programming language with low-level features.
- Bjarne Stroustrup developed C++ (C with Classes) in 1983 at Bell Labs as an enhancement to the C programming language.
- A first C++ programming language standard was ratified in 1998 as ISO/IEC 14882:1998. The current extending version is ISO/IEC 14882:2011 (informally known as C++11, C++14, C++17, C++20,...).
- C++11 differs greatly from C++98. C++14 differs ...
- C++11 introduces move semantics and lambda expressions (revised in C++14)

# Design Philosophy of C++

- C++ is a **hybrid, statically-typed, general-purpose** language that is as efficient and portable as C.
- C++ directly supports **multiple programming styles** like procedural programming, object-oriented programming, or generic programming.
- C++ gives the programmer choice, even if this makes it possible for the programmer to choose incorrectly!
- C++ avoids features that are platform specific or not general purpose, but is itself **platform-dependent**.
- C++ does not incur overhead for features that are not used.
- C++ functions without an integrated and sophisticated programming environment.

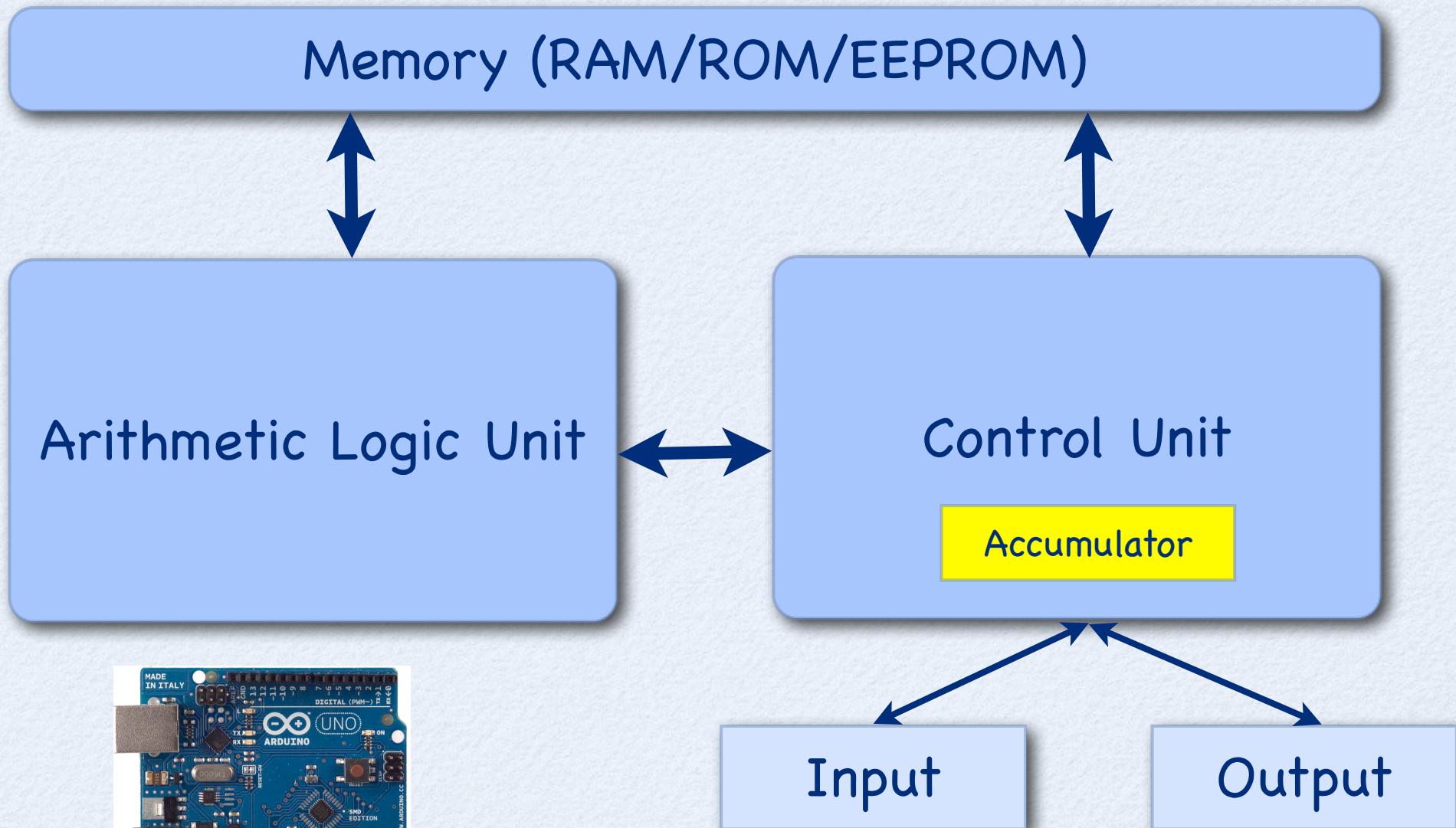
# C++ Paradigms

- C++ is a multi-paradigm language.
- C++ provides natural support for
  - the imperative paradigm and
  - the object-oriented paradigm.
- Paradigms must be mixed in any non-trivial project.

# Imperative Programming

- This is the oldest style of programming, in which the algorithm for the computation is expressed explicitly in terms of instructions such as assignments, tests, branching and so on.
- Execution of the algorithm requires data values to be held in variables which the program can access and modify.
- Imperative programming corresponds naturally to the earliest, basic and still used model for the architecture of the computer, the von Neumann model.

# The von Neumann Architecture



Source: <http://arduino.cc/en/Main/ArduinoBoardUno>

# Object-Oriented Programming

- In general, object-oriented languages are based on the concepts of **class** and **inheritance**, which may be compared to those of **type** and **variable** respectively in a language like Pascal and C.
- A class describes the characteristics common to all its instances, in a form similar to the record of Pascal (structures in C), and thus defines a set of fields.
- In object-oriented programming, instead of applying global procedures or functions to variables, we invoke the **methods** associated with the **instances** (i.e., objects), an action called "**message passing**."
- The basic concept inheritance is used to derive new classes from exiting ones by modifying or extending the inherited class(es).

# The Simplest Possible C++ Program

```
.text
    .align 1,0x90
.globl _main
_main:
LFB2:
    pushl %ebp
LCFI0:
    movl %esp, %ebp
LCFI1:
    subl $8, %esp
LCFI2:
    movl $0, %eax
    leave
    ret
LFE2:
    .globl _main.eh
_main.eh = 0
.no_dead_strip _main.eh
.constructor
.destructor
.align 1
.subsections_via_symbols
```

```
int main()
{
    return 0;
}
```

```
Kamala:COS30008 Markus$ g++ -o Simple Simple.cpp
Kamala:COS30008 Markus$ ./Simple
Kamala:COS30008 Markus$ echo $?
0
Kamala:COS30008 Markus$
```

# Let's make the program more responsive!



```
#include <iostream>
using namespace std;
int main()
{
    cout << "Enter two numbers:" << endl;
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1 << " and " << v2
        << " is " << v1 + v2 << endl;

    return 0;
}
```

Kamala:COS30008 Markus\$ g++ -o SimpleIO SimpleIO.cpp  
Kamala:COS30008 Markus\$ ./SimpleIO  
Enter two numbers:  
7  
5  
The sum of 7 and 5 is 12  
Kamala:COS30008 Markus\$

- C++ does not directly define any I/O primitives.
- I/O operations are provided by standard libraries.

# The Standard Input Stream `cin`

- `cin` is an object of class `istream` that represents the standard input stream. It corresponds to `stdin` in C.
- `cin` is a globally visible object that is readily available to any C++ compilation unit (i.e., a `.cpp`-file) that includes the library `iostream`.
- `cin` receives input either from the keyboard or a stream associated with the standard input stream.
- We use the operator `>>` to fetch formatted data or use the methods `read` or `get` to retrieve unformatted data from the standard input stream.

# The Standard Output Stream cout

- cout is an object of class ostream that represents the standard output stream. It corresponds to stdout in C.
- cout is a globally visible object that is readily available to any C++ compilation unit (i.e., a .cpp-file) that includes the library iostream.
- cout sends data either to the console (as text) or a stream associated with the standard output stream.
- We use the operator << to push formatted data or use the methods write or put to send unformatted data to the standard output stream.

# Visual Studio

2019

The screenshot shows the Visual Studio 2022 start page. The top navigation bar includes File, Edit, View, Debug, Analyze, Tools, Extensions, Window, Help, and a Search (Ctrl+Q) field. Below the navigation bar, there are several icons for quick access, followed by dropdown menus for Debug (set to x86), Attach..., Auto, and a folder icon.

# What would you like to do?

**Open recent**

As you use Visual Studio, any projects, folders, or files that you open will show up here for quick access.

You can pin anything that you open frequently so that it's always at the top of the list.

**Get started**

- Clone a repository**  
Get code from an online repository like GitHub or Azure DevOps
- Open a project or solution**  
Open a local Visual Studio project or .sln file
- Open a local folder**  
Navigate and edit code within any folder
- Create a new project**  
Choose a project template with code scaffolding to get started

SIGN IN  -   

File Edit View Debug Analyze Tools Extensions Window Help Search (Ctrl+Q) 

Debug x86 Attach... Auto  

# Create a new project

Recent project templates

Windows Desktop Wizard C++

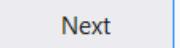
Empty Project  
Start from scratch with C++ for Windows. Provides no starting files.  
Console C++ Windows

Console App  
Run code in a Windows terminal. Prints "Hello World" by default.  
Console C++ Windows

Windows Desktop Wizard  
Create your own Windows app using a wizard.  
Console C++ Desktop Library Windows

Windows Desktop Application  
A project for an application with a graphical user interface that runs on Windows.  
C++ Desktop Windows

Shared Items Project  
A Shared Items project is used for sharing files between multiple projects.  
Android Console C++ Desktop Games iOS Library



# Configure

## Windows Desktop

Project name

SimpleIO

Location

E:\COS30008\

Solution name i

SimpleIO

Place solution and project in same folder

## Windows Desktop Project

### Application type

Console Application (.exe)

### Additional options:

Empty project

Precompiled header

Export symbols

MFC headers



Tip: You can also use the Empty Project template to create this kind of project.

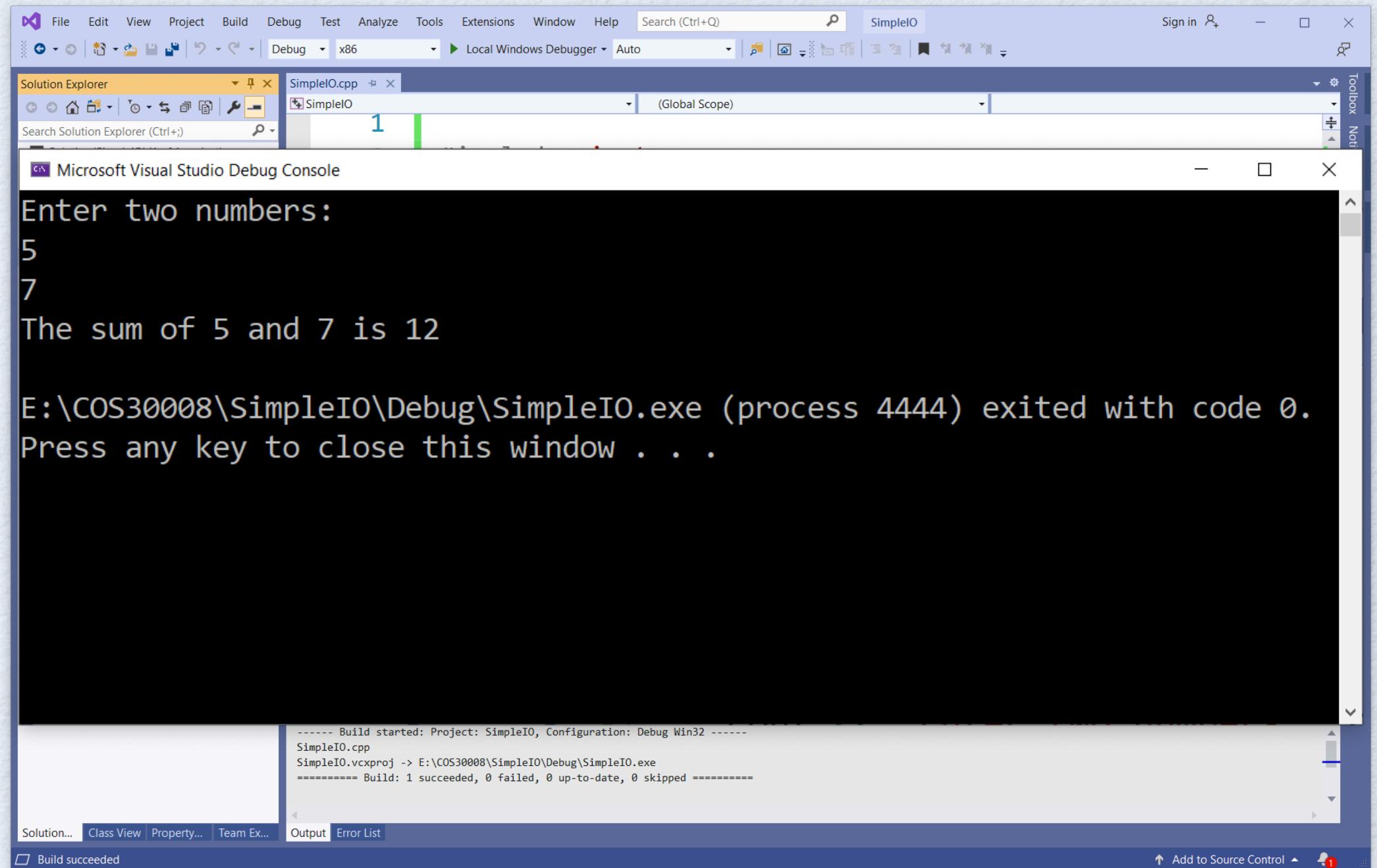
OK

Cancel

Create

The screenshot shows the Microsoft Visual Studio IDE interface. The main window displays a code editor with the file `SimpleIO.cpp` open. The code implements a simple program that prompts the user for two integers, calculates their sum, and prints the result. The code is written in C++ using standard input-output streams (`<iostream>`). The code editor features color-coded syntax highlighting, including red for strings, blue for keywords like `#include` and `int`, and green for comments. The Solution Explorer on the left shows a single project named `SimpleIO` with one source file, `SimpleIO.cpp`. The Output and Error List tabs at the bottom indicate that no issues were found in the code.

```
1 #include <iostream>
2
3 using namespace std;
4
5
6 int main()
7 {
8     cout << "Enter two numbers:" << endl;
9     int v1, v2;
10    cin >> v1 >> v2;
11    cout << "The sum of " << v1 << " and " << v2
12        << " is " << v1 + v2 << endl;
13
14    return 0;
15
16 }
```



# A Simple Particle Simulation

- Before we can write a program in a new language, we need to know some of its basic features. C++ is no exception.
- Let's start with a simple particle simulation (based on ideas presented on [www.codingmath.com](http://www.codingmath.com)).
- The simulation program requires us to
  - define variables
  - perform input and output
  - define two data structures (i.e., classes) to hold the data we are managing: Vector2D and Particle2D
  - implement basic vector math operations for Vector2D and Particle2D
  - write control code to simulate a particle object

# Technical References

- Eric Lengyel: Mathematics for 3D Game Programming and Computer Graphics, Course Technology (2012)
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)
- Kenneth H. Rosen: Discrete Mathematics and Its Application. 7th Edition. McGraw-Hill (2012)
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein: Introduction to Algorithms. 3rd Edition. The MIT Press (2009)
- Paul Orland: Math for Programmers – 3D graphics, machine learning, and simulations with Python. Manning Publications (2020)
- [www.codingmath.com](http://www.codingmath.com)

# 2D Vector Operations

Length:

$$\|\vec{v}\| = \sqrt{x^2 + y^2}$$

Dot Product:

$$\vec{v} \cdot \vec{u} = v_x u_x + v_y u_v = \|\vec{v}\| \|\vec{u}\| \cos \alpha$$

Cross Product\*:

$$\vec{v} \times \vec{u} = \det \begin{pmatrix} v_x & u_x \\ v_y & u_y \end{pmatrix} = v_x u_y - u_x v_y$$

Align Orientation:

$$\vec{v}' = \begin{bmatrix} \cos \alpha \\ \sin \alpha \end{bmatrix} \|\vec{v}\|$$

Orientation/Direction:

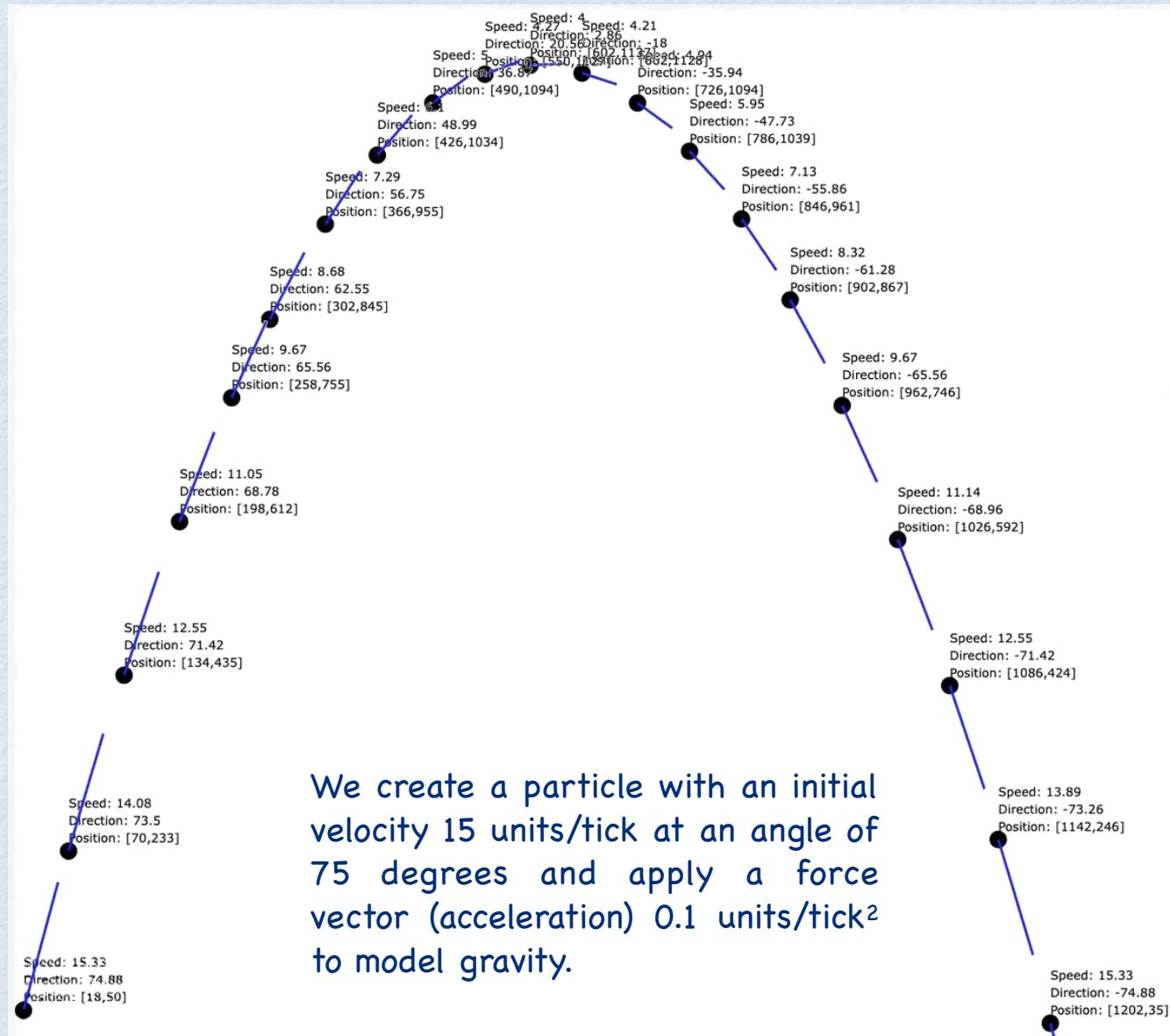
$$\theta = \tan^{-1}(v_y/v_x)$$

Normalization:

$$\vec{n}_v = \frac{\vec{v}}{\|\vec{v}\|}, \|\vec{n}_v\| = 1$$

\* The cross product is a three-dimensional concept. In 2D, we use it to determine whether consecutive line segments turn left or right.

# Visualization of the Particle Simulation



# Preliminaries

# Coding Conventions

# Coding Conventions

- Coding conventions establish guidelines for a specific programming language that recommend programming style, practices, and methods for each aspect of a program written in this language.
- Coding conventions are only applicable to the human maintainers and peer reviewers of a software project.
- Conventions may be formalized in a documented set of rules that an entire team or company follows, or may be as informal as the habitual coding practices of an individual.
- Coding conventions are not enforced by compilers.

# CamelCase

- CamelCase (or medial capitals) is a practice of writing names of variable or functions with some inner uppercase letters to denote embedded words:
  - `InputStreamReader`: character input stream
  - `getEncoding`: getter method for data encoding
  - `MyIntegerArray`: array variable
  - `CreateWindowEx`: Windows function
- Two popular variants:
  - Pascal InfixCaps - the first letter should be a capital, and any embedded words in an identifier should be in caps, as well as any acronym that is embedded.
  - Java - variables are mixed case with a lowercase first letter, methods should be verbs, in mixed case with the first letter lowercase, and classes should be nouns, in mixed case with the first letter of each internal word capitalized.

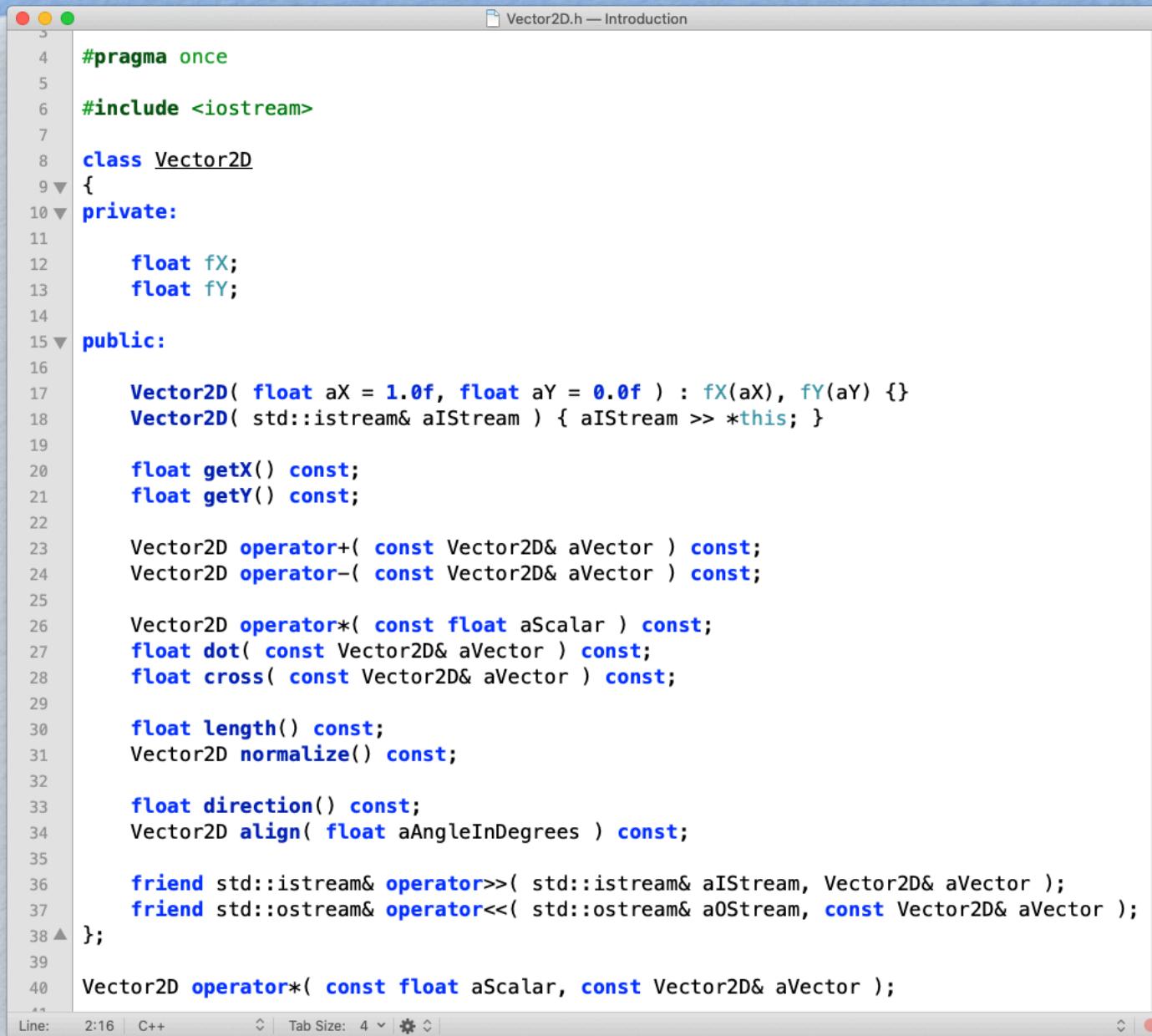
# Borland-Inspired Style Guide Elements

- Field names should start with the letter 'f'.
- Local variable names should start with the letter 'l'.
- Parameter names should start with the letter 'a'.
- Global variable names should start with the letter 'g'.

# Which Coding Standard To Use

- Coding conventions are not enforced by compilers.
- Not following some or all of the rules has no impact on the executable programs created from the source code.
- Code standards facilitate program comprehension and make program maintenance easier.
- Every organization may enforce some sort of coding standards as part of organization's quality assurance process.

# Vector2D: A Basic 2D Vector Class



The screenshot shows a code editor window titled "Vector2D.h — Introduction". The code is written in C++ and defines a class named Vector2D. The class has private members fX and fY, and public methods for construction, reading from an istream, getting coordinates, performing arithmetic operations, calculating length and direction, normalizing, aligning, and writing to an ostream.

```
#pragma once
#include <iostream>
class Vector2D
{
private:
    float fX;
    float fY;
public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
    Vector2D( std::istream& aInputStream ) { aInputStream >> *this; }

    float getX() const;
    float getY() const;

    Vector2D operator+( const Vector2D& aVector ) const;
    Vector2D operator-( const Vector2D& aVector ) const;

    Vector2D operator*( const float aScalar ) const;
    float dot( const Vector2D& aVector ) const;
    float cross( const Vector2D& aVector ) const;

    float length() const;
    Vector2D normalize() const;

    float direction() const;
    Vector2D align( float aAngleInDegrees ) const;

    friend std::istream& operator>>( std::istream& aInputStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOutputStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

# The Structure of a Class

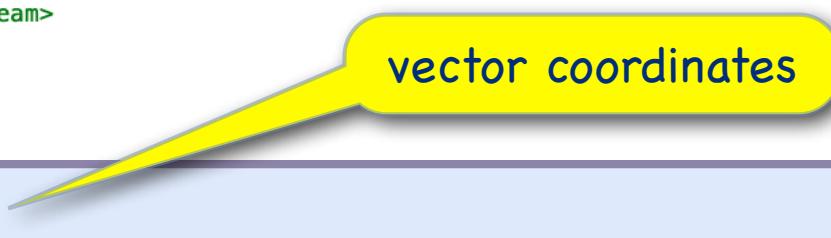
```
class X
{
    private:
        // private members
    protected:
        // protected members
    public:
        // public members
};
```

Don't forget the semicolon!

# Access Modifiers

- **public:**
  - Public members can be accessed anywhere, including outside of the class itself.
- **protected:**
  - Protected members can be accessed within the class in which they are declared and within derived classes.
- **private:**
  - Private members can be accessed only within the class in which they are declared.

# Vector2D: Private Members



```
#pragma once
#include <iostream>
class Vector2D
{
private:
    float fX;
    float fY;

public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
    Vector2D( std::istream& aInputStream ) { aInputStream >> *this; }

    float getX() const;
    float getY() const;

    Vector2D operator+( const Vector2D& aVector ) const;
    Vector2D operator-( const Vector2D& aVector ) const;

    Vector2D operator*( const float aScalar ) const;
    float dot( const Vector2D& aVector ) const;
    float cross( const Vector2D& aVector ) const;

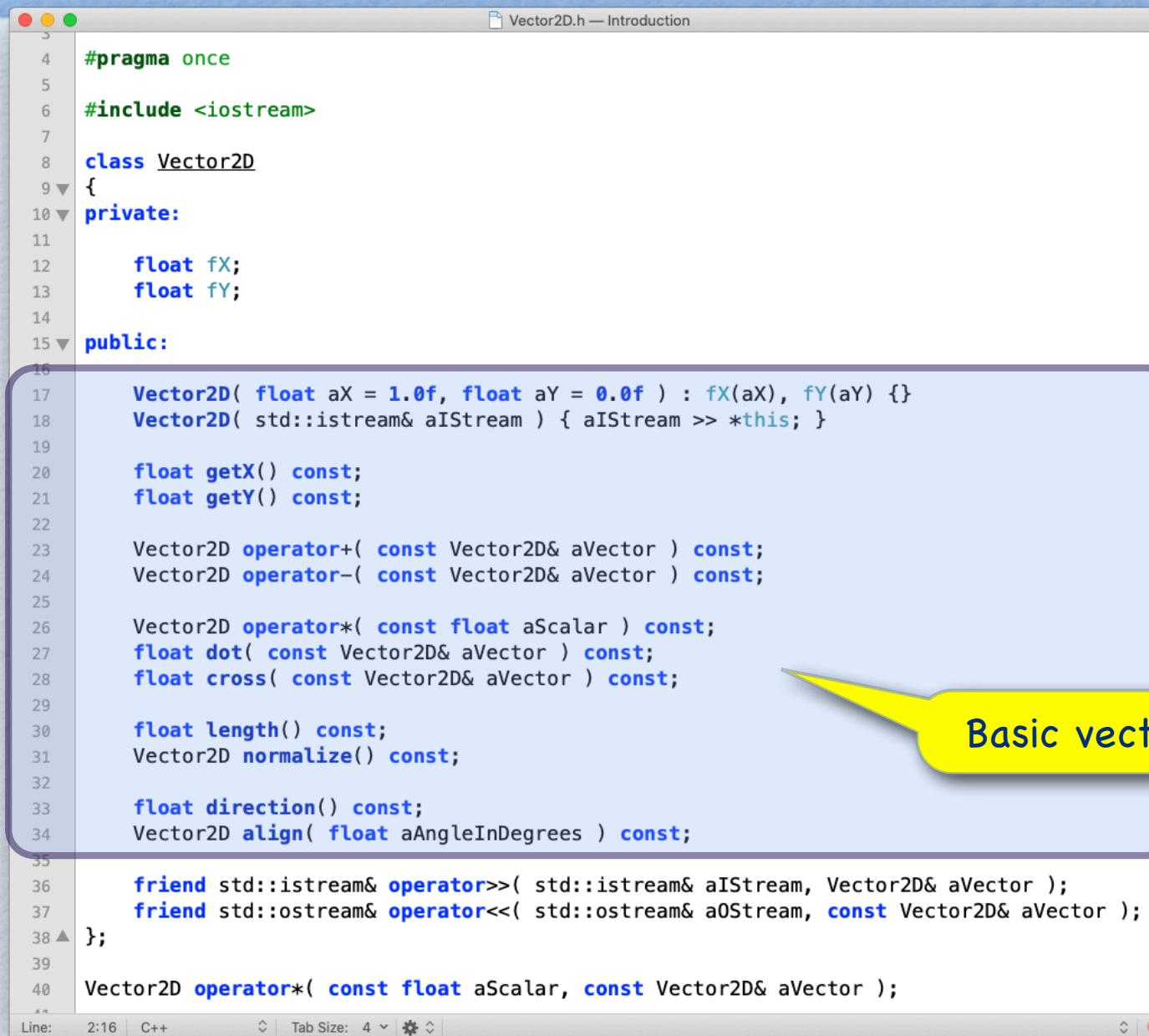
    float length() const;
    Vector2D normalize() const;

    float direction() const;
    Vector2D align( float aAngleInDegrees ) const;

    friend std::istream& operator>>( std::istream& aInputStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOutputStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

# Vector2D: Public Members



```
#pragma once
#include <iostream>
class Vector2D
{
private:
    float fX;
    float fY;
public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
    Vector2D( std::istream& aInputStream ) { aInputStream >> *this; }

    float getX() const;
    float getY() const;

    Vector2D operator+( const Vector2D& aVector ) const;
    Vector2D operator-( const Vector2D& aVector ) const;

    Vector2D operator*( const float aScalar ) const;
    float dot( const Vector2D& aVector ) const;
    float cross( const Vector2D& aVector ) const;

    float length() const;
    Vector2D normalize() const;

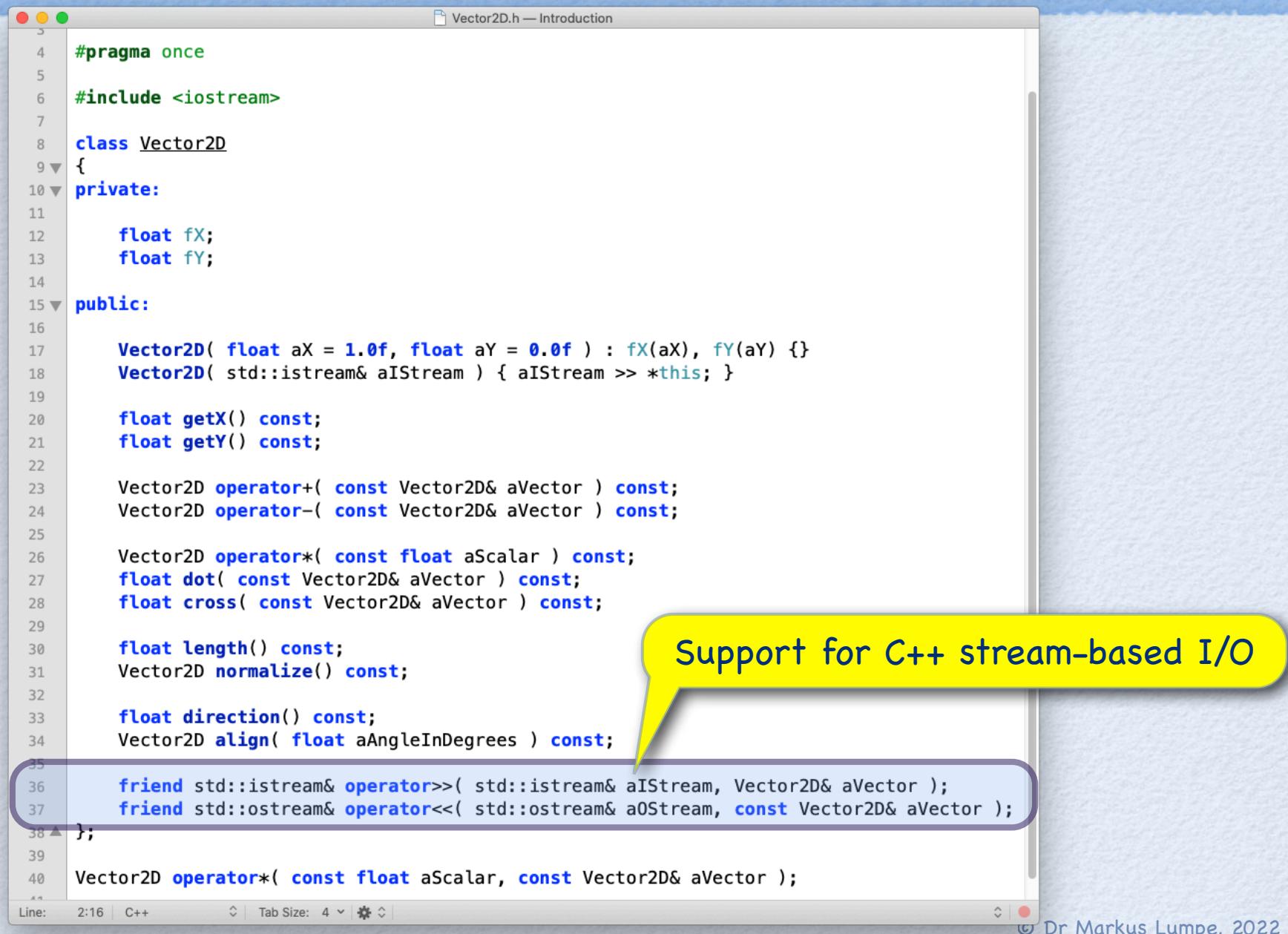
    float direction() const;
    Vector2D align( float aAngleInDegrees ) const;

    friend std::istream& operator>>( std::istream& aInputStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOutputStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

Basic vector operations

# Vector2D: Friends



```
#pragma once
#include <iostream>
class Vector2D
{
private:
    float fX;
    float fY;
public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
    Vector2D( std::istream& aInputStream ) { aInputStream >> *this; }

    float getX() const;
    float getY() const;

    Vector2D operator+( const Vector2D& aVector ) const;
    Vector2D operator-( const Vector2D& aVector ) const;

    Vector2D operator*( const float aScalar ) const;
    float dot( const Vector2D& aVector ) const;
    float cross( const Vector2D& aVector ) const;

    float length() const;
    Vector2D normalize() const;

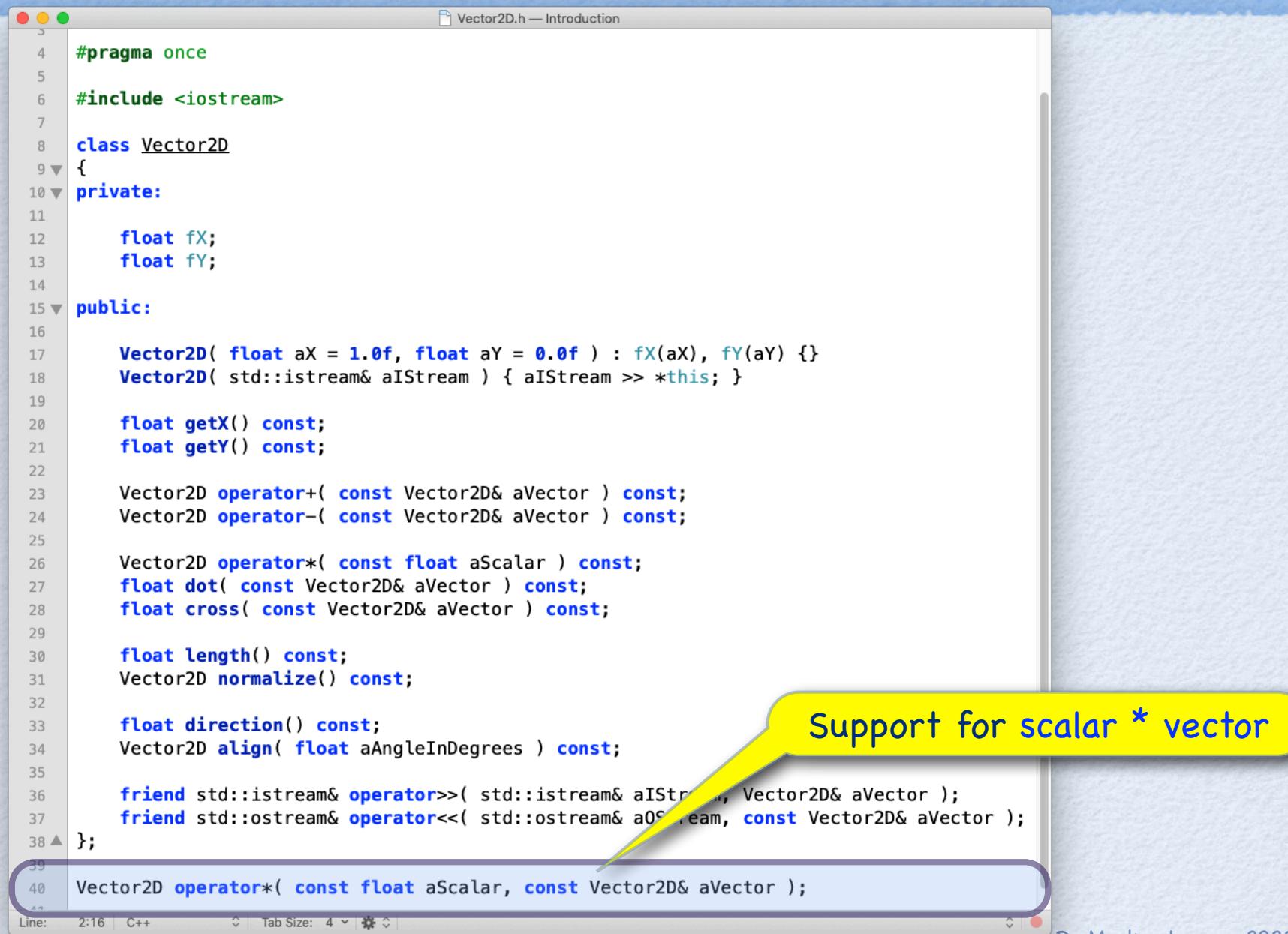
    float direction() const;
    Vector2D align( float aAngleInDegrees ) const;

    friend std::istream& operator>>( std::istream& aInputStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOutputStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

Support for C++ stream-based I/O

# Vector2D: Ad hoc Definitions



```
#pragma once
#include <iostream>
class Vector2D
{
private:
    float fX;
    float fY;
public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
    Vector2D( std::istream& aInputStream ) { aInputStream >> *this; }

    float getX() const;
    float getY() const;

    Vector2D operator+( const Vector2D& aVector ) const;
    Vector2D operator-( const Vector2D& aVector ) const;

    Vector2D operator*( const float aScalar ) const;
    float dot( const Vector2D& aVector ) const;
    float cross( const Vector2D& aVector ) const;

    float length() const;
    Vector2D normalize() const;

    float direction() const;
    Vector2D align( float aAngleInDegrees ) const;

    friend std::istream& operator>>( std::istream& aInputStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOutputStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

Support for scalar \* vector

# const

# Constants and Const Qualifier

- What are the problems with

```
for ( int index = 0; index < 128; index++ ) { ... }
```

What is 128?

- We can do better

```
for ( int index = 0; index < BufferSize; index++ ) { ... }
```

Is it safe?

- Defining a const object:

```
const int BufferSize = 128; // initialized at compile time
```

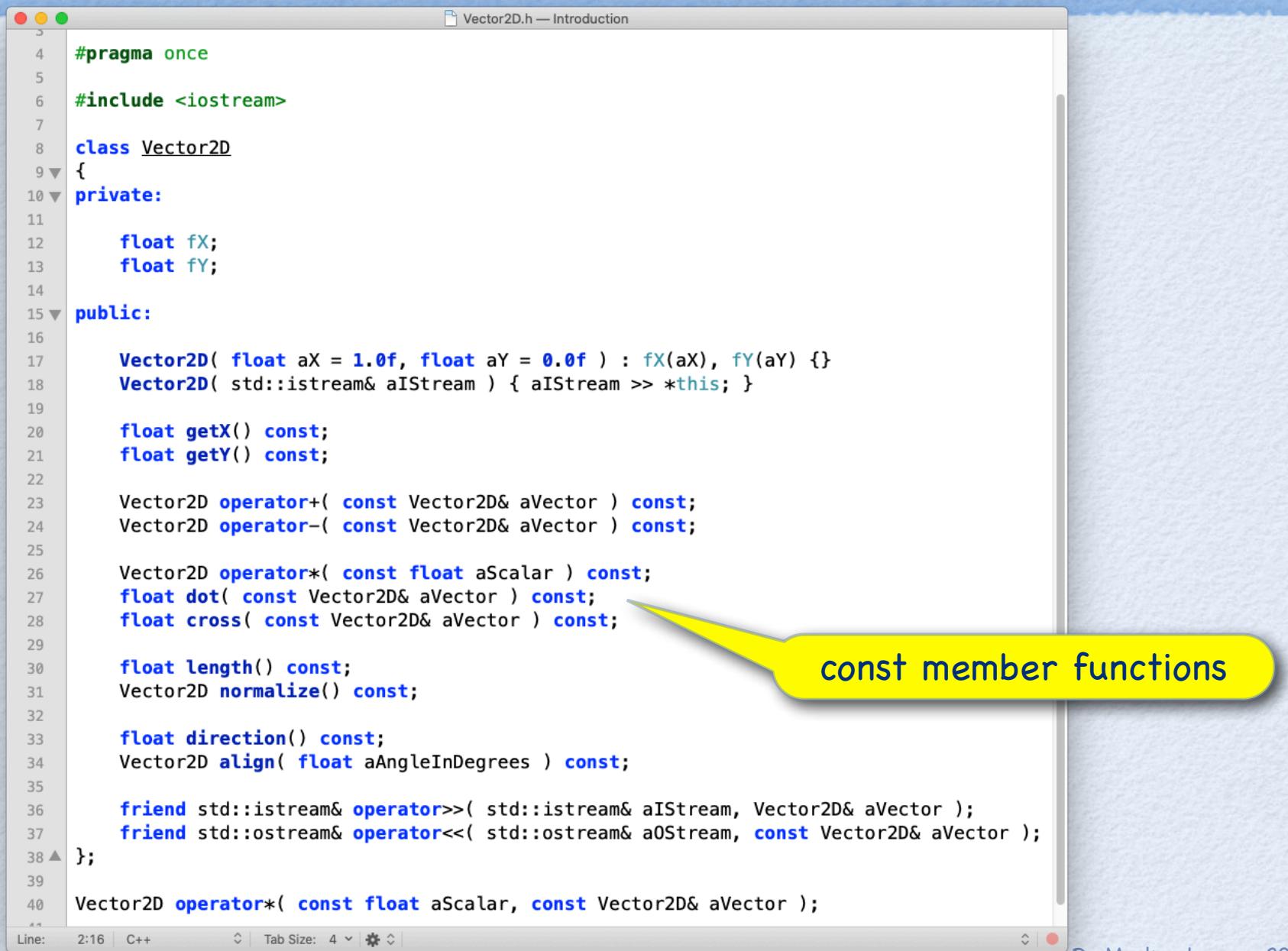
- Unlike macro definitions, const objects have an address!

```
#define BUF_SIZE 128 // macro definition
```

```
const int BufferSize = BUF_SIZE; // initialized at compile time
```

Everything that should or  
must not change is marked  
with the **const** keyword.

# Initialized Vector2D objects are read-only.



```
#pragma once
#include <iostream>
class Vector2D
{
private:
    float fX;
    float fY;
public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
    Vector2D( std::istream& aInputStream ) { aInputStream >> *this; }

    float getX() const;
    float getY() const;

    Vector2D operator+( const Vector2D& aVector ) const;
    Vector2D operator-( const Vector2D& aVector ) const;

    Vector2D operator*( const float aScalar ) const;
    float dot( const Vector2D& aVector ) const;
    float cross( const Vector2D& aVector ) const;

    float length() const;
    Vector2D normalize() const;

    float direction() const;
    Vector2D align( float aAngleInDegrees ) const;

    friend std::istream& operator>>( std::istream& aInputStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOutputStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

const member functions

# References (C++-98)

- A reference introduces a new name (alias) for an existing object:

```
int BlockSize = 512;
```

```
int& BufferSize = BlockSize;
```



BufferSize is an  
alias for BlockSize

# Constant References (C++-98)

- A constant reference yields a new name for a constant object:

```
const int FixedBlockSize = 512;
```

```
const int& FixedBufferSize = FixedBlockSize;
```

- A constant reference defines an immutable alias to an existing object.

# References prevent copies from being made.

# Reference Parameters (C++-98)

- C++ uses call-by-value as default parameter passing mechanism.

```
void Assign( int aPar, int aVal ) { aPar = aVal; }

Assign( val, 3 );           // val unchanged
```

- A reference parameter yields call-by-reference:

```
void AssignR( int& aPar, int aVal ) { aPar = aVal; }

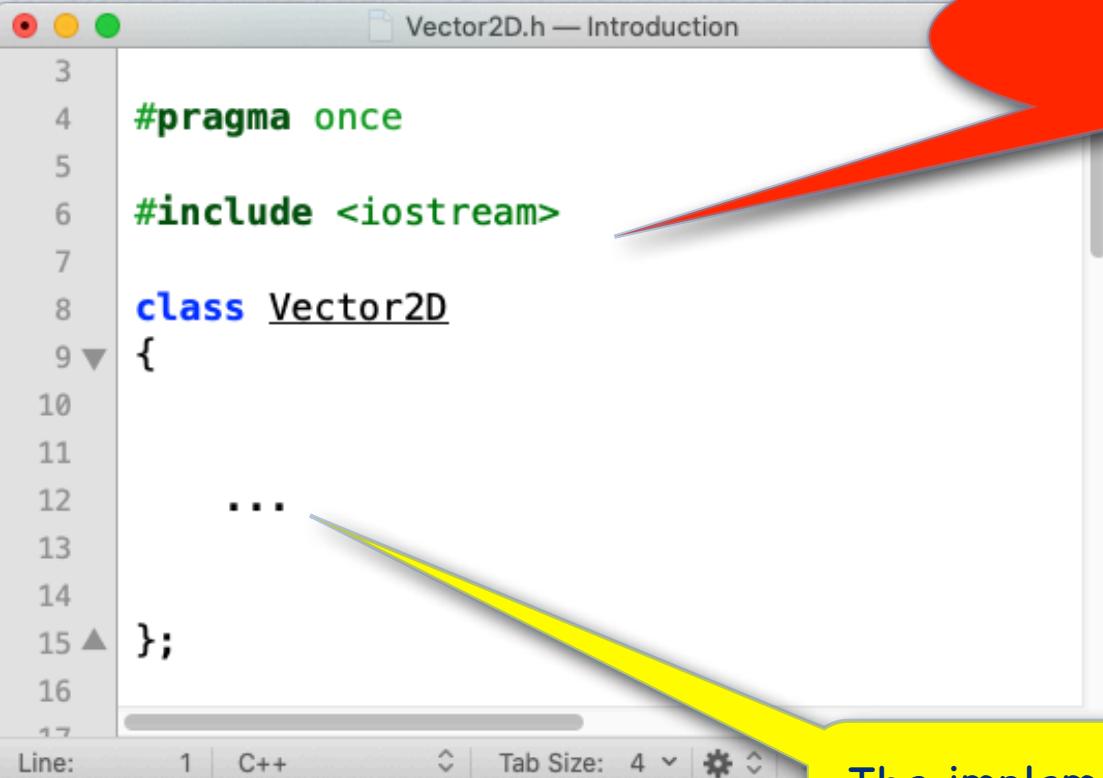
AssignR( val, 3 );          // val is set to 3
```

- A const reference parameter yields call-by-reference, but the value of the parameter is read-only:

```
void AssignCR( const int& aPar, int aVal ) { aPar = aVal; } // error
```

# Class Implementation

# Include File: Vector2D.h



```
3
4 #pragma once
5
6 #include <iostream>
7
8 class Vector2D
9 {
10
11
12     ...
13
14 };
15
16
17
```

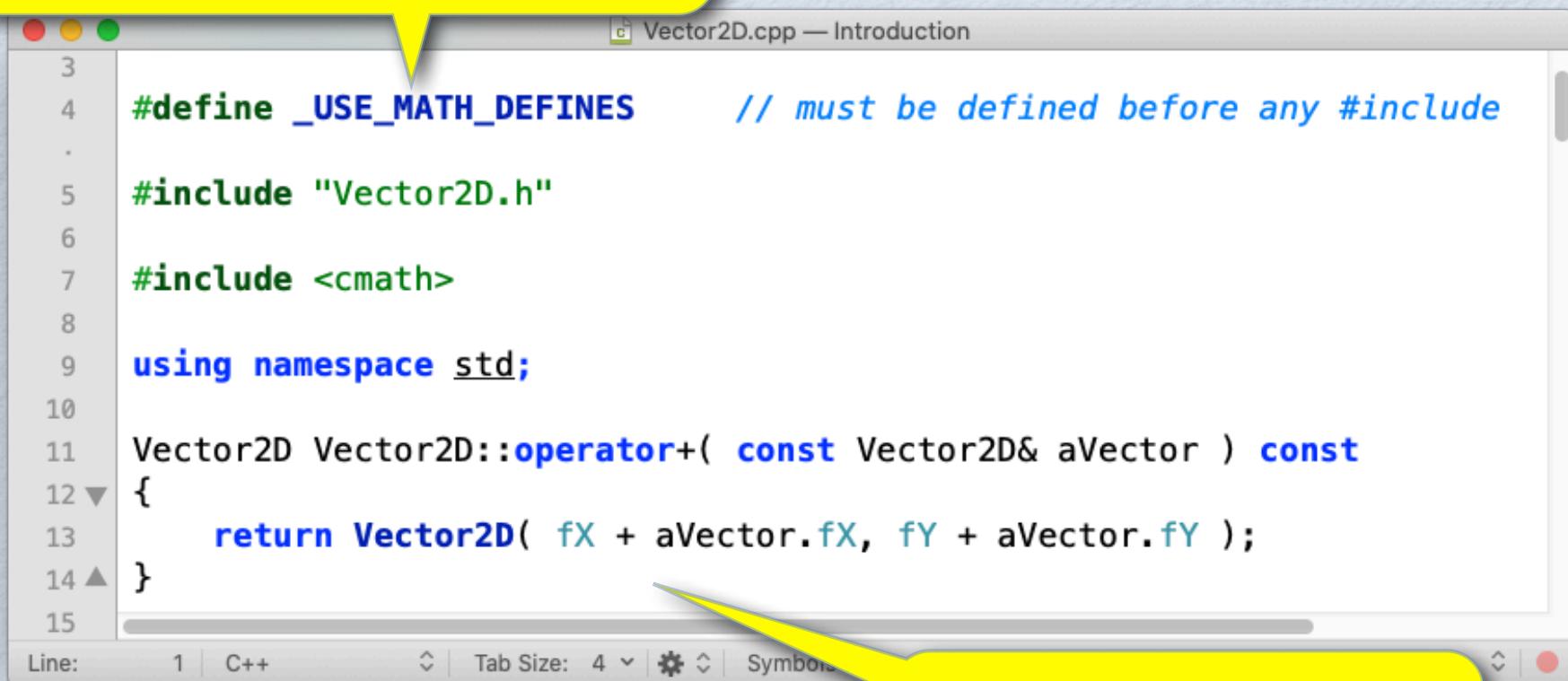
The code editor window shows the beginning of a C++ class definition. The file is titled "Vector2D.h — Introduction". The code includes a `#pragma once` directive, an `#include <iostream>` directive, and a class definition starting with `class Vector2D`. The implementation part of the class, indicated by the ellipsis (...), is shown in a yellow speech bubble.

We do not select any namespace yet!

The implementation goes to Vector2D.cpp

# Implementation File: Vector2D.cpp

Macro definition to make math definitions available (e.g., M\_PI)



```
3
4 #define _USE_MATH_DEFINES      // must be defined before any #include
.
5 #include "Vector2D.h"
6
7 #include <cmath>
8
9 using namespace std;
10
11 Vector2D Vector2D::operator+( const Vector2D& aVector ) const
12 {
13     return Vector2D( fX + aVector.fX, fY + aVector.fY );
14 }
15
```

Line: 1 | C++ | Tab Size: 4 | Symbols

Implementation

# Member Implementation

- When implementing a member function of a class in C++ we must explicitly specify the class name using a scope identifier within the signature of the member function.
- A scope identifier is a name followed by two colons (e.g. `Vector2D::`).

The screenshot shows a code editor window with the following code:

```
11
12 float Vector2D::getX() const
13 {
14     return fx;
15 }
16
17 float Vector2D::getY() const
18 {
19     return fy;
20 }
```

A yellow callout bubble points to the scope identifier `Vector2D::` in the first line of code, with the text "Scope identifier `Vector2D::`". Another yellow callout bubble points to the two `get` functions, with the text "Getters for `Vector2D`".

# C++ Code Organization

- Classes are **defined** in include files (i.e., .h).
- Class members are **implemented** in source files (i.e., .cpp).
- There are **exceptions** when working with templates.

# Standard Boilerplate Code

Guard  
against  
repeated  
inclusion

```
#ifndef HEADER_H_
#define HEADER_H_

/* Body of Header */

#endif /* HEADER_H_ */
```

# #pragma once (Visual Studio)

```
#pragma once
```

Guard against  
repeated inclusion

```
/* Body of Header */
```

# Object Initialization

- A **class** defines a new **data type**. Instances of this data type are **objects** that need initialization.
- Each class defines explicitly (or implicitly) some special member functions, called **constructors**, that are executed whenever we create new objects of a class.
- The job of a constructor is to ensure that the data members of each objects are set to some sensible initial values.

# Constructors

- Constructors may be overloaded.
- The concrete constructor arguments determine which constructor to use.
- Constructors are executed automatically whenever a new object is created.

# Default Constructor

- A default constructor is one that does not take any arguments.
- The compiler will synthesize a default constructor, when no other constructors have been specified.
- There are situations when the compiler needs to create objects in an environment agnostic way (e.g., array of objects). In this case the compiler relies on the existence of the default constructor.
- If some data members have built-in or compound types, then the class should not rely on the synthesized default constructor!

# Class Vector2D - Constructors

```
#pragma once
#include <iostream>
class Vector2D
{
private:
    float fX;
    float fY;

public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
    Vector2D( std::istream& aInputStream ) { aInputStream >> *this; }

    float getX() const;
    float getY() const;

    Vector2D operator+( const Vector2D& aVector ) const;
    Vector2D operator-( const Vector2D& aVector ) const;

    Vector2D operator*( const float aScalar ) const;
    float dot( const Vector2D& aVector ) const;
    float cross( const Vector2D& aVector ) const;

    float length() const;
    Vector2D normalize() const;

    float direction() const;
    Vector2D align( float aAngleInDegrees ) const;

    friend std::istream& operator>>( std::istream& aInputStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOutputStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

Line: 2:16 | C++ | Tab Size: 4 |

Default argument

Note, we use inlined trivial constructors here.

# Constructor Initializer

- A constructor initializer is a comma-separated list of member initializers, which is declared between the signature of the constructor and its body.
- Constructor initializers take the form of function calls where the name of the function coincides with name of the instance variable being initialized.

```
Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
```

# Implicit Class-Type Conversions

- A constructor that can be called with a single argument defines an implicit conversion from the parameter type to the class type.

```
Vector2D( std::istream& aIStream ) { aIStream >> *this; }
```

Conversion from std::istream to Vector2D

- Accordingly, we can use an `istream` where an object of type `Vector2D` is expected:

```
Vector2D( std::istream& aIStream ) { aIStream >> *this; }
```

```
Vector2D operator+( const Vector2D& aVector ) const;
```

```
Vector2D result = vec + cin;
```

`std::istream` → `Vector2D`

# Suppressing Implicit Conversions

- When a constructor is declared **explicit**, the compiler will **not** use it as a conversion operator.

```
explicit Vector2D( std::istream& aIStream ) { aIStream >> *this; }
```

- Accordingly, we **cannot** use an **istream** where an object of type **Vector2D** is expected:

```
explicit Vector2D( std::istream& aIStream ) { aIStream >> *this; }
```

```
Vector2D operator+( const Vector2D& aVector ) const;
```



```
Vector2D result = vec + cin;
```

Error

**It is application-dependent  
whether implicit conversion  
needs to be supported.**

# Rule of Thumb

- Keep C++ headers free of implementations, unless you want the implementations to be inlined or they are templates.

# Basic 2D vector operations

# Operator Overloading

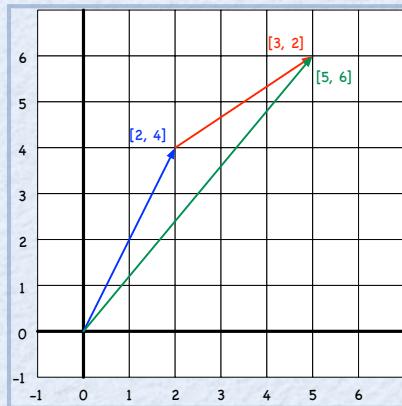
- C++ supports operator overloading.
- Overloaded operators are like normal functions, but are defined using a pre-defined operator symbol.
- You cannot change the priority and associativity of an operator.
- Operators are selected by the compiler based on the static types of the specified operands.

# Member Operators vs Ad hoc Operators

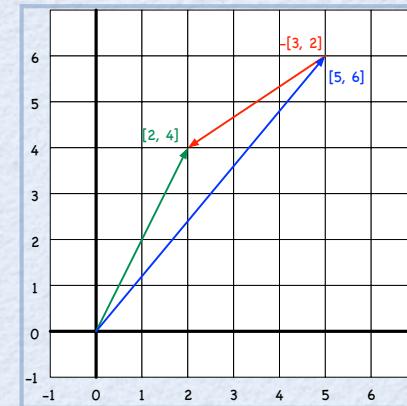
- There are two forms of operator overloading:
  - member operator
  - ad hoc operator
- A member operator receives a first implicit argument - this object. In other words, an overloaded operator in C++ is like any other non-static method function defined for a class. Non-static member functions receive as first argument "this" object. As a consequence, we only specify the second argument for binary operators like '+' or '-'.
- Ad hoc operators are not members of a class. Their signature has to match the signature of the operator to be defined. That is, an ad hoc definition of a binary operator '+' requires two arguments: the left-hand side of '+' and the right-hand side of '+'.

# Addition & Subtraction

$$[5,6] = [2,4] + [3,2]$$



$$[2,4] = [5,6] - [3,2]$$



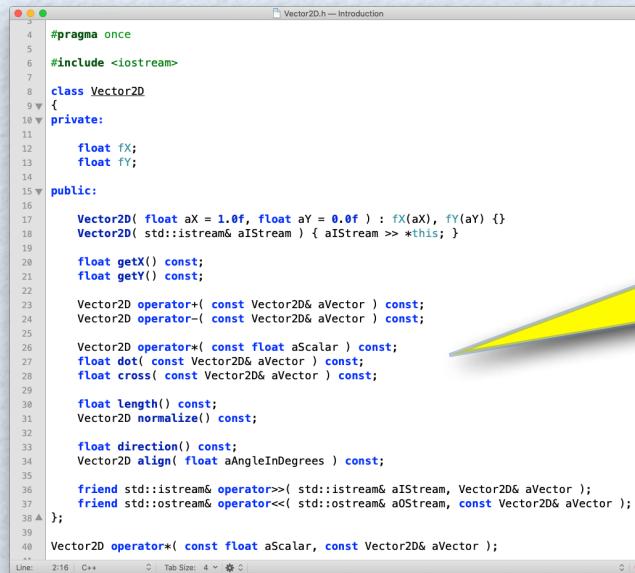
```
Vector2D Vector2D::operator+( const Vector2D& aVector ) const
{
    return Vector2D( fX + aVector.fX, fY + aVector.fY );
}

Vector2D Vector2D::operator-( const Vector2D& aVector ) const
{
    return Vector2D( fX - aVector.fX, fY - aVector.fY );
}
```

Member operators

# Closures – Lexical Scoped Name Binding

- Classes provide a lexical scoped name binding for methods. This allows methods of a class to access instance variables and other methods of the class that are defined outside the scope of a given method.
- Operationally, classes create closures that are records storing a function (or a set of functions) together with an environment. The environment associates each free variable of the function(s) to values to which the name was bound when the closure was created.



```
#pragma once
#include <iostream>
class Vector2D
{
private:
    float fx;
    float fy;
public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) : fx(aX), fy(aY) {}
    Vector2D( std::istream& aInputStream ) { aInputStream >> *this; }

    float getX() const;
    float getY() const;

    Vector2D operator+( const Vector2D& aVector ) const;
    Vector2D operator-( const Vector2D& aVector ) const;

    Vector2D operator*( const float aScalar ) const;
    float dot( const Vector2D& aVector ) const;
    float cross( const Vector2D& aVector ) const;

    float length() const;
    Vector2D normalize() const;

    float direction() const;
    Vector2D align( float aAngleInDegrees ) const;

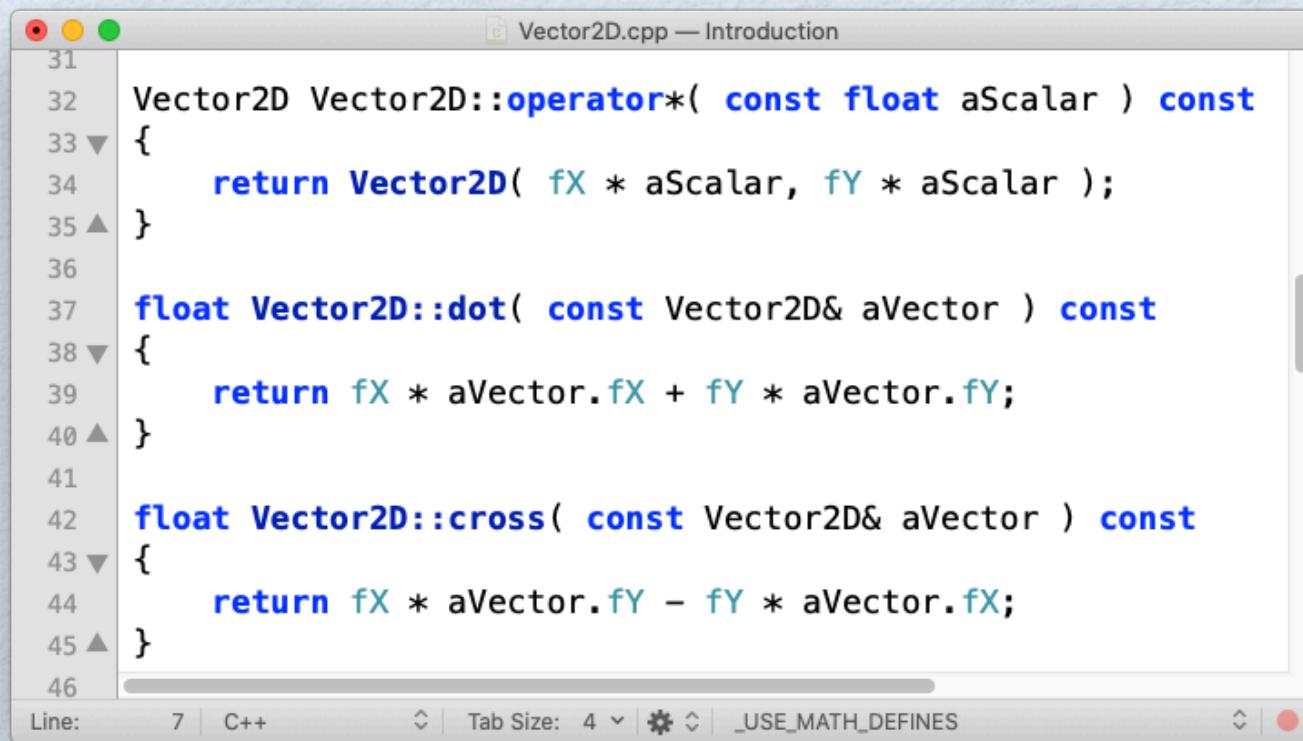
    friend std::istream& operator>>( std::istream& aInputStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOutputStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

A Vector2D closure

# Scalar Multiplication, Dot Product, and Cross Product

- We use scalar multiplication to scale a vector uniformly.
- The dot product (inner product) is a measure of the difference between the directions in which the two vectors point.
- The 2D cross product yields a scalar that we use it to determine whether consecutive line segments turn left or right.



The screenshot shows a code editor window titled "Vector2D.cpp — Introduction". The code defines three member functions for the `Vector2D` class:

```
31
32     Vector2D Vector2D::operator*( const float aScalar ) const
33 {
34     return Vector2D( fX * aScalar, fY * aScalar );
35 }
36
37     float Vector2D::dot( const Vector2D& aVector ) const
38 {
39     return fX * aVector.fX + fY * aVector.fY;
40 }
41
42     float Vector2D::cross( const Vector2D& aVector ) const
43 {
44     return fX * aVector.fY - fY * aVector.fX;
45 }
```

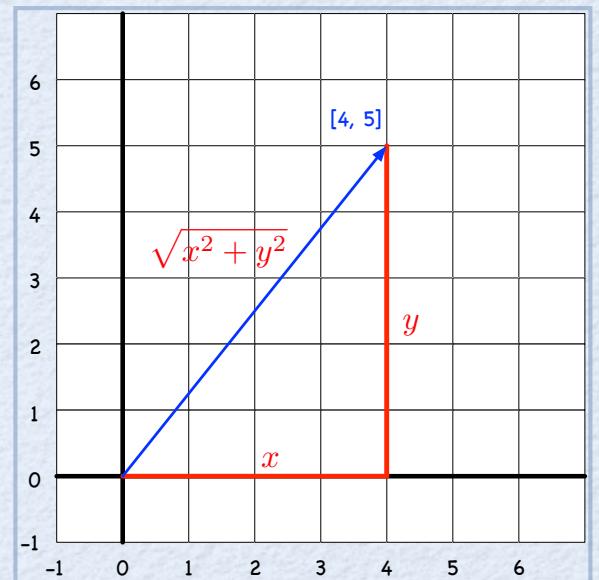
The code uses standard C++ syntax with `const` qualifiers and floating-point arithmetic. The editor interface includes line numbers, code folding arrows, and a status bar at the bottom.

# Vector Length and Unit Vector of a Vector

- The **length** of a vector (magnitude) is the hypotenuse of the right-angled triangle formed by the vector coordinates x and y.
- The **unit vector** of a vector is a vector with length 1. (In the code below `*this` refers to the `this` object, that is, the vector object for which we calculate the unit vector.)

```
Vector2D.cpp — Introduction
47 float Vector2D::length() const
48 {
49     float val = sqrt(fX * fX + fY * fY);
50
51     return round( val * 100.0f ) / 100.0f;
52 }
53
54 Vector2D Vector2D::normalize() const
55 {
56     return *this * (1.0f/length());
57 }
```

vector length



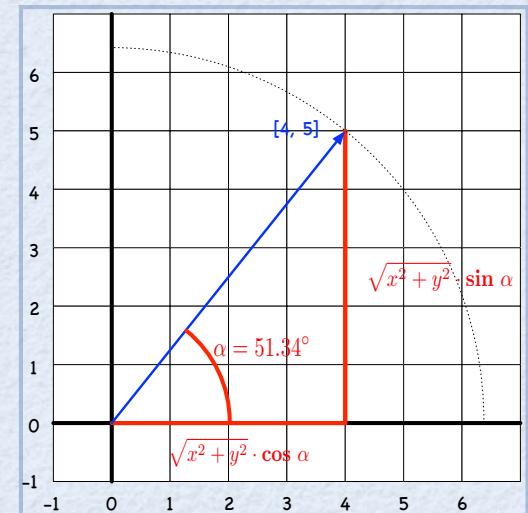
# Direction and Align

- The direction of a vector is the arctangent of the right-angled triangle formed by the vector coordinates  $x$  and  $y$ .
- We can align/rotate a vector, without changing its length, by multiplying its length with the sine and the cosine of the direction angle to obtain the new  $x$  and  $y$  coordinates, respectively.

C++ cast M\_PI to float

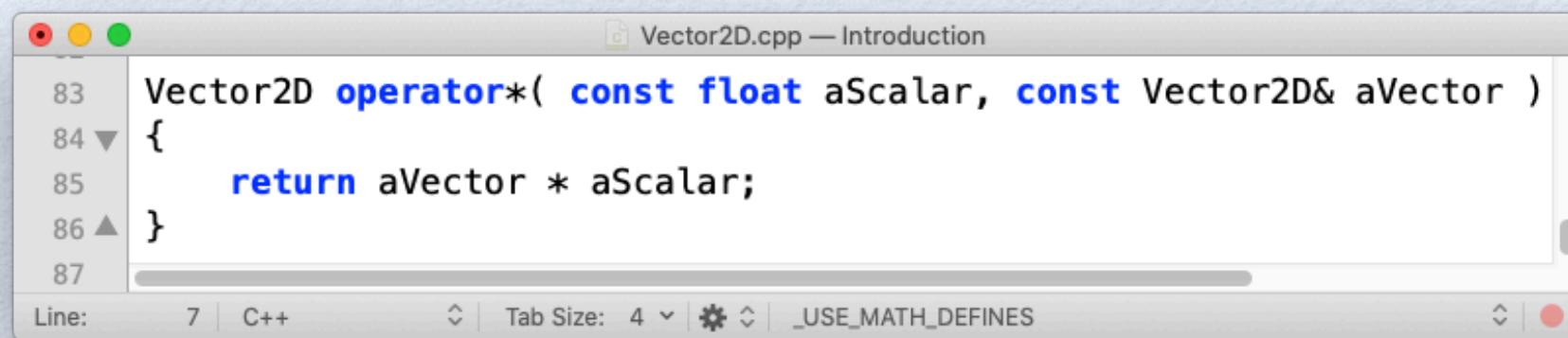
```
58
59     float Vector2D::direction() const
60 {
61     float val = atan2( fY, fX ) * 180.0f / static_cast<float>(M_PI);
62
63     return round( val * 100.0f ) / 100.0f;
64 }
65
66 Vector2D Vector2D::align( float aAngleInDegrees ) const
67 {
68     float lRadians = aAngleInDegrees * static_cast<float>(M_PI) / 180.0f;
69
70     return length() * Vector2D( cos( lRadians ), sin( lRadians ) );
71 }
72
```

direction/align



# Ad hoc Operator \*

- The member operator for scalar multiplication only allows for `vector * scalar`. However, multiplication is commutative, that is changing the order of the operands does not change the result.
- We can recover the commutativity of scalar multiplication by defining an ad hoc multiplication operator that takes a scalar as first argument and a vector as the second:



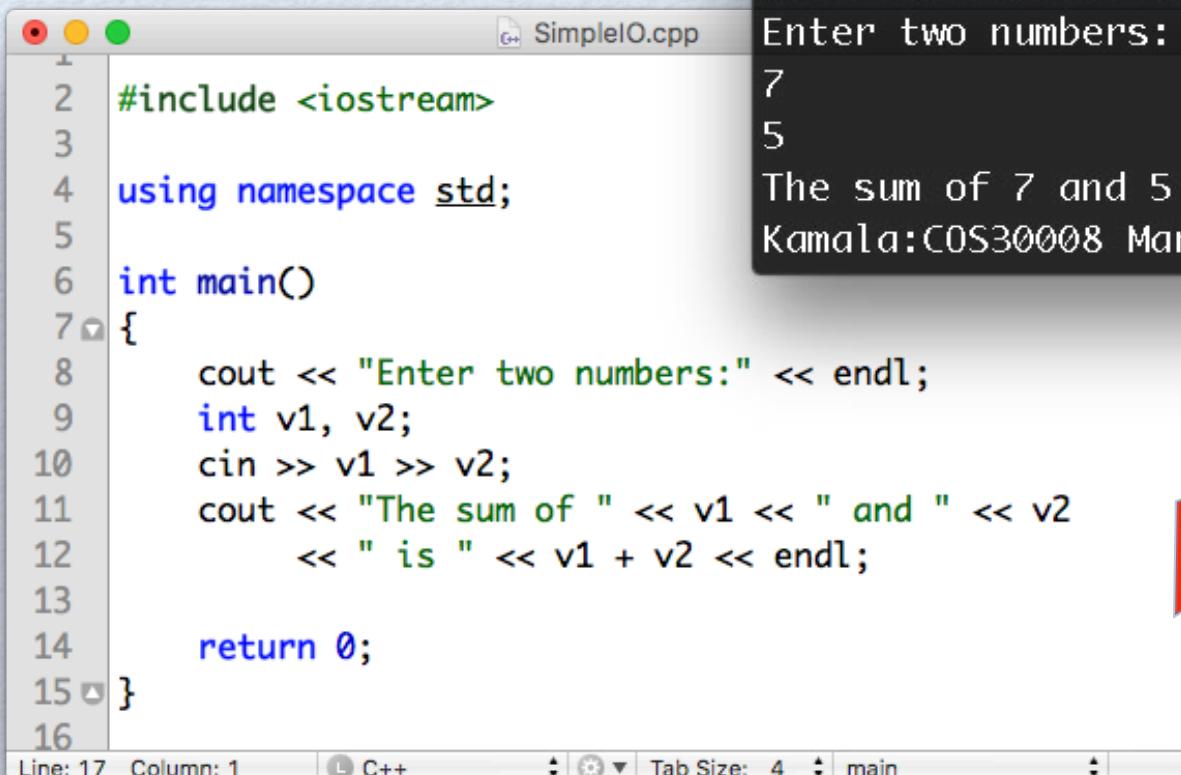
The screenshot shows a code editor window titled "Vector2D.cpp — Introduction". The code is as follows:

```
83 Vector2D operator*( const float aScalar, const Vector2D& aVector )
84 {
85     return aVector * aScalar;
86 }
87
```

The code defines a member function `operator*` for the `Vector2D` class. It takes a `const float` parameter `aScalar` and a `const Vector2D&` parameter `aVector`. The function returns the result of multiplying `aVector` by `aScalar`. The code editor interface includes line numbers, syntax highlighting, and standard toolbars at the bottom.

# Data I/O

# I/O in C++ is operator based.



```
SimpleIO.cpp
2 #include <iostream>
3
4 using namespace std;
5
6 int main()
7 {
8     cout << "Enter two numbers:" << endl;
9     int v1, v2;
10    cin >> v1 >> v2;
11    cout << "The sum of " << v1 << " and " << v2
12    << " is " << v1 + v2 << endl;
13
14    return 0;
15 }
16
```

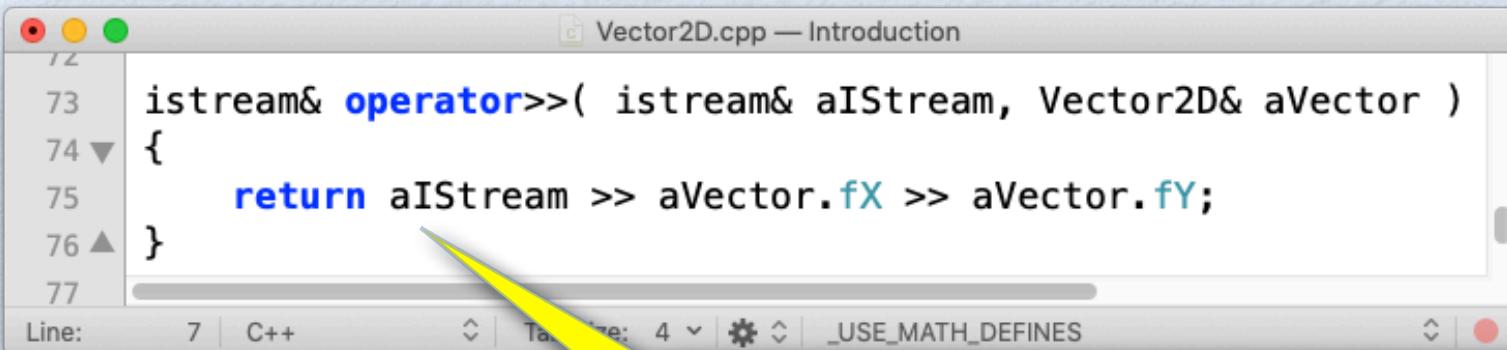
Line: 17 Column: 1    C++    Tab Size: 4    main

```
COS30008
Kamala:COS30008 Markus$ g++ -o SimpleIO SimpleIO.cpp
Kamala:COS30008 Markus$ ./SimpleIO
Enter two numbers:
7
5
The sum of 7 and 5 is 12
Kamala:COS30008 Markus$
```

- C++ relies on the binary operators `<<` and `>>` to perform formatted input and output, respectively.

# The Vector2D Input Operator >>

- The input operator performs formatted input of the vector coordinates, that is, we try to fetch two floating point values from the input stream.
- If the input fails, either of the coordinates or both are set to 0.0.
- At the end, the input operator returns the stream object to allow for a “chaining” of input operations.



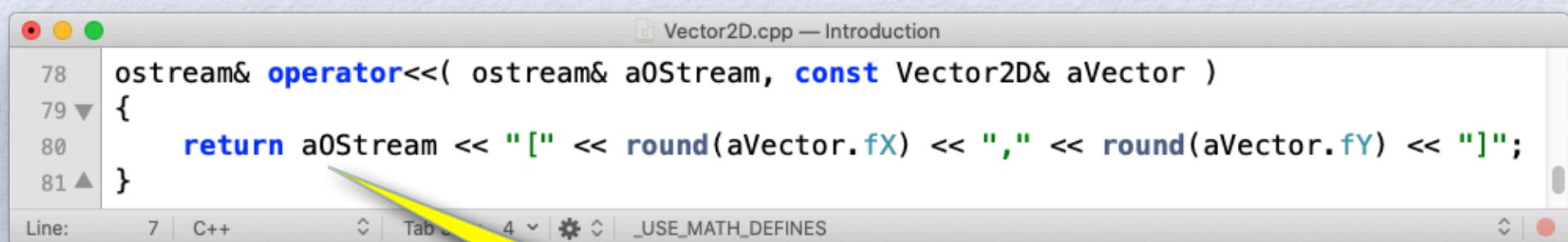
```
Vector2D.cpp — Introduction
73 istream& operator>>( istream& aIStream, Vector2D& aVector )
74 {
75     return aIStream >> aVector.fX >> aVector.fY;
76 }
77
```

A screenshot of a Mac OS X-style code editor window titled "Vector2D.cpp — Introduction". The code shows the implementation of the input operator (>>) for a Vector2D class. Line 73 defines the operator with its parameters. Line 74 begins a block brace. Line 75 contains the return statement, which chains two input operations (>>) for the fX and fY members of the Vector2D object. Line 76 ends the block brace. Line 77 is a blank line. The status bar at the bottom shows "Line: 7 | C++" and various tool icons.

Return reference to input stream.

# The Vector2D Output Operator <<

- The output operator assigns a “textual representation” to objects of type Vector2D.
- It performs formatted output, that is, the coordinate values are rendered as integer values in the output. The round function yields the nearest integral value.
- At the end, the input operator returns the stream object to allow for a “chaining” of output operations.

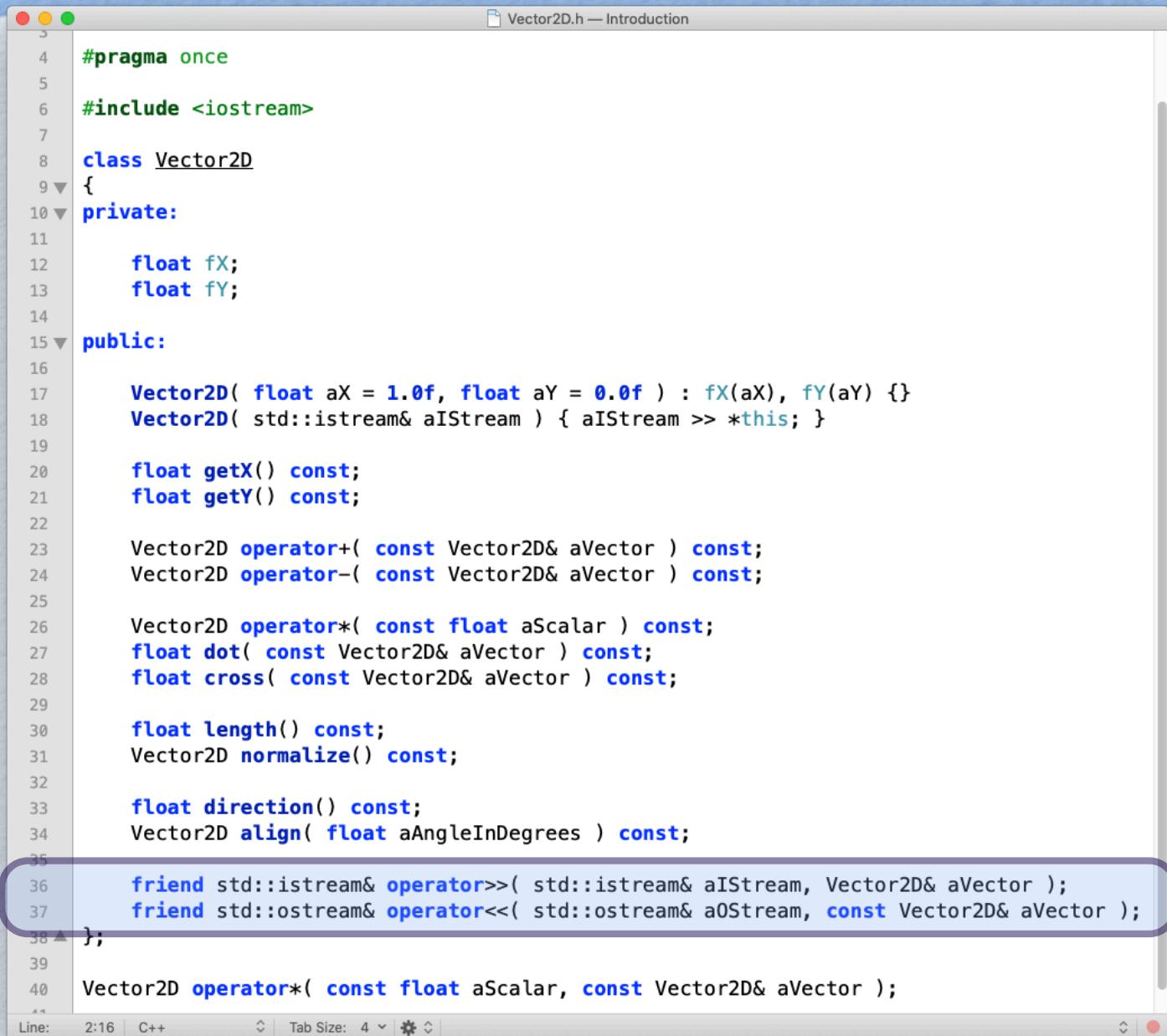


```
Vector2D.cpp — Introduction
78 ostream& operator<<( ostream& aOutputStream, const Vector2D& aVector )
79 {
80     return aOutputStream << "[" << round(aVector.fX) << "," << round(aVector.fY) << "]";
81 }
```

A screenshot of a C++ code editor window titled "Vector2D.cpp — Introduction". The code shows the implementation of the output operator (<<) for the Vector2D class. The operator takes an output stream (ostream&) and a Vector2D object (const Vector2D&) as parameters. It returns the output stream itself after inserting a "[", the rounded x-value (fX), a ",", the rounded y-value (fY), and a "]". The code editor interface includes a toolbar at the top, a status bar at the bottom, and a yellow callout bubble pointing to the return statement.

Return reference to output stream.

# Class Vector2D - The Friends



```
#pragma once
#include <iostream>
class Vector2D
{
private:
    float fX;
    float fY;
public:
    Vector2D( float aX = 1.0f, float aY = 0.0f ) : fX(aX), fY(aY) {}
    Vector2D( std::istream& aInputStream ) { aInputStream >> *this; }

    float getX() const;
    float getY() const;

    Vector2D operator+( const Vector2D& aVector ) const;
    Vector2D operator-( const Vector2D& aVector ) const;

    Vector2D operator*( const float aScalar ) const;
    float dot( const Vector2D& aVector ) const;
    float cross( const Vector2D& aVector ) const;

    float length() const;
    Vector2D normalize() const;

    float direction() const;
    Vector2D align( float aAngleInDegrees ) const;

    friend std::istream& operator>>( std::istream& aInputStream, Vector2D& aVector );
    friend std::ostream& operator<<( std::ostream& aOutputStream, const Vector2D& aVector );
};

Vector2D operator*( const float aScalar, const Vector2D& aVector );
```

# Friends

- Friends are allowed to access private members of classes.
- A class declares its friends explicitly.
- Friends enable uncontrolled access to members.
- The friend mechanism induces a particular programming (C++) style.
- The friend mechanism is not object-oriented!
- I/O depends on the friend mechanism.

# The Friend Mechanism

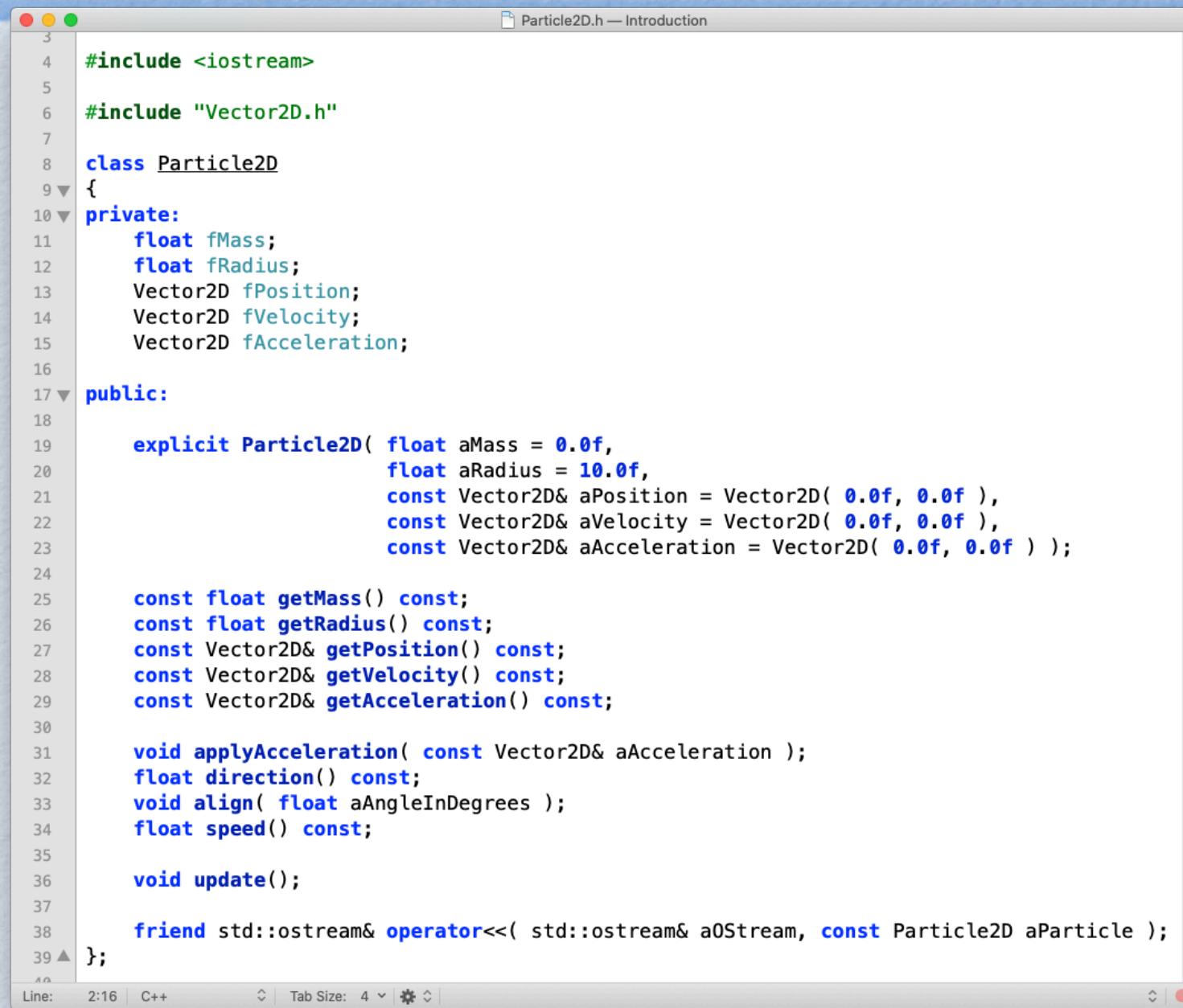
Friends are self-contained procedures (or functions) that do not belong to a specific class, but have access to the members of a class, when this class declares those procedures as friends.

# Class Composition: Particle2D

# Requirements for a particle class

- Particles represent physical model entities with the following attributes:
  - Mass (the particle's resistance – inertia – to change in its velocity)
  - Radius (particles can be rendered as filled circles)
  - Position, a 2D vector to assign a location in 2D space.
  - Velocity, a 2D vector to capture the speed of a particle in a given direction
  - Acceleration, a 2D vector representing the acceleration of a particle in a given direction due to the application of some force (e.g., gravity)
- References
  - Leonard Susskind and George Hrabovsky: The Theoretical Minimum – What You Need to Know to Start Doing Physics. Allen Lane (2013)
  - [www.codingmath.com](http://www.codingmath.com)

# Particle2D: A Simple 2D Particle Class



```
3 #include <iostream>
4
5 #include "Vector2D.h"
6
7 class Particle2D
8 {
9 private:
10     float fMass;
11     float fRadius;
12     Vector2D fPosition;
13     Vector2D fVelocity;
14     Vector2D fAcceleration;
15
16 public:
17     explicit Particle2D( float aMass = 0.0f,
18                           float aRadius = 10.0f,
19                           const Vector2D& aPosition = Vector2D( 0.0f, 0.0f ),
20                           const Vector2D& aVelocity = Vector2D( 0.0f, 0.0f ),
21                           const Vector2D& aAcceleration = Vector2D( 0.0f, 0.0f ) );
22
23     const float getMass() const;
24     const float getRadius() const;
25     const Vector2D& getPosition() const;
26     const Vector2D& getVelocity() const;
27     const Vector2D& getAcceleration() const;
28
29     void applyAcceleration( const Vector2D& aAcceleration );
30     float direction() const;
31     void align( float aAngleInDegrees );
32     float speed() const;
33
34     void update();
35
36     friend std::ostream& operator<<( std::ostream& aOutputStream, const Particle2D aParticle );
37
38 };
39
```

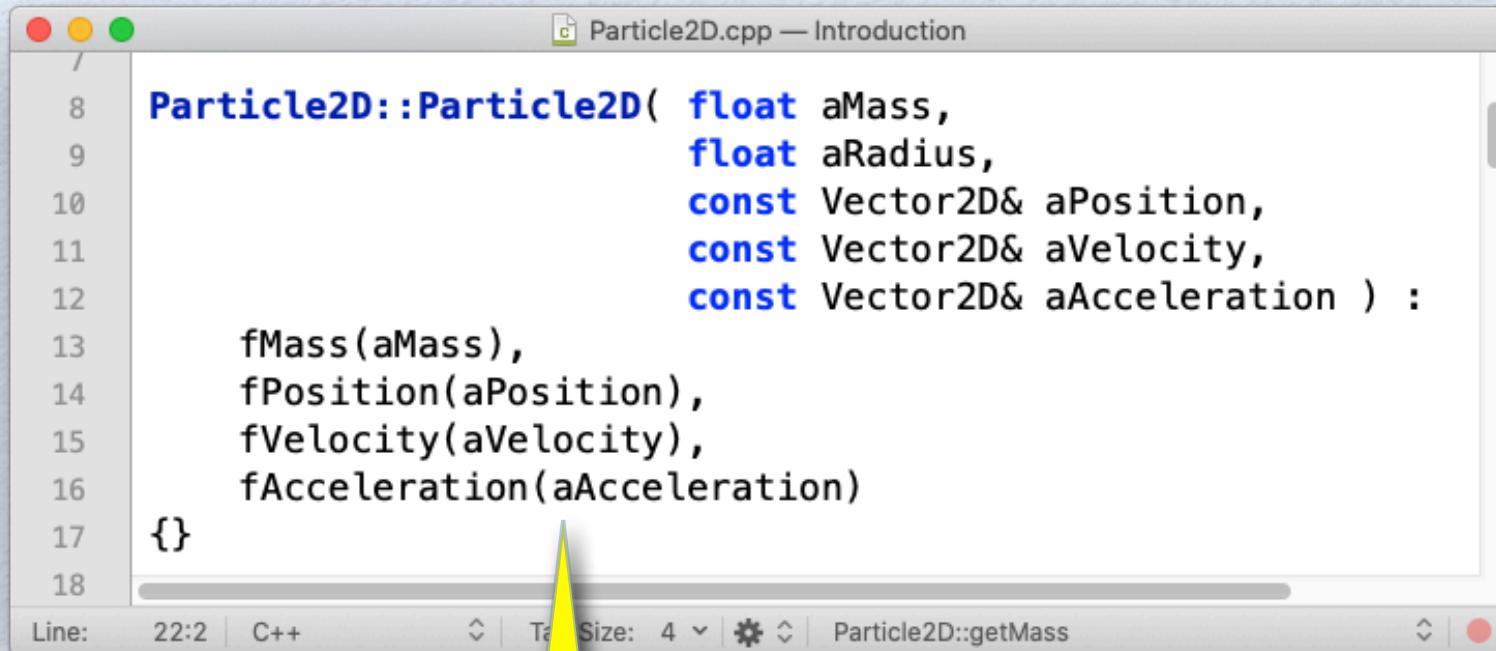
Line: 2:16 | C++ | Tab Size: 4 | 

# Explicit Particle2D Constructor

```
3 #include <iostream>
4
5 #include "Vector2D.h"
6
7 class Particle2D
8 {
9 private:
10     float fMass;
11     float fRadius;
12     Vector2D fPosition;
13     Vector2D fVelocity;
14     Vector2D fAcceleration;
15
16 public:
17     explicit Particle2D( float aMass = 0.0f,
18                           float aRadius = 10.0f,
19                           const Vector2D& aPosition = Vector2D( 0.0f, 0.0f ),
20                           const Vector2D& aVelocity = Vector2D( 0.0f, 0.0f ),
21                           const Vector2D& aAcceleration = Vector2D( 0.0f, 0.0f ) );
22
23     const float getMass() const;
24     const float getRadius() const;
25     const Vector2D& getPosition() const;
26     const Vector2D& getVelocity() const;
27     const Vector2D& getAcceleration() const;
28
29     void applyAcceleration( const Vector2D& aAcceleration );
30     float direction() const;
31     void align( float aAngleInDegrees );
32     float speed() const;
33
34     void update();
35
36     friend std::ostream& operator<<( std::ostream& aOutputStream, const Particle2D aParticle );
37
38 };
39
```

The constructor is marked `explicit` to prevent automatic type conversion from `float` to `Particle2D` when all other parameters are omitted (use default).

# Constructor Implementation



```
Particle2D::Particle2D( float aMass,
                        float aRadius,
                        const Vector2D& aPosition,
                        const Vector2D& aVelocity,
                        const Vector2D& aAcceleration ) :
    fMass(aMass),
    fPosition(aPosition),
    fVelocity(aVelocity),
    fAcceleration(aAcceleration)
{}
```

We use a constructor initializer list to prevent a default initialization of the Vector2D member variables. Instead, we employ the compiler-generated copy constructor to initialize the Vector2D member variables. (The Qt framework uses this approach extensively.)

# The Particle2D Operations

A screenshot of a C++ code editor showing the file `Particle2D.cpp`. The code defines a class `Particle2D` with several member functions:

```
43 void Particle2D::applyAcceleration( const Vector2D& aAcceleration )
44 {
45     fAcceleration = fAcceleration + aAcceleration;
46 }
47
48 float Particle2D::direction() const
49 {
50     return fVelocity.direction();
51 }
52
53
54 void Particle2D::align( float aAngleInDegrees )
55 {
56     fVelocity = fVelocity.align( aAngleInDegrees );
57 }
58
59 float Particle2D::speed() const
60 {
61     return fVelocity.length();
62 }
63
64 void Particle2D::update()
65 {
66     fVelocity = fVelocity + fAcceleration;
67     fPosition = fPosition + fVelocity;
68 }
```

Annotations with yellow callouts explain the functionality of some of these methods:

- A callout points to the `align` method with the text "Adjust the direction of the velocity".
- A callout points to the update loop at the bottom with the text "Update velocity and position".

The code editor interface shows the following details at the bottom:

Line: 22:2 | C++ | Tab Size: 4 | Particle2D::getMass

# The Simulation: Main

The image shows a comparison between a code editor and a terminal window.

**Code Editor (Left):** A screenshot of a Mac OS X-style application window titled "Main.cpp — Introduction". It contains the following C++ code:

```
#include <iostream>
#include "Particle2D.h"

using namespace std;

int main()
{
    cout << "A simple particle simulation\n" << endl;

    Particle2D obj( 0.0f,
                    10.0f,
                    Vector2D( 10.0f, 20.0f ),
                    Vector2D( 4.0f, 15.0f ),
                    Vector2D( 0.0f, -0.1f )
    );

    do
    {
        cout << obj << endl;
        obj.update();
    } while ( obj.getPosition().getY() >= 20.0f );

    cout << obj << endl;

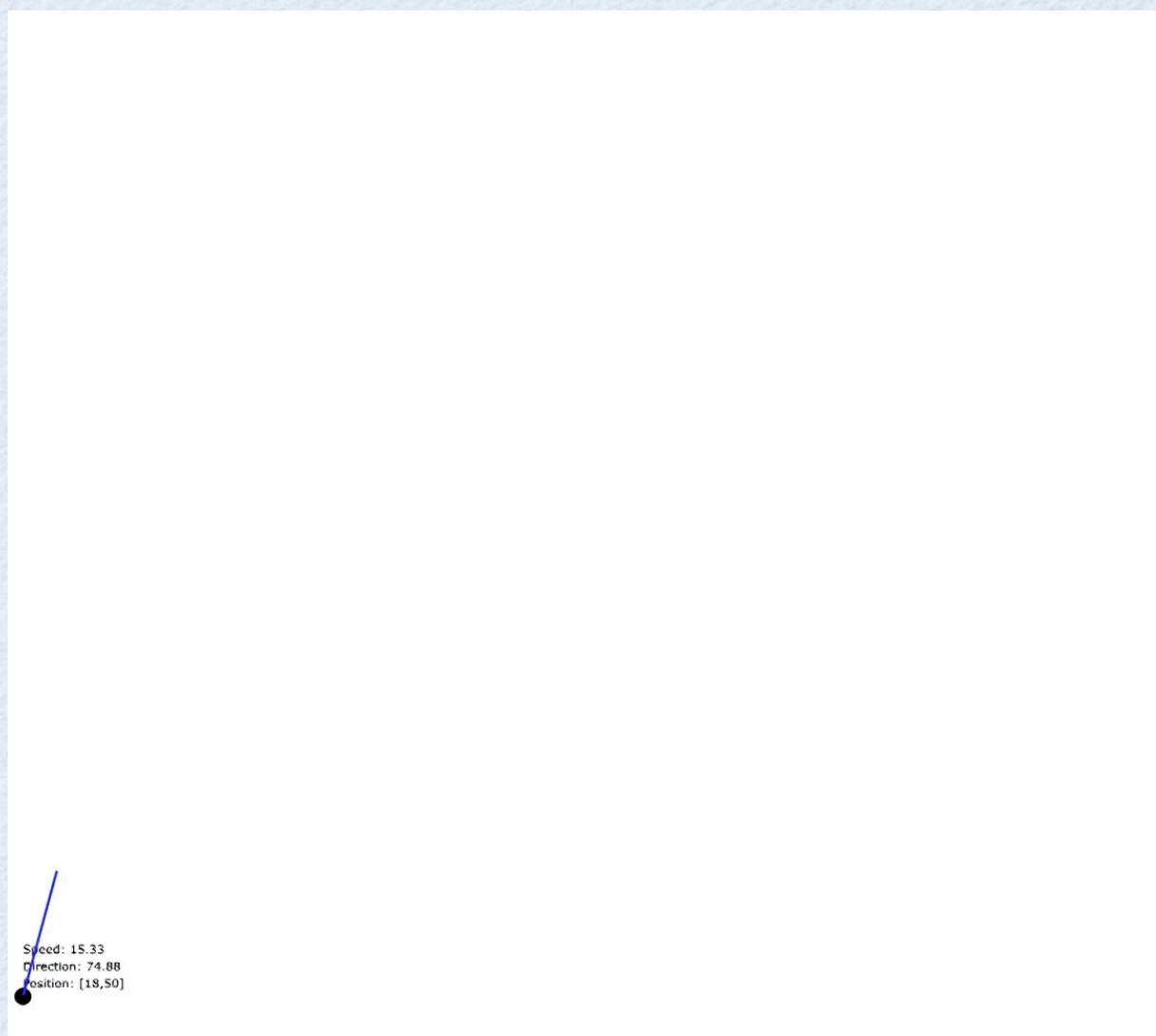
    return 0;
}
```

**Terminal (Right):** A screenshot of a Microsoft Visual Studio Debug C++ terminal window. It displays the output of the program, which is a series of particle states:

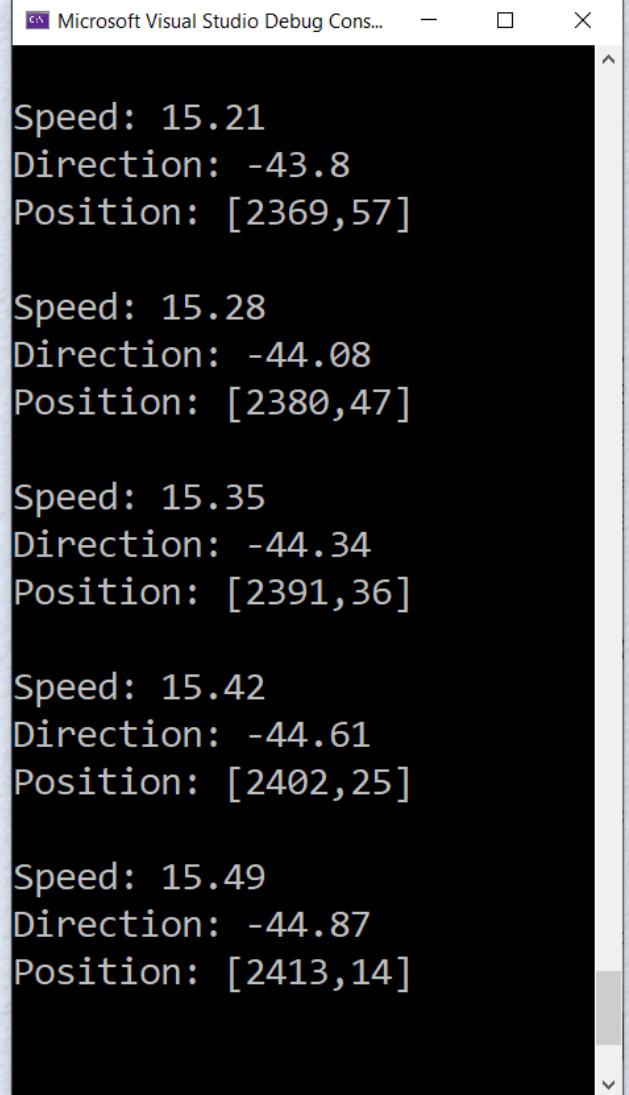
Speed	Direction	Position
14.95	-74.48	[1186,93]
15.04	-74.58	[1190,79]
15.14	-74.68	[1194,64]
15.23	-74.78	[1198,50]
15.33	-74.88	[1202,35]
15.43	-74.97	[1206,20]

A yellow callout bubble points from the terminal window to the condition in the loop: "Loop until particle is less than 20 units above the base line."

# Simulation Visualization



# Best Angle: 45 Degrees



```
Line: 1 | C++ | Tab Size: 4 | ⚙ | 90
```

```
Main.cpp — Introduction

2 #include <iostream>
3
4 #include "Particle2D.h"
5
6 using namespace std;
7
8 int main()
9 {
10     cout << "A simple particle simulation\n" << endl;
11
12     Particle2D obj( 0.0f,
13                      10.0f,
14                      Vector2D( 10.0f, 20.0f ),
15                      Vector2D( 4.0f, 15.0f ),
16                      Vector2D( 0.0f, -0.1f )
17                      );
18
19     obj.align( 45.0f );    // best angle
20
21     do
22     {
23         cout << obj << endl;
24
25         obj.update();
26     } while ( obj.getPosition().getY() >= 20.0f );
27
28     cout << obj << endl;
29
30     return 0;
31 }
```

```
Speed: 15.21
Direction: -43.8
Position: [2369,57]

Speed: 15.28
Direction: -44.08
Position: [2380,47]

Speed: 15.35
Direction: -44.34
Position: [2391,36]

Speed: 15.42
Direction: -44.61
Position: [2402,25]

Speed: 15.49
Direction: -44.87
Position: [2413,14]
```

# Object Models & Inheritance

# C++ Object Models

- C++ supports:
  - A **value-based object model**
  - A **reference-based object model**

This can make programming in C++ difficult. Java, for example, has only one model - everything is a reference!

# The Value-based Object Model

- Value-based object model:
  - Objects are stored on the stack.
  - Object are accessed through object variables.
  - An object's memory is implicitly released.

# Valued-based Objects

- Value-based objects look and feel like records (or structs):

```
Card AceOfDiamond( Diamond, 14 );
```

```
Card TestCard( Diamond, 14 );
```

Structural  
Equivalence

```
if ( TestCard == AceOfDiamond )
```

```
    cout << "The test card is " << TestCard.getName() << endl;
```

Member Selection

# The Reference-based Object Model

- Reference-based object model:
  - Objects are stored on the heap.
  - Objects are accessed through pointer variables.
  - An object's memory must be explicitly released.

# Reference-based Objects

- Reference-based objects require pointer variables and an explicit new and delete:

```
Card* AceOfDiamond = new Card( Diamond, 14 );
```

```
Card* TestCard = new Card( Diamond, 14 );
```

Pointer Comparison

```
if ( TestCard == AceOfDiamond )
```

```
cout << "The test card is " << TestCard->getName() << endl;
```

```
delete AceOfDiamond;
```

```
delete TestCard;
```

Member Dereference

# Cardinal Rule

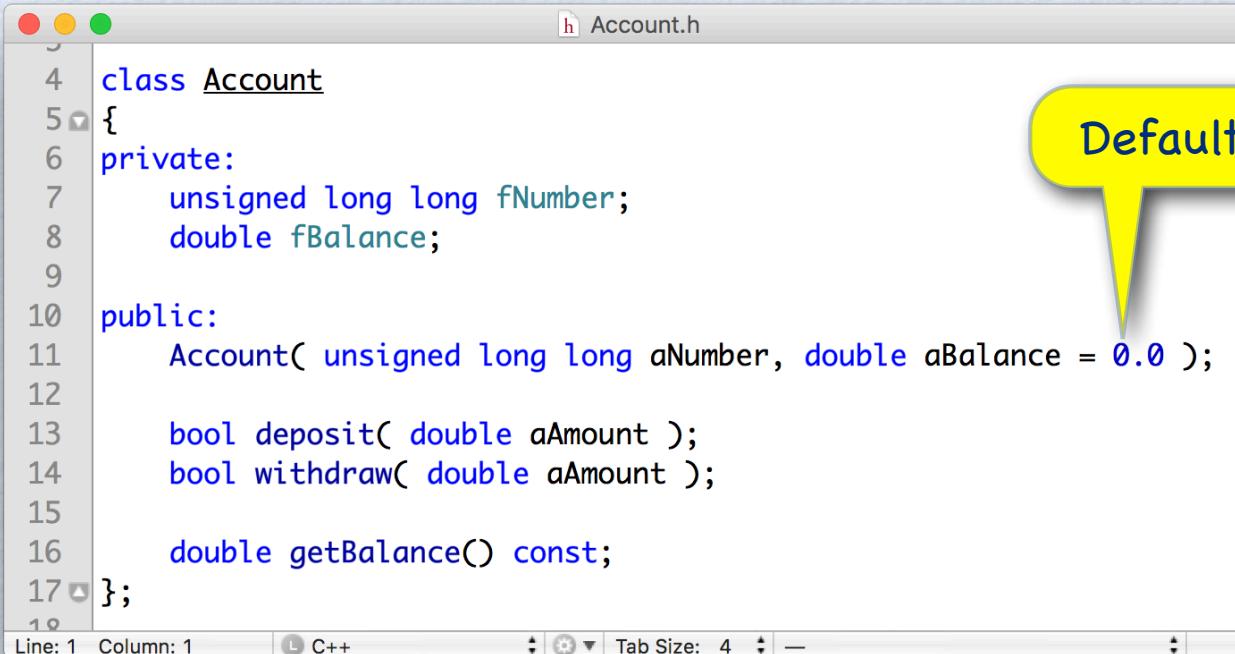
- We find both models in C++ code.
- Value semantics provides us with objects that behave like arithmetic types:

Every object is unique, but it may be known under numerous aliases (aka references).

# Inheritance

- Inheritance lets us define classes that model relationships among classes, sharing what is common, and specializing only that which is inherently different.
- Inheritance is
  - A mechanism for specialization
  - A mechanism for reuse
  - Fundamental to supporting polymorphism

# An Account Class



```
4 class Account
5 {
6     private:
7         unsigned long long fNumber;
8         double fBalance;
9
10    public:
11        Account( unsigned long long aNumber, double aBalance = 0.0 );
12
13        bool deposit( double aAmount );
14        bool withdraw( double aAmount );
15
16        double getBalance() const;
17    };
18
```

Line: 1 Column: 1 | L C++ | Tab Size: 4 | —

Default argument

- An account has a number ( $2^{64} - 1$  values) and a balance.
- To create an account we need a number. Funds can be credited to an account once it has been created.

# Account Implementation

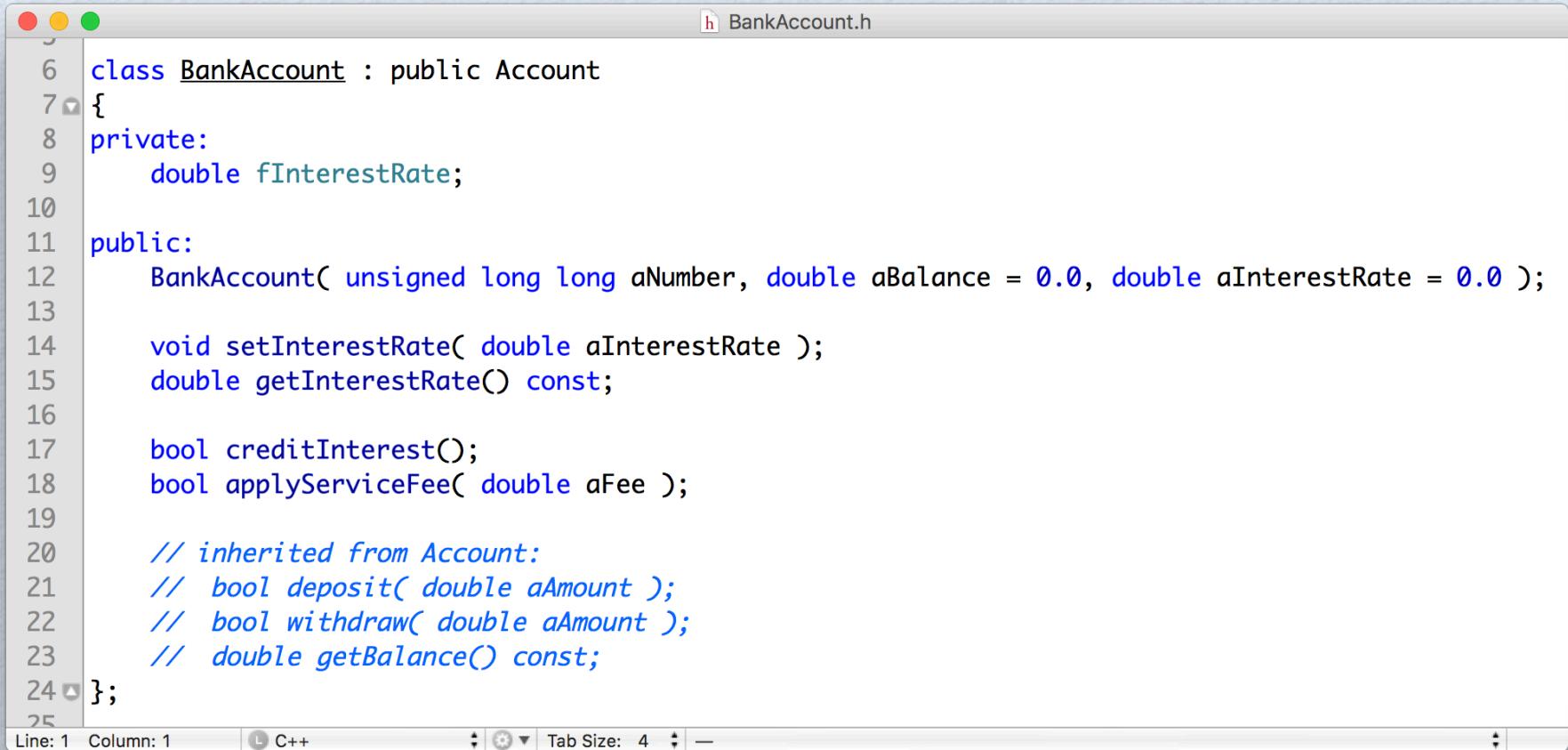
```
4 Account::Account( unsigned long long aNumber, double aBalance )
5 {
6     fNumber = aNumber;
7     fBalance = aBalance;
8 }
9
10 bool Account::deposit( double aAmount )
11 {
12     fBalance += aAmount;
13
14     return true;
15 }
16
17 bool Account::withdraw( double aAmount )
18 {
19     fBalance -= aAmount;
20
21     return true;
22 }
23
24 double Account::getBalance() const
25 {
26     return fBalance;
27 }
28
```

Default in class specification

Signal success

Balance can become negative

# A BankAccount Class



The screenshot shows a code editor window with the title "BankAccount.h". The code is written in C++ and defines a class named "BankAccount" that inherits from "Account". The class has private members "fInterestRate" and a public constructor that initializes the account number, balance, and interest rate. It also includes methods to set and get the interest rate, as well as methods to credit interest and apply service fees. The code is annotated with comments indicating inheritance from "Account". The code editor interface at the bottom shows "Line: 1 Column: 1", "C++", "Tab Size: 4", and other standard editor controls.

```
6 class BankAccount : public Account
7 {
8     private:
9         double fInterestRate;
10
11    public:
12        BankAccount( unsigned long long aNumber, double aBalance = 0.0, double aInterestRate = 0.0 );
13
14        void setInterestRate( double aInterestRate );
15        double getInterestRate() const;
16
17        bool creditInterest();
18        bool applyServiceFee( double aFee );
19
20        // inherited from Account:
21        // bool deposit( double aAmount );
22        // bool withdraw( double aAmount );
23        // double getBalance() const;
24    };
25
```

- A bank account is a **subtype** of account. A bank account **includes** all features of an account and offers bank account specific additional features (**incremental refinement**).

# Access Levels for Inheritance

- **public:**

- Public members in the base class remain public.
- Protected members in the base class remain protected.
- Yields a “is a” relationship, that is, a subtype.

- **protected:**

- Public and protected members in the base class are protected in the derived class.
- Yields a “implemented in terms of” relationship, that is, a new type.

- **private:**

- Public and protected members in the base class become private in the derived class.
- Yields a stricter “implemented in terms of” relationship, that is, a new type.

# Public Inheritance

- Public inheritance enables inclusion polymorphism: A subclass is a subtype.
- The subtype relationship is a key ingredient in contemporary object-oriented software development.
- An object of a subclass can be used safely anywhere an object of a superclass is expected. Example, we can use a BankAccount in lieu of an Account object.

# A Grain of Salt

- Public inheritance is not without flaws. It can result in an unwieldy and confusingly large set of public methods.
- Public inheritance is based on additive refinement. New classes are defined by extending an old class with new features.
- Protected and private inheritance allow for a more fine-grained control of the set of public methods. Unfortunately, these mechanisms do not yield subclasses and is often viewed controversial.

# Constructors and Inheritance

- Whenever an object of a derived class is instantiated, multiple constructors are called so that each class in the inheritance chain can initialize itself.
- The constructor for each class in the inheritance chain is called beginning with the base class at the top of the inheritance chain and ending with the most recent derived class.

# Base Class Initializer

```
Account.cpp
4 Account::Account( unsigned long long aNumber, double aBalance )
5 {
6     fNumber = aNumber;
7     fBalance = aBalance;
8 }
```

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

```
BankAccount.cpp
4 BankAccount::BankAccount( unsigned long long aNumber, double aBalance, double aInterestRate ) : Account( aNumber, aBalance )
5 {
6     fInterestRate = aInterestRate;
7 }
```

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

Direct super class constructor

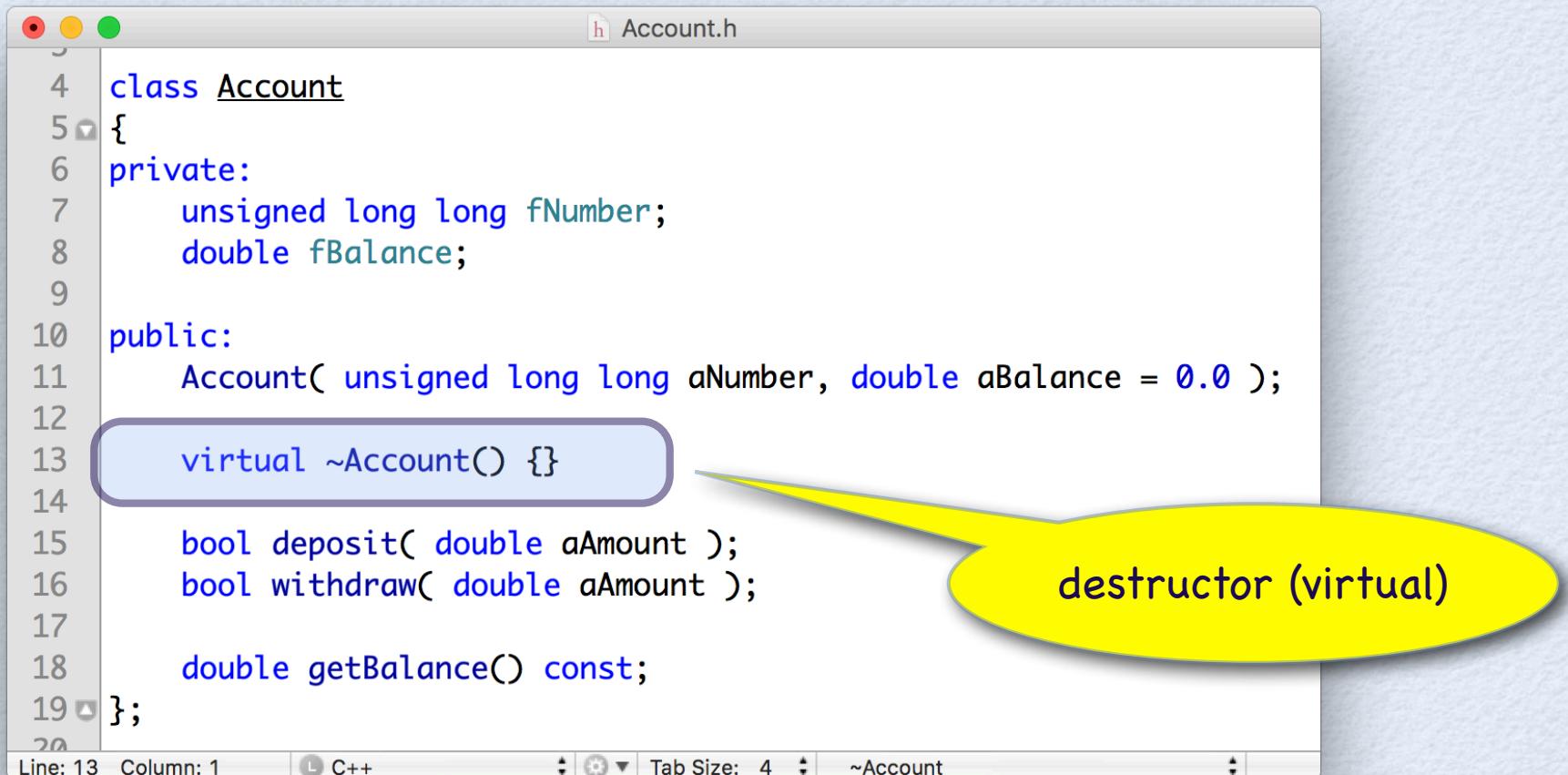
# Facts About Base Class Initializers

- If a base class does not have a default constructor, the derived class must provide a base class initializer for it.
- Base class initializers frequently appear alongside member initializers, which use similar syntax.
- If more than one argument is required by a base class constructor, the arguments are separated by comma.
- Reference members need to be initialized using a member initializer.

# Destructors and Inheritance

- Whenever an object of a derived class is destroyed, the destructor for each class in the inheritance chain, if defined, is called.
- The destructor for each class in the inheritance chain is called beginning with the most recent derived class and ending with the base class at the top of the inheritance chain.

# Account & BankAccount Constructors



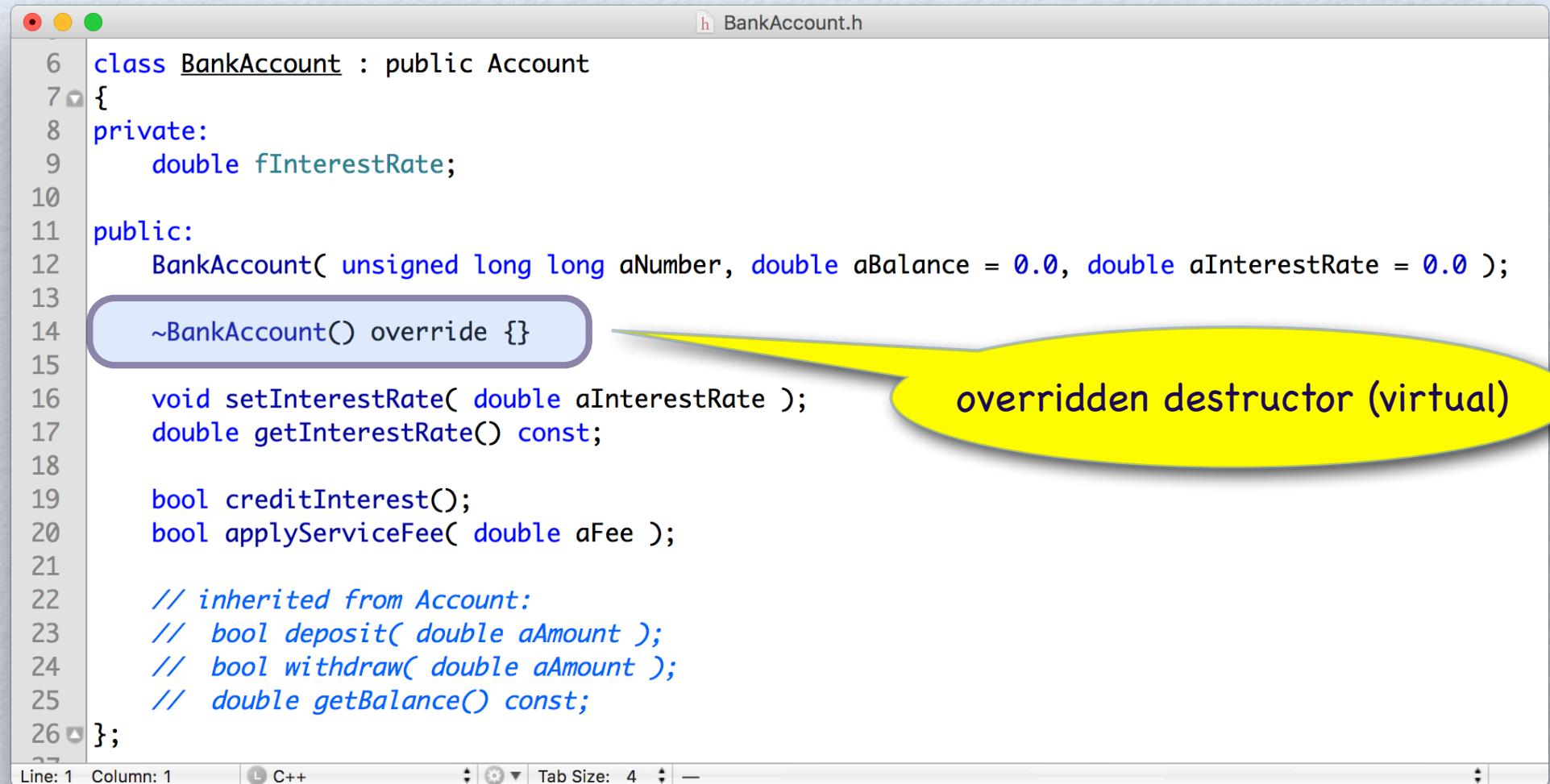
```
4 class Account
5 {
6     private:
7         unsigned long long fNumber;
8         double fBalance;
9
10    public:
11        Account( unsigned long long aNumber, double aBalance = 0.0 );
12
13        virtual ~Account() {}
14
15        bool deposit( double aAmount );
16        bool withdraw( double aAmount );
17
18        double getBalance() const;
19
20};
```

Line: 13 Column: 1    C++    Tab Size: 4    ~Account

virtual ~Account() {}

destruct (virtual)

# Account & BankAccount Constructors



```
6 class BankAccount : public Account
7 {
8     private:
9         double fInterestRate;
10
11    public:
12        BankAccount( unsigned long long aNumber, double aBalance = 0.0, double aInterestRate = 0.0 );
13
14        ~BankAccount() override {} // overridden destructor (virtual)
15
16        void setInterestRate( double aInterestRate );
17        double getInterestRate() const;
18
19        bool creditInterest();
20        bool applyServiceFee( double aFee );
21
22        // inherited from Account:
23        // bool deposit( double aAmount );
24        // bool withdraw( double aAmount );
25        // double getBalance() const;
26    };
27
```

The screenshot shows a code editor window with the file "BankAccount.h" open. The code defines a class BankAccount that inherits from Account. It includes private members for a number and interest rate, and public methods for setting and getting the interest rate, as well as credit and apply service fees. An overridden destructor `~BankAccount()` is also present. A callout bubble highlights this destructor with the text "overridden destructor (virtual)".

# Virtual Member Functions

- To give a member function from a base class new behavior in a derived class, one overrides it.
- To allow a member function in a base class to be overridden, one must declare the member function `virtual`.
- Note, non-virtual member functions are resolved with respect to the declared type of the object.
- In Java all member functions are virtual.
- Explicitly defining a (virtual) destructor affects which special member functions the compiler synthesizes (C++-11).

# The C++11 `override` Keyword

- With C++11, we have a new keyword, `override`, to signal overridden virtual methods (including destructors).
- It has the same semantics as in C#, but you do not need to explicitly specify it.
- The `virtual` specifier can still be used in these cases, but `override` is the new standard.

# Virtual Destructors

- When deleting an object using a base class pointer of reference, it is essential that the destructors for each of the classes in the inheritance chain get a chance to run:

```
BankAccount *BAptr;
```

```
Account *Aprt;
```

```
BAptr = new BankAccount( 2.25 );
```

```
Aprt = BAptra;
```

```
...
```

```
delete Aprt;
```

# Virtual Account Destructor

```
class Account
{
public:
    virtual ~Account() { ... }
};
```

Not declaring a destructor virtual  
is a common source of memory  
leaks in C++ code.

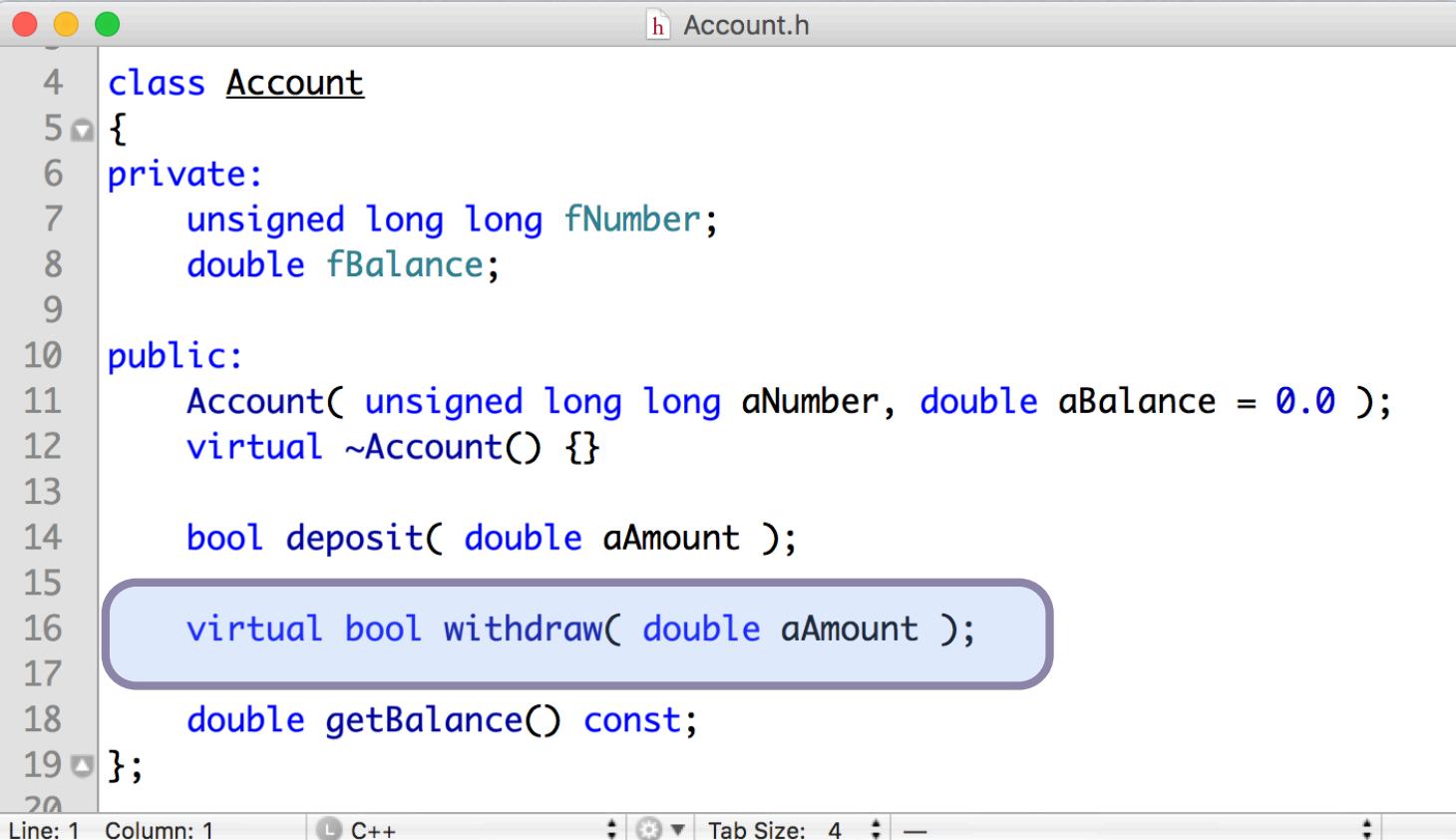
# Method Overriding

- Method overriding is an object-oriented programming mechanism that allows a subclass to provide a more specific implementation of a method, which is also present in one of its superclasses.
- The implementation in the subclass overrides (replaces) the implementation in the superclass by providing a method that has the same name, parameter signature, and return type as the method in the parent class. We say these methods belong to the same method family.
- Which (overridden) method is selected at runtime is determined by the receiver object used to invoke it. In general, the most recent definition is chosen, if possible.

# Method Family

- A member function of a class always belongs to a specific set, called **method family**. If the elements of this set are **virtual**, then their invocation is governed by **dynamic binding**, a technique that makes polymorphism real.

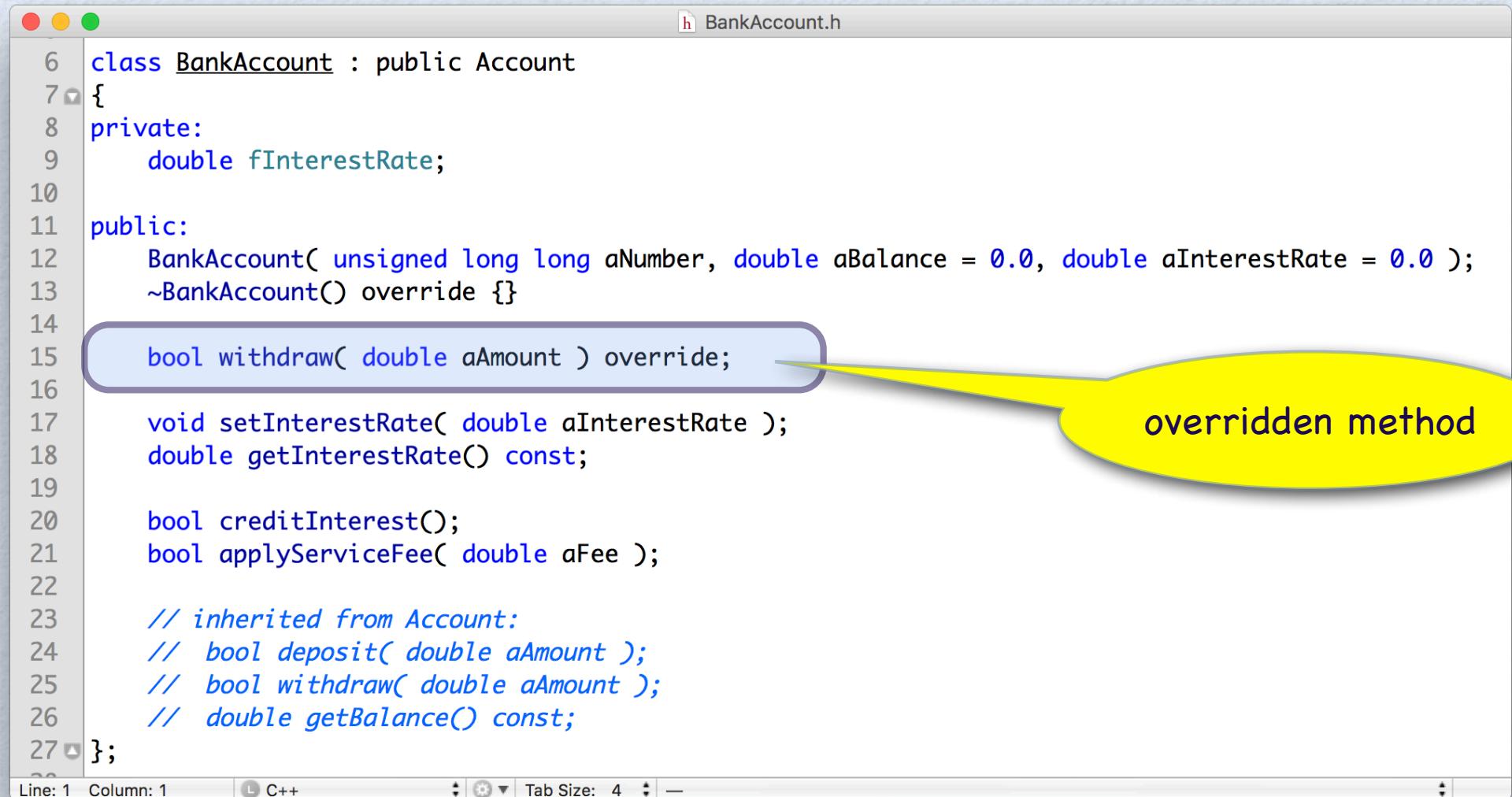
# Virtual withdraw Method



```
4 class Account
5 {
6     private:
7         unsigned long long fNumber;
8         double fBalance;
9
10    public:
11        Account( unsigned long long aNumber, double aBalance = 0.0 );
12        virtual ~Account() {}
13
14        bool deposit( double aAmount );
15
16        virtual bool withdraw( double aAmount );
17
18        double getBalance() const;
19    };
20
```

The code editor shows the `Account.h` header file. The `withdraw` method is highlighted with a purple rounded rectangle. The code defines a class `Account` with private members `fNumber` and `fBalance`, a constructor taking `aNumber` and `aBalance` (with a default value of `0.0`), a destructor, a `deposit` method, a `withdraw` method (which is virtual), and a `getBalance` method marked as `const`. The `withdraw` method is highlighted to emphasize its importance.

# Virtual withdraw Method



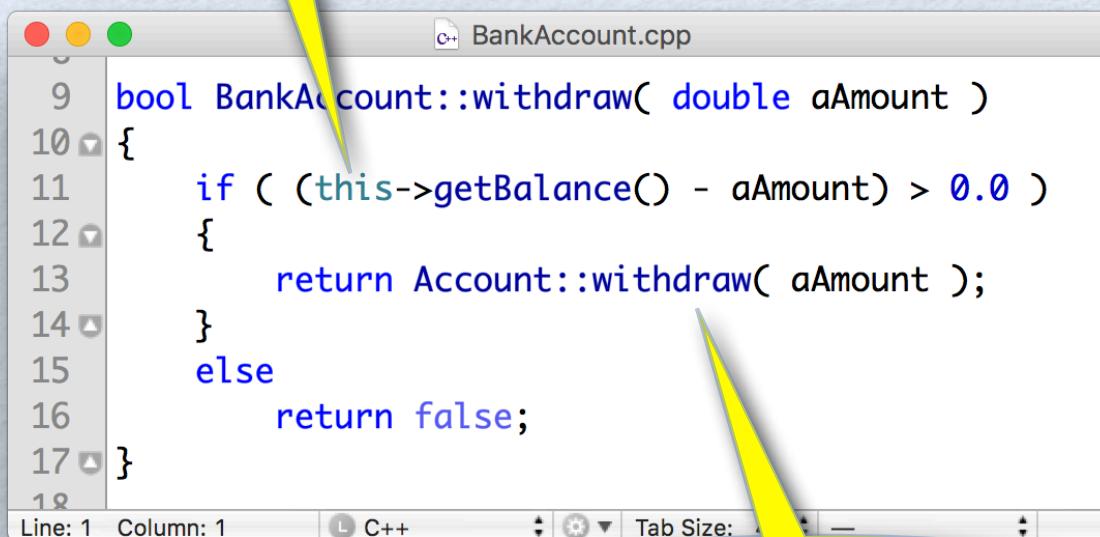
```
6 class BankAccount : public Account
7 {
8     private:
9         double fInterestRate;
10
11    public:
12        BankAccount( unsigned long long aNumber, double aBalance = 0.0, double aInterestRate = 0.0 );
13        ~BankAccount() override {}
14
15        bool withdraw( double aAmount ) override; // overridden method
16
17        void setInterestRate( double aInterestRate );
18        double getInterestRate() const;
19
20        bool creditInterest();
21        bool applyServiceFee( double aFee );
22
23        // inherited from Account:
24        // bool deposit( double aAmount );
25        // bool withdraw( double aAmount );
26        // double getBalance() const;
27};
```

# The withdraw Specialization

- In class Account, the method withdraw implements the behavior required to take funds out of an account. The Account class does not do any range checks. Hence, the execution of the withdraw method could result in a negative account balance.
- In general, bank accounts do not allow for negative balances (unless it is a credit account). Hence, a withdrawal can only succeed if there are sufficient funds in the account. The BankAccount's withdraw methods must guard a bank account against requests exceeding the current account balance.

# Overriding the withdraw Method

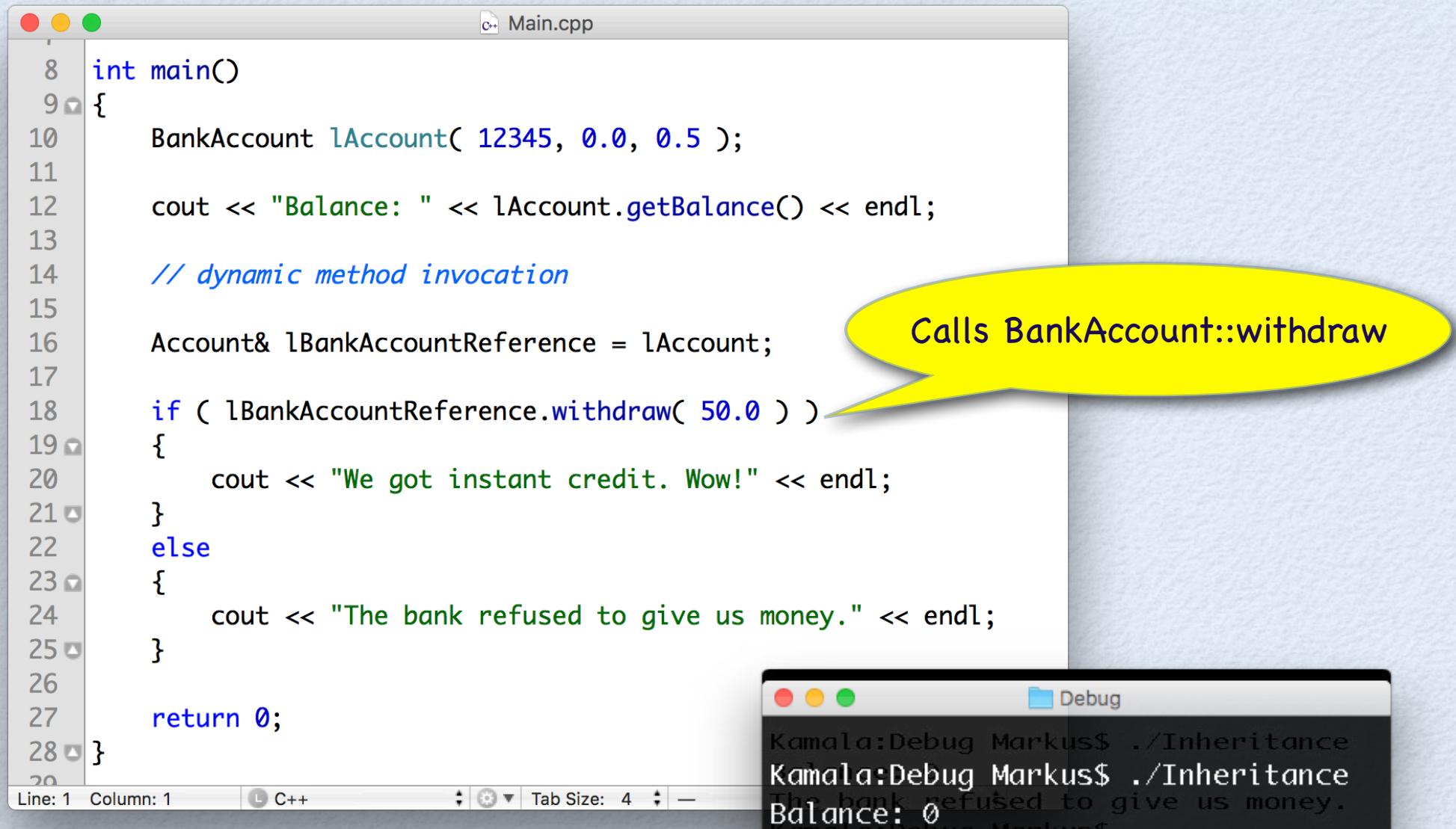
Call inherited method (`this->` optional)



```
9  bool BankAccount::withdraw( double aAmount )
10 { 
11     if ( (this->getBalance() - aAmount) > 0.0 )
12     {
13         return Account::withdraw( aAmount );
14     }
15     else
16         return false;
17 }
18 
```

Call overridden method

# Calling a Virtual Method



```
8 int main()
9 {
10    BankAccount lAccount( 12345, 0.0, 0.5 );
11
12    cout << "Balance: " << lAccount.getBalance() << endl;
13
14    // dynamic method invocation
15
16    Account& lBankAccountReference = lAccount;
17
18    if ( lBankAccountReference.withdraw( 50.0 ) )
19    {
20        cout << "We got instant credit. Wow!" << endl;
21    }
22    else
23    {
24        cout << "The bank refused to give us money." << endl;
25    }
26
27
28    return 0;
29 }
```

Calls BankAccount::withdraw

```
Kamala:Debug Markus$ ./Inheritance
Kamala:Debug Markus$ ./Inheritance
The bank refused to give us money.
Kamala:Debug Markus$
```

# Facts About Virtual Members

- Constructors cannot be virtual.
- Declaring a member function `virtual` does not require that this function must be overridden in derived classes, except the member function is declared `pure virtual`.
- Once a member function has been declared `virtual`, it remains `virtual`.
- Parameter and result types must match to properly override a `virtual` member function.
- If one declares a non-`virtual` member function `virtual` in a derived class, the new member function hides the inherited member function.

# Note on Virtual Member Functions

- If a virtual member function is called from inside a constructor or destructor, then the version that is run is the one defined for the type of the constructor or destructor itself - static method invocation.
- C++ is a very complex language.

# Data Structures – Basic Concepts

## Overview

- Programming Paradigms
- Values, Sets, and Arrays
- Indexer, Iterators, and Pattern Structures

## References

- Bruno R. Preiss: Data Structures and Algorithms with Object-Oriented Design Patterns in C++. John Wiley & Sons, Inc. (1999)
- Richard F. Gilberg and Behrouz A. Forouzan: Data Structures - A Pseudocode Approach with C. 2nd Edition. Thomson (2005)
- Russ Miller and Laurence Boxer: Algorithms Sequential & Parallel. 2nd Edition. Charles River Media Inc. (2005)
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)

# Programming Paradigms

- Imperative style:

program = algorithms + data

- Functional style:

program = function • function

- Logic programming style:

program = facts + rules

- Object-oriented style:

program = objects + messages

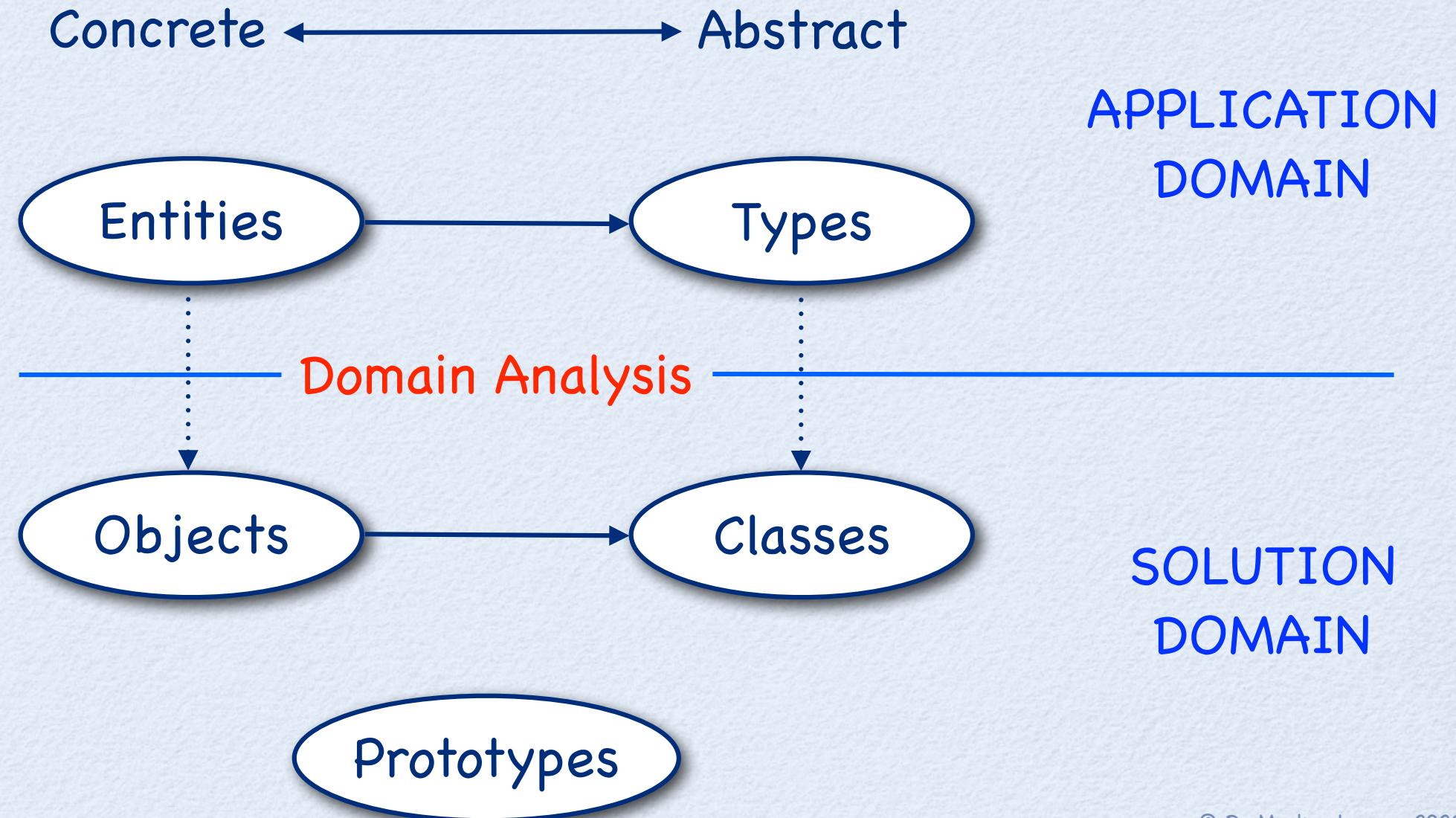
- Other styles and paradigms:

blackboard, events, pipes and filters, constraints, lists, ...

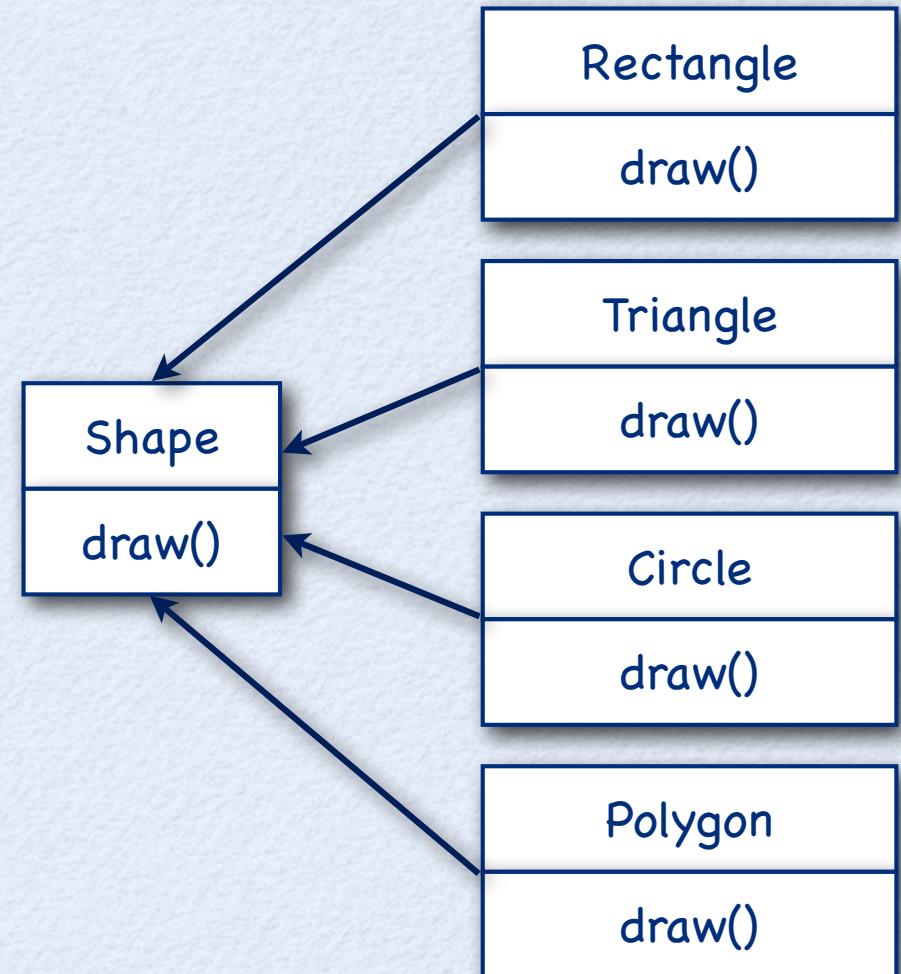
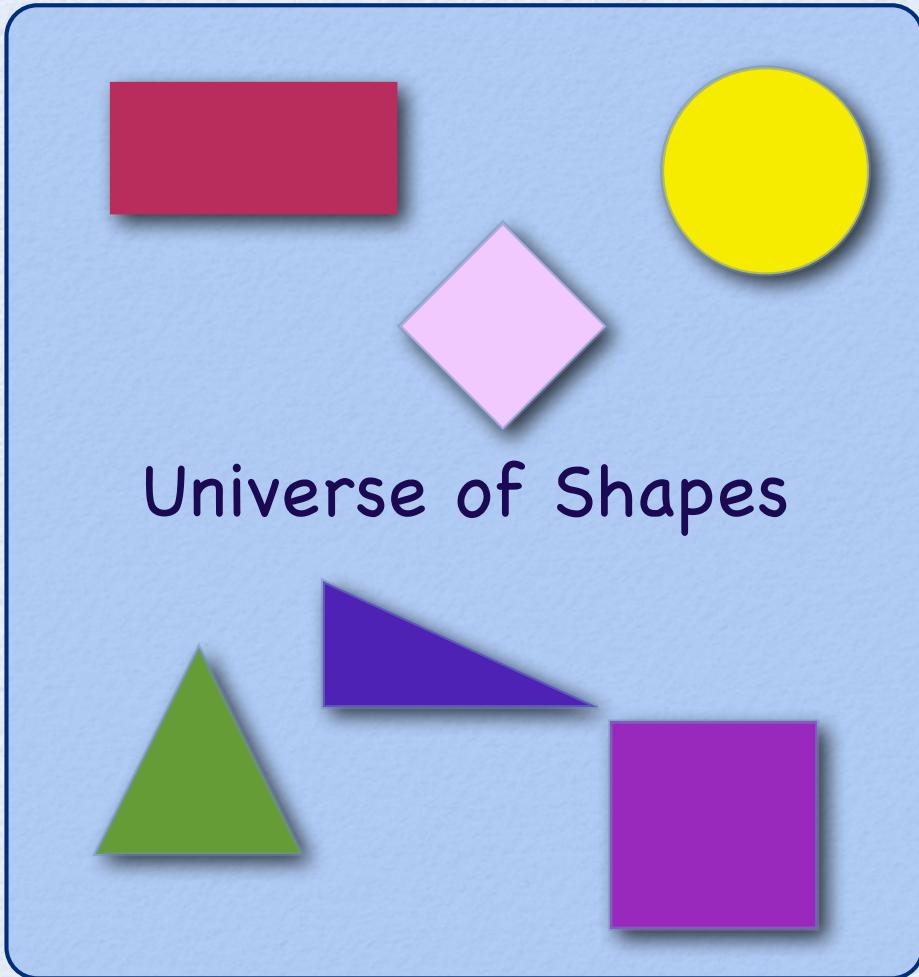
# Object-Oriented Software Development

- Object-oriented programming is about
  - Object-oriented software development
  - Using an object-oriented programming language
- Object-oriented software development is
  - An evolutionary step refining earlier techniques
  - A revolutionary idea perfecting earlier methods

# Object-Oriented Design



# Concrete vs. Abstract



# Why is object-oriented software development popular?

- The object-oriented development approach
  - Naturally captures real life
  - Scales well from trivial to complex tasks
  - Focuses on responsibilities, reuse, and composition

# Values

- In computer science we classify as a value everything that may be evaluated, stored, incorporated in a data structure, passed as an argument to a procedure or function, returned as a function result, and so on.
- In computer science, as in mathematics, an “expression” is used (solely) to denote a value.
- Which kinds of values are supported by a specific programming environment depends heavily on the underlying paradigm and its application domain.
- Most programming environments provide support for some basic sets of values like truth values, integers, real number, records, lists, etc.

# Constants

- Constants are named abstractions of values.
- Constants are used to assign an user-defined meaning to a value.
- Examples:
  - `EOF = -1`
  - `TRUE = 1`
  - `FALSE = 0`
  - `PI = 3.1415927`
  - `MESSAGE = "Welcome to DSP"`
- Constants do not have an address, that is, they do not have a location.
- At compile time, applications of constants are substituted by their corresponding definition.

# Primitive Values

- Primitive values are values whose representation cannot be further decomposed. We find that some of these values are implementation and platform dependent.
- Examples:

- Truth values,
- Integers,
- Characters,
- Strings,
- Enumerands,
- Real numbers.

-1

“Hello World!”

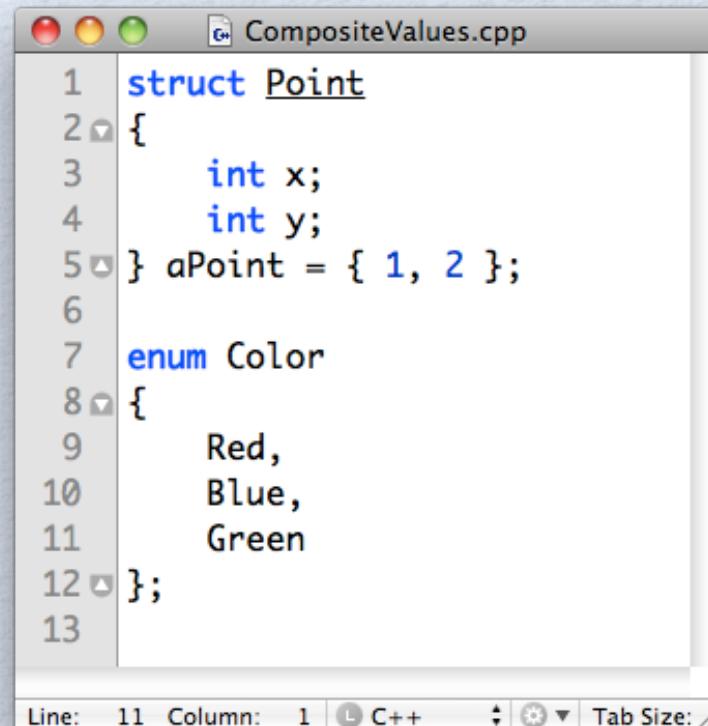
3.14159

false

Red

# Composite Values

- Composite values are built up using primitive values and composite values. The layout of composite values is in general implementation dependent.
- Examples:
  - Records
  - Arrays
  - Enumerations
  - Sets
  - Lists
  - Tuples
  - Files



```
1 struct Point
2 {
3     int x;
4     int y;
5 } aPoint = { 1, 2 };
6
7 enum Color
8 {
9     Red,
10    Blue,
11    Green
12 };
13
```

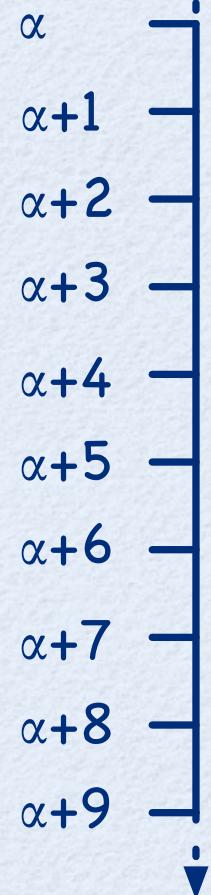
The screenshot shows a code editor window titled "CompositeValues.cpp". The code defines a struct named "Point" with two integer members, x and y. It also defines an enum named "Color" with three values: Red, Blue, and Green. A variable "aPoint" is initialized with the value { 1, 2 }. The code editor interface includes a status bar at the bottom with "Line: 11 Column: 1", "C++", and "Tab Size: 4".

# Pointers

- Pointers are references to values, i.e., they denote locations of a values.
- Pointers are used to store the address of a value (variable or function) – pointer to a value, and pointers are also used to store the address of another pointer – pointer to pointer.
- In general, it not necessary to define pointers with a greater reference level than pointer to pointer.
- In modern programming environments, we find pointers to variables, pointers to pointer, function pointers, and object pointers, but not all programming languages provide means to use pointers directly (e.g., Java).

# Memory, Values, and Pointers

Memory:



Values:

`pivar ==  $\alpha+3$`

`pivar == 3`

`pfvar ==  $\alpha+8$`

`fvar == 3.14`

Deference/Address :

`*pivar == 3`

`&pivar ==  $\alpha+3$`

`*pfvar == 3.14`

`&fvar ==  $\alpha+8$`

# Sets

- A set is a collection of elements (or values), possibly empty.
- All elements satisfy a possibly complex characterizing property. Formally, we write:

$$\{ x \mid P(x) = \text{True} \}$$

to define a set, where all elements satisfy the property  $P$ .

- The basic axiom of set theory is that there exists an empty set,  $\emptyset$ , with no elements. Formally,

$$\forall x, x \notin \emptyset$$

In words, “for every  $x$ ,  $x$  is not an element of  $\emptyset$ .”

# Sets are collections of values.

# Inductive Reasoning

- To define a set and to capture what qualifies values to be members of the set, we can use inductive reasoning and formally verify properties about members of the set.
- Algebraically, we can define a set using induction on the structure of expressions and induction on the length or structure of expressions as a means to verify (prove) properties of the set and the elements thereof.
- **Note:** We can construct infinitely many values from a given finite recipe – inductive specification.

# Inductive Specification

- Sometimes it is difficult to define a set explicitly, in particular if the elements of the set have a complex structure.
- However, it may be easy to define the set in terms of itself. This process is called inductive specification or recursion.
- Example:

Let the set  $S$  be the smallest set of natural numbers satisfying the following two properties:

- $0 \in S$ , and
- Whenever  $x \in S$ , then  $x + 3 \in S$ .

The first property is called base clause and the second property is called inductive/recursive clause. An inductive specification may have multiple base and inductive clauses.

# The “Smallest Set”

- If we use inductive specification, we always define the smallest set that satisfies all given properties. That is, inductive specification is free of redundancy.
- It is easy to see that there can be only one such set:

If  $S_1$  and  $S_2$  both satisfy all given properties, and both are the smallest, then we have  $S_1 \subseteq S_2$  (since  $S_1$  is the smallest), and  $S_2 \subseteq S_1$  (since  $S_2$  is the smallest), hence  $S_1 = S_2$ .

# The Set of Strings

$S = \epsilon \mid aS$ , where

- $\epsilon$  is the empty string and
  - $a \in \Sigma$ , with  $\Sigma$  being the alphabet over  $S$ .
- 
- Examples:
    - $\epsilon, \epsilon a, \epsilon aaaaaaaaaa$  where  $a$  is some character in the alphabet  $\Sigma$  ( $a \in \Sigma$ )

# Regular Sets of Strings

- Operations for building sets of strings:

- Alternation

$$S_1 \mid S_2 = \{ s \mid s \in S_1 \vee s \in S_2 \}$$

- Concatenation

$$S_1 \cdot S_2 = \{ s_1 s_2 \mid s_1 \in S_1, s_2 \in S_2 \}$$

- Iteration

$$S^* = \{ \epsilon \} \mid S \mid S \cdot S \mid S \cdot S \cdot S \mid \dots$$

$$= S_0 \mid S_1 \mid S_3 \mid S_3 \mid \dots$$

- A set of strings over  $\Sigma$  is said to be regular if it can be built from the empty set  $\emptyset$  and the singleton set  $\{a\}$  (for each  $a \in \Sigma$ ), using just the operations of alternation, concatenation, and iteration.

# Indexed Sets

- Sets are unordered collections of data elements.
- In order to obtain an ordering relation over the elements of a given set, we can assign each element in that set a unique element of another ordered set I:

$$S_I = \{ a_i \mid a \in S, i \in I \}$$

$S_I$  is called the “indexed set” of  $S$ .

# Some Indexed Sets

- Let  $A = \{ a, b, c, d \}$  and  $I = \mathcal{N}$ , then

$$A_I = \{ a_1, b_2, c_3, d_4 \}$$

- Let  $A = \{ a, b, c, d \}$  and  $I = (S \times S, <)$ , then

$$A_I = \{ a^{<_1}, b^{<_2}, c^{<_3}, d^{<_4} \}$$

# 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:

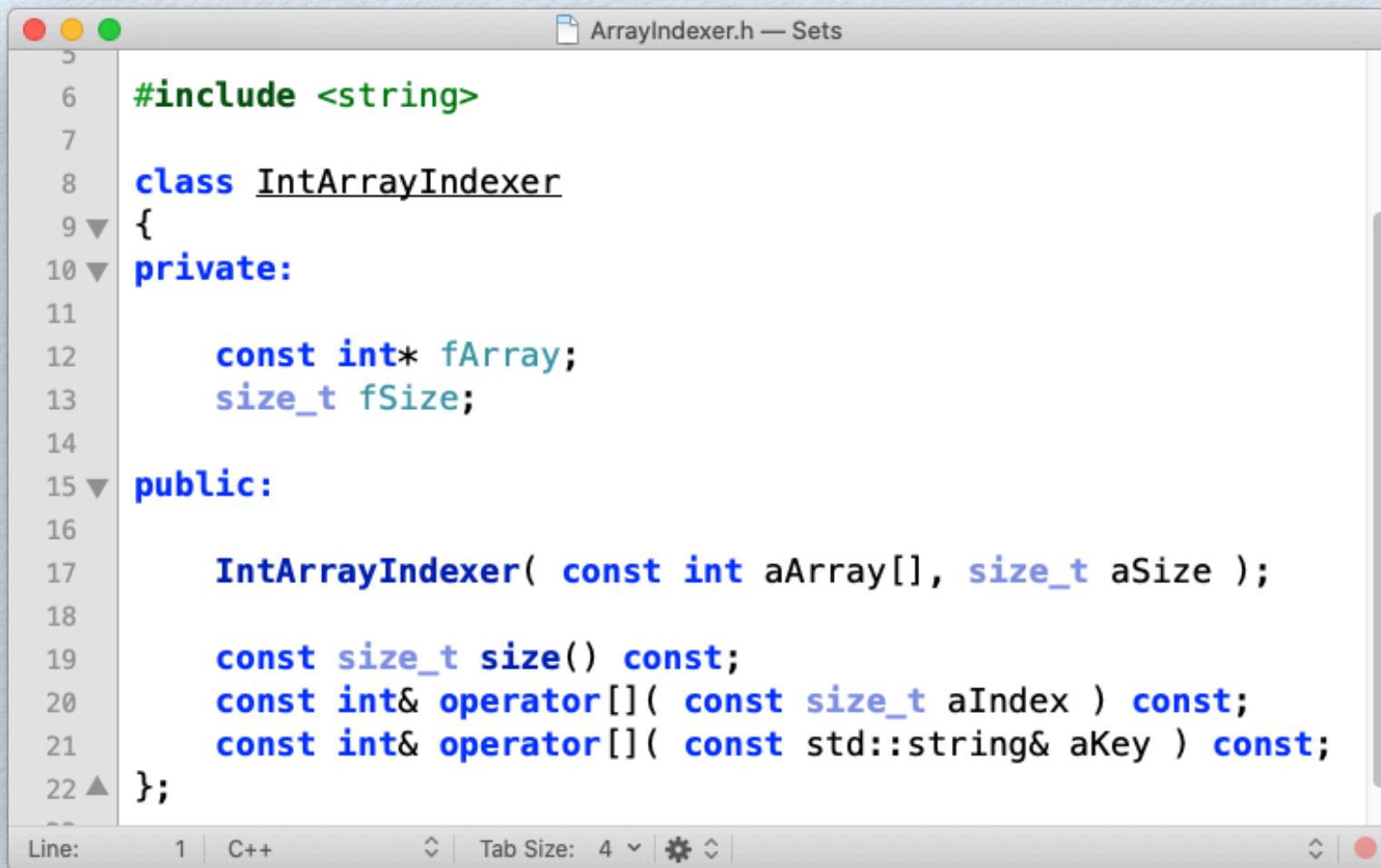
$a = \{ ("u" \mapsto 345), ("v" \mapsto 2), ("w" \mapsto 39), ("x" \mapsto 5) \}$

$a["w"] = 39$

$a["z"] = \perp$

# From Indices to Keys

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



The screenshot shows a code editor window with the title "ArrayIndexer.h — Sets". The code defines a class `IntArrayIndexer` with private members `fArray` and `fSize`, and public methods `size()`, `operator[](size_t aIndex)`, and `operator[](std::string& aKey)`.

```
#include <string>

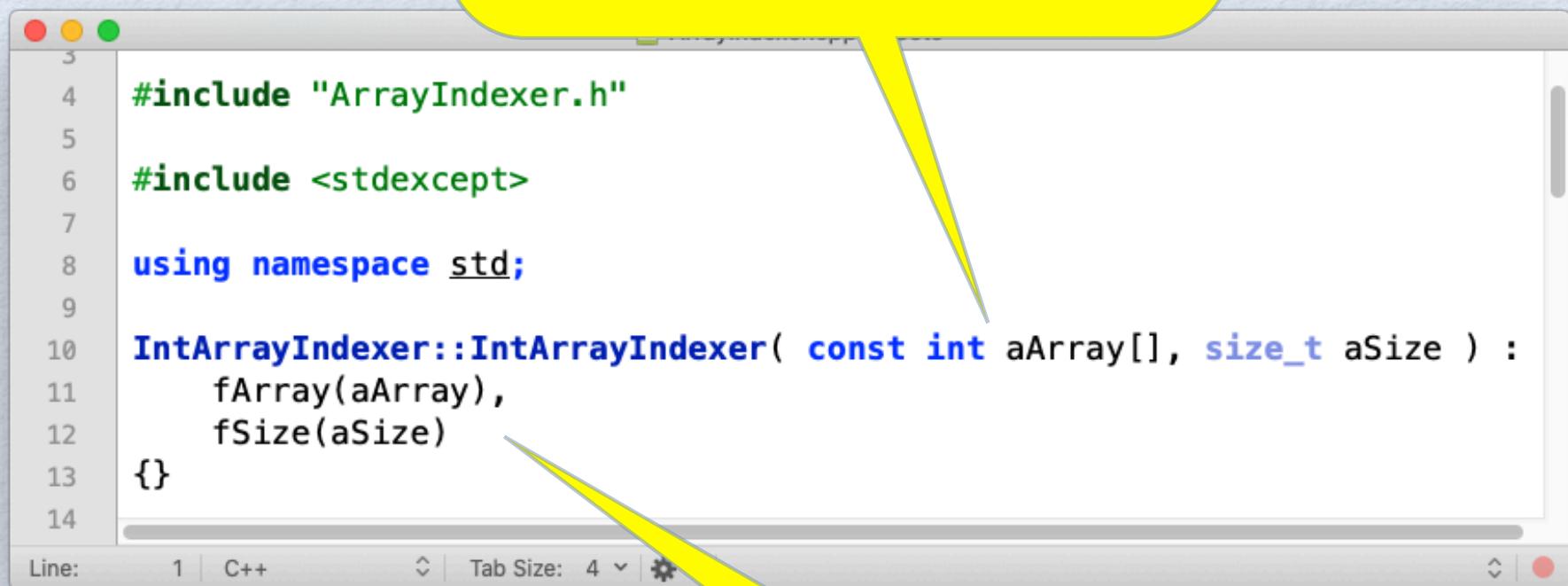
class IntArrayIndexer
{
private:
    const int* fArray;
    size_t fSize;

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;
};
```

# 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
```

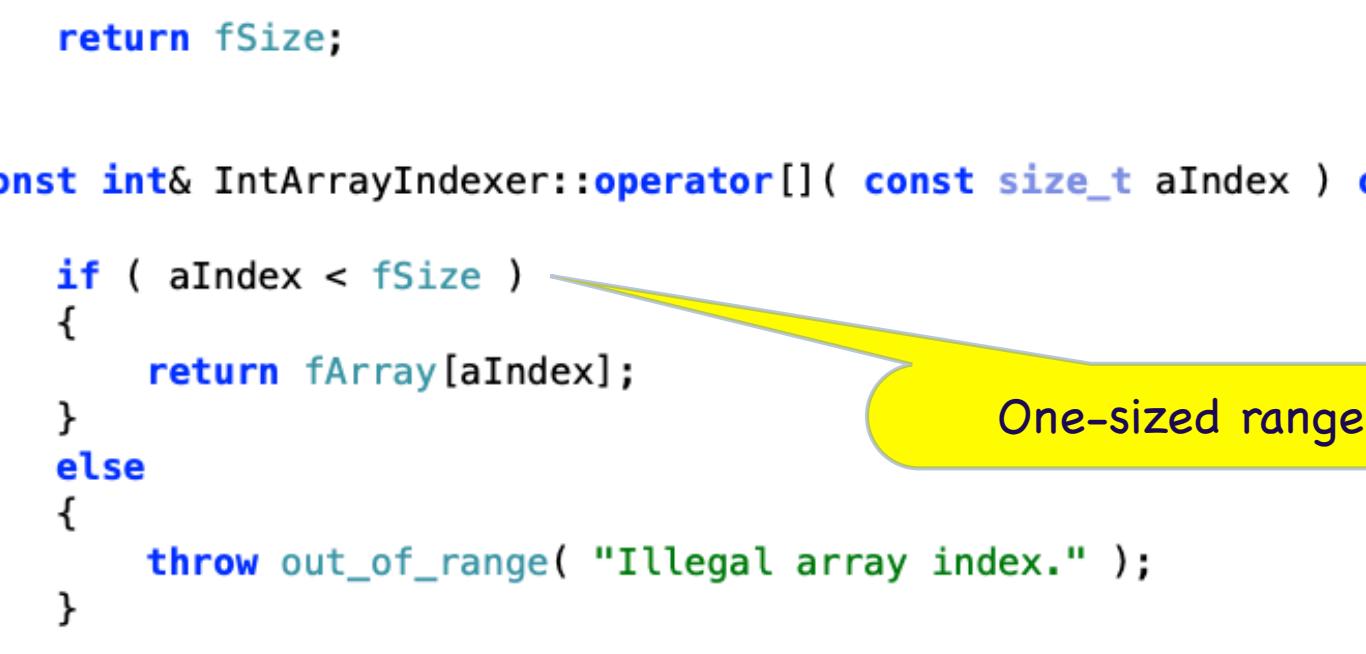
The screenshot shows a code editor window with the following C++ code:

```
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
```

The code defines a constructor for a class named `IntArrayIndexer`. It takes a `const int` array `aArray` and a `size_t` `aSize` as parameters. Inside the constructor, two member variables `fArray` and `fSize` are initialized with the passed values. The code editor interface includes tabs for Line, 1, C++, and Tab Size: 4, along with standard window controls.

We must use member initializer to initialize `const` instance variables!

# Basic Indexer Operations

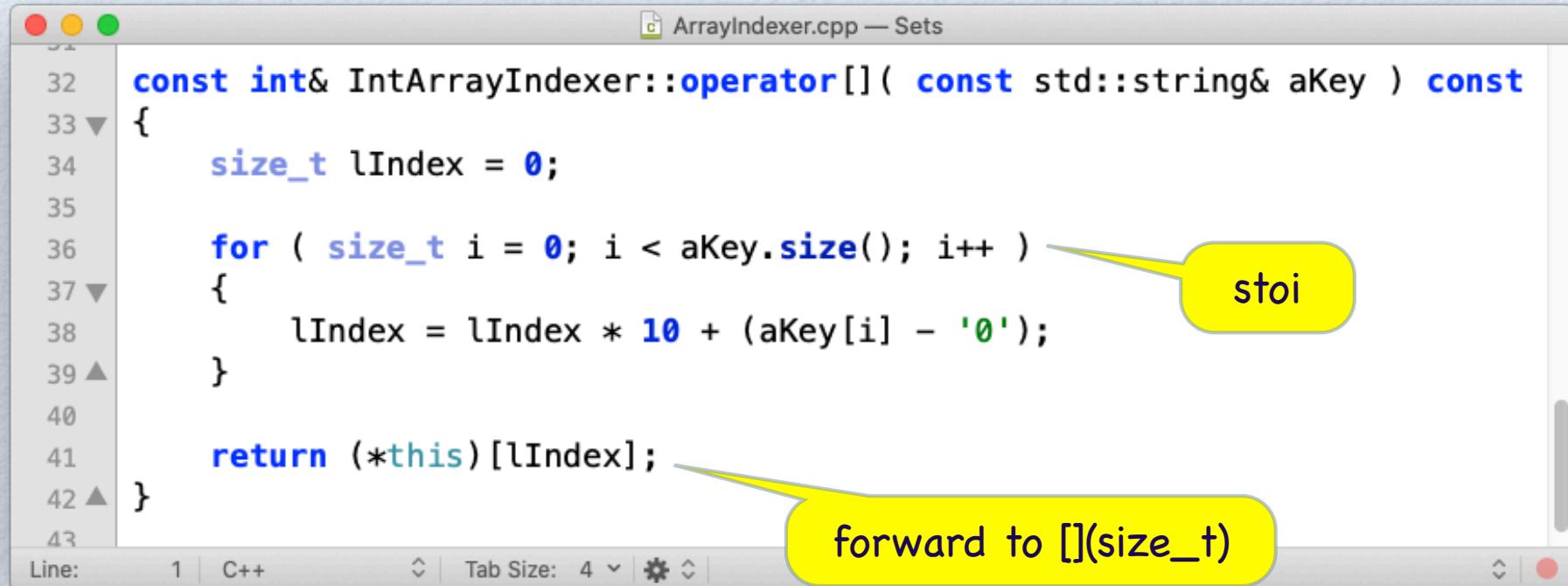


```
15 const size_t IntArrayIndexer::size() const
16 {
17     return fSize;
18 }
19
20 const int& IntArrayIndexer::operator[]( const size_t aIndex ) const
21 {
22     if ( aIndex < fSize )
23     {
24         return fArray[aIndex];
25     }
26     else
27     {
28         throw out_of_range( "Illegal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

One-sized range test.

## 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 }
```

Line: 1 | C++ | Tab Size: 4 |  |  | 

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

The image shows a Mac OS X desktop environment. In the foreground, a code editor window titled "Main.cpp — Sets" displays C++ code. In the background, a terminal window titled "COS3008" shows the output of a program run.

```
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 | ☰ |

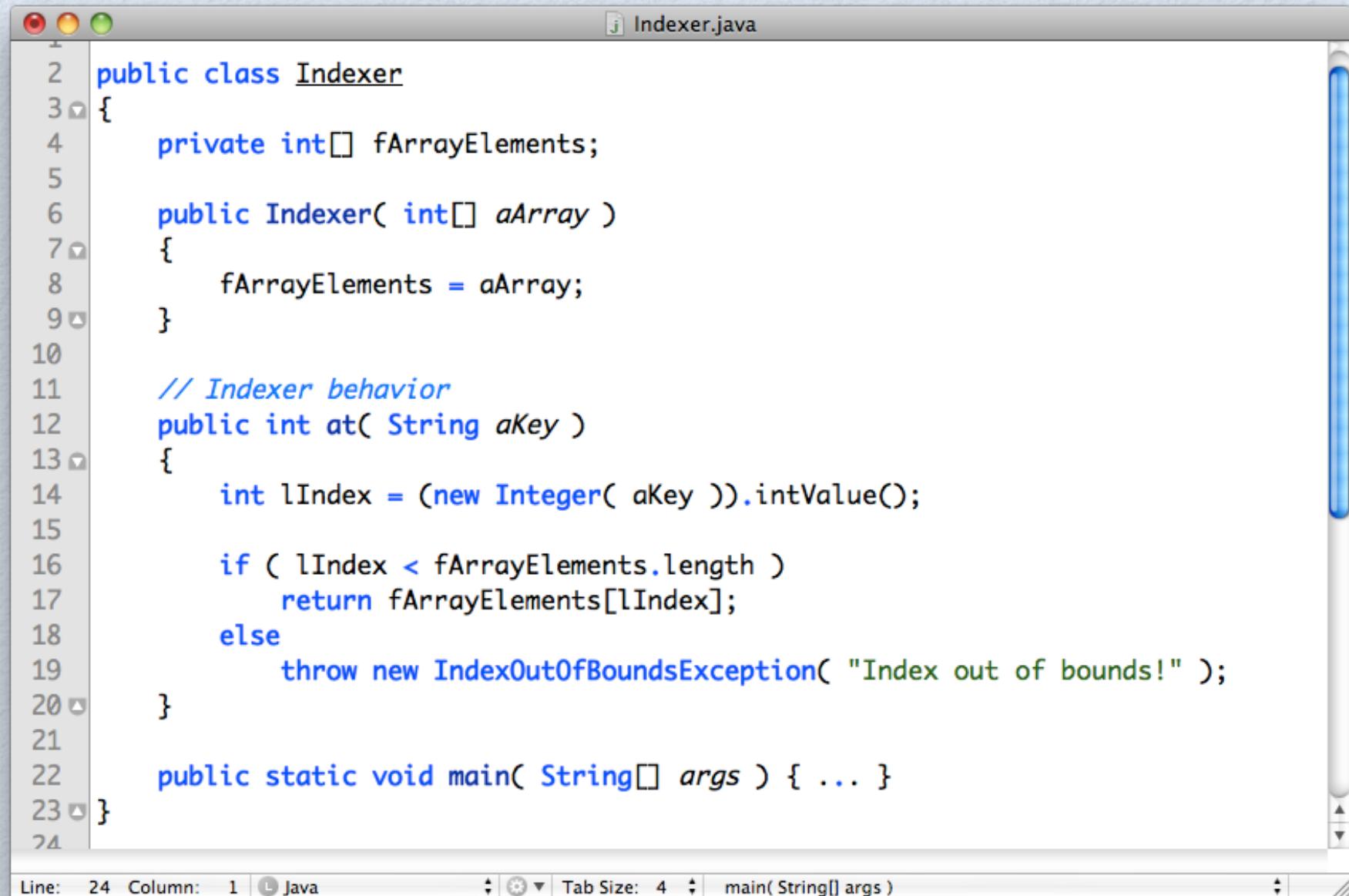
```
Kamala:COS3008 Markus$ ./ArrayIndexer
Indexed sum of [1,2,3,4,5] is 15
Kamala:COS3008 Markus$ _
```

# 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

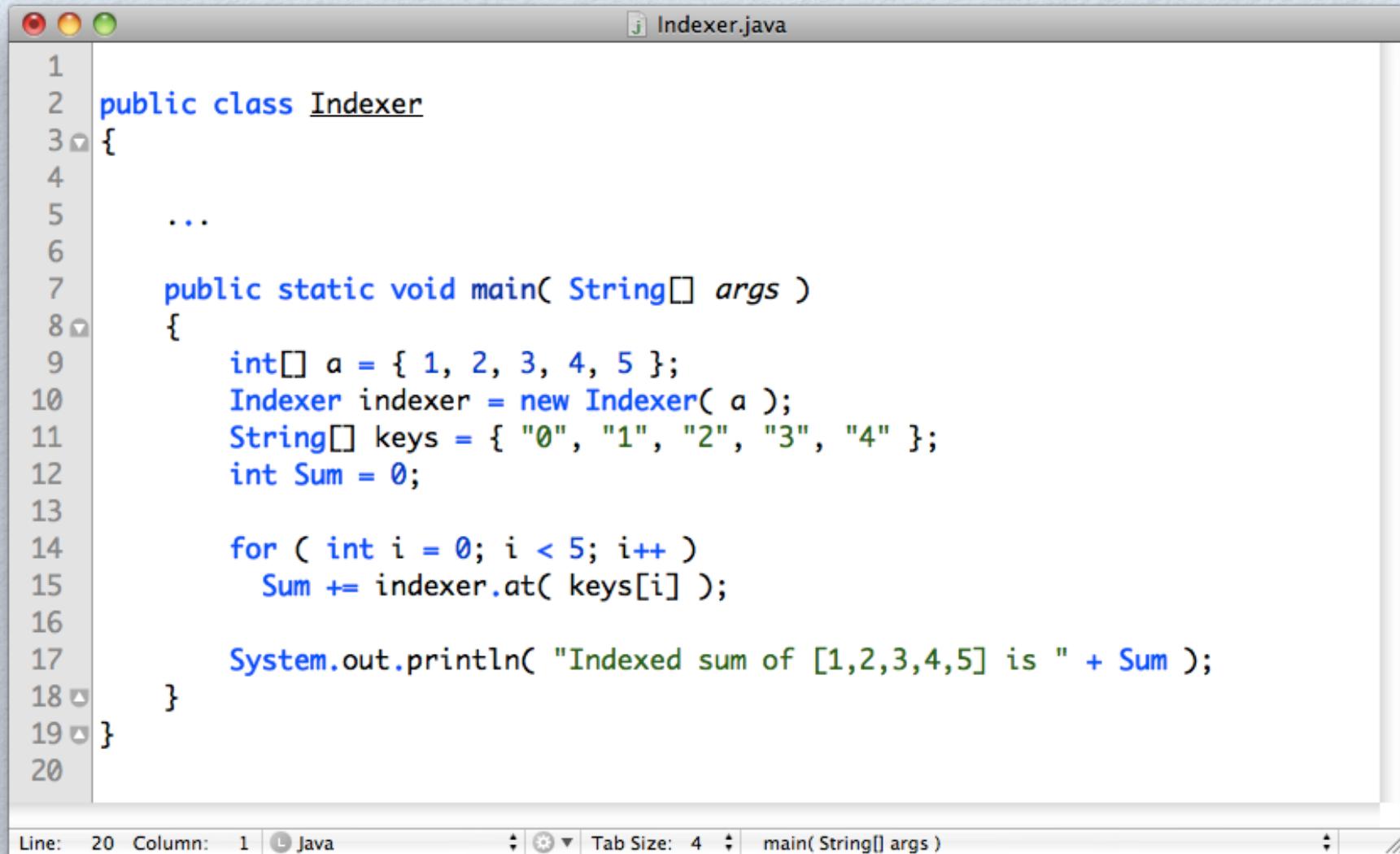


The screenshot shows a Java code editor window titled "Indexer.java". The code defines a class named "Indexer" with a constructor that takes an array of integers and initializes a private field "fArrayElements". It also contains a public method "at" that takes a string key, converts it to an integer index, and returns the corresponding array element. If the index is out of bounds, it throws an "IndexOutOfBoundsException". The code editor interface includes a toolbar, status bar, and scroll bars.

```
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



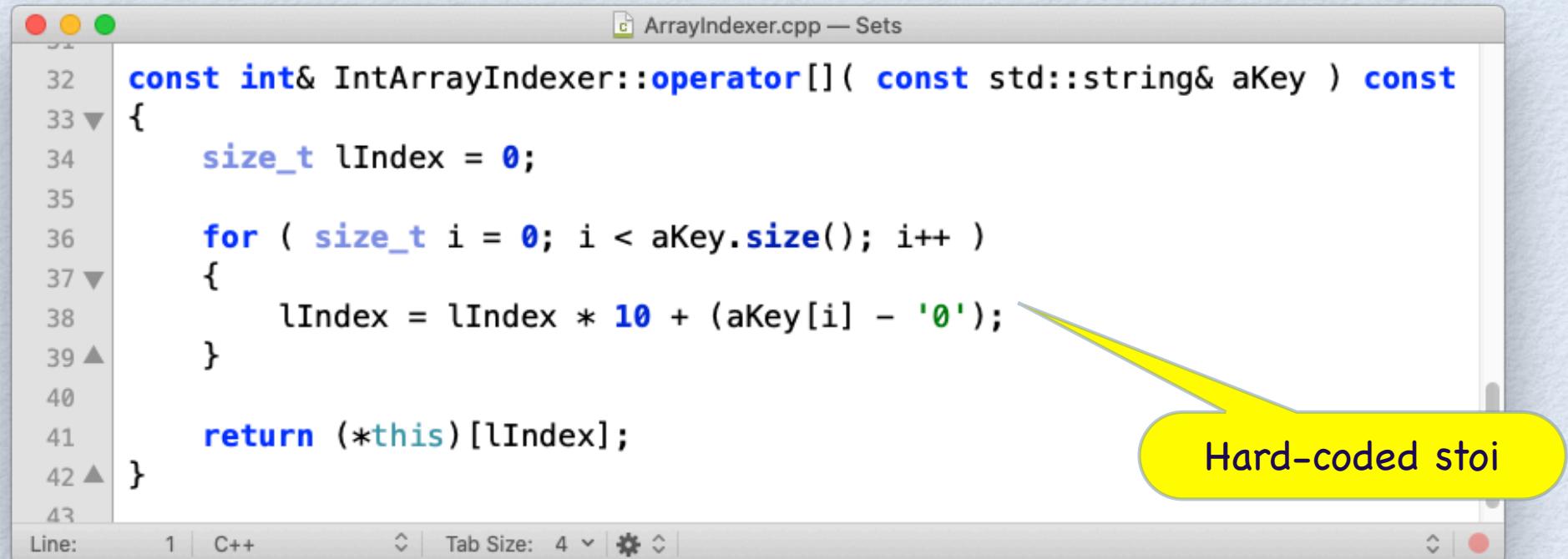
A screenshot of a Java code editor window titled "Indexer.java". The code is as follows:

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

The status bar at the bottom shows "Line: 20 Column: 1 Java Tab Size: 4 main( String[] args )".

# Additional Flexibility: Lambda Expressions

# Hard-coded Conversion



A screenshot of a Mac OS X-style code editor window titled "ArrayIndexer.cpp — Sets". The code in the editor is:

```
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
```

The line "lIndex = lIndex \* 10 + (aKey[i] - '0');" is highlighted with a yellow arrow pointing to a callout bubble. The callout bubble contains the text "Hard-coded stoi".

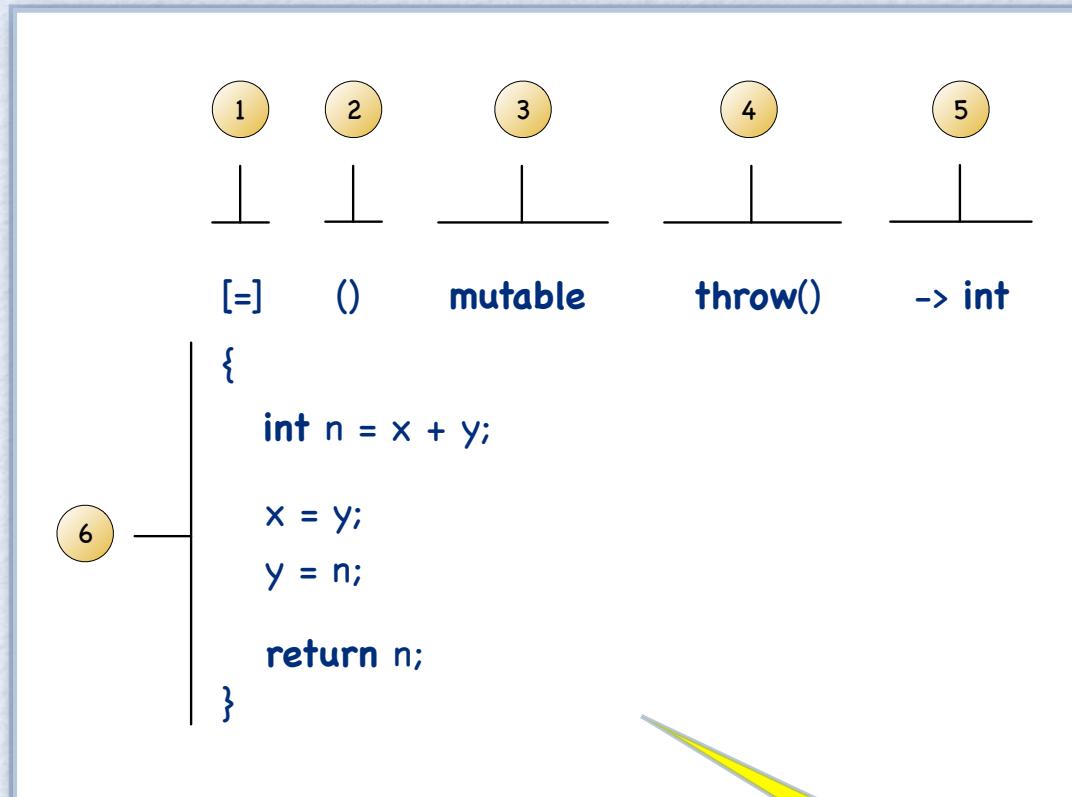
At the bottom of the editor window, there is a toolbar with buttons for Line: 1, C++, Tab Size: 4, and other settings.

- 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

[ ]	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 no 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 <b>this</b> and all names must be preceded by &.

# Keep Lambda Captures Simple

# Indexer with Lambda

The screenshot shows a code editor window with the file `ArrayIndexerWithLambda.h`. The code defines a class `IntArrayIndexer` with a public `get` method. The `get` method takes a string key and returns an integer index by iterating through a string number and converting it to an integer. A yellow callout bubble points to the `get` method with the text "conversion function as lambda and default values the header". Another yellow callout bubble points to the `operator[]` declaration with the text "new get function, operator[] expects one argument only".

```
23
24
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
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
778
779
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
824
825
826
827
827
828
829
829
830
831
832
833
834
835
836
837
837
838
839
839
840
841
842
843
844
845
846
847
847
848
849
849
850
851
852
853
854
855
856
857
857
858
859
859
860
861
862
863
864
865
866
866
867
868
868
869
869
870
871
872
873
874
875
875
876
876
877
877
878
878
879
879
880
881
882
883
884
885
885
886
886
887
887
888
888
889
889
890
891
892
893
894
895
895
896
896
897
897
898
898
899
899
900
901
902
903
904
905
905
906
906
907
907
908
908
909
909
910
911
912
913
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1580
1581
1581
1582
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1590
1591
1591
1592
1592
1593
1593
1594
1594
1595
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1600
1601
1601
1602
1602
1603
1603
1604
1604
1605
1605
1606
1606
1607
1607
1608
1608
1609
1609
1610
1610
1611
1611
1612
1612
1613
1613
1614
1614
1615
1615
1616
1616
1617
1617
1618
1618
1619
1619
1620
1620
1621
1621
1622
1622
1623
1623
1624
1624
1625
1625
1626
1626
1627
1627
1628
1628
1629
1629
1630
1
```

# Implementation of get()



A screenshot of a C++ IDE showing the code for the `get` method. The code is as follows:

```
const int& IntArrayIndexer::get( const std::string& aKey, StringMap aFunc ) const
{
    return (*this)[aFunc(aKey)];
}
```

The IDE interface includes a title bar "ArrayIndexerWithLambda.cpp — SetDefault", a status bar with "Line: 30 | C++ | Tab Size: 4 | IntArrayIndexer::size", and a toolbar with various icons.

forward to indexer

call lambda for conversion from string to size\_t

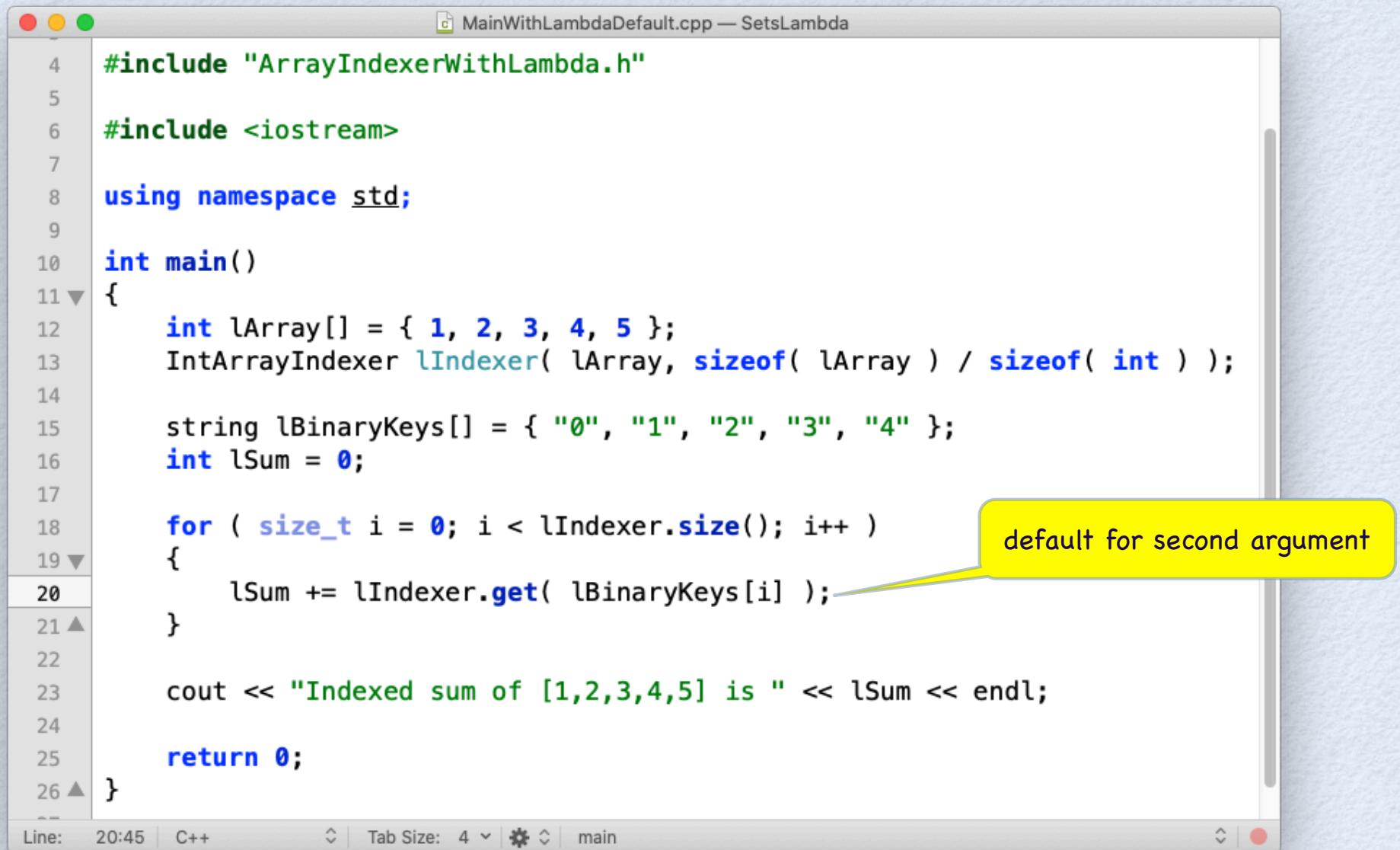
# `std::function<class Ret, class... Args>`

- 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



```
>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 }
```

A yellow callout box points to the line of code `lIndexer.get( lBinaryKeys[i] );`. The text "default for second argument" is contained within the callout box.

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 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

```
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.**

# Systematic Traversal of Sets

# Iterators

- We can use a loop statement and a loop counter to traverse all elements of an array in sequence.
- However, not all data types are arrays and simple indexing may not suffice.
- Iterators offer programmers a suitable alternative to define traversal in a data type agnostic way.
- Conceptually, iterators are objects that implement the necessary infrastructure to iterate over elements of a sequence. They do this via a common interface.
- The C++ ecosystem has popularized and uses five types of iterators. Iterator objects have the look-and-feel of pointers, and advancing an iterator means to increment/decrement a pointer-like object.

# C++ Iterators

Input Iterator

Output Iterator

Forward Iterator

Bidirectional Iterator

Random Access Iterator

# Abilities of Iterators

Iterator Category	Ability	Provider
Input Iterator	Read forward	istream
Output Iterator	Write forward	ostream, inserter
Forward Iterator	Read and write forward	
Bidirectional Iterator	Read and write forward and backward	list, set, multiset, map, multimap, vector, deque, string, array
Random Access Iterator	Read and write with random access	

# Input Iterator

Expression	Effect
<code>*iter</code>	Provides read access to the actual element
<code>iter-&gt;member</code>	Provides read access to a member of the actual element
<code>++iter</code>	Steps forward (returns new position)
<code>iter++</code>	Steps forward (returns old position)
<code>iter1 == iter2</code>	Returns whether iter1 and iter2 are equal
<code>iter1 != iter2</code>	Returns whether iter1 and iter2 are not equal

# Output Iterator

Expression	Effect
<code>*iter = value</code>	Provides write access to the actual element
<code>++iter</code>	Steps forward (returns new position)
<code>iter++</code>	Steps forward (returns old position)

An output iterator is like a “black hole.”

# Forward Iterator

Expression	Effect
<code>*iter</code>	Provides read access to the actual element
<code>iter-&gt;member</code>	Provides read access to a member of the actual element
<code>++iter</code>	Steps forward (returns new position)
<code>iter++</code>	Steps forward (returns old position)
<code>iter1 == iter2</code>	Returns whether iter1 and iter2 are equal
<code>iter1 != iter2</code>	Returns whether iter1 and iter2 are not equal
<code>iter1 = iter2</code>	Assigns an iterator

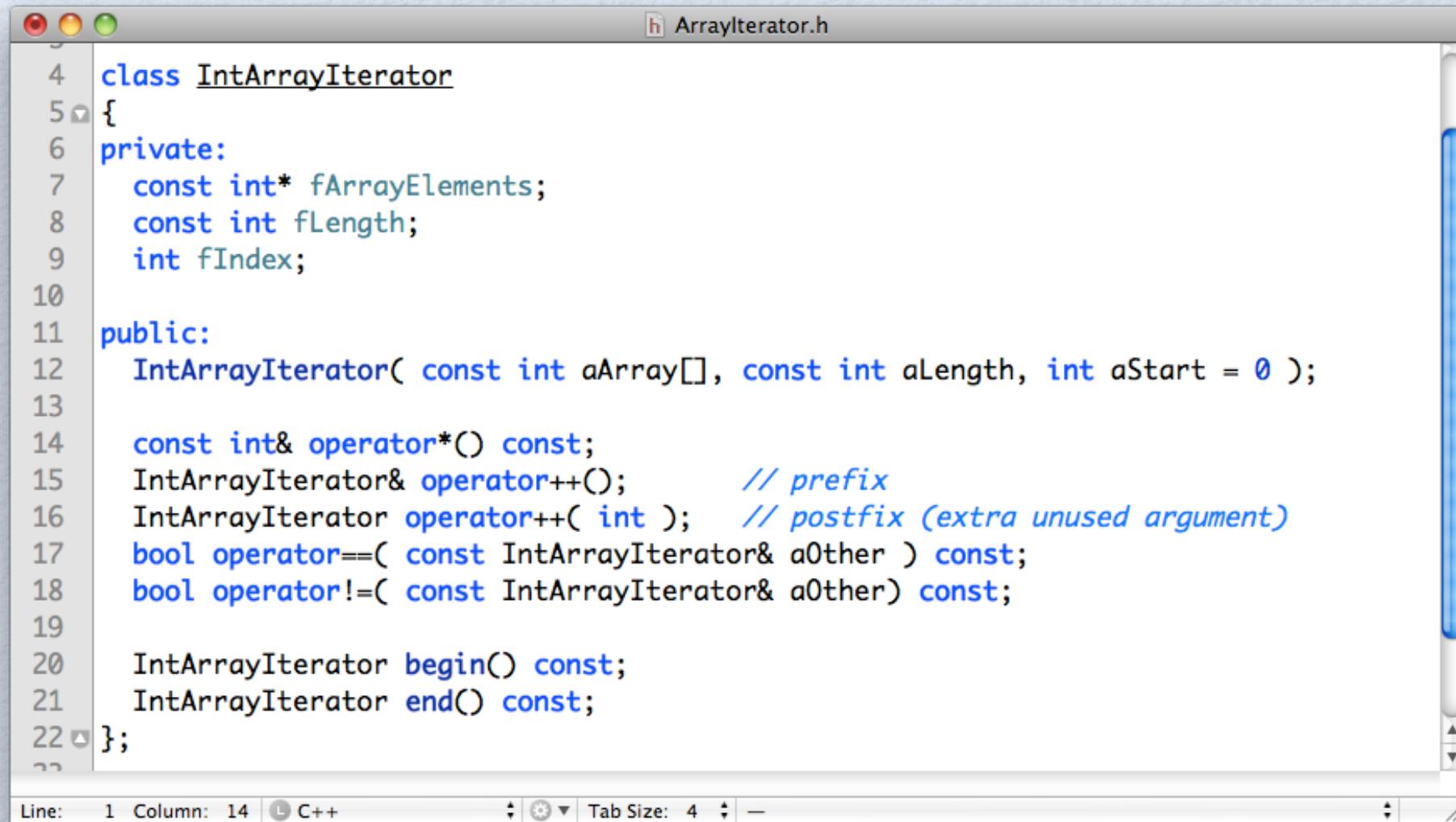
# Bidirectional Iterator

Expression	Effect
<code>*iter</code>	Provides read access to the actual element
<code>iter-&gt;member</code>	Provides read access to a member of the actual element
<code>++iter</code>	Steps forward (returns new position)
<code>iter++</code>	Steps forward (returns old position)
<code>--iter</code>	Steps backward (returns new position)
<code>iter--</code>	Steps backward (returns old position)
<code>iter1 == iter2</code>	Returns whether iter1 and iter2 are equal
<code>iter1 != iter2</code>	Returns whether iter1 and iter2 are not equal
<code>iter1 = iter2</code>	Assigns an iterator

# Random Access Iterator

Expression	Effect
<code>iter[n]</code>	Provides read access to the element at index n
<code>iter += n</code>	Steps n elements forward or backward
<code>iter -= n</code>	Steps n elements forward or backward
<code>n+iter</code>	Returns the iterator of the nth next element
<code>n-iter</code>	Returns the iterator of the nth previous element
<code>iter - iter2</code>	Returns disjoint distance between iter1 and iter2
<code>iter1 &lt; iter2</code>	Returns whether iter1 is before iter2
<code>iter1 &gt; iter2</code>	Returns whether iter1 is after iter2
<code>iter1 &lt;= iter2</code>	Returns whether iter1 is not after iter2
<code>iter1 &gt;= iter2</code>	Returns whether iter1 is not before iter2

# A Read-Only Forward Iterator



The screenshot shows a code editor window titled "ArrayIterator.h". The code defines a class `IntArrayIterator` with private members `fArrayElements`, `fLength`, and `fIndex`. It includes a constructor, several operator overloads (prefix and postfix ++, ==, !=), and two const member functions `begin()` and `end()`. The code is written in C++ and uses standard syntax highlighting.

```
4 class IntArrayIterator
5 {
6     private:
7         const int* fArrayElements;
8         const int fLength;
9         int fIndex;
10
11    public:
12        IntArrayIterator( const int aArray[], const int aLength, int aStart = 0 );
13
14        const int& operator*() const;
15        IntArrayIterator& operator++();           // prefix
16        IntArrayIterator operator++( int );      // postfix (extra unused argument)
17        bool operator==( const IntArrayIterator& aOther ) const;
18        bool operator!=( const IntArrayIterator& aOther) const;
19
20        IntArrayIterator begin() const;
21        IntArrayIterator end() const;
22};
```

Line: 1 Column: 14 C++ Tab Size: 4 —

# Forward Iterator Constructor

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

We must not repeat the default value.

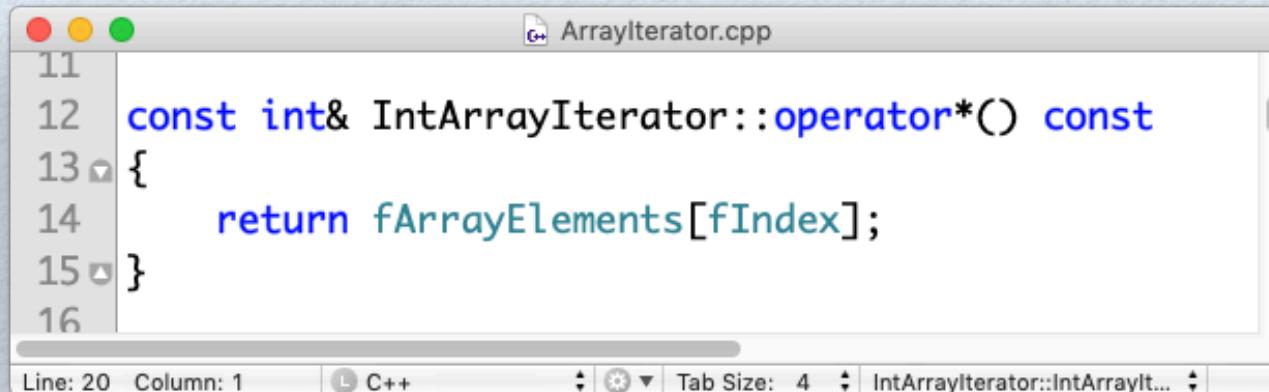
The screenshot shows a code editor window titled "ArrayIterator.cpp". The code is as follows:

```
1 #include <iostream>
2 #include "ArrayIterator.h"
3
4 using namespace std;
5
6 IntArrayIterator::IntArrayIterator( const int aArray[], const int aLength, int aStart ) :
7     fArrayElements(aArray), fLength(aLength)
8 {
9     fIndex = aStart;
10}
11
```

The code editor interface includes a toolbar at the top with icons for file operations, and a status bar at the bottom showing "Line: 48 Column: 28 C++". Three yellow callout bubbles point from the text in the first two paragraphs to the corresponding parts of the code: one points to the parameter declarations, another to the member initializer, and a third to the assignment statement.

We must use member initializer to initialize const instance variables!

# The Dereference Operator



A screenshot of a Mac OS X-style code editor window titled "ArrayIterator.cpp". The code implements the dereference operator for an iterator class. The code is as follows:

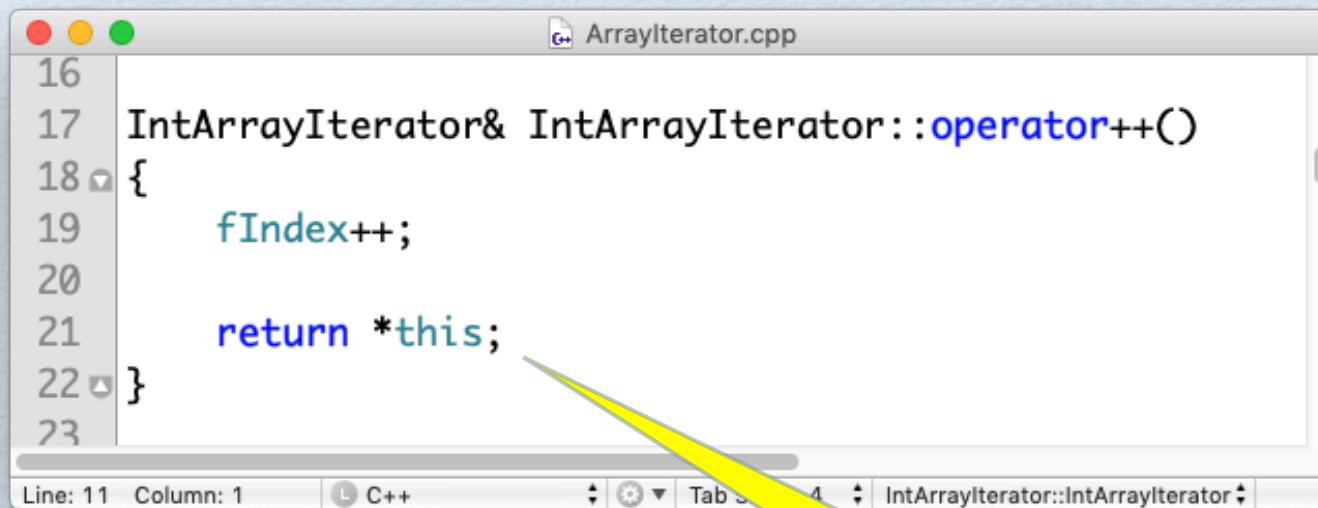
```
11
12 const int& IntArrayIterator::operator*() const
13 {
14     return fArrayElements[fIndex];
15 }
16
```

The editor shows syntax highlighting for C++ code, with blue for keywords like `const`, `int`, and `operator*`. The file tab at the bottom right shows "IntArrayIterator::IntArrayIt...".

- The dereference operator returns the element the iterator is currently positioned on.
- The dereference operator is a `const` operation, that is, it does not change any instance variables of the iterator.
- We use a `const` reference to avoid copying the original value stored in the underlying collection.
- No range check is required. The `operator*()` should only be called if the iterator has not yet reached the end of the underlying collection.

# Prefix Increment

- The prefix increment operator advances the iterator and returns a reference of this iterator.



```
16
17 IntArrayIterator& IntArrayIterator::operator++()
18 {
19     fIndex++;
20
21     return *this;
22 }
23
```

The screenshot shows a code editor window titled "ArrayIterator.cpp". The code implements the prefix increment operator for a class named "IntArrayIterator". The operator increments the private member variable "fIndex" and returns a reference to the current object ("\*this").

Return a reference to the current iterator (set forward).

# Postfix Increment

- The postfix increment operator advances the iterator and returns a copy of the old iterator.

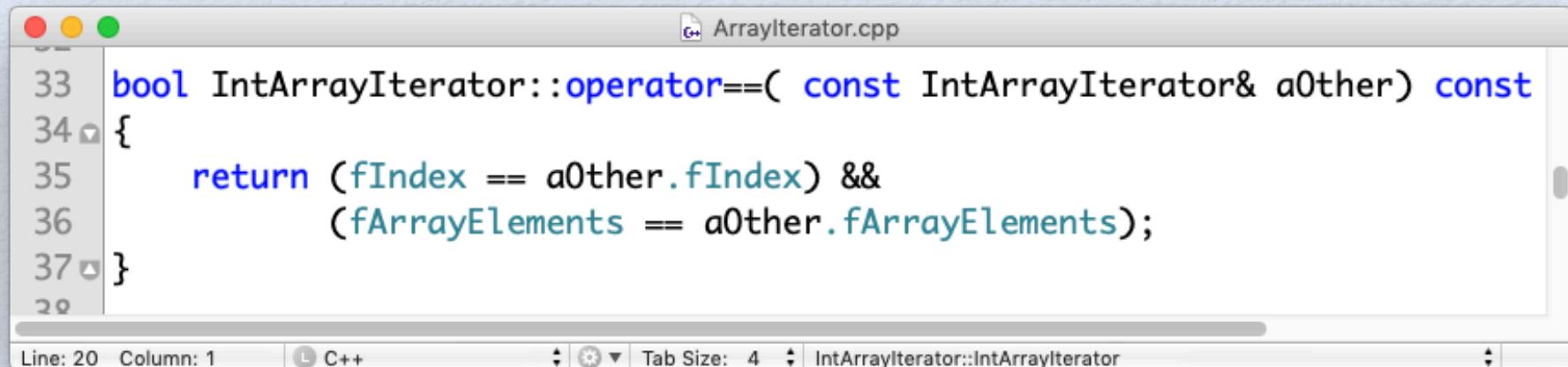
A screenshot of a Mac OS X-style code editor window titled "ArrayIterator.cpp". The code implements the postfix increment operator for an iterator class:

```
24 IntArrayIterator IntArrayIterator::operator++( int )
25 {
26     IntArrayIterator temp = *this;
27
28     ++(*this); // reuse implementation
29
30     return temp;
31 }
```

The code uses a temporary variable `temp` to store the current value of `*this`. It then increments `*this` using the prefix increment operator `++(*this)`. Finally, it returns a copy of the original iterator `temp`.

Return a copy of the old iterator  
(position unchanged).

# Iterator Equivalence



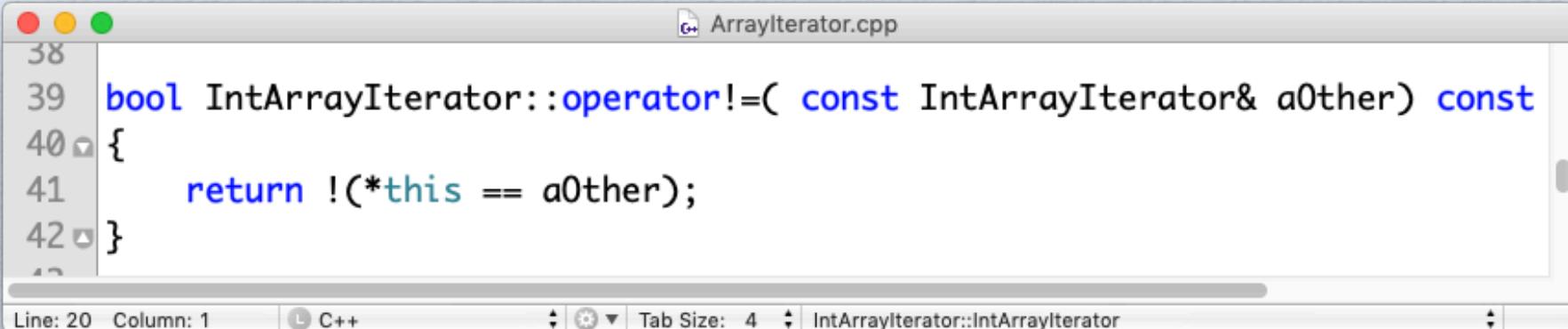
```
33 bool IntArrayIterator::operator==( const IntArrayIterator& aOther) const
34 {
35     return (fIndex == aOther.fIndex) &&
36         (fArrayElements == aOther.fArrayElements);
37 }
```

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

Two iterators are equal if and only if they refer to the same element (this may require considering the context of ==):

- fIndex is the current index into the array
- Arrays are passed as a pointer to the first element (arrays decay to pointers) that is constant throughout runtime.

# Iterator Inequality

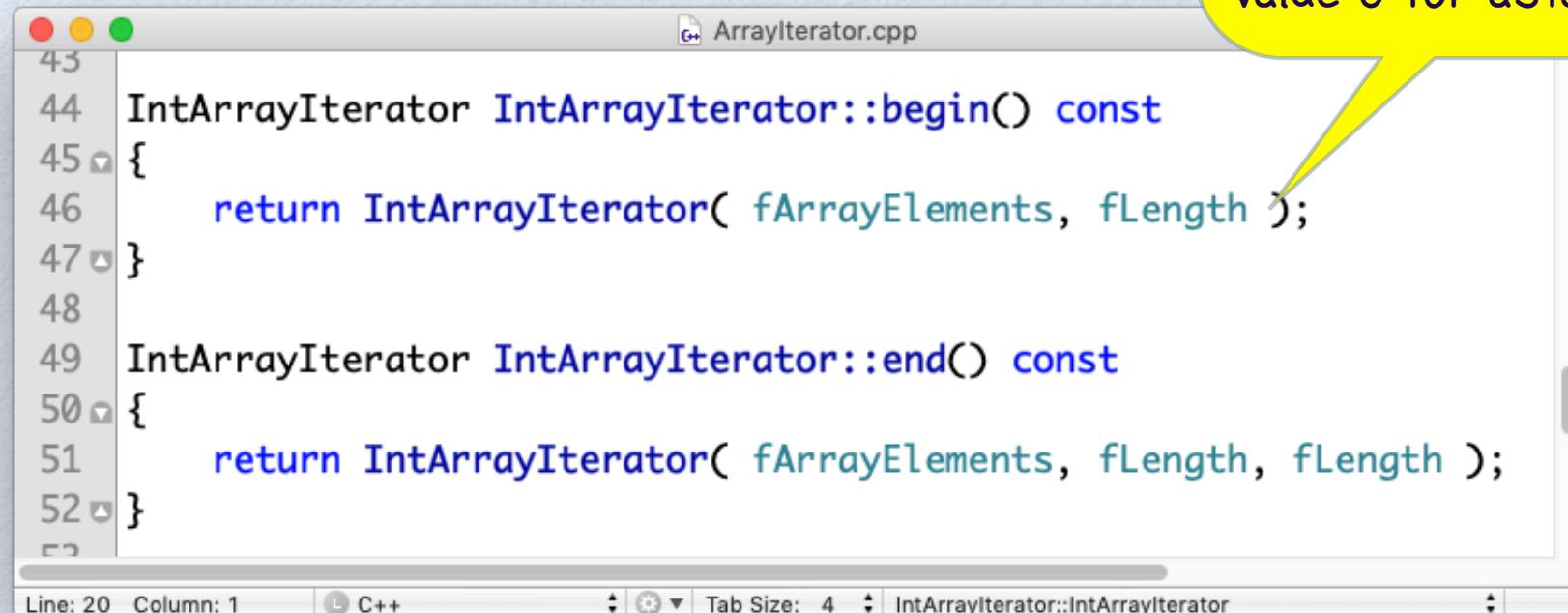


```
38
39 bool IntArrayIterator::operator!=( const IntArrayIterator& aOther) const
40 {
41     return !(*this == aOther);
42 }
```

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

We implement != in terms of ==.

# Auxiliary Methods



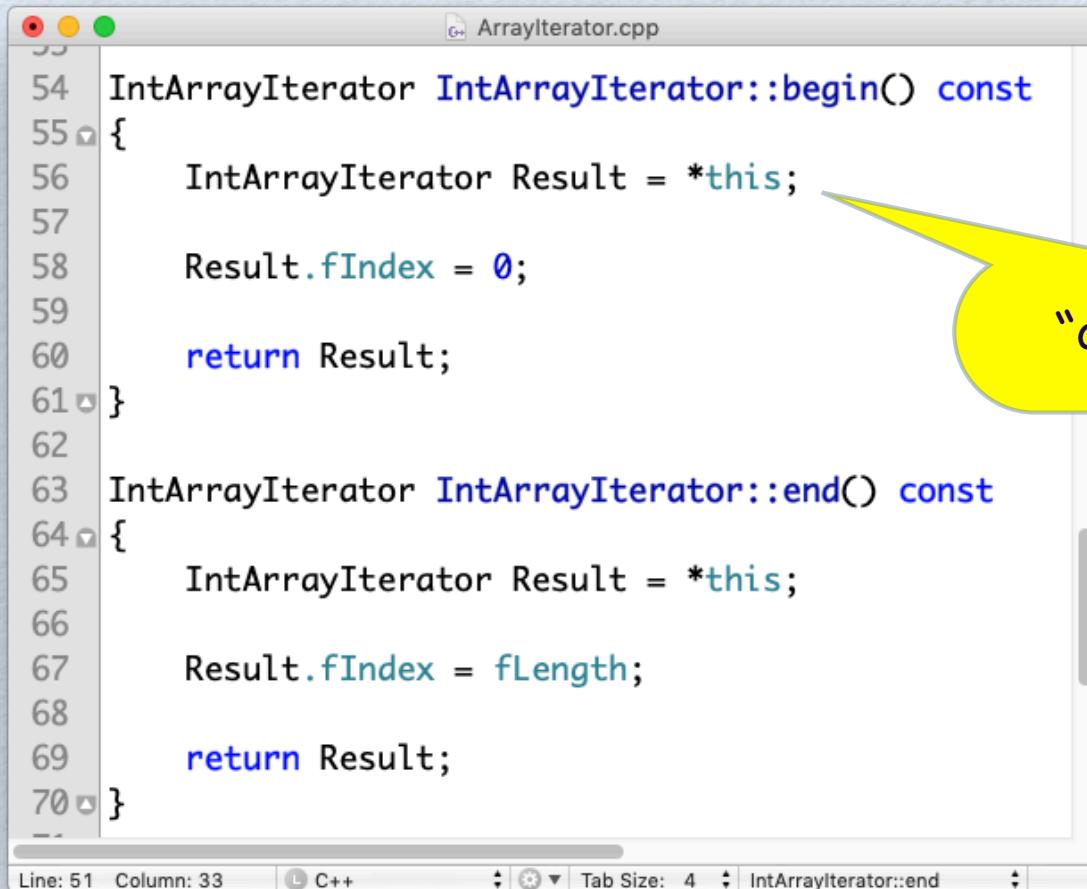
The screenshot shows a code editor window titled "ArrayIterator.cpp". The code contains two methods: `IntArrayIterator IntArrayIterator::begin() const` and `IntArrayIterator IntArrayIterator::end() const`. Both methods return an `IntArrayIterator` object initialized with `fArrayElements` and `fLength`. A yellow speech bubble points to the first method with the text: "We use the default value 0 for aStart here."

```
43
44 IntArrayIterator IntArrayIterator::begin() const
45 {
46     return IntArrayIterator( fArrayElements, fLength );
47 }
48
49 IntArrayIterator IntArrayIterator::end() const
50 {
51     return IntArrayIterator( fArrayElements, fLength, fLength );
52 }
```

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

- The methods `begin()` and `end()` return fresh iterators set to the first element and past-the-end element, respectively.
- The names and implementation of these auxiliary methods follow standard practices. The compiler may look for them when you use a `for-each` loop.

# Auxiliary Methods: Copy This Object



```
55
54     IntArrayIterator IntArrayIterator::begin() const
55     {
56         IntArrayIterator Result = *this;
57
58         Result.fIndex = 0;
59
60         return Result;
61     }
62
63     IntArrayIterator IntArrayIterator::end() const
64     {
65         IntArrayIterator Result = *this;
66
67         Result.fIndex = fLength;
68
69         return Result;
70     }
```

Line: 51 Column: 33 C++ Tab Size: 4 IntArrayIterator::end

“clone” this object

- The methods “clone” this iterator object and set the position accordingly.

# Putting Everything Together

The screenshot shows a Mac OS X desktop environment. In the foreground, a code editor window titled "ArrayIterator.cpp" displays the following C++ code:

```
52
53 int main()
54 {
55     int a[] = { 1, 2, 3, 4, 5 };
56     int Sum = 0;
57
58     for ( IntArrayIterator iter( a, 5 ); iter != iter.end(); iter++ )
59         Sum += *iter;
60
61     cout << "Iterated sum of [1,2,3,4,5] is " << Sum << endl;
62
63     return 0;
64 }
65
```

Below the code editor, the status bar shows: Line: 45 Column: 1 C++ Tab Size: 4 IntArrayIterator::begin.

In the background, a terminal window titled "COS3008" is open, showing the output of the program:

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

# Can we do better?

# C++11: For-Each-Loop

- The traditional for statement in C++ reads:

```
for ( init-statement; condition; expression )  
    statement
```

This form uses explicit loop variables, conditions, and increments over loop variables.

- C++11 introduces a simpler form, called range for statement, to iterate through the elements of a container or other sequence:

```
for ( declaration : expression )  
    statement
```

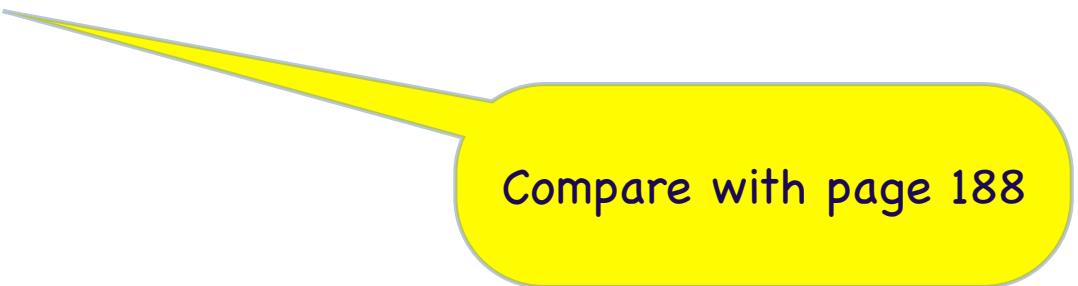
This form is called for-each, and expression must denote a sequence and declaration defines a variable, set in each step of the iteration.

# Understanding C++11's range loop

## for ( declaration : expression ) statement

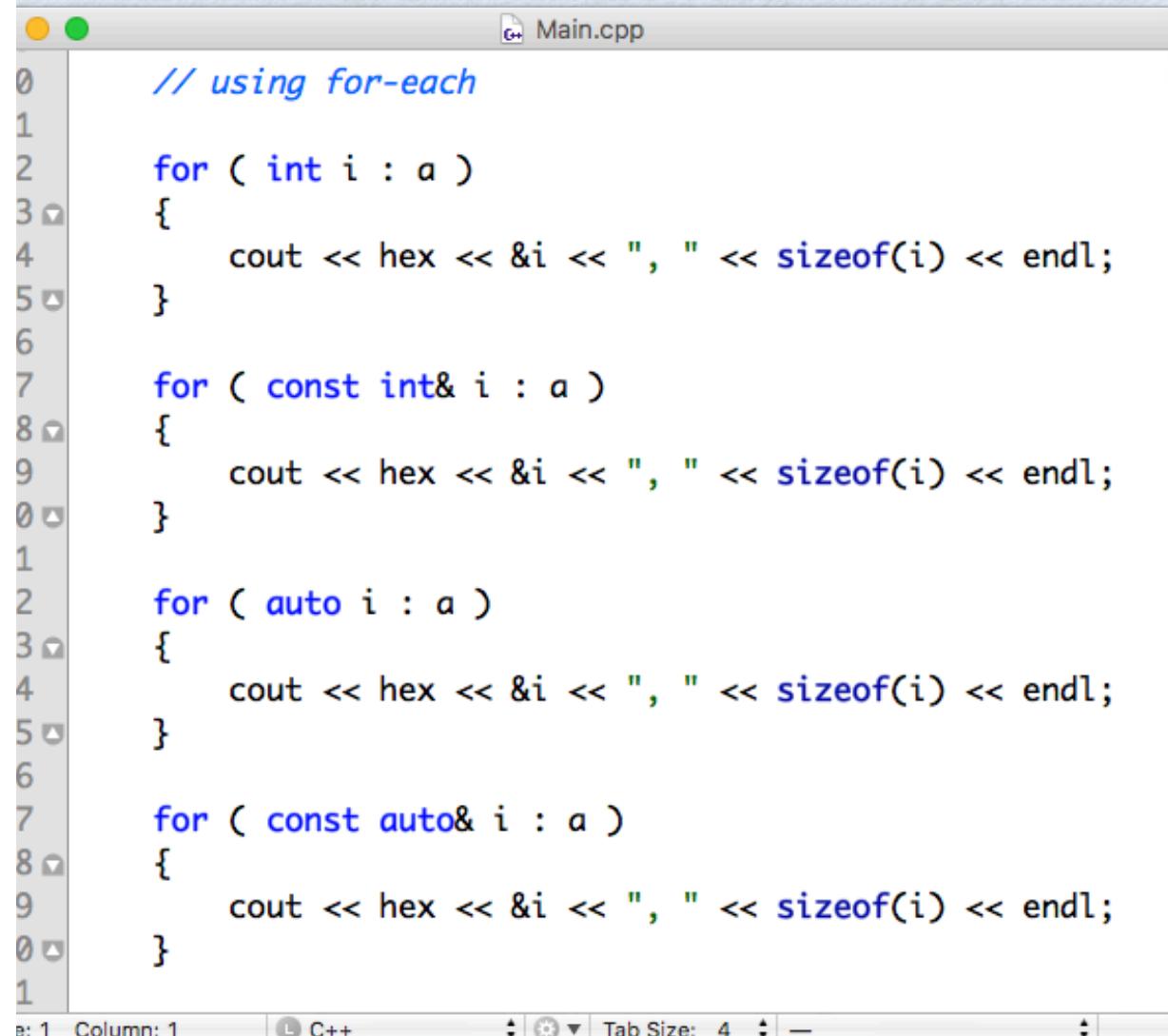
- According to the standard, this is equivalent to the following plain for loop:

```
auto&& __range = expression;           // C++11 forwarding (move)
for ( auto __begin = begin-expression,   // begin()
      __end = end-expression;           // end()
      __begin != __end;
      ++__begin )
{
    declaration = *__begin;
    statement;
}
```



Compare with page 188

# Using C++11's For-Each-Loop



```
// using for-each

0 for ( int i : a )
1 {
2     cout << hex << &i << ", " << sizeof(i) << endl;
3 }
4
5 for ( const int& i : a )
6 {
7     cout << hex << &i << ", " << sizeof(i) << endl;
8 }
9
10 for ( auto i : a )
11 {
12     cout << hex << &i << ", " << sizeof(i) << endl;
13 }
14
15 for ( const auto& i : a )
16 {
17     cout << hex << &i << ", " << sizeof(i) << endl;
18 }
```

- Case 1: read-write variable
- Case 2: constant reference
- Case 3: auto variable
- Case 4: constant auto reference

# For-Each-Loop Behavior

The screenshot shows a C++ code editor window titled "Main.cpp". The code contains five for-each loops, each printing the memory address and size of its iteration variable. The loops use different iterator types: standard int, const int&, auto, const auto&, and const int&.

```
30 // using for-each
31
32     for ( int i : a )
33     {
34         cout << hex << &i << ", " << sizeof(i) << endl;
35     }
36
37     for ( const int& i : a )
38     {
39         cout << hex << &i << ", " << sizeof(i) << endl;
40     }
41
42     for ( auto i : a )
43     {
44         cout << hex << &i << ", " << sizeof(i) << endl;
45     }
46
47     for ( const auto& i : a )
48     {
49         cout << hex << &i << ", " << sizeof(i) << endl;
50     }
51 
```

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

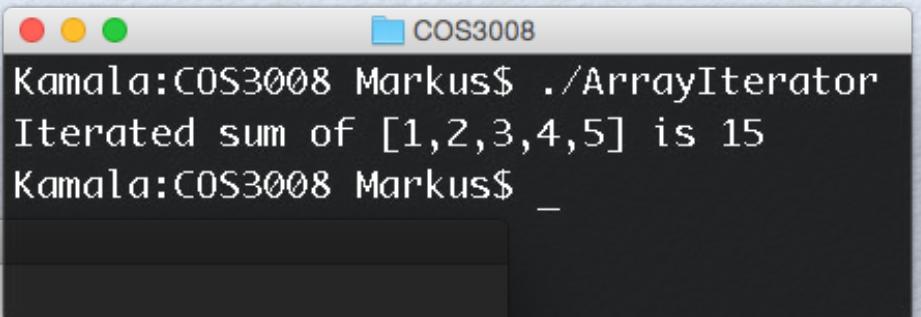
The screenshot shows a terminal window titled "Debug" running the command "./ArrayIterator". The output displays 16 lines of memory addresses and their sizes, all starting with 0x7fff5a95e5dc, which corresponds to the starting address of the integer array "a".

```
./ArrayIterator
0x7fff5a95e5dc, 4
0x7fff5a95e5dc, 4
0x7fff5a95e5dc, 4
0x7fff5a95e5dc, 4
0x7fff5a95e5dc, 4
0x7fff5a95e5dc, 4
0x7fff5a95e6f0, 4
0x7fff5a95e6f4, 4
0x7fff5a95e6f8, 4
0x7fff5a95e6fc, 4
0x7fff5a95e700, 4
0x7fff5a95e59c, 4
0x7fff5a95e59c, 4
0x7fff5a95e59c, 4
0x7fff5a95e59c, 4
0x7fff5a95e59c, 4
0x7fff5a95e6f0, 4
0x7fff5a95e6f4, 4
0x7fff5a95e6f8, 4
0x7fff5a95e6fc, 4
0x7fff5a95e700, 4
```

# Which declaration to use?

- In a for-each loop always use a reference variable. This avoids unnecessary copies.
- Prefer auto to explicit type declarations. Iterator types can be quite complex and hard to express. Using auto – automatic type deduction – simplifies things dramatically.
- You still need to understand what type deduction means and what the results are. Code becomes less readable, as fewer explicit detail is available.

# Iterator Idiom at Work



```
9 int main()
10 {
11     int a[] = { 1, 2, 3, 4, 5 };
12     int Sum = 0;
13
14     for( const auto& i : IntArrayIterator( a, 5 ) )
15     {
16         Sum += i;
17     }
18
19     cout << "Iterated sum of [1,2,3,4,5] is " << Sum << endl;
20
21     return 0;
22 }
```

Line: 42 Column: 6 C++ Tab Size: 4 main

# A Note on `auto`

- 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.
- The use of `auto` can hamper program comprehension. We may have to perform an in-depth study of the code base to understand what type we are dealing with and which methods can be safely invoked on a given object.
- **Using `auto` can produce undefined behavior.** The code still compiles fine, but there is a chance the result will be unpredictable. In this case an explicit type specification is required.

# How can we define an iterator in Java?

# Iterator Interface - `java.util.Iterator`

- `boolean hasNext():`

- Returns true if the iteration has more elements. In other words, returns true if `next()` would return an element rather than throwing an exception.

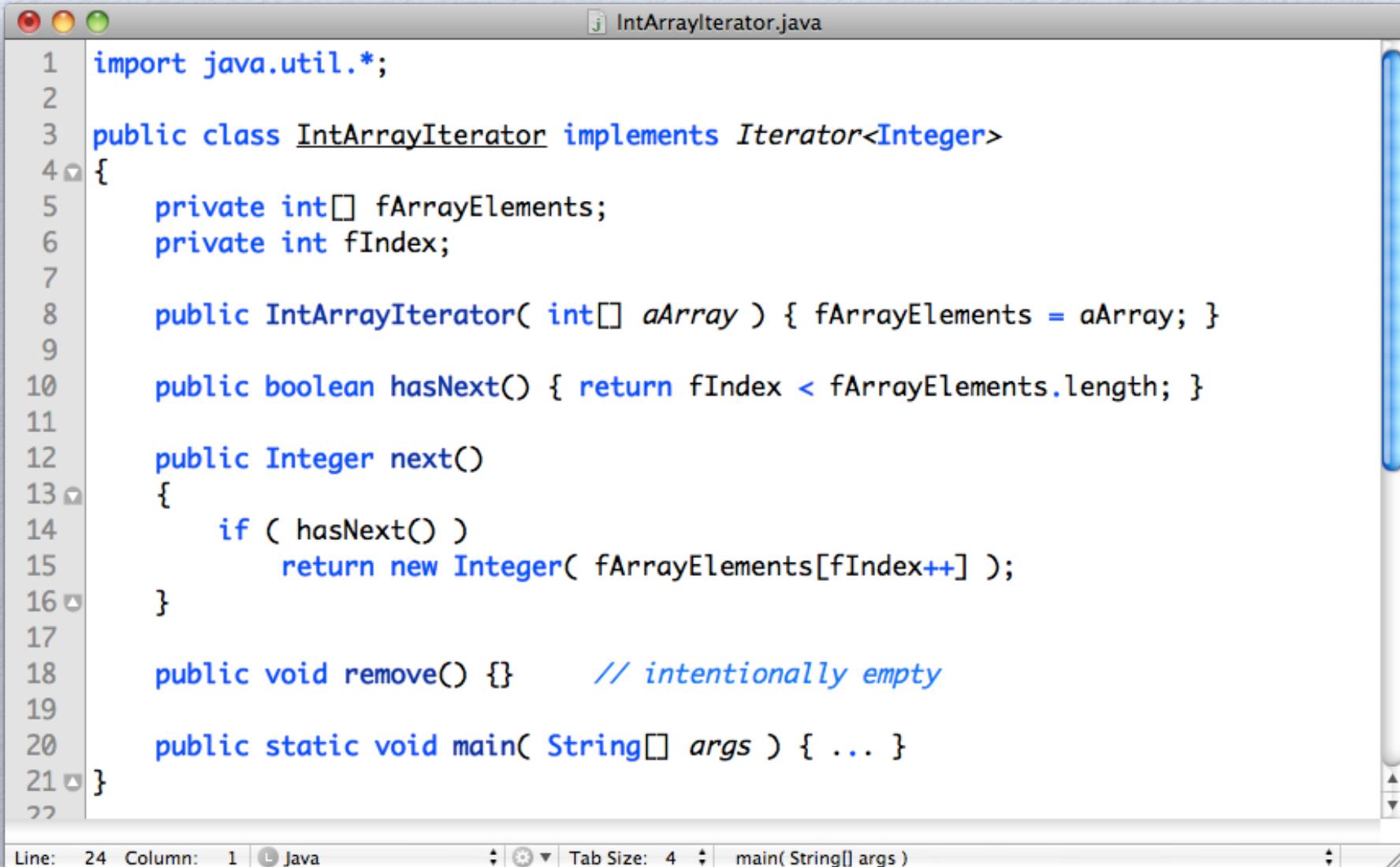
- `E next():`

- Returns the next element in the iteration. Calling this method repeatedly until the `hasNext()` method returns false will return each element in the underlying collection exactly once.

- `void remove():`

- Removes the last element returned from the underlying collection. This is an optional operation.

# IntArrayIterator in Java

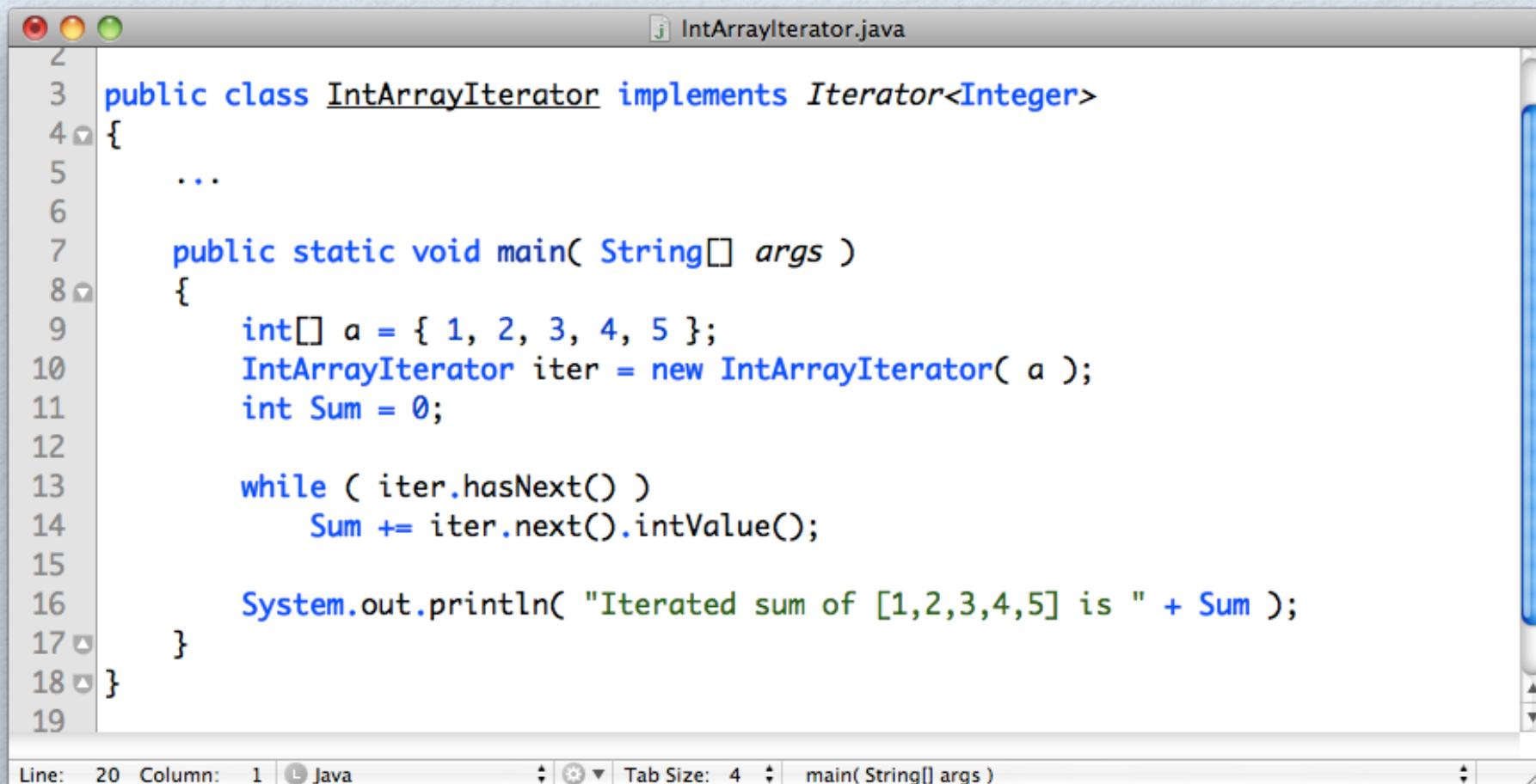


The screenshot shows a Java code editor window titled "IntArrayIterator.java". The code implements an iterator for an integer array. It includes methods for checking if there are more elements ("hasNext") and retrieving the next element ("next"). The "remove" method is marked as intentionally empty. A main method is also present.

```
1 import java.util.*;
2
3 public class IntArrayIterator implements Iterator<Integer>
4 {
5     private int[] fArrayElements;
6     private int fIndex;
7
8     public IntArrayIterator( int[] aArray ) { fArrayElements = aArray; }
9
10    public boolean hasNext() { return fIndex < fArrayElements.length; }
11
12    public Integer next()
13    {
14        if ( hasNext() )
15            return new Integer( fArrayElements[fIndex++] );
16    }
17
18    public void remove() {}      // intentionally empty
19
20    public static void main( String[] args ) { ... }
21}
22
```

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

# The Iterator's main Method



A screenshot of a Java code editor window titled "IntArrayIterator.java". The code implements the Iterator interface for an integer array. It includes a main method that initializes an array [1, 2, 3, 4, 5], creates an iterator, and calculates the sum of its elements using a while loop and the next() method. The code is color-coded, and the editor shows line numbers from 1 to 19.

```
1 public class IntArrayIterator implements Iterator<Integer>
2 {
3     ...
4
5     public static void main( String[] args )
6     {
7         int[] a = { 1, 2, 3, 4, 5 };
8         IntArrayIterator iter = new IntArrayIterator( a );
9         int Sum = 0;
10
11         while ( iter.hasNext() )
12             Sum += iter.next().intValue();
13
14         System.out.println( "Iterated sum of [1,2,3,4,5] is " + Sum );
15     }
16
17 }
18
19
```

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

# Design Pattern

# What is a Design Pattern?

Christopher Alexander says:

"... [A] pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."



# Essential Design Pattern Elements

A pattern has four essential elements:

- The pattern name that we use to describe a design problem,
- The problem that describes when to apply the pattern,
- The solution that describes the elements that make up the design, and
- The consequences that are the results and trade-offs of applying the pattern.

# Design Patterns Are Not About Design

- Design patterns are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is.
- Design patterns are not complex, domain-specific designs for an entire application or subsystem.
- Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

# Creational Patterns

- Creational patterns abstract the instantiation process. They help to make a system independent of how its objects are created, composed, and represented.
- Main forms:
  - Creational patterns for classes use inheritance to vary the class that is instantiated.
  - Creational patterns for objects delegate instantiation to another object.

# Example: Factory Method

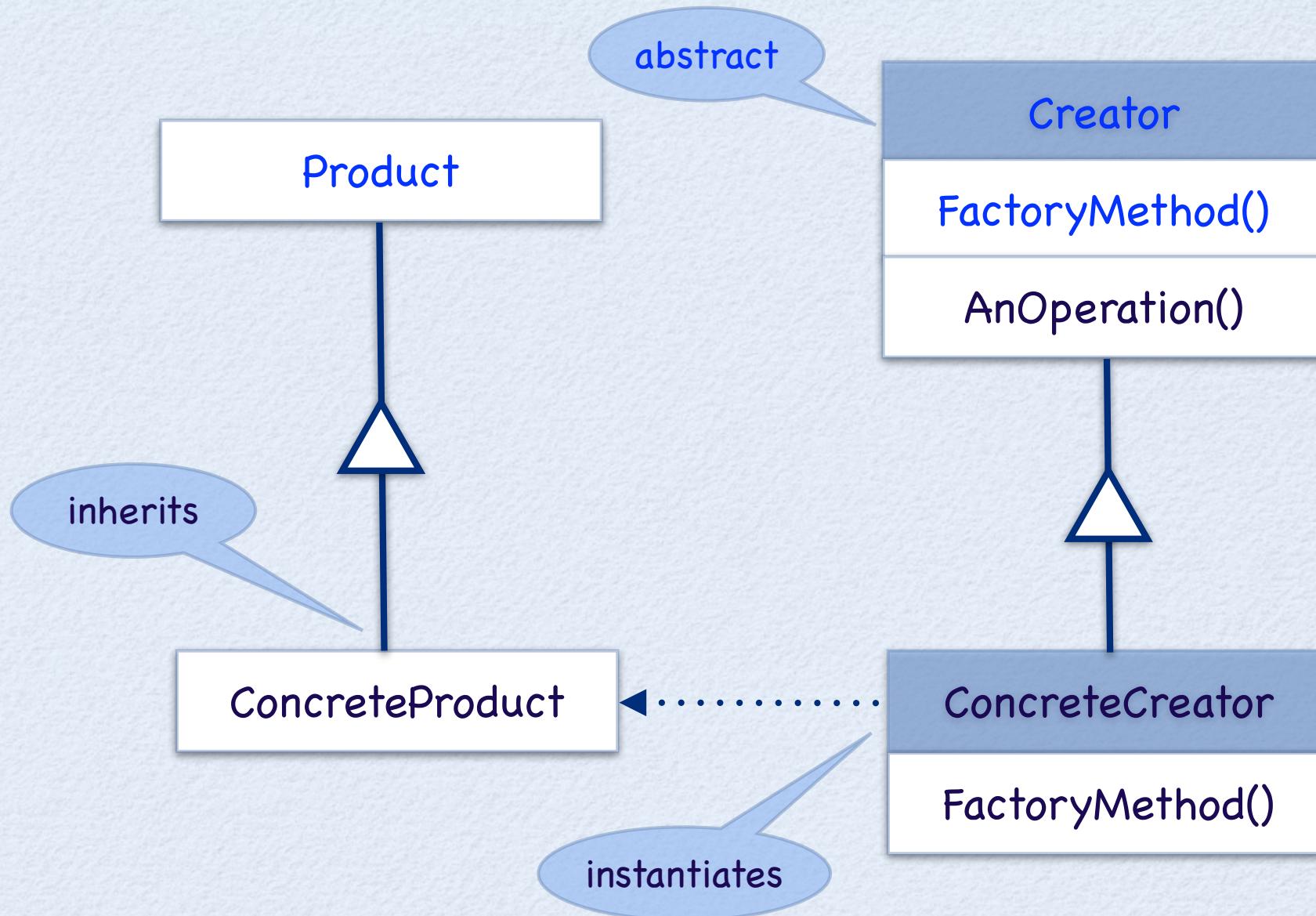
- Intent:

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

- Collaborations:

- Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate `ConcreteProduct`.

# Structure of Factory Method



# Classical Example

- A classical example of factory method is that of iterators.
- An iterator provides access to elements of a collection. A concrete iterator methods isolate the caller from knowing which class to instantiate.

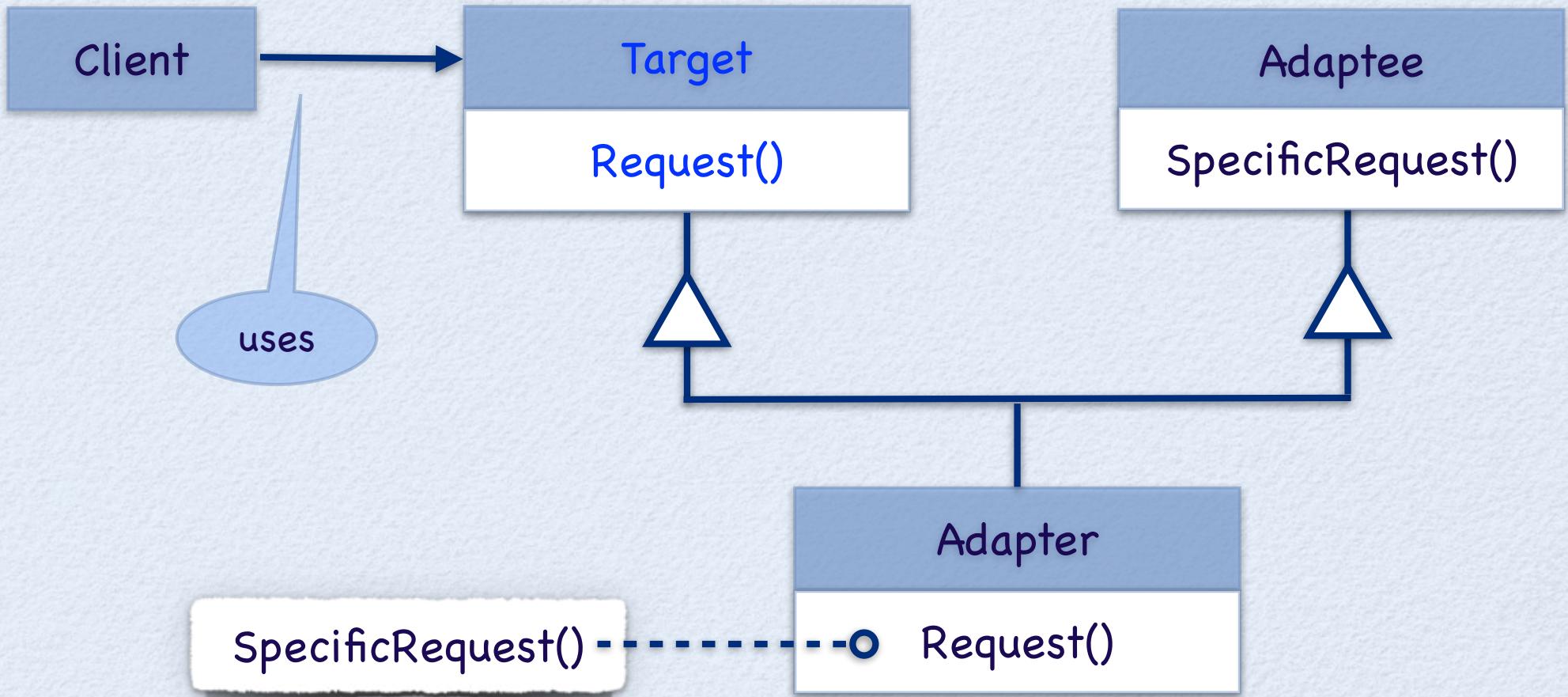
# Structural Patterns

- Structural patterns are concerned with how classes and objects are composed to form larger structures:
  - Structural class patterns use inheritance to compose interfaces or implementations.
  - Structural object patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at runtime, which is impossible with static class composition.

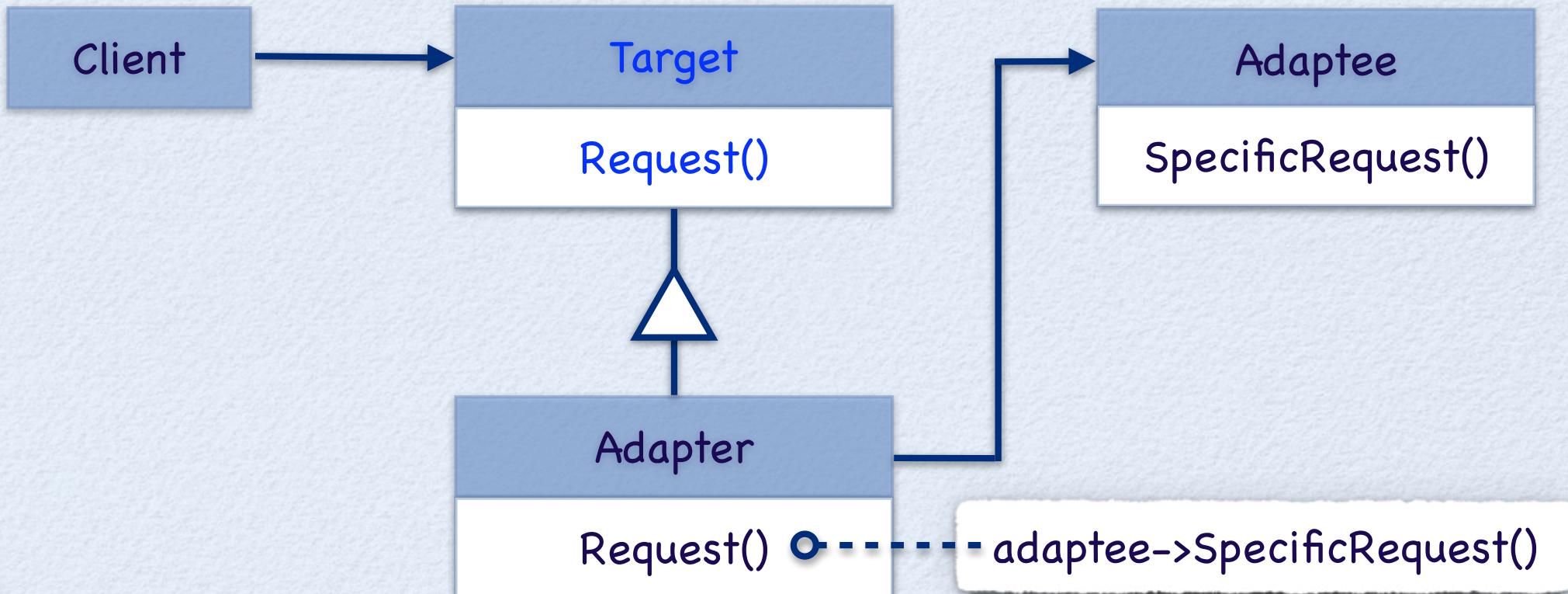
# Example: Adapter

- Intent:
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.
- Collaborations:
  - Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

# Structure of a Class Adapter



# Structure of an Object Adapter



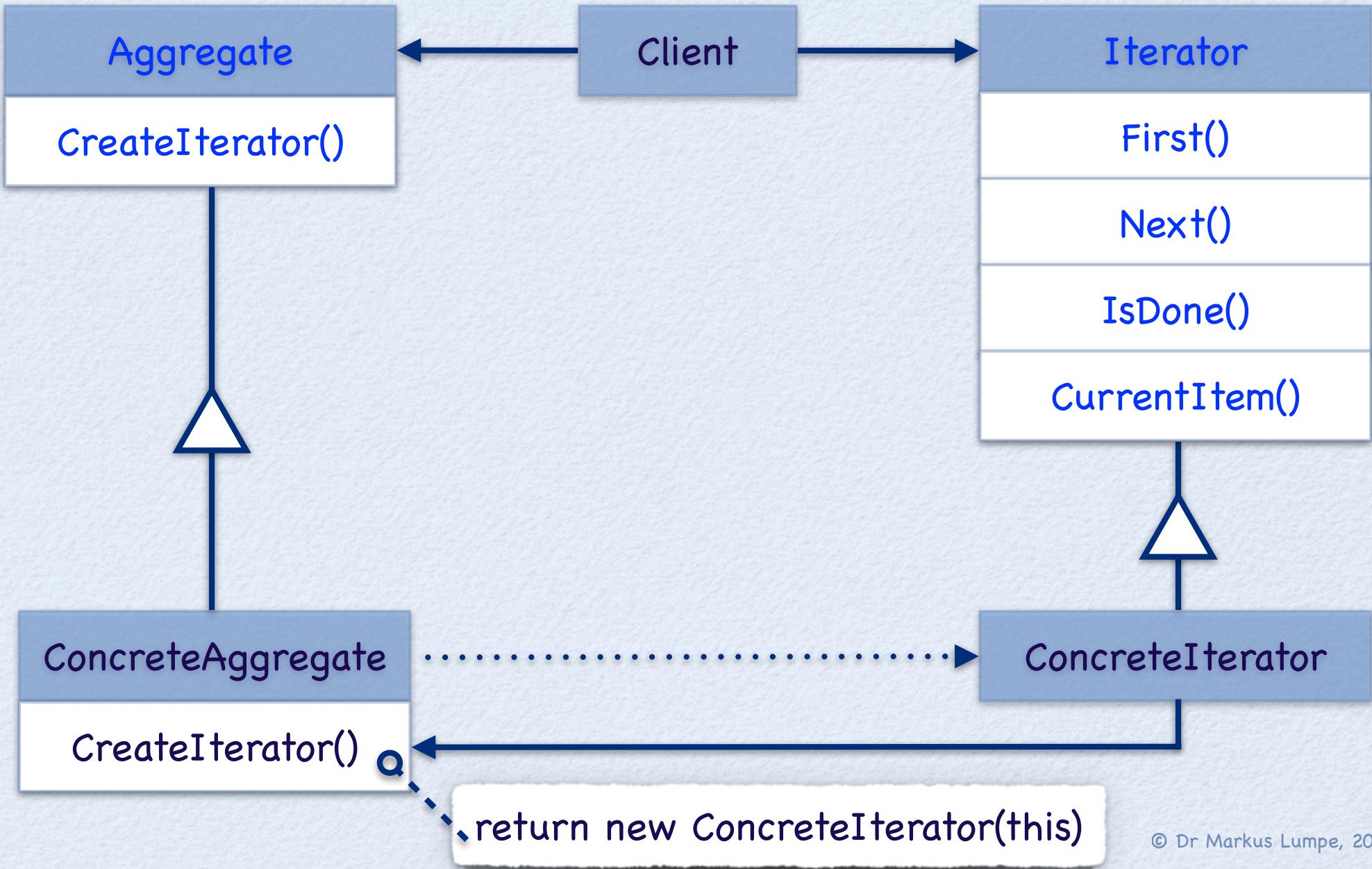
# Behavioral Patterns

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between classes and objects:
  - Behavioral class patterns use inheritance to distribute behavior between classes.
  - Behavioral object patterns use composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself.
- The classic example of a behavioral pattern is Model-View-Controller (MVC), where all views of the model are notified whenever the model's state changes.

# Example: Iterator

- Intent:
  - Provide a way to access the elements of n aggregate object sequentially without exposing its underlying representation.
- Collaborations:
  - A `ConcreteIterator` keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

# Structure of Iterator



# Recursion, Linked Lists, and ADTs

## Overview

- Recursion
- Singly-Linked Lists
- Abstract Data Types

## References

- Bruno R. Preiss: Data Structures and Algorithms with Object-Oriented Design Patterns in C++. John Wiley & Sons, Inc. (1999)
- Richard F. Gilberg and Behrouz A. Forouzan: Data Structures - A Pseudocode Approach with C. 2nd Edition. Thomson (2005)
- Russ Miller and Laurence Boxer: Algorithms Sequential & Parallel. 2nd Edition. Charles River Media Inc. (2005)
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)

# Recursion

- If a procedure contains within its body calls to itself, then this procedure is said to be **recursively defined**.
- This approach of program specification is called recursion and is found not only in programming.
- If we define a procedure recursively, then there must exist at least one sub-problem that can be solved directly, that is without calling the procedure again.
- A recursively defined procedure must always contain a directly solvable sub-problem. Otherwise, this procedure does not terminate.

# Problem-Solving with Recursion

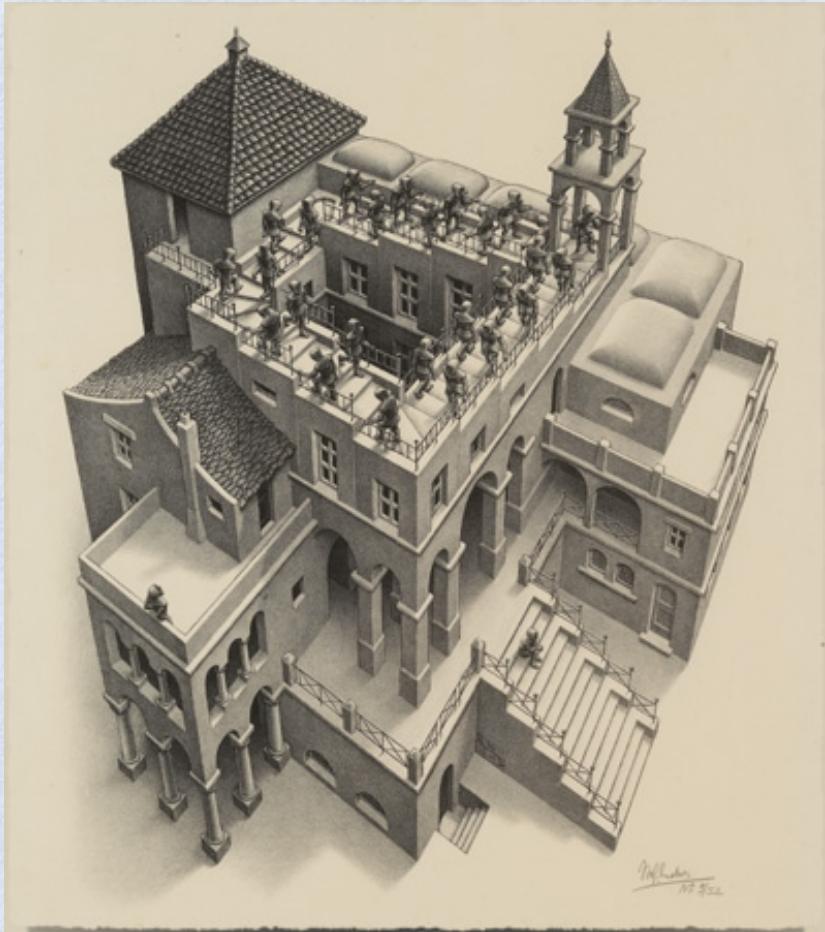
- Recursion is an important problem-solving technique in which a given problem is reduced to smaller instances of the same problem.
- The general structure of a recursive definition is

$$X = \dots X \dots$$

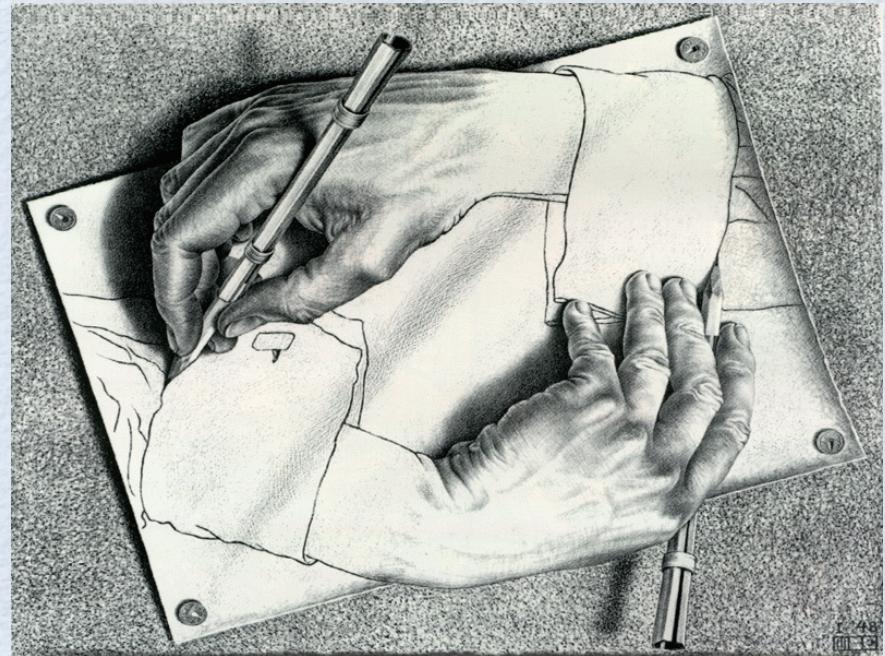
Left-hand side

Right-hand side

# Impossible Recursive Structures



<http://www.mcescher.com/>



Fair use of material subject to copyright: Low resolution images for educational presentation on recursion.

# Impossible Structures in C++:

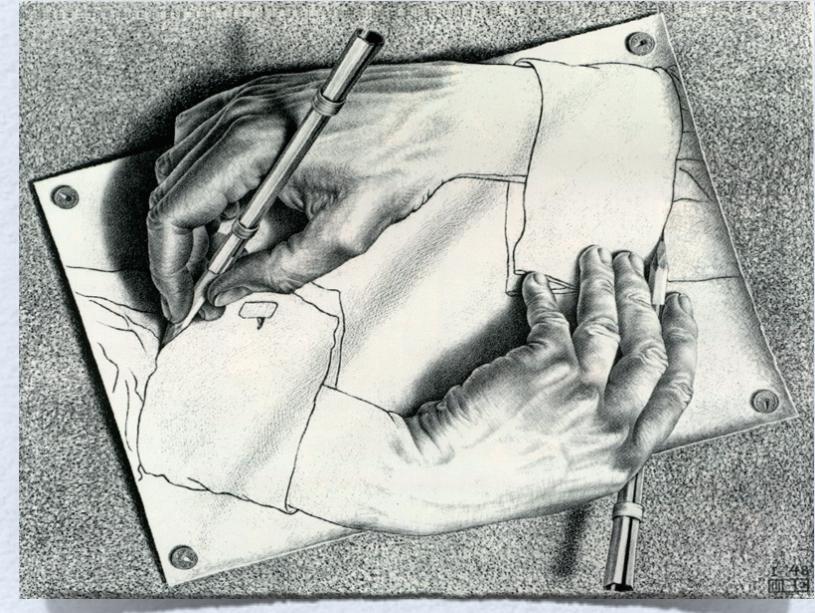
```
#include "ClassB.h"

class ClassA
{
    use ClassB
};
```



```
#include "ClassA.h"

class ClassB
{
    use ClassA
};
```

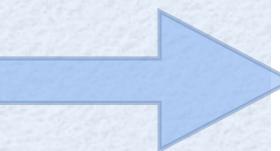


# Forward Declaration

- Unlike in Java or C#, we cannot define mutual recursive classes in C++.
- We must resolve the dependency manually and use forward declarations in specifications.
- In the implementations, we must include all specifications, so that the compiler can resolve the dependencies.

```
class ClassB;  
  
class ClassA  
{  
    use ClassB  
};
```

```
class ClassA;  
  
class ClassB  
{  
    use ClassA  
};
```



```
#include "ClassA.h"  
#include "ClassB.h"  
  
implement ClassA  
implement ClassB
```

# Basic Recursive Problems

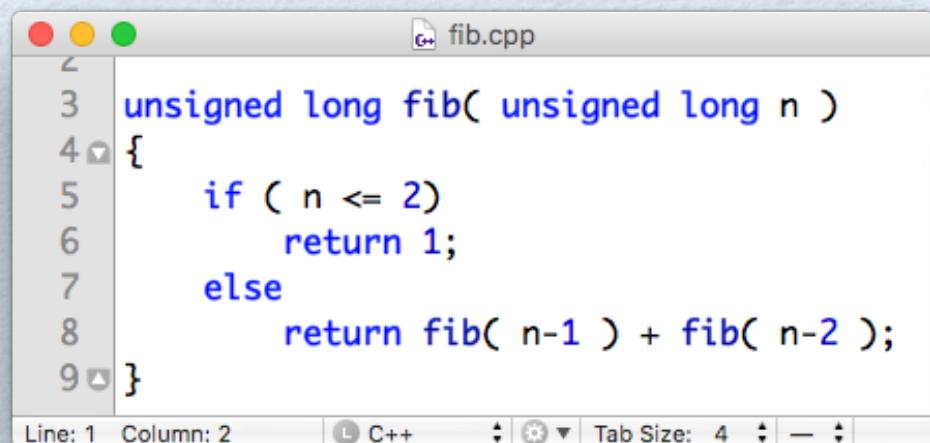
# Fibonacci

- The Fibonacci numbers are the following sequence of numbers: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- In mathematical terms, the sequence  $F(n)$  of Fibonacci numbers is defined recursively as follows:

$$F(1) = 1$$

$$F(2) = 1$$

$$F(n) = F(n-1) + F(n-2)$$



A screenshot of a C++ code editor window titled "fib.cpp". The code defines a recursive function "fib" that takes an unsigned long integer "n" as a parameter and returns an unsigned long integer. The function uses an if-else statement to check if "n" is less than or equal to 2. If true, it returns 1. Otherwise, it returns the sum of the results of calling "fib" with "n-1" and "n-2". The code editor interface includes a toolbar at the top, line numbers on the left, and status bars at the bottom showing "Line: 1 Column: 2", "C++", "Tab Size: 4", and other settings.

```
unsigned long fib( unsigned long n )
{
    if ( n <= 2)
        return 1;
    else
        return fib( n-1 ) + fib( n-2 );
}
```

# Recursive Problem-Solving: Factorials

- The factorial for positive integers is

$$n! = n * (n - 1) * \dots * 1$$

- The recursive definition:

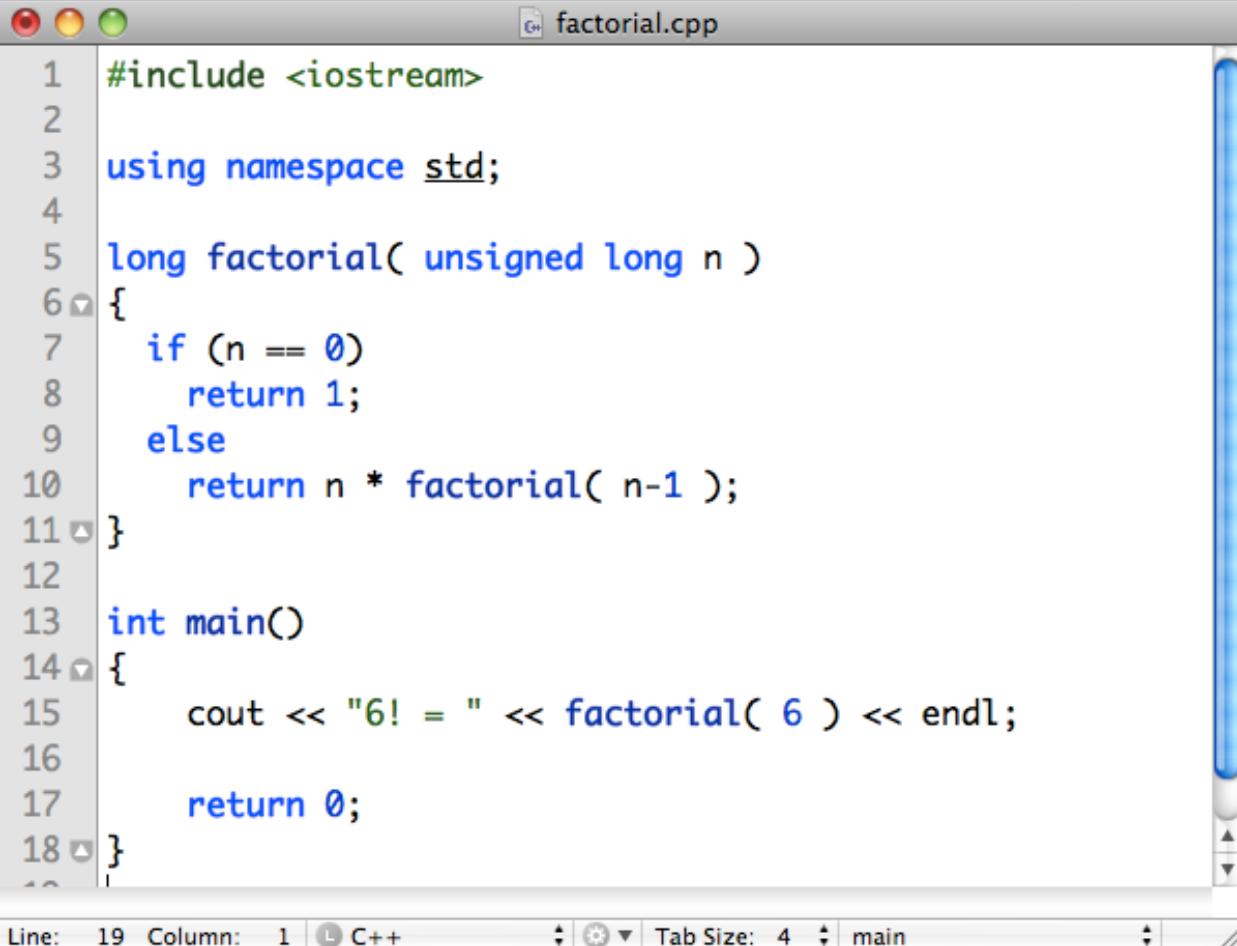
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{if } n > 0 \end{cases}$$

# Calculating Factorials

- The recursive definition tells us exactly how to calculate a factorial:

$$\begin{aligned} 4! &= 4 * 3! && \text{Recursive step: } n=4 \\ &= 4 * (3 * 2!) && \text{Recursive step: } n=3 \\ &= 4 * (3 * (2 * 1!)) && \text{Recursive step: } n=2 \\ &= 4 * (3 * (2 * (1 * 0!))) && \text{Recursive step: } n=1 \\ &= 4 * (3 * (2 * (1 * 1))) && \text{Stop condition: } n=0 \\ &= 4 * (3 * (2 * 1)) \\ &= 4 * (3 * 2) \\ &= 4 * 6 \\ &= 24 \end{aligned}$$

# Recursive Factorial

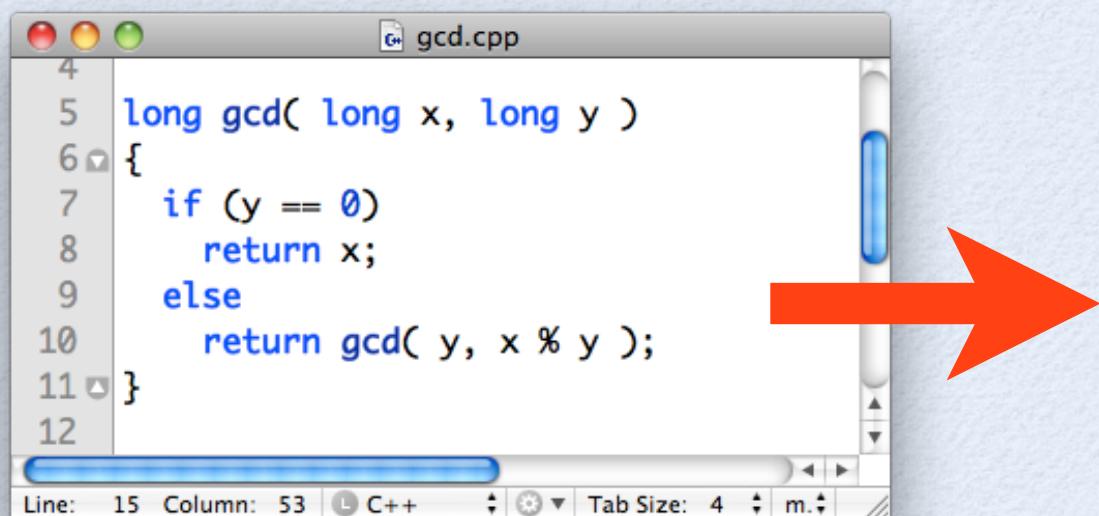


A screenshot of a C++ code editor window titled "factorial.cpp". The code implements a recursive factorial function. The editor shows syntax highlighting for C++ keywords and comments. The status bar at the bottom displays "Line: 19 Column: 1 C++" and "Tab Size: 4 main".

```
1 #include <iostream>
2
3 using namespace std;
4
5 long factorial( unsigned long n )
6 {
7     if (n == 0)
8         return 1;
9     else
10        return n * factorial( n-1 );
11 }
12
13 int main()
14 {
15     cout << "6! = " << factorial( 6 ) << endl;
16
17     return 0;
18 }
```

# Tail-Recursion

- A function is called tail-recursive if it ends in a recursive call that does not build-up any deferred operations.



```
gcd.cpp
4
5 long gcd( long x, long y )
6 {
7     if (y == 0)
8         return x;
9     else
10        return gcd( y, x % y );
11 }
12

Line: 15 Column: 53 C++ Tab Size: 4 m.
```

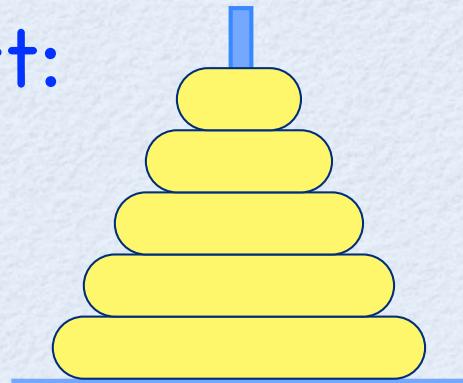
gcd( 1246, 234 ):  
↳ gcd( 234, 76 )  
↳ gcd( 76, 6 )  
↳ gcd( 6, 4 )  
↳ gcd( 4, 2 )  
↳ gcd( 2, 0 )  
↳ 2

# Towers of Hanoi

- Problem:
  - Move disks from a start peg to a target peg using a middle peg.
- Challenge:
  - All disks have a unique size and at no time must a bigger disk be placed on top of a smaller one.

# Towers of Hanoi: Configuration

Start:



Start

Middle

Target

Finish:



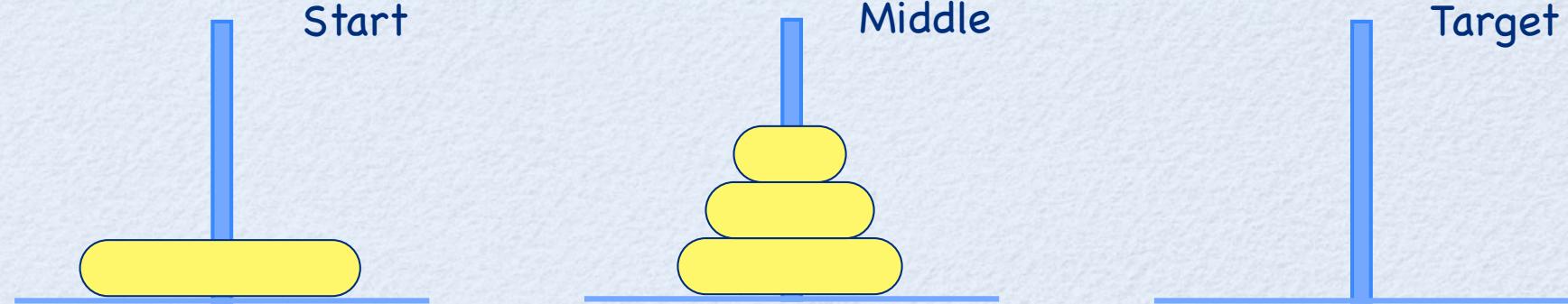
Start

Middle

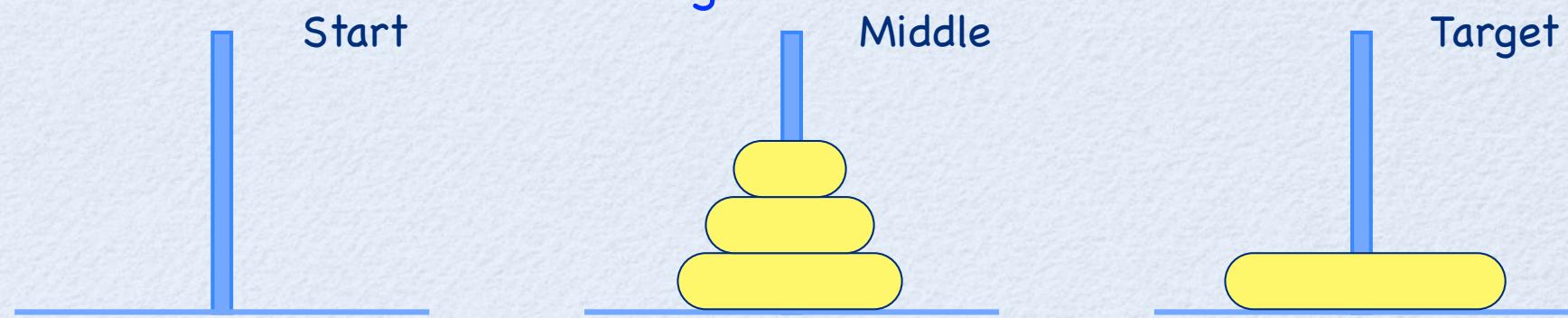
Target

# A Recursive Solution

1. Move  $n-1$  disks from Start to Middle:



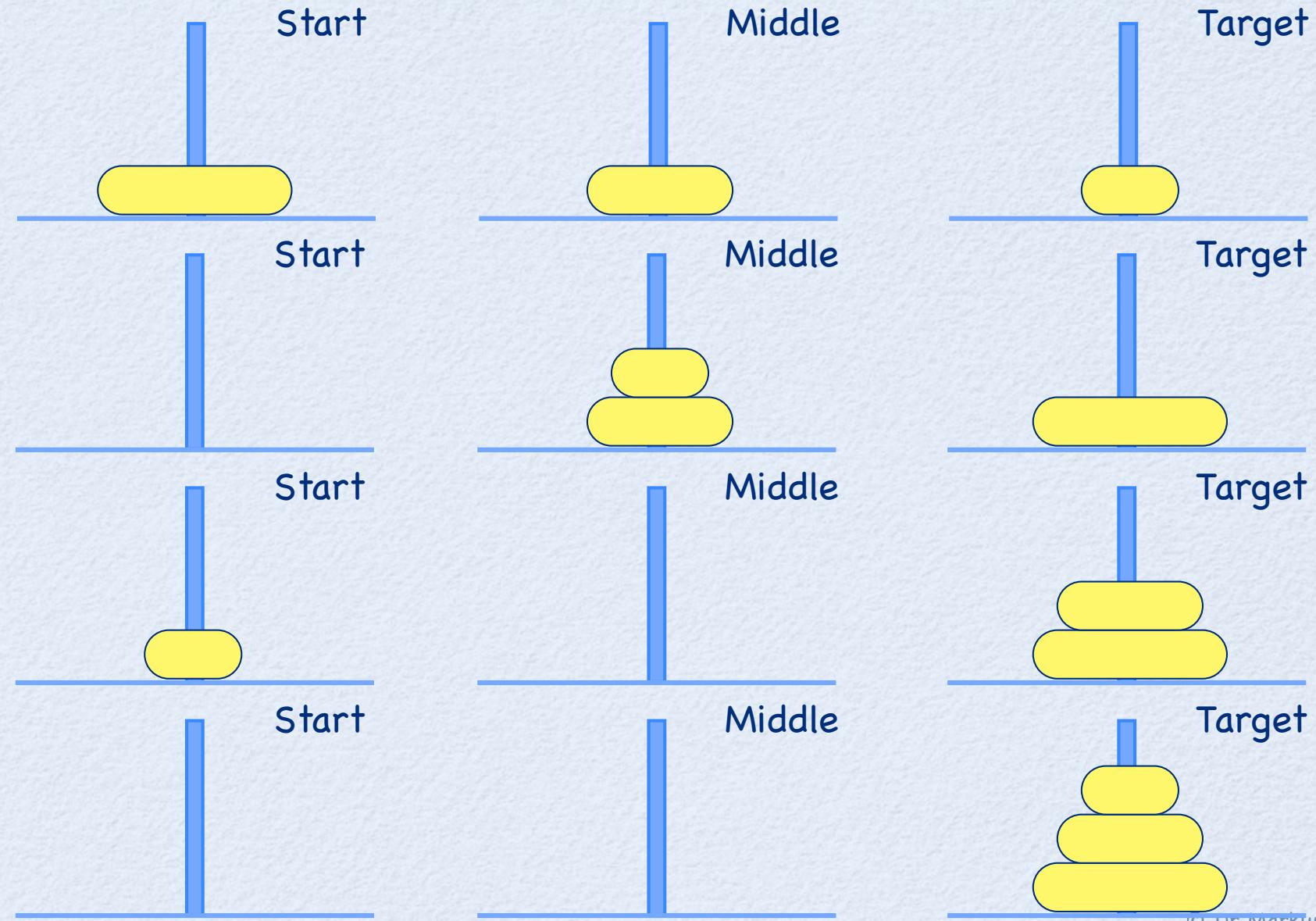
2. Move 1 disk from Start to Target:



3. Move  $n-1$  disks from Middle to Target:



# A Recursive Solution: Intermediate



# The Recursive Procedure

A screenshot of a Mac OS X desktop environment. On the left is a code editor window titled "hanoi.cpp" containing C++ code for the Tower of Hanoi problem. On the right is a terminal window titled "COS30008" showing the output of running the program.

```
1 #include <iostream>
2
3 using namespace std;
4
5 void move( int n, string start, string target, string middle )
6 {
7     if ( n > 0 )
8     {
9         move( n-1, start, middle, target );
10        cout << "Move disk " << n << " from " << start
11            << " to " << target << "." << endl;
12        move( n-1, middle, target, start );
13    }
14 }
15
16 int main()
17 {
18     move( 3, "Start", "Target", "Middle" );
19
20     return 0;
21 }
```

Line: 22 Column: 1 C++ Tab Size: 4 main

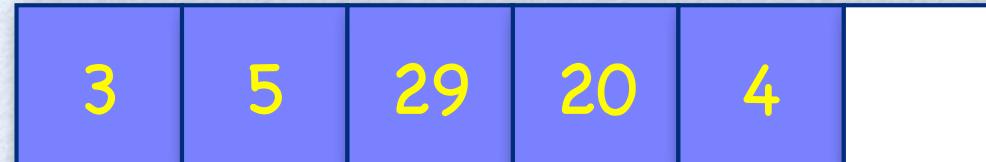
```
Kamala:COS30008 Markus$ ./hanoi
Move disk 1 from Start to Target.
Move disk 2 from Start to Middle.
Move disk 1 from Target to Middle.
Move disk 3 from Start to Target.
Move disk 1 from Middle to Start.
Move disk 2 from Middle to Target.
Move disk 1 from Start to Target.
Kamala:COS30008 Markus$ _
```

# Recursion is a prerequisite for linked lists!

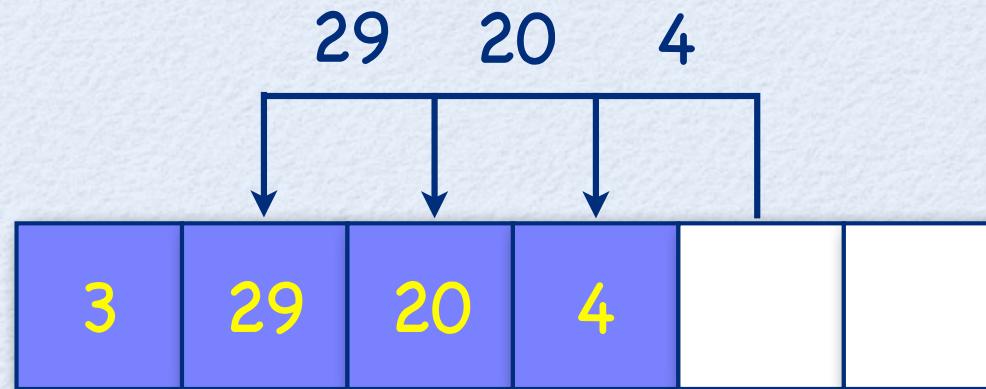
# Problems with Arrays

- An array is a contiguous storage that provides insufficient abstractions for handling addition and deletion of elements.
- Addition and deletion require  $n/2$  shifts on average.
- The computation time is  $O(n)$ .
- Resizing affects performance.

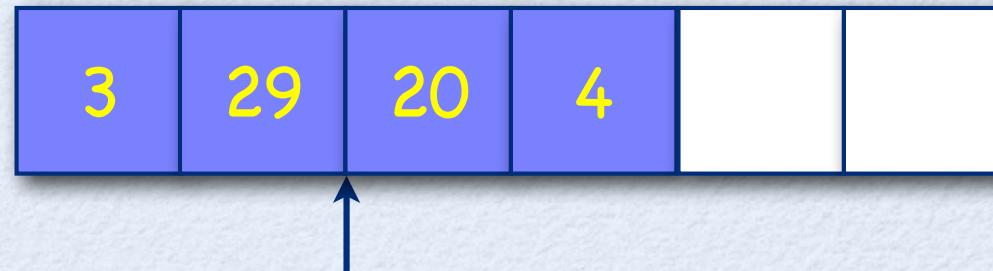
# Deletion Requires Relocation



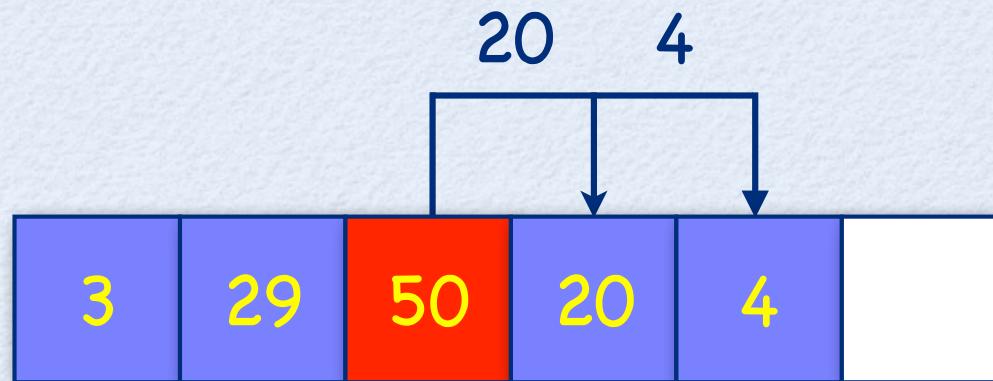
Delete 5



# Insertion Requires Relocation



Insert 50 after 29



# Singly-Linked Lists

- A singly-linked list is a sequence of data items, each connected to the next by a pointer called `next`.



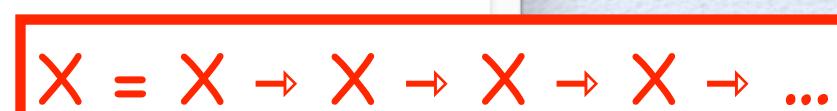
- A data item may be a primitive value, a composite value, or even another pointer.
- A singly-linked list is a recursive data structure whose nodes refers to nodes of the same type.

# Impossible Singly-Linked List Structure in C++:

```
// impossible singly-linked list
struct SinglyLinkedList
{
    DataType fData;
    SinglyLinkedList fNext;

    SinglyLinkedList( const DataType& aData,
                      const SinglyLinkedList& aNext ) :
        fData(aData),
        fNext(aNext)
    {}
};

Line: 4 Column: 22 | C++ | Tab Size: 4
```



Field fNext has incomplete type.

# Singly-Linked List Using Pointers

The screenshot shows a code editor window with the file `SinglyLinkedList.h` open. The code defines a singly-linked list structure with a constructor that initializes the data and the next pointer.

```
6 struct SinglyLinkedList
7 {
8     DataType fData;
9     SinglyLinkedList* fNext;
10
11    SinglyLinkedList( const DataType& aData,
12                      SinglyLinkedList* aNext = nullptr ) :
13        fData(aData),
14        fNext(aNext)
15    {}
16};
```

Two yellow callout bubbles are present:

- A bubble pointing to the `DataType` type in the declaration: **DataType is application-specific**.
- A bubble pointing to the `fNext` pointer member: **We need to use pointers**.

At the bottom of the editor window, the status bar displays: Line: 9 Column: 2 C++ Tab Size: 4 SinglyLinkedList

- A list manages a collection of elements.
- The class `SinglyLinkedList` defines a value-based sequence container for values of type `DataType`.
- To break infinite recursion, we have to use pointers to the next elements. This way, the compiler can deduce the size of a singly-linked list element and compile the definition.

# Singly-Linked List with R-Values

```
6 struct SinglyLinkedList
7 {
8     DataType fData;
9     SinglyLinkedList* fNext;
10
11    SinglyLinkedList( const DataType& aData, SinglyLinkedList* aNext = nullptr ) :
12        fData(aData),
13        fNext(aNext)
14    {}
15
16    SinglyLinkedList( DataType&& aData, SinglyLinkedList* aNext = nullptr ) :
17        fData(std::move(aData)),
18        fNext(aNext)
19    {}
20};
```

- The r-value constructor can “steal” the memory of the argument `aData` to initialize the payload `fData`. To use this constructor, the parameter `aData` must be a temporary of literal value.

# R-Value References (C++-11)

# L-Values and L-Value References &

- The references that we have seen so far are l-value references, that is, references to l-values.
- The term l-value refers to a thing that can occur on the left side of an assignment, named objects that have a defined storage location (i.e., an address). L-value references can only be bound to l-values (exception: we can bind an r-value to a const l-value reference):

```
int var = 42;           // l-value: initialized variable declaration  
int& ref = var;        // l-value reference to an l-value  
int& ref2 = 42;         // error: non-const l-value reference cannot bind to temporary  
const int& ref3 = 42;    // const l-value reference to an r-value
```

# R-Values and R-Value References &&

- The term r-value refers to things that can occur on the right side of an assignment, literals and temporaries that do not have a defined storage locations.
- R-value references only bind to r-values:

```
int&& ref = 42;      // r-value reference to an r-value
```

```
int val = 42;        // l-value: initialized variable declaration
```

```
int&& ref2 = val;    // error: r-value reference cannot bind to l-value
```

# Move Semantics (std::move)

- R-values are typically temporary and so can be freely modified: if you know that your function parameter is an r-value, you can use it as temporary storage, or “steal” its contents without affecting program correctness.
- This means that rather than copying the contents (expensive) of an r-value, you can move the contents (cheap).

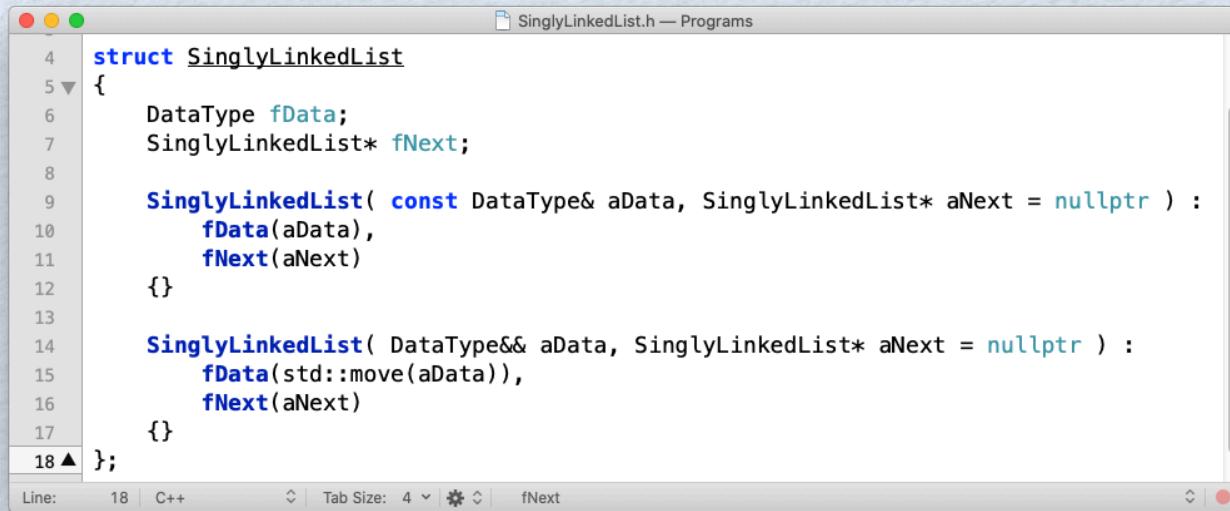
```
4 struct SinglyLinkedList
5 {
6     DataType fData;
7     SinglyLinkedList* fNext;
8
9     SinglyLinkedList( const DataType& aData, SinglyLinkedList* aNext = nullptr ) :
10        fData(aData),
11        fNext(aNext)
12    {}
13
14     SinglyLinkedList( DataType&& aData, SinglyLinkedList* aNext = nullptr ) :
15        fData(std::move(aData)),
16        fNext(aNext)
17    {}
18}
```

copy l-value

move r-value

**std::move** is a function that performs a type cast of its argument to an r-value.

# Using L-Values and R-Values



```
struct SinglyLinkedList
{
    DataType fData;
    SinglyLinkedList* fNext;

    SinglyLinkedList( const DataType& aData, SinglyLinkedList* aNext = nullptr ) :
        fData(aData),
        fNext(aNext)
    {}

    SinglyLinkedList( DataType&& aData, SinglyLinkedList* aNext = nullptr ) :
        fData(std::move(aData)),
        fNext(aNext)
    {}
};
```

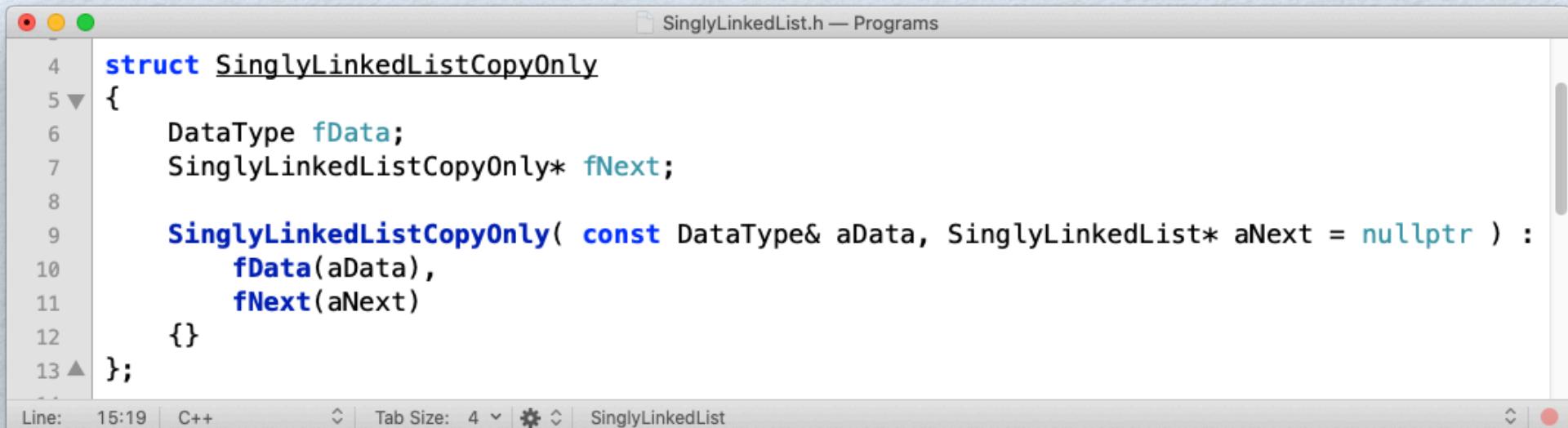
- Using the different constructors, we can write:

```
string lValue = "COS30008";
SinglyLinkedList lNodeWithCopy( lValue );
SinglyLinkedList lNodeWithMove( "COS30008" );
```

copy  
l-value

move  
r-value

# Using L-Values and R-Values: No Move



The screenshot shows the Xcode IDE with a file named "SinglyLinkedList.h" open. The code defines a struct named "SinglyLinkedListCopyOnly" with two members: "fData" of type "DataType" and a pointer "fNext" to another "SinglyLinkedListCopyOnly" object. A constructor is provided that takes a reference to a "(DataType&)" and a pointer to the next node ("SinglyLinkedList\*"). The code is color-coded, with "SinglyLinkedListCopyOnly" in blue, "fData" in teal, and "fNext" in light blue.

```
4 struct SinglyLinkedListCopyOnly
5 {
6     DataType fData;
7     SinglyLinkedListCopyOnly* fNext;
8
9     SinglyLinkedListCopyOnly( const DataType& aData, SinglyLinkedList* aNext = nullptr ) :
10        fData(aData),
11        fNext(aNext)
12    {}
13};
```

- If a class does not support move semantics, then the compiler will use copy semantics. That is, an r-value decays to an l-value.

```
string lValue = "COS30008";
SinglyLinkedListCopyOnly lNodeWithLValue( lValue );
SinglyLinkedListCopyOnly lNodeWithRValue( "COS30008" );
```

copy l-value

copy  
r-value

We discuss more details  
later when we study  
memory management.

# A Simple List of Integers

```
2 #include <iostream>
3 #include <string>
4
5 using namespace std;
6
7 using DataType = string;
8
9 #include "SinglyLinkedList.h"
10
11 int main()
12 {
13     string lA = "AAAA";
14     string lC = "CCCC";
15
16     SinglyLinkedList One( lA );
17     SinglyLinkedList Two( "BBBB", &One );
18     SinglyLinkedList Three( lC, &Two );
19
20     SinglyLinkedList* lTop = &Three;
21
22     for ( ; lTop != nullptr; lTop = lTop->fNext )
23     {
24         cout << "Value: " << lTop->fData << endl;
25     }
26
27
28     return 0;
29 }
```

Line: 36:2 | C++ | Tab Size: 4 | main | 250

define DataType a synonym for string

```
Microsoft Visual Studio Deb...
Value: CCCC
Value: BBBB
Value: AAAA
```

# Using Declaration – C++11 Type Aliases

- A type alias is a name that refers to a previously defined type.

```
using identifier = type;
```

- Type aliases are commonly used for three purposes:
  - To hide the implementation of a given type.
  - To streamline complex type definitions making them easier to understand, and
  - To allow a single type to be used in different contexts under different names.
- Type aliases establish a **nominal equivalence between types**.
- Type aliases are similar to **typedef**. However, type aliases are better suited when creating alias templates.

# Can we do better?

# Templates - C++'s Generic Types

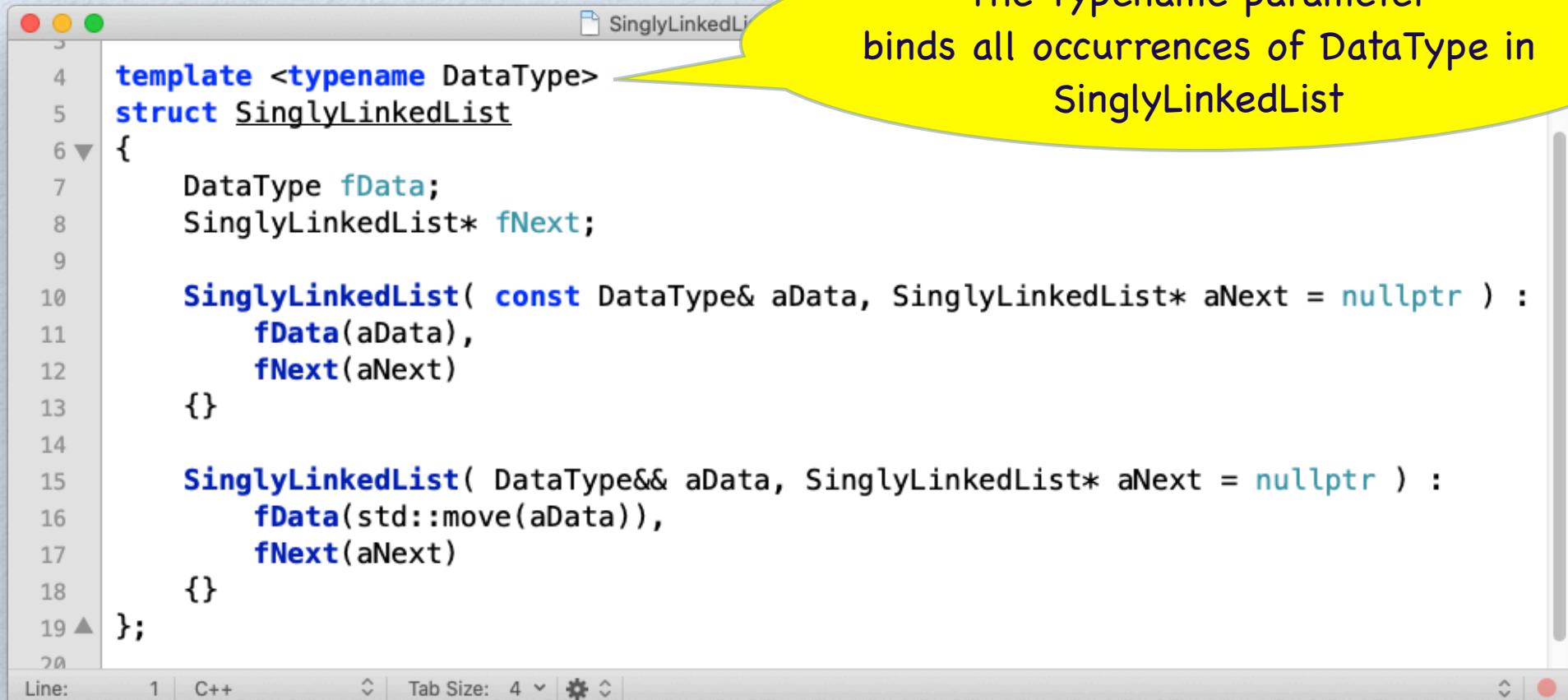
- Templates are blueprints from which classes and/or functions are automatically generated by the compiler based on a set of parameters.
- Note, every time a given template is being instantiated with type parameters that have not been used before, a new version of the class or function is generated.
- A new version of a class or function is called specialization of the template. Specializations are not mutually compatible.

# Class Template

```
template<typename T1, ..., typename Tn>
class AClassTemplate
{
    // class specification
};
```

- A template is a parameterized abstraction over a class.
- From the language-theoretical perspective, templates are 2nd order functions from types to classes/functions.
- To instantiate a class template we supply the desired types, as actual template parameters, so that the C++ compiler can synthesize a specialized class for the template.

# Singly-Linked List Class Template



```
template <typename DataType>
struct SinglyLinkedList
{
    DataType fData;
    SinglyLinkedList* fNext;

    SinglyLinkedList( const DataType& aData, SinglyLinkedList* aNext = nullptr ) :
        fData(aData),
        fNext(aNext)
    {}

    SinglyLinkedList( DataType&& aData, SinglyLinkedList* aNext = nullptr ) :
        fData(std::move(aData)),
        fNext(aNext)
    {}

};
```

The `typename` parameter binds all occurrences of `DataType` in `SinglyLinkedList`

# The New Main

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 #include "SinglyLinkedListTemplate.h"
7
8
9 int main()
10 {
11     using StringList = SinglyLinkedList<string>;
12
13     string lA = "AAAA";
14     string lC = "CCCC";
15
16     StringList One( lA );
17     StringList Two( "BBBB", &One );
18     StringList Three( lC, &Two );
19
20     StringList* lTop = &Three;
21
22     for ( ; lTop != nullptr; lTop = lTop->fNext )
23     {
24         cout << "Value: " << lTop->fData << endl;
25     }
26
27
28     return 0;
29 }
```

We instantiate the template  
SinglyLinkedList to SinglyLinkedList<string>.

# Class Template Instantiation

```
using IntegerList = SinglyLinkedList<int>;
```

```
using ListOfIntegerLists = SinglyLinkedList<IntegerList>;
```

- Types used as arguments cannot be classes with local scope.
- Once instantiated, a class template can be used as any other class.

# List Iterator Template

# SinglyLinkedList Iterator Specification

```
#pragma once

#include "SinglyLinkedListTemplate.h"

template <typename T>
class SinglyLinkedListIterator
{
private:
    using ListNode = SinglyLinkedList<T>;
    const ListNode* fList;
    const ListNode* fIndex;

public:
    using Iterator = SinglyLinkedListIterator<T>;
    SinglyLinkedListIterator( const ListNode* aList );
    const T& operator*() const;
    Iterator& operator++();      // prefix
    Iterator operator++(int);   // postfix
    bool operator==( const Iterator& aRHS ) const;
    bool operator!=( const Iterator& aRHS ) const;
    Iterator begin();           // for-range feature
    Iterator end();             // for-range feature
};

Line: 32 | C++ | Tab Size: 4 | end
```

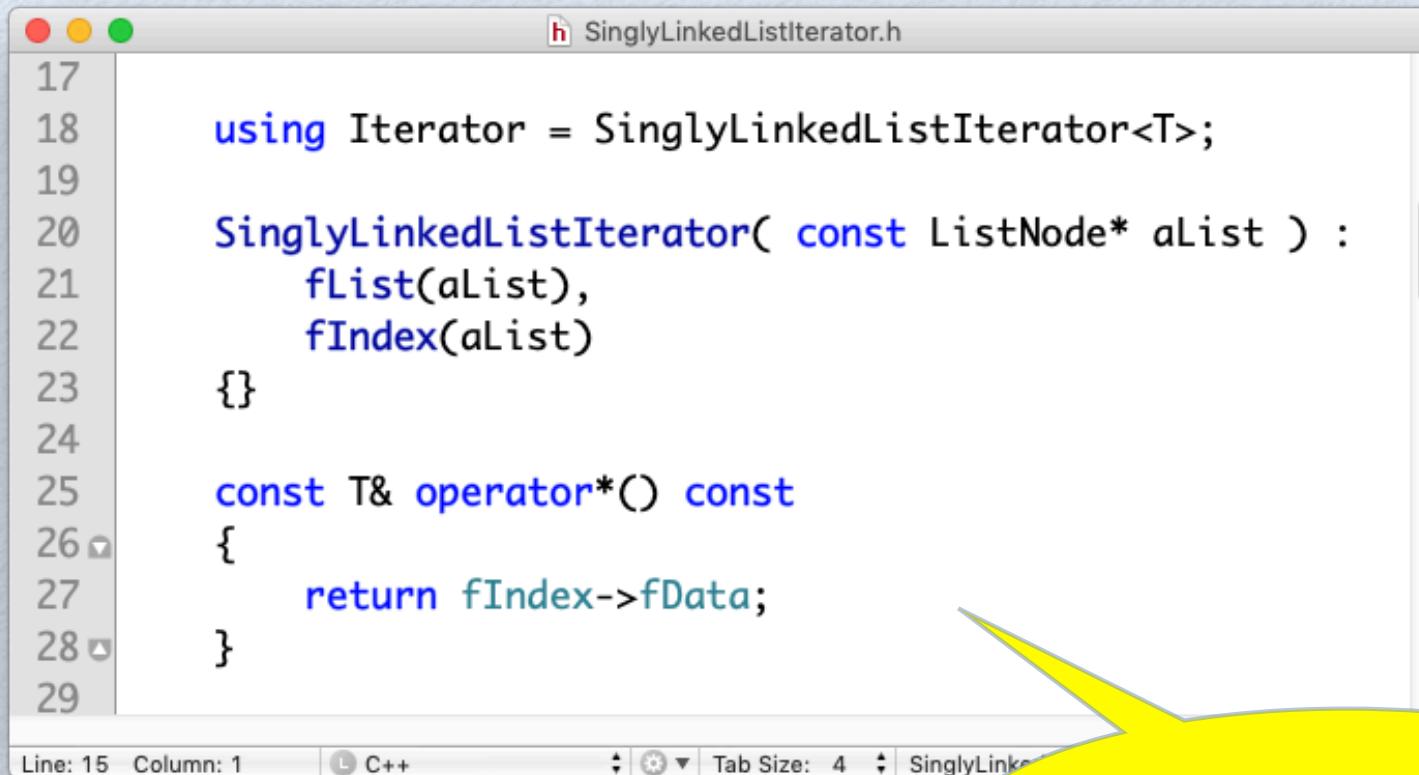
We maintain a pointer to  
read-only list elements

# Notes on Templates

- When using templates, the C++ compiler must have access not only the specification of a template class but also to all method implementations in order to properly instantiate the template.
- Templates are not to be confused with library classes. Templates are blueprint that have to be instantiated for each separate type application.
- Think of templates as special forms of macros.
- When defining a template class, we need to implement, like in Java, all methods within the class definition, usually in a header file.

# **Templates have no .cpp file!**

# Constructor & Deference



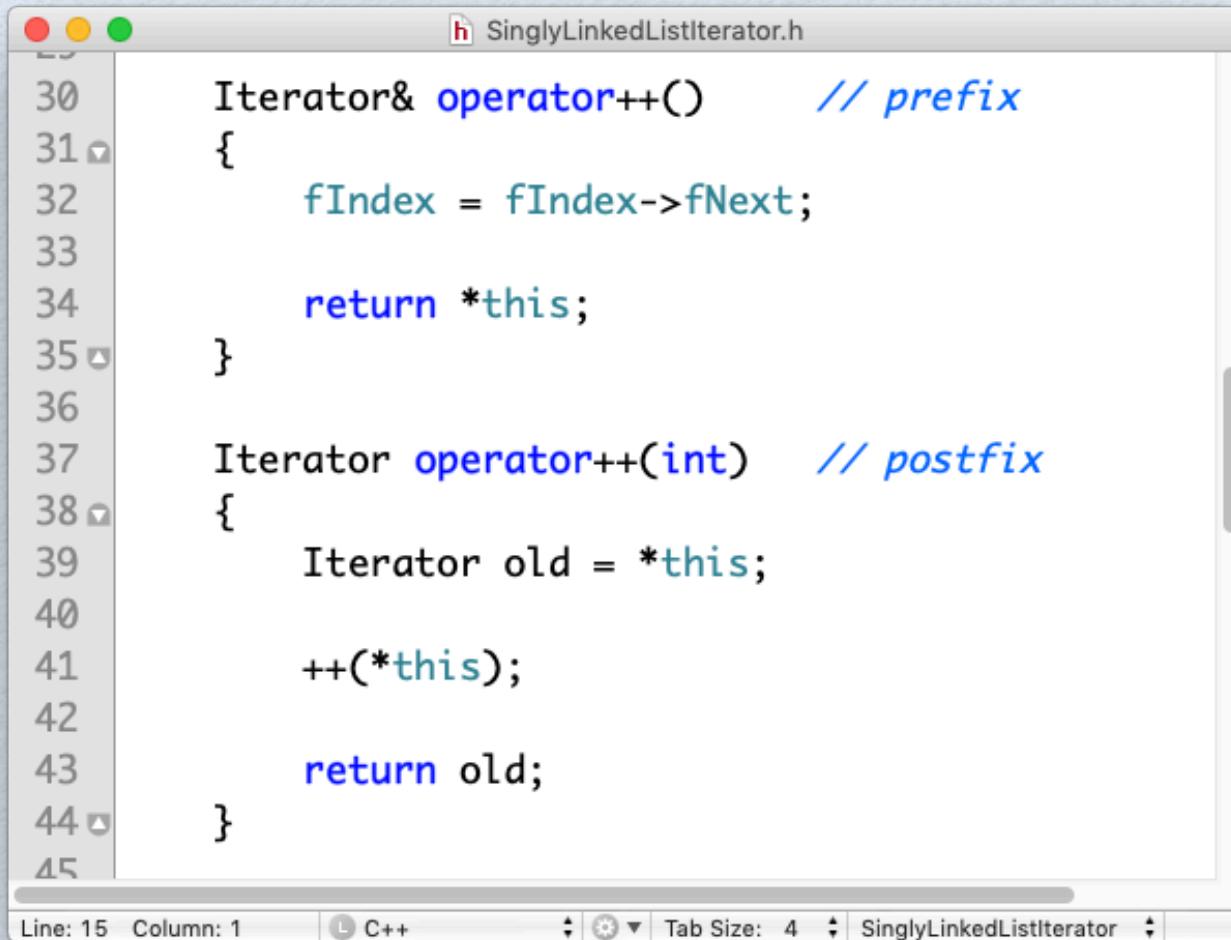
The screenshot shows a code editor window titled "SinglyLinkedListIterator.h". The code implements an iterator for a singly linked list. It includes a constructor that initializes pointers to the list and index nodes, and an operator\*() const method that returns a reference to the data at the current index. The code is annotated with line numbers from 17 to 29.

```
17  
18     using Iterator = SinglyLinkedListIterator<T>;  
19  
20     SinglyLinkedListIterator( const ListNode* aList ) :  
21         fList(aList),  
22         fIndex(aList)  
23     {}  
24  
25     const T& operator*() const  
26     {  
27         return fIndex->fData;  
28     }  
29
```

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

For iterator implementation in header file.

# Increments

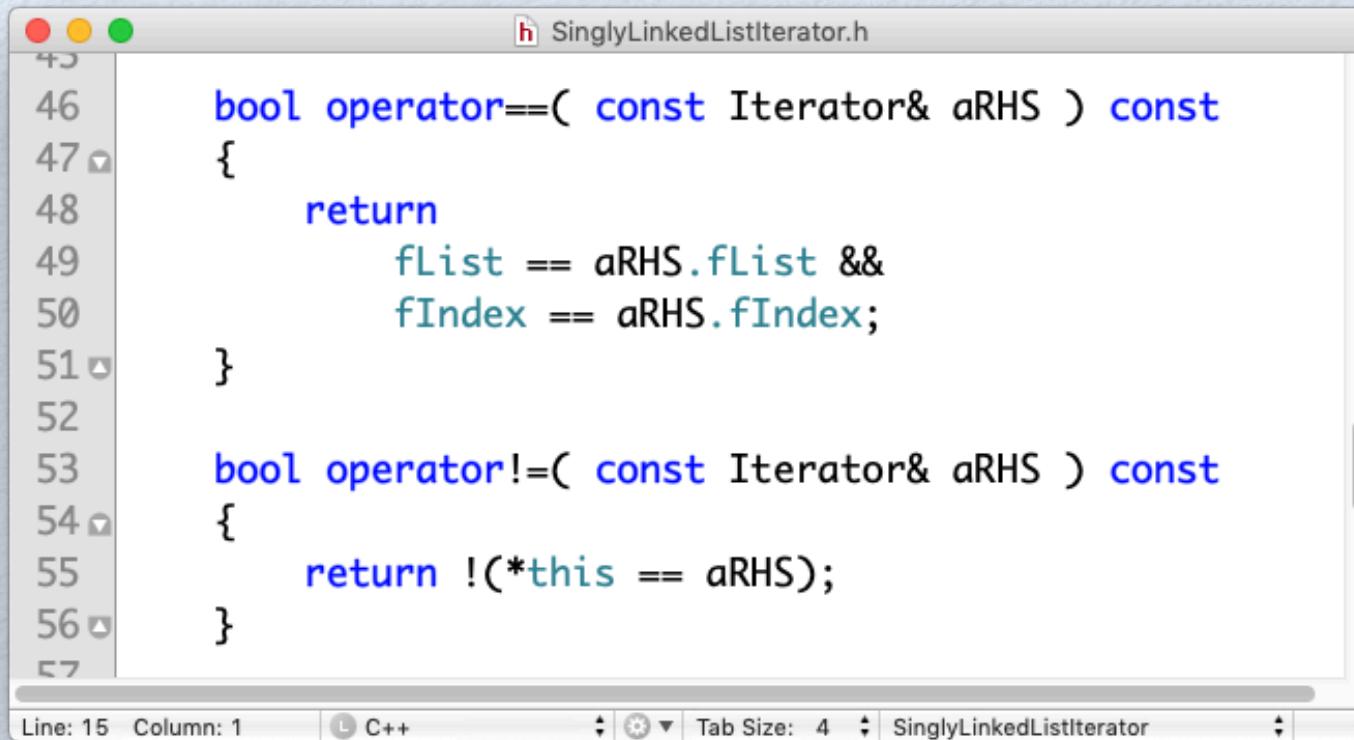


A screenshot of a code editor window titled "SinglyLinkedListIterator.h". The code implements two increment operators for an iterator class:

```
30     Iterator& operator++()      // prefix
31 {
32     fIndex = fIndex->fNext;
33
34     return *this;
35 }
36
37 Iterator operator++(int)    // postfix
38 {
39     Iterator old = *this;
40
41     ++(*this);
42
43     return old;
44 }
45
```

The code uses standard C++ syntax for class members and operators. The prefix increment operator increments the iterator's index and returns the current value. The postfix increment operator creates a copy of the current iterator before incrementing it.

# Equivalence



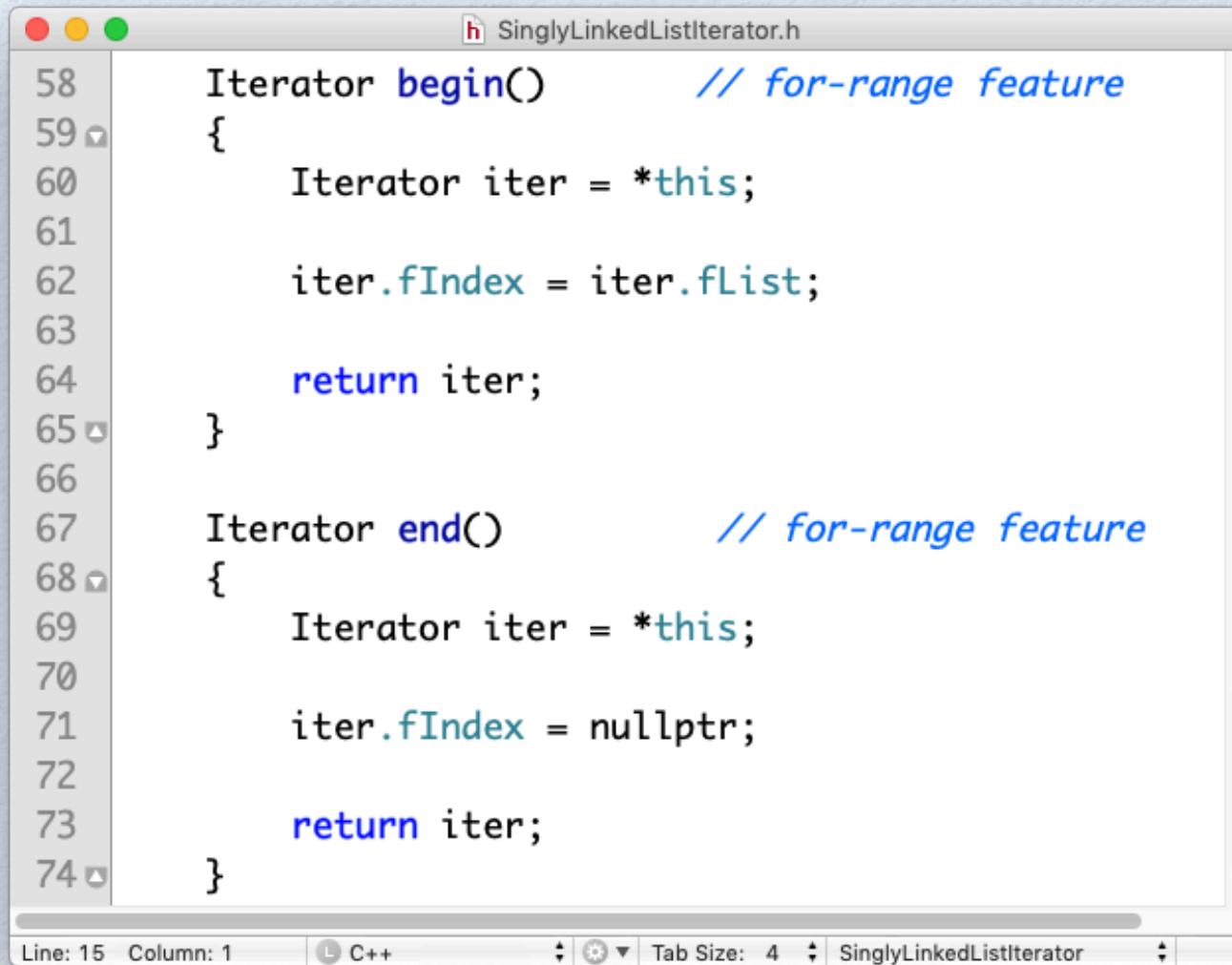
The screenshot shows a code editor window with the file `SinglyLinkedListIterator.h` open. The code implements comparison operators for two iterator objects.

```
45
46     bool operator==( const Iterator& aRHS ) const
47 {
48     return
49         fList == aRHS.fList &&
50         fIndex == aRHS.fIndex;
51 }
52
53     bool operator!=( const Iterator& aRHS ) const
54 {
55     return !(*this == aRHS);
56 }
```

The code editor interface includes:

- File menu icons (red, yellow, green) in the top-left corner.
- Title bar: `SinglyLinkedListIterator.h`.
- Code area with line numbers 45 to 57.
- Status bar at the bottom: "Line: 15 Column: 1", "C++", "Tab Size: 4", and the file name "SinglyLinkedListIterator".

# Auxiliaries (For-Range)

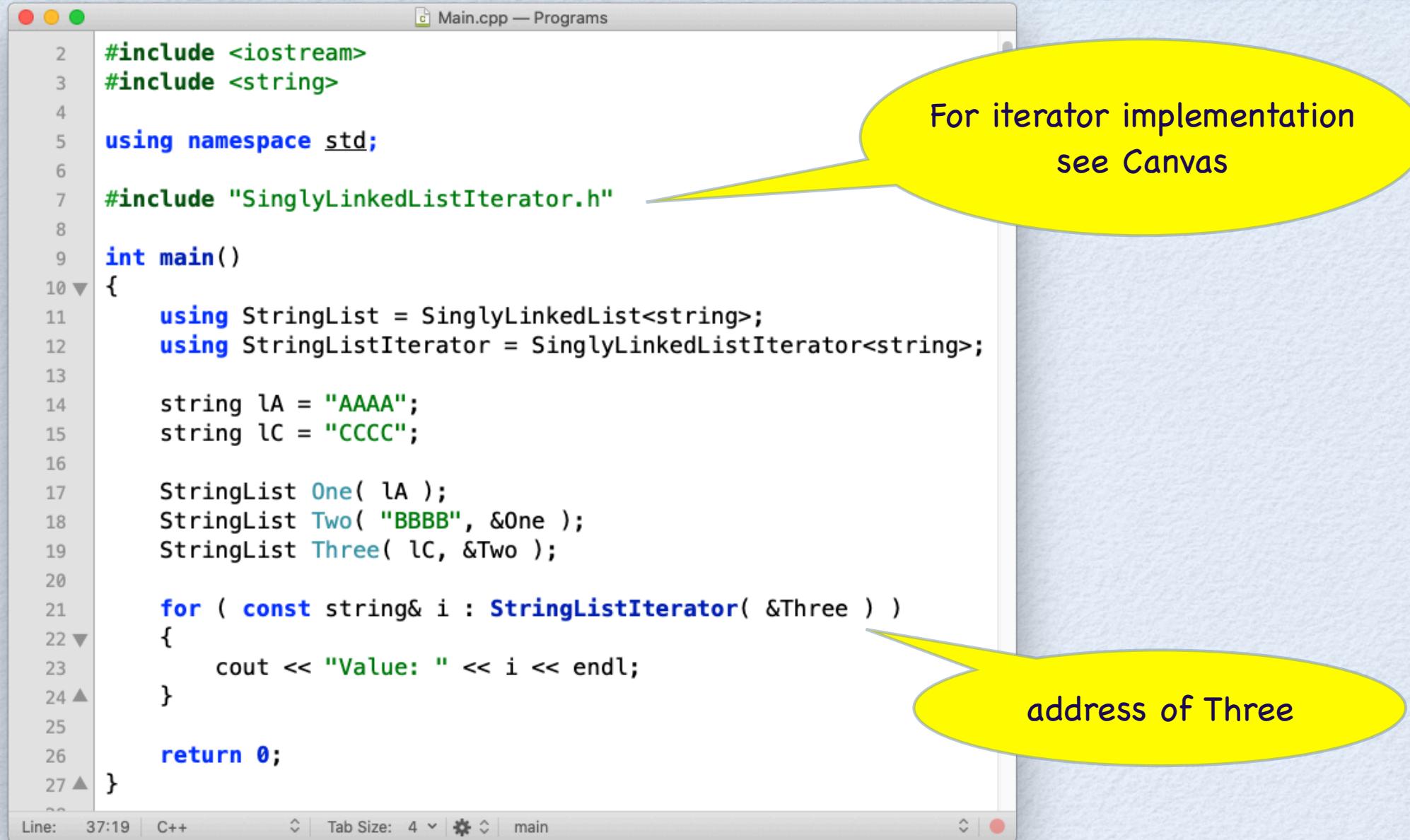


The screenshot shows a code editor window titled "SinglyLinkedListIterator.h". The code defines two iterator methods: `begin()` and `end()`. Both methods return an `Iterator` object. The `begin()` method initializes the iterator to point to the start of the list, while the `end()` method initializes it to point to the end of the list. The code uses the `// for-range feature` comment to indicate the purpose of these methods.

```
58     Iterator begin()          // for-range feature
59 {
60     Iterator iter = *this;
61
62     iter.fIndex = iter.fList;
63
64     return iter;
65 }
66
67     Iterator end()           // for-range feature
68 {
69     Iterator iter = *this;
70
71     iter.fIndex = nullptr;
72
73     return iter;
74 }
```

Line: 15 Column: 1 | C++ | Tab Size: 4 | SinglyLinkedListIterator

# SinglyLinkedList Iterator Test



```
2 #include <iostream>
3 #include <string>
4
5 using namespace std;
6
7 #include "SinglyLinkedListIterator.h"
8
9 int main()
10 {
11     using StringList = SinglyLinkedList<string>;
12     using StringListIterator = SinglyLinkedListIterator<string>;
13
14     string lA = "AAAA";
15     string lC = "CCCC";
16
17     StringList One( lA );
18     StringList Two( "BBBB", &One );
19     StringList Three( lC, &Two );
20
21     for ( const string& i : StringListIterator( &Three ) )
22     {
23         cout << "Value: " << i << endl;
24     }
25
26     return 0;
27 }
```

Line: 37:19 | C++ | Tab Size: 4 | main

For iterator implementation  
see Canvas

address of Three

# SinglyLinkedList Iterator: Specification B

```
1 #include "SinglyLinkedListTemplate.h"
2
3 template <typename T>
4 class SinglyLinkedListIterator
5 {
6     private:
7
8         using ListNode = SinglyLinkedList<T>;
9
10        const ListNode& fList;
11        const ListNode* fIndex;
12
13    public:
14
15        using Iterator = SinglyLinkedListIterator<T>;
16
17        SinglyLinkedListIterator( const ListNode& aList );
18
19        const T& operator*() const;
20        Iterator& operator++();      // prefix
21        Iterator operator++(int);   // postfix
22        bool operator==( const Iterator& aRHS ) const;
23        bool operator!=( const Iterator& aRHS ) const;
24
25        Iterator begin();           // for-range feature
26        Iterator end();            // for-range feature
27
28    };
29
```

Line: 31 | C++ | Tab Size: 4 | end

We maintain a read-only  
reference to list elements

# Reference Data Members

```
class ClassWithRefMember
{
private:
    SomeType& fRef;
public:
    ClassWithRefMember( SomeType& aRef ) : fRef(aRef)
    { ... }
};
```

Reference data members must be initialized before the constructor body is entered!

- Reference member variables store references to data outside an object. These references are established upon object creation via a member initializer.
- In case of iterators this might be an attractive option to avoid coping the underlying collection.
- Important: Reference data members require a constructor initializer.

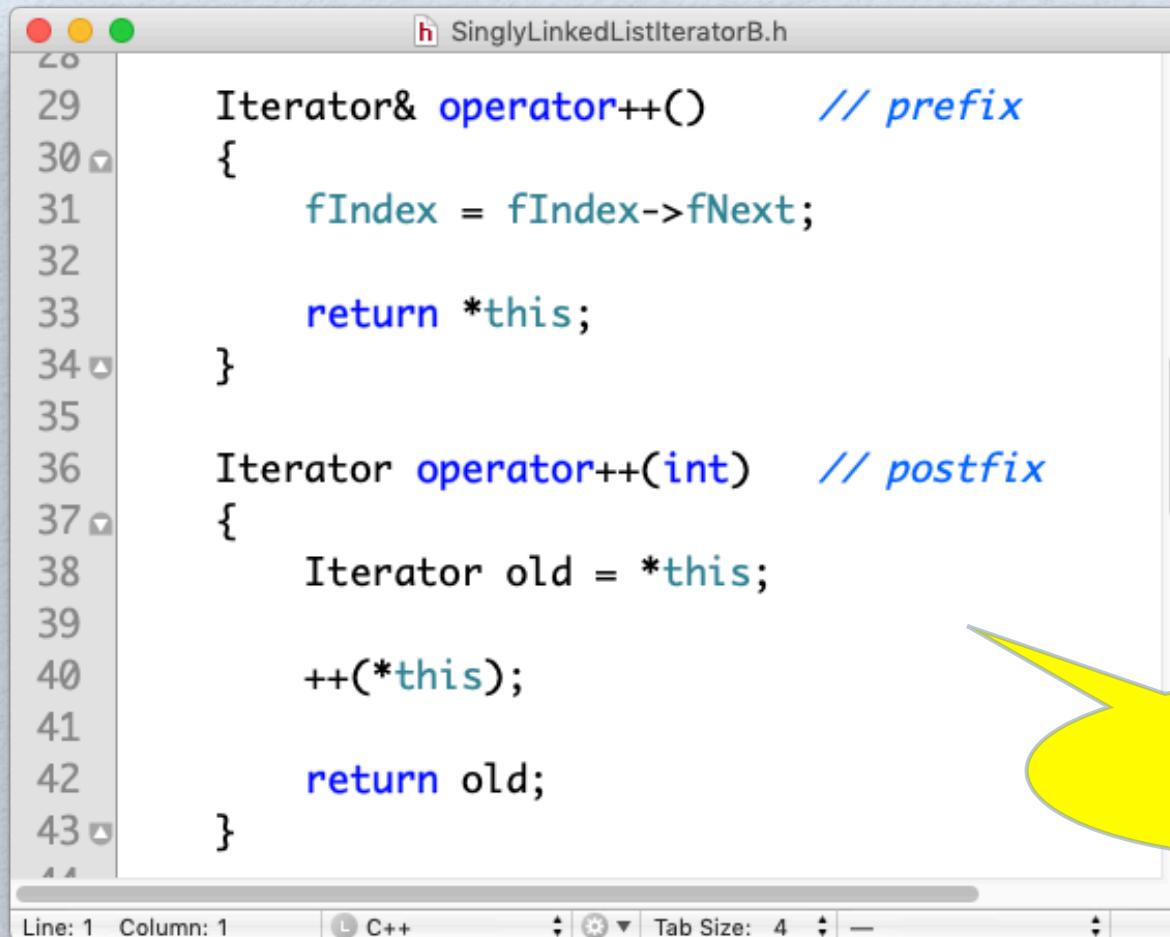
# Constructor & Deference B

```
using Iterator = SinglyLinkedListIterator<T>;  
  
SinglyLinkedListIterator( const ListNode& aList ) :  
    fList(aList),  
    fIndex(&aList)  
{}  
  
const T& operator*() const  
{  
    return fIndex->fData;  
}
```

Establish reference

fIndex is a pointer

# Increments B



A screenshot of a C++ code editor window titled "SinglyLinkedListIteratorB.h". The code defines two operator overloads for the iterator class:

```
29     Iterator& operator++()      // prefix
30 {
31     fIndex = fIndex->fNext;
32
33     return *this;
34 }
35
36     Iterator operator++(int)    // postfix
37 {
38     Iterator old = *this;
39
40     ++(*this);
41
42     return old;
43 }
```

The code uses color-coded syntax highlighting: red for keywords like `operator`, blue for types like `Iterator`, green for identifiers like `fIndex`, and teal for comments like `// prefix`. Line numbers are visible on the left.

unchanged

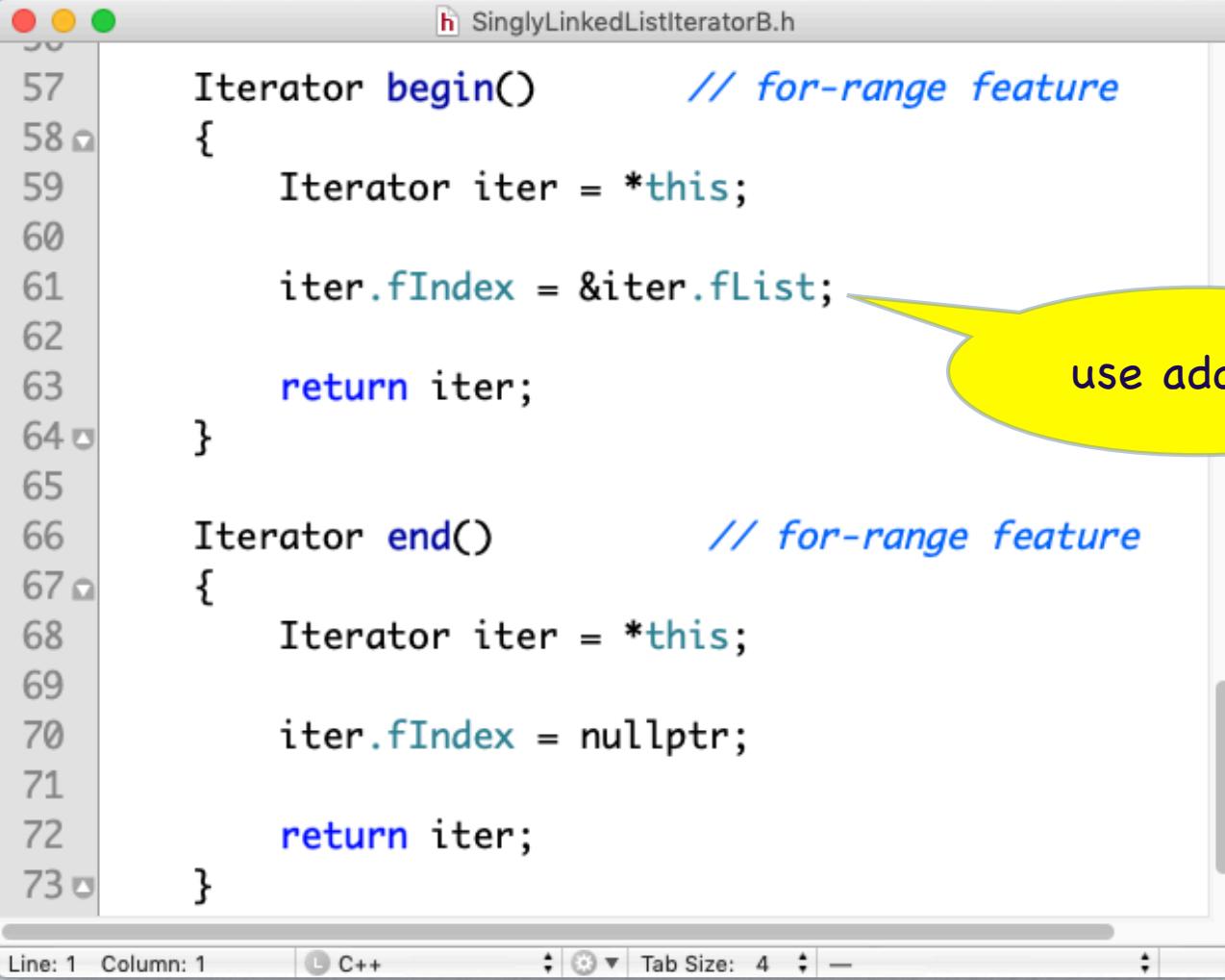
# Equivalence B



```
h SinglyLinkedListIteratorB.h
45     bool operator==( const Iterator& aRHS ) const
46     {
47         return
48             &fList == &aRHS.fList &&
49                 fIndex == aRHS.fIndex;
50     }
51
52     bool operator!=( const Iterator& aRHS ) const
53     {
54         return !(*this == aRHS);
55     }
56
```

Line: 1 Column: 1 C++ Tab Size: 4 —

# Auxiliaries (For-Range) B



The screenshot shows a code editor window titled "SinglyLinkedListIteratorB.h". The code contains two methods: `begin()` and `end()`. Both methods return an `Iterator` object. The `begin()` method initializes the `Iterator` to point to the current object (`*this`) and sets its `fIndex` to a pointer to its internal list (`&fList`). The `end()` method initializes the `Iterator` to point to the current object (`*this`) and sets its `fIndex` to `nullptr`. A yellow callout bubble points to the line `iter.fIndex = &iter.fList;` with the text "use address".

```
57     Iterator begin()          // for-range feature
58 {
59     Iterator iter = *this;
60
61     iter.fIndex = &iter.fList;
62
63     return iter;
64 }
65
66     Iterator end()          // for-range feature
67 {
68     Iterator iter = *this;
69
70     iter.fIndex = nullptr;
71
72     return iter;
73 }
```

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

# SinglyLinkedList Iterator Test B

```
2 #include <iostream>
3 #include <string>
4
5 using namespace std;
6
7 #include "SinglyLinkedListIteratorB.h"
8
9 int main()
10 {
11     using StringList = SinglyLinkedList<string>;
12     using StringListIterator = SinglyLinkedListIterator<string>;
13
14     string lA = "AAAA";
15     string lC = "CCCC";
16
17     StringList One( lA );
18     StringList Two( "BBBB", &One );
19     StringList Three( lC, &Two );
20
21     for ( const string& i : StringListIterator( Three ) )
22     {
23         cout << "Value: " << i << endl;
24     }
25
26     return 0;
27 }
```

For iterator implementation  
see Canvas

Three passed as  
l-value reference

# Pointers

# The Need for Pointers

- A linked-list is a dynamic data structure with a varying number of nodes.
- Access to a linked-list is through a pointer variable in which the **base type** is the same as the node type:

```
IntegerList<int>* pListOfIntegers = &Three;
```

```
IntegerList<int>* Nil = nullptr;
```

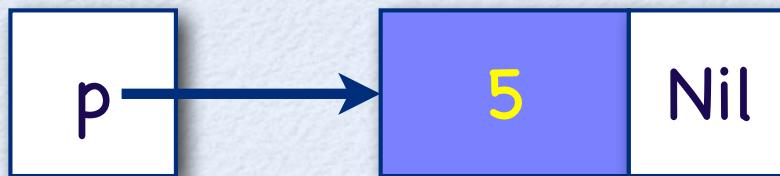
Nil means “empty list.”

# Node Construction

```
IntegerList *p, *q;
```

```
p = new IntegerList( 5 );
```

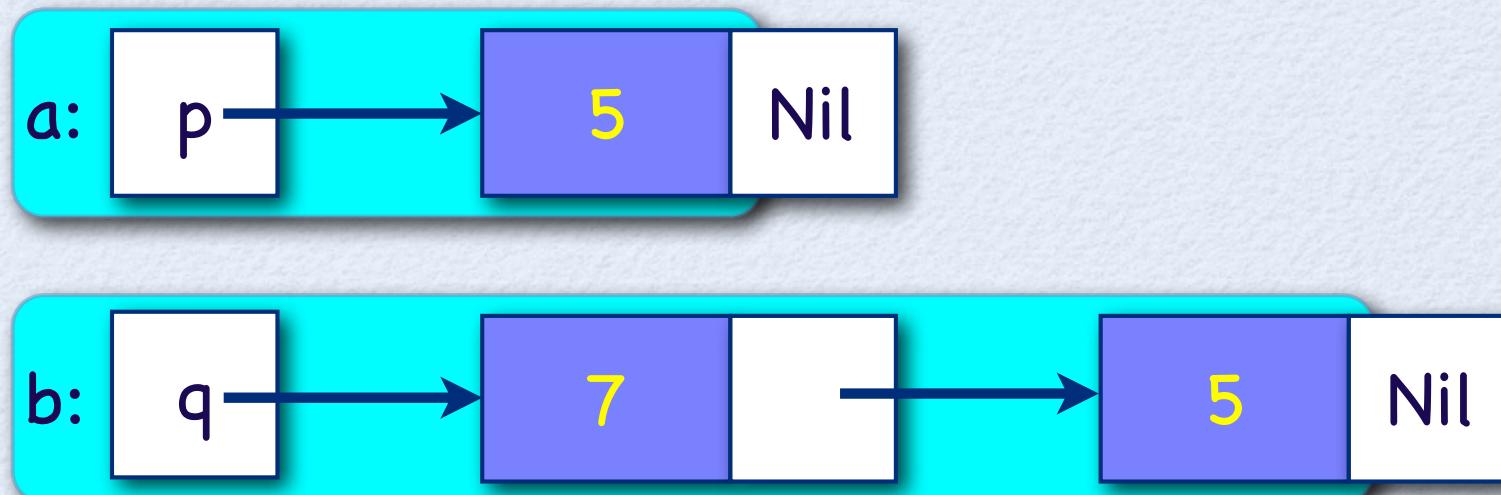
```
q = new IntegerList( 7, p );
```



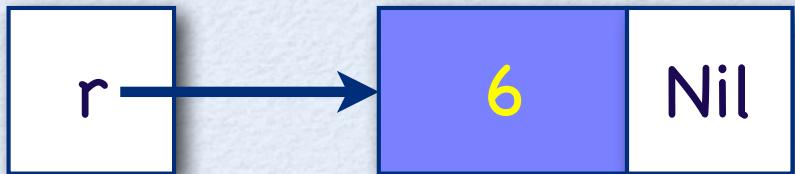
# Node Access

```
int a = p->fData;
```

```
int b = q->fNext->fData;
```

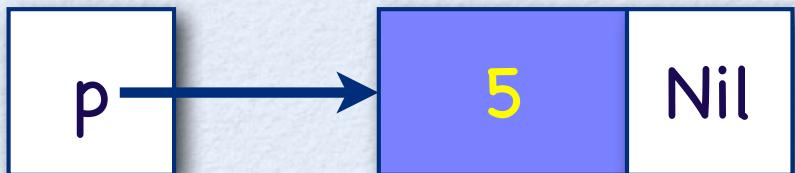


# Inserting a Node

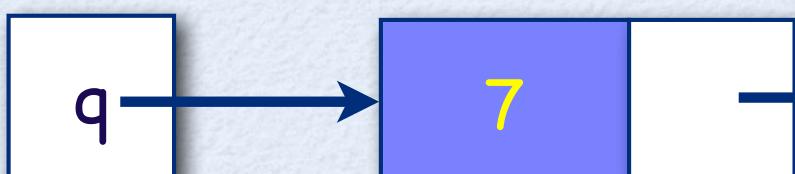


```
IntegerList *r;
```

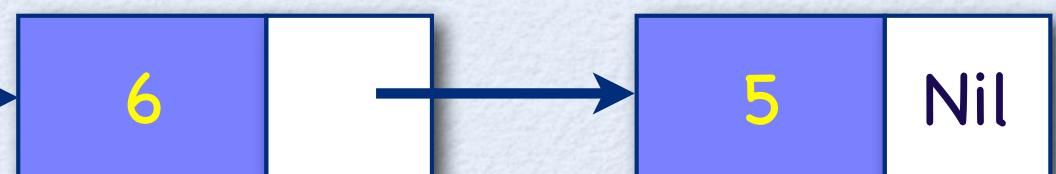
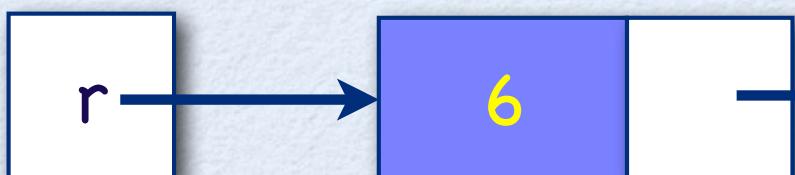
```
r = new IntegerList( 6 );
```



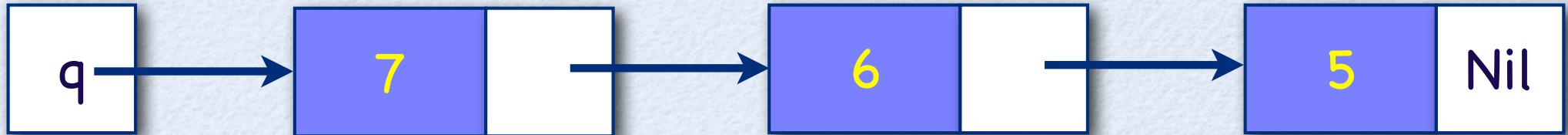
```
r->fNext = p;
```



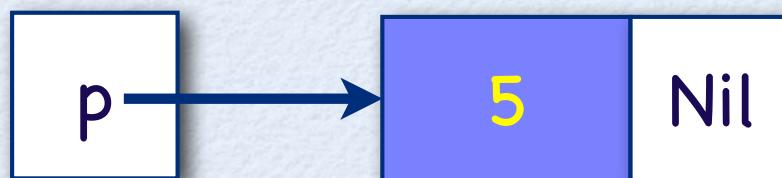
```
q->fNext = r;
```



# Deleting a Node

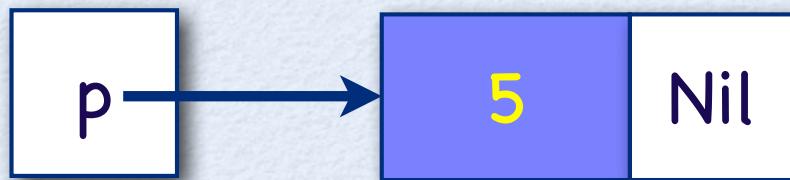


*q->fNext = q->fNext->fNext;*



# Insert at the Top

```
IntegerList *p = nullptr;  
p = new IntegerList( 5, p );
```

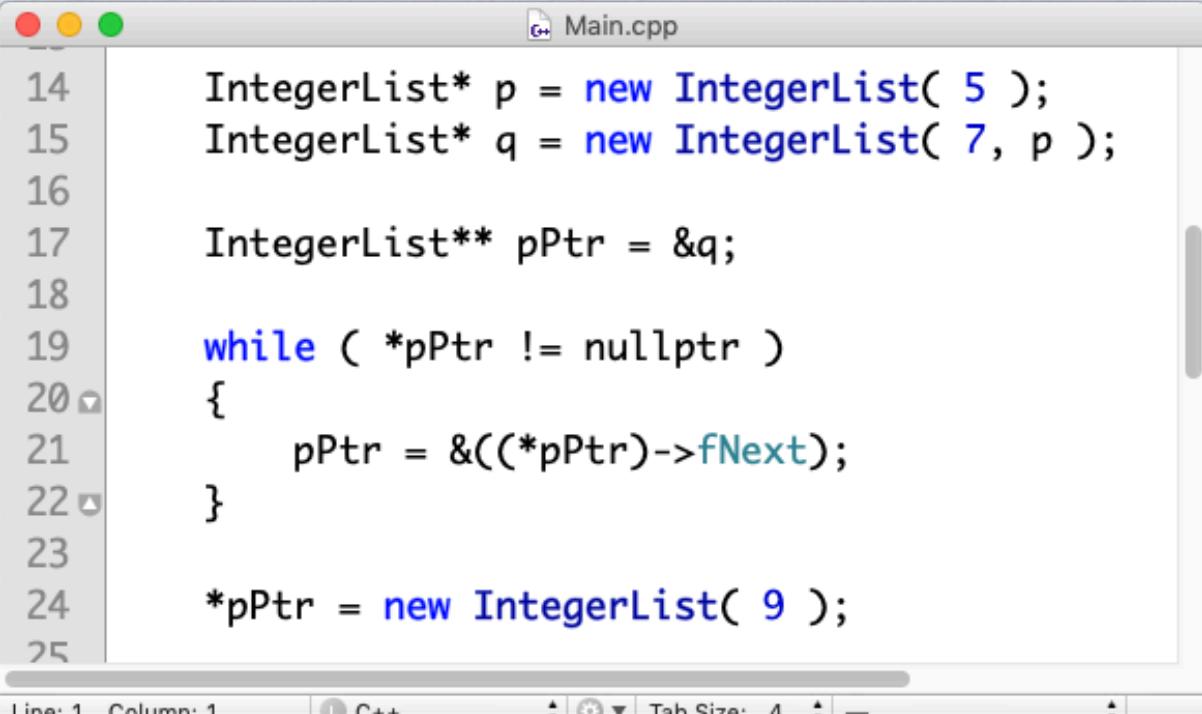


```
p = new IntegerList( 7, p );
```



# Insert at the End

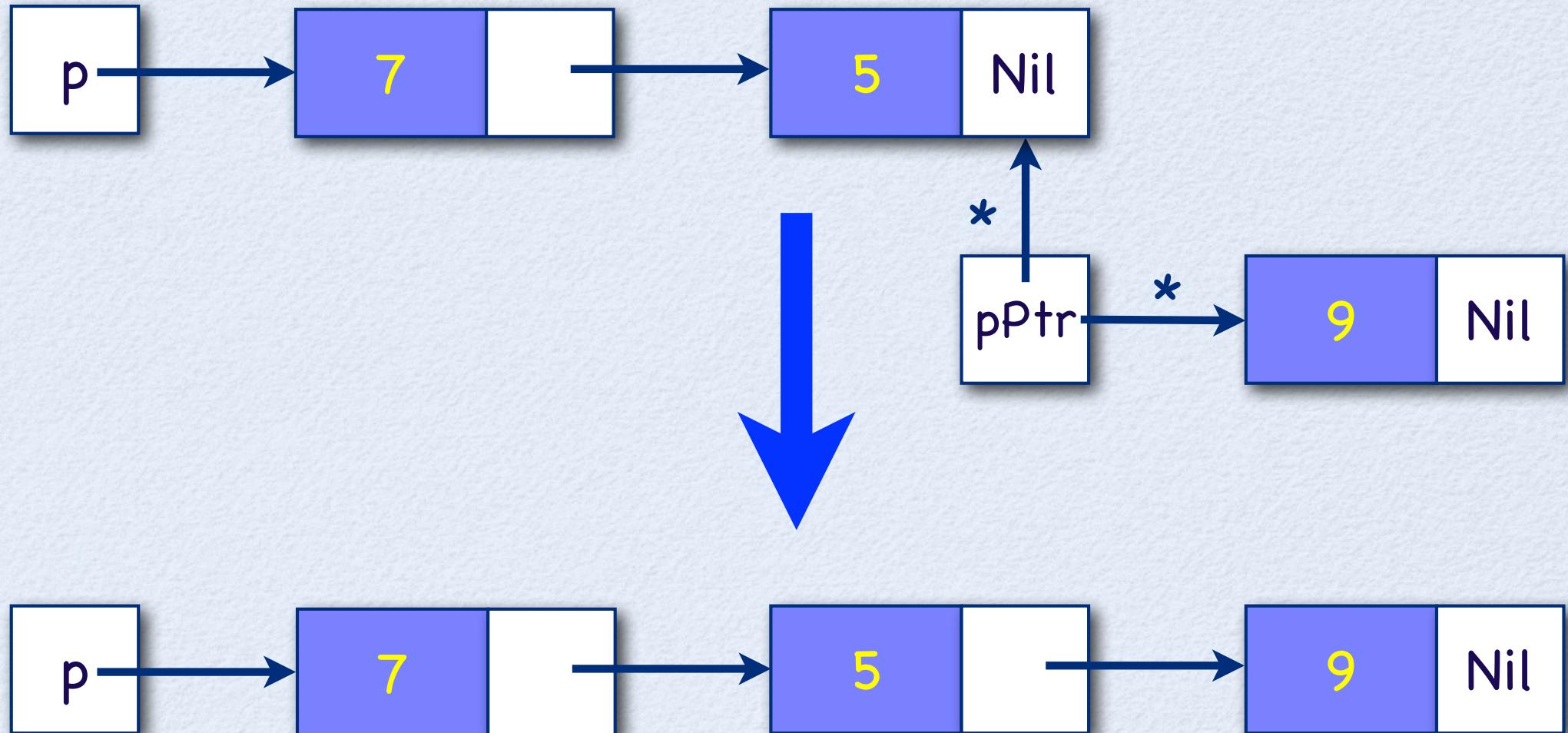
- To insert a new node at the end of a linked list we need to search for the end:



```
14 IntegerList* p = new IntegerList( 5 );
15 IntegerList* q = new IntegerList( 7, p );
16
17 IntegerList** pPtr = &q;
18
19 while ( *pPtr != nullptr )
20 {
21     pPtr = &(*pPtr)->fNext;
22 }
23
24 *pPtr = new IntegerList( 9 );
25
```

The screenshot shows a Mac OS X application window titled "Main.cpp". The window contains a text editor with C++ code. The code defines two pointers, p and q, both pointing to objects of type IntegerList. The pointer p is created with a value of 5, and q is created with a value of 7 and points to p. A loop then iterates, moving the pointer pPtr from p to the next node in the list until it reaches a null pointer. Finally, a new node with a value of 9 is inserted at the end of the list, pointing to the previous last node.

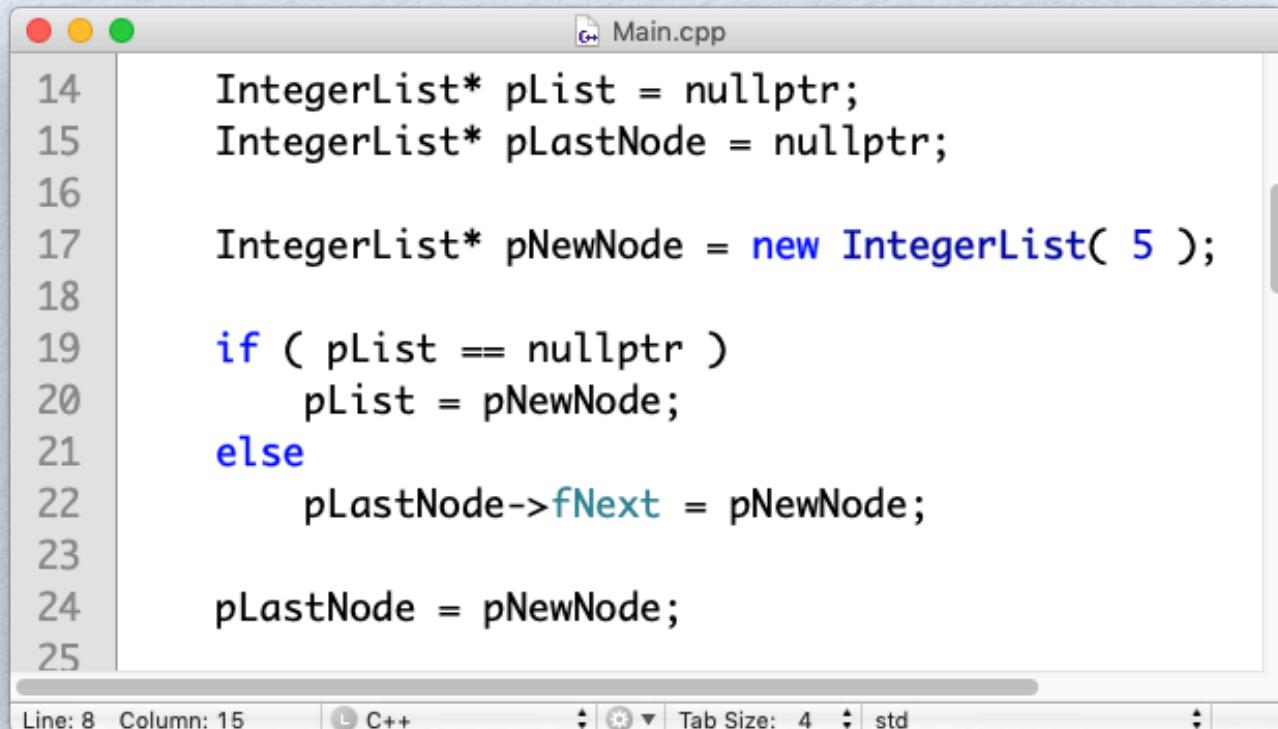
# Insert at the End: The Pointers



**Insert at the end preserves  
the order of list nodes.**

# Insert at the End with Aliasing

- Rather than using a Pointer-to-Pointer we can just record the last next pointer.



The screenshot shows a code editor window titled "Main.cpp". The code implements the insertion of a new node at the end of a singly linked list. The list is defined by pointers `pList` and `pLastNode`. A new node `pListNode` is created with value 5. If the list is empty (`pList == nullptr`), `pList` is set to `pListNode`. Otherwise, the `fNext` pointer of the previous last node (`pLastNode`) is updated to point to `pListNode`. Finally, `pLastNode` is set to `pListNode`. The code uses standard C++ syntax with `new` and `delete` operators for dynamic memory management.

```
14 IntegerList* pList = nullptr;
15 IntegerList* pLastNode = nullptr;
16
17 IntegerList* pListNode = new IntegerList( 5 );
18
19 if ( pList == nullptr )
20     pList = pListNode;
21 else
22     pLastNode->fNext = pListNode;
23
24 pLastNode = pListNode;
25
```

# Complications with Singly-Linked Lists

- The deletion of a node at the end of a list requires a search from the top to find the new last node.

# Abstraction

# New Sets of Values

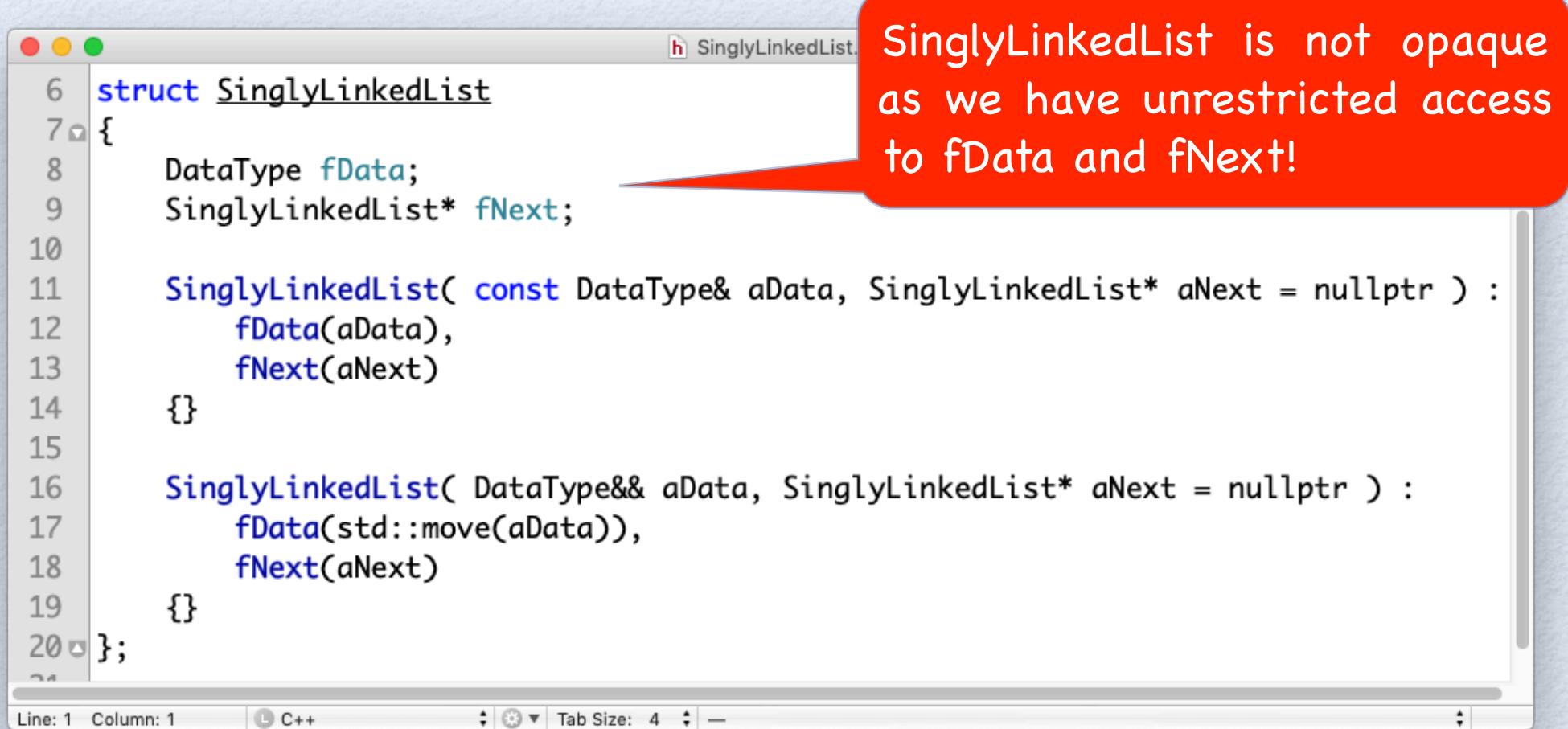
- The definition of a new data type (i.e., a new set of values) consists of two ingredients:
  - Some set, called the **interface**, that serves as representation of the newly define data type, and
  - Some set of procedures, called the **implementation**, that provides the operations, which can be used to manipulate the newly defined data type.

# Representation Independence

- The representation of new data types can be often very complex.
- When working with new data types, we usually do not want to be concerned with their actual representation. In fact, programs become more reliable and robust, if they do not depend on the actual representation of data type. Data types that do not expose their actual representation are called opaque. Otherwise, they are called transparent.
- Data types in C/C++ are in general transparent (e.g. the size of integers in C/C++ is platform dependent).
- Data types in Java are basically opaque (arrays are an exception, since they are represented by objects).

# Opaque Representation

- A data type is opaque if there is no way to find out its representation, even by printing.



SinglyLinkedList is not opaque as we have unrestricted access to fData and fNext!

```
6 struct SinglyLinkedList
7 {
8     DataType fData;
9     SinglyLinkedList* fNext;
10
11    SinglyLinkedList( const DataType& aData, SinglyLinkedList* aNext = nullptr ) :
12        fData(aData),
13        fNext(aNext)
14    {}
15
16    SinglyLinkedList( DataType&& aData, SinglyLinkedList* aNext = nullptr ) :
17        fData(std::move(aData)),
18        fNext(aNext)
19    {}
20};
21
```

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

# Object-Oriented Encapsulation

- Object-oriented encapsulation is a principle that provides the means to obtain an opaque data type:
  - All instance variables have private visibility.
  - All member functions have public visibility.
- The extent to which this scheme is used can differ!
- Opaque here does not necessarily mean that you cannot see the implementation, just that there are no pragmatic approaches to exploit this knowledge.

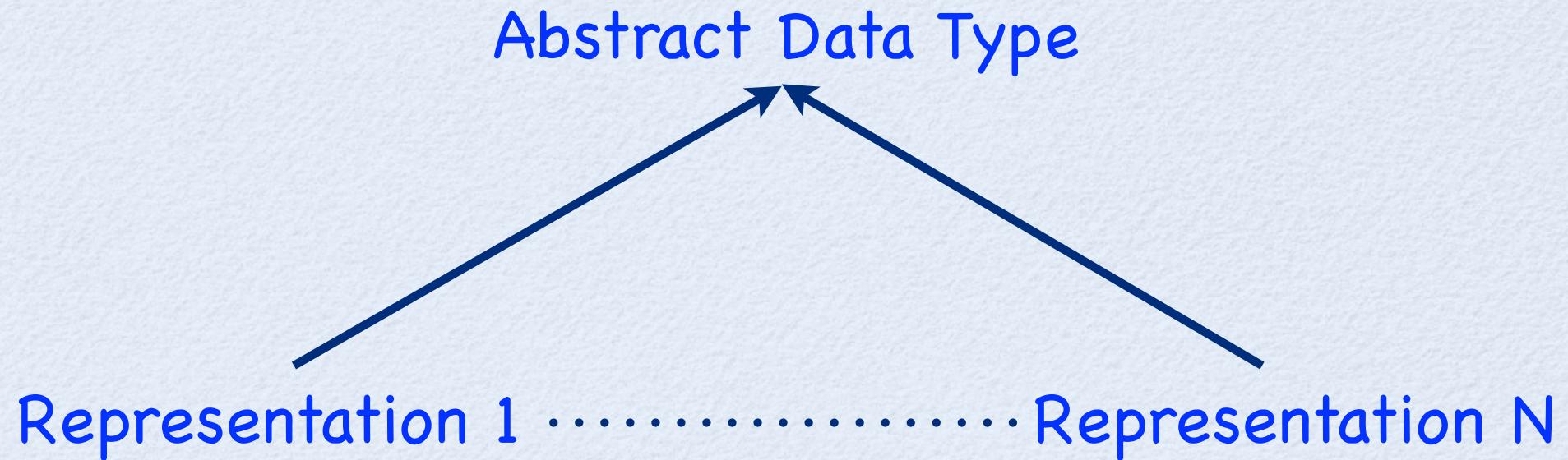
# Pros & Cons

- Opaque data types enforce the use of defining procedure (i.e., a constructor).
- Opaque data types are more secure. Access to values of opaque data types is only possible by means of access procedures defined in an interface.
- Transparent data types are easier to debug and to extend.
- The fact that transparent data types expose their internal representation is also a disadvantage (limited security).

# Abstract Data Type

- The technique used to define new data types independently of their actual representation is called **data abstraction**.
- A data type, which has been defined in this way is called **abstract data type**. A client (program) can use values of an abstract data type by means of the interface without knowing their actual representation (which can change over time).

# Representation Strategies for ADTs



- Given an interface for a data type we can change the underlying representation if needed using different strategies.

# Data Abstraction

- Data abstraction enforces representation independence.
- Data abstraction divides the data types in interfaces and implementations:
  - Interfaces are used to specify the set of values the data types represents, the operations, which are available for that data type, and properties these operations may be guaranteed to have.
  - Implementations provide a specific representation of the data and code for the operations.

# Examples of Abstract Data Types

- Files
- Lists, hash tables, vectors, bags
- Strings, records, arrays
- Objects with private instance variables and public methods
- Standardized integers (e.g. in Java the type int is represented using 32 bits and big endian format, network byte order, on every platform)

# Constructors and Access Procedures

- In order to create, manipulate, and verify that a given value is of the desired data type, we need the following ingredients:
  - Constructors that allow us to build values of a given data type,
  - A predicate that tests whether a given value is a representation of a particular data type, and
  - Some access procedures that allow us to extract a particular information from a given representation of a data type.

# Can we represent lists as abstract data type?

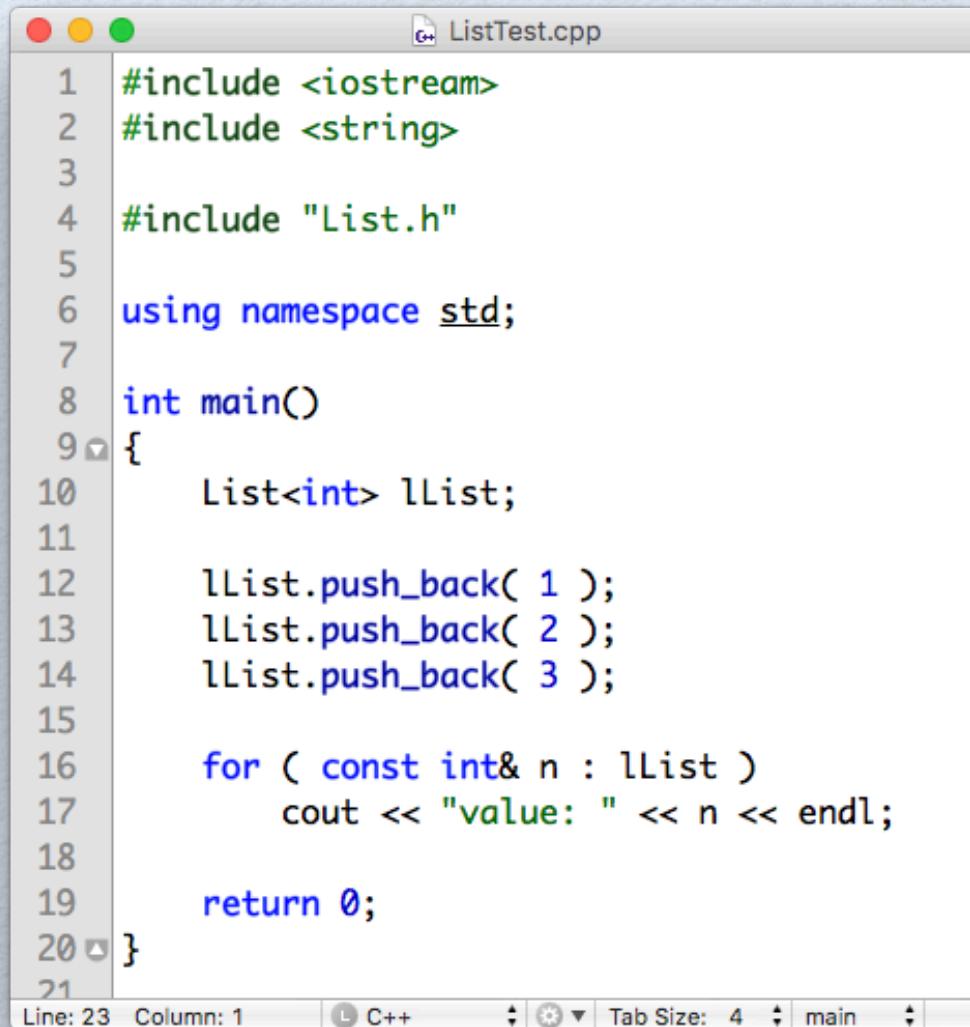
```

6 #include "DoublyLinkedList.h"
7 #include "DoublyLinkedListIterator.h"
8
9 #include <stdexcept>
10
11 template<class T>
12 class List
13 {
14 private:
15     // auxiliary definition to simplify node usage
16     using Node = DoublyLinkedList<T>;
17
18     Node* fRoot;    // the first element in the list
19     int fCount;     // number of elements in the list
20
21 public:
22     // auxiliary definition to simplify iterator usage
23     using Iterator = DoublyLinkedListIterator<T>;
24
25     List();                      // default constructor - creates empty list
26     List( const List& aOtherList ); // copy constructor
27     List( List&& aOtherList );   // move constructor
28     List& operator=( const List& aOtherList ); // assignment operator
29     List& operator=( List&& aOtherList );   // move assignment operator
30     ~List();                     // destructor - frees all nodes
31
32     bool isEmpty() const;        // Is list empty?
33     int size() const;           // list size
34
35     void push_front( const T& aElement ); // adds aElement at front (copy)
36     void push_front( T&& aElement );   // adds aElement at front (move)
37     void push_back( const T& aElement ); // adds aElement at back (copy)
38     void push_back( T&& aElement );   // adds aElement at back (move)
39     void remove( const T& aElement );  // remove first match from list
40
41     const T& operator[]( size_t aIndex ) const; // list indexer
42
43     Iterator begin() const;        // return a forward iterator
44     Iterator end() const;         // return a forward end iterator
45     Iterator rbegin() const;      // return a backwards iterator
46     Iterator rend() const;       // return a backwards end iterator
47 };

```

r-value variant

# List Test



```
1 #include <iostream>
2 #include <string>
3
4 #include "List.h"
5
6 using namespace std;
7
8 int main()
9 {
10     List<int> lList;
11
12     lList.push_back( 1 );
13     lList.push_back( 2 );
14     lList.push_back( 3 );
15
16     for ( const int& n : lList )
17         cout << "value: " << n << endl;
18
19     return 0;
20 }
```

Line: 23 Column: 1    C++    Tab Size: 4    main

The type `list<T>` is part of the standard template library.

# 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)

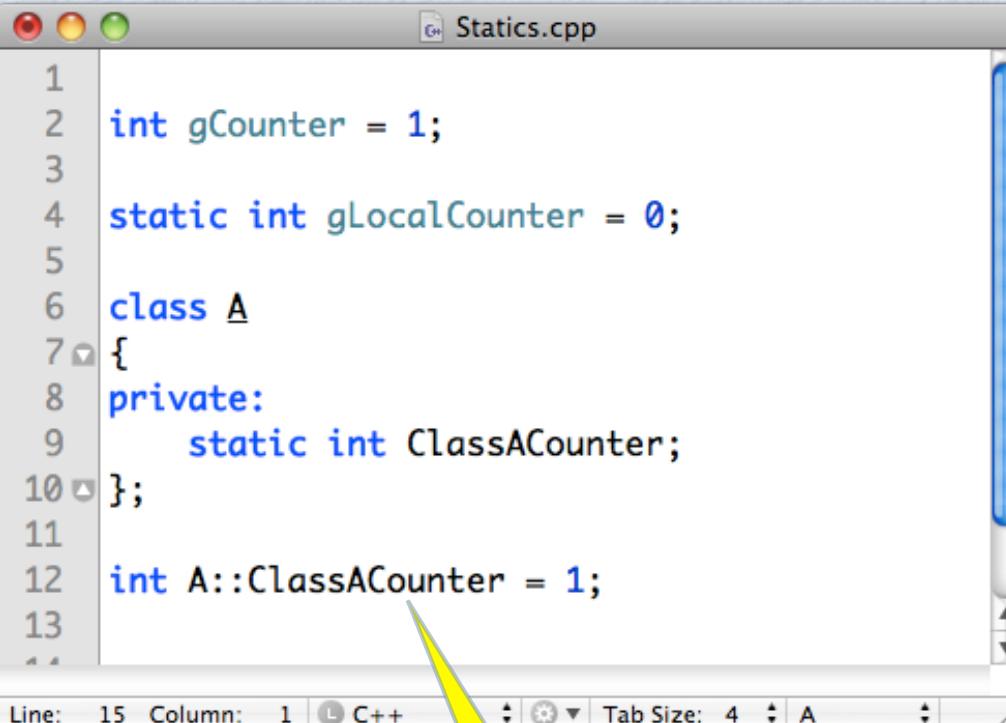
# Static Read-Write Memory

- 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 contains the following C++ code:

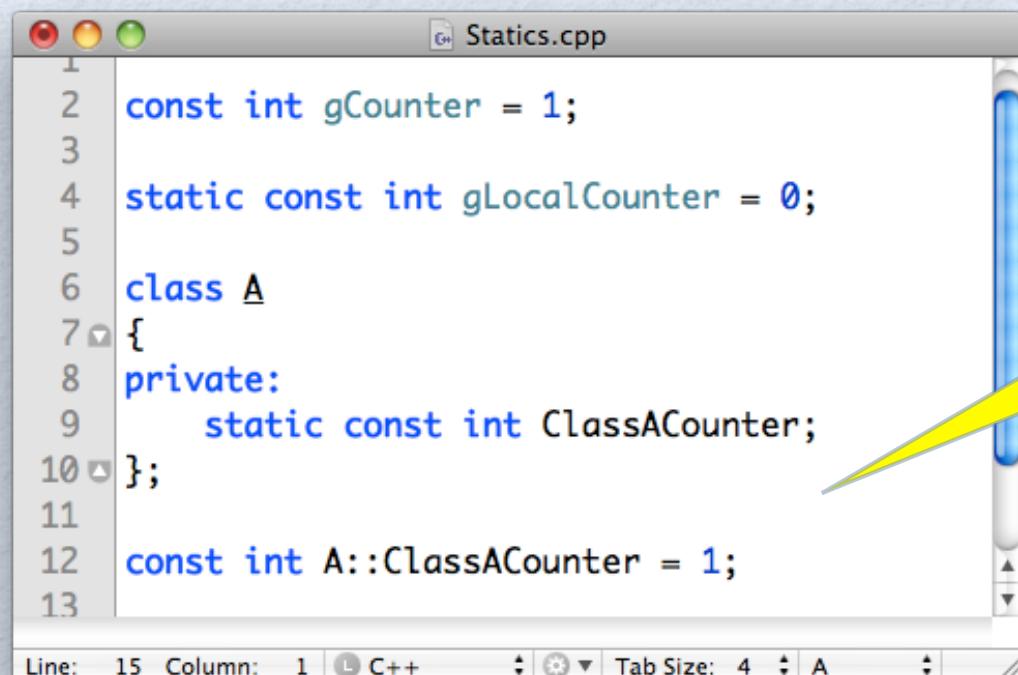
```
1 int gCounter = 1;
2
3 static int gLocalCounter = 0;
4
5
6 class A
7 {
8 private:
9     static int ClassACounter;
10}
11
12 int A::ClassACounter = 1;
```

The code illustrates the declaration and initialization of static variables. The global variable `gCounter` is initialized at line 1. The static local variable `gLocalCounter` is initialized at line 3. The static class variable `ClassACounter` is declared within class `A` at line 6 and initialized outside the class at line 12.

Static class variables must be initialized outside the class.

# Static Read-Only Memory

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



```
1  const int gCounter = 1;
2
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
```

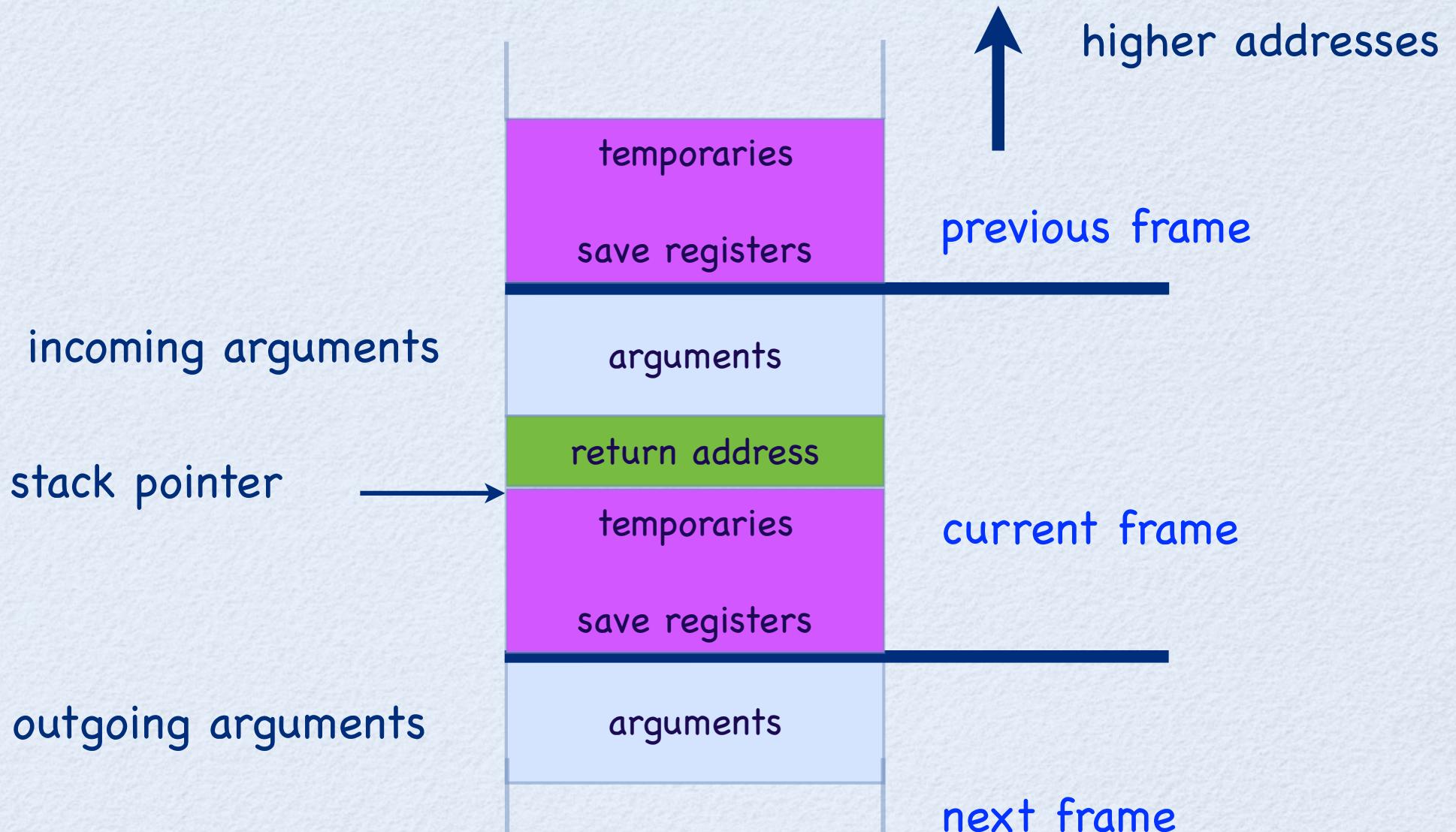
The screenshot shows a code editor window titled "Statics.cpp". The code contains several static const variables and a class definition. Line 12 shows a static const variable being defined within a class scope.

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()

The screenshot shows a code editor window titled "List.h" containing the destructor definition for the "List" class. The code uses a while loop to iterate through the list until the root node is null. If there is more than one element (i.e., the root's previous pointer is not null), it creates a temporary node lTemp pointing to the previous node of the root, isolates it, and then deletes it. This process removes the node from the list and frees its memory. If there is only one element (root's previous pointer is null), it simply deletes the root node and breaks out of the loop. A yellow callout bubble points to the delete lTemp; line, indicating that this is where memory associated with the list node object on the heap is released.

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

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

Release memory associated with  
list node object on the heap.

# The Dominion Over Objects

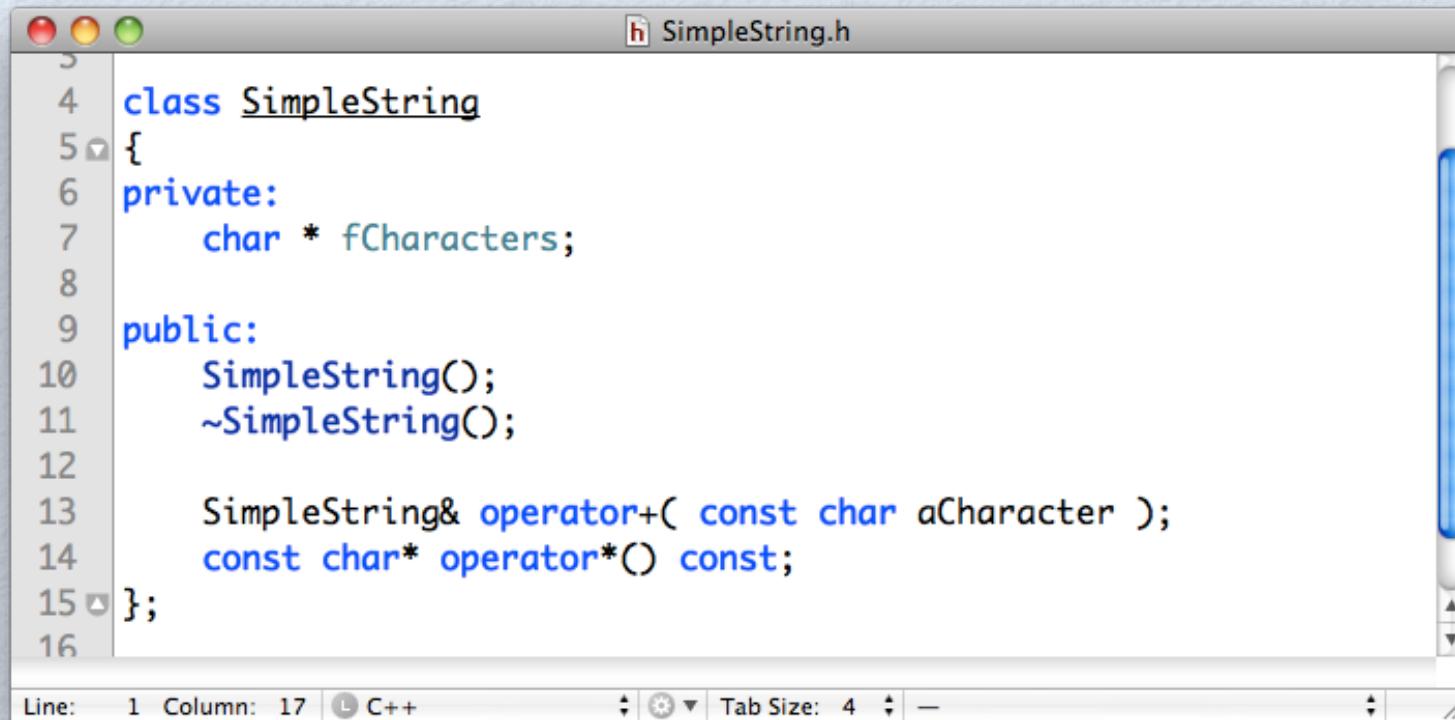
- Alias control is one of the most difficult problems to master in object-oriented programming.
- Aliases are the default in reference-based object models used, for example, in Java and C#.
- To guarantee uniqueness of value-based objects in C++, we are required to define copy constructors.

# 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

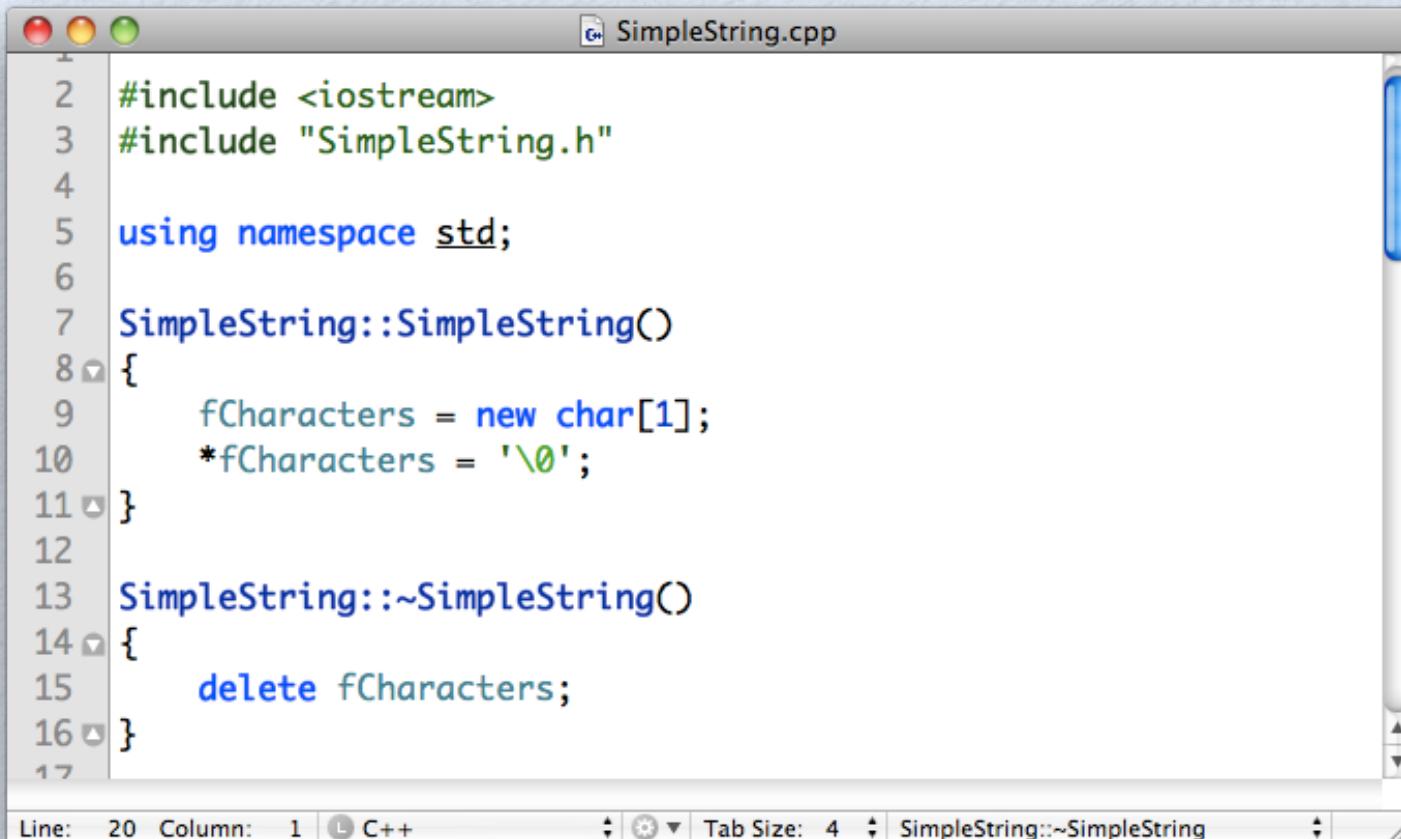


The screenshot shows a code editor window titled "SimpleString.h". The code defines a class `SimpleString` with private member `fCharacters` and public constructor/destructor and operators.

```
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
```

The status bar at the bottom indicates "Line: 1 Column: 17 C++" and "Tab Size: 4".

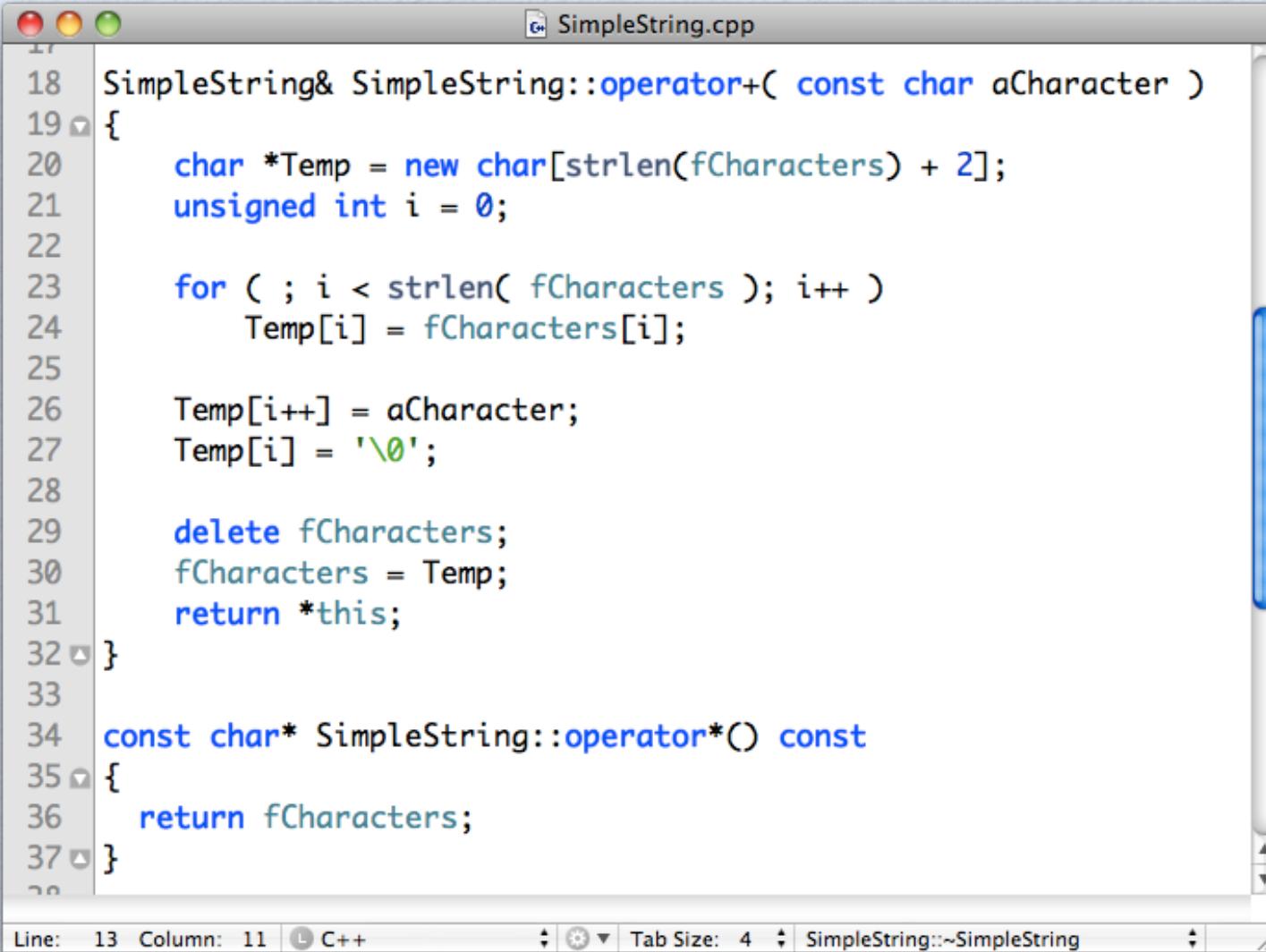
# SimpleString: Constructor & Destructor



```
SimpleString.cpp
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 implements two operators: the assignment operator and the dereference operator.

```
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 }
```

The status bar at the bottom indicates "Line: 13 Column: 11 C++" and "Tab Size: 4".

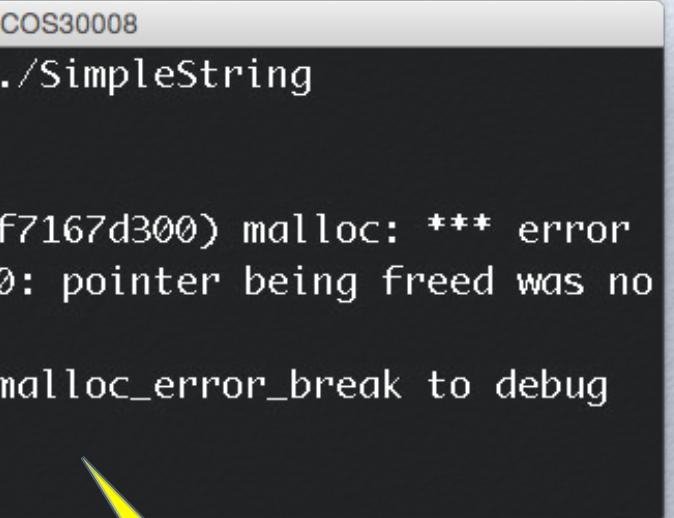
# Implicit Copy Constructor

The screenshot shows a Xcode interface. On the left is a code editor with the file `SimpleString.cpp` containing the following code:

```
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 }
```

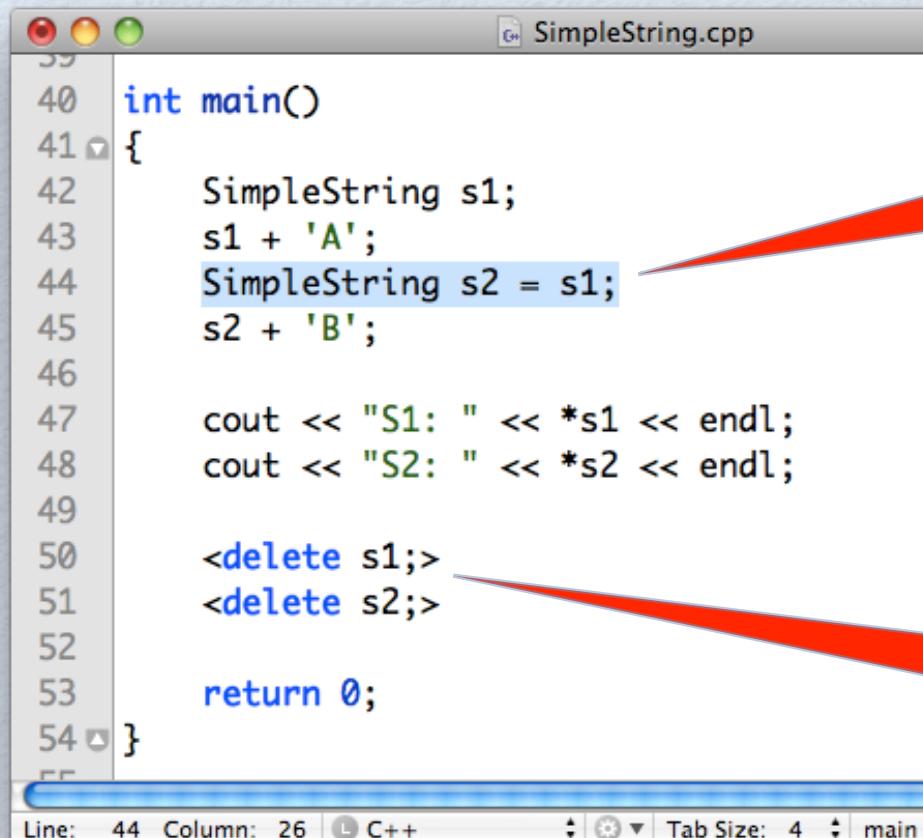
On the right is a terminal window titled `COS30008` showing the output of the program:

```
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 s1;
s1 + 'A';
SimpleString s2 = s1;
s2 + 'B';

cout << "S1: " << *s1 << endl;
cout << "S2: " << *s2 << endl;

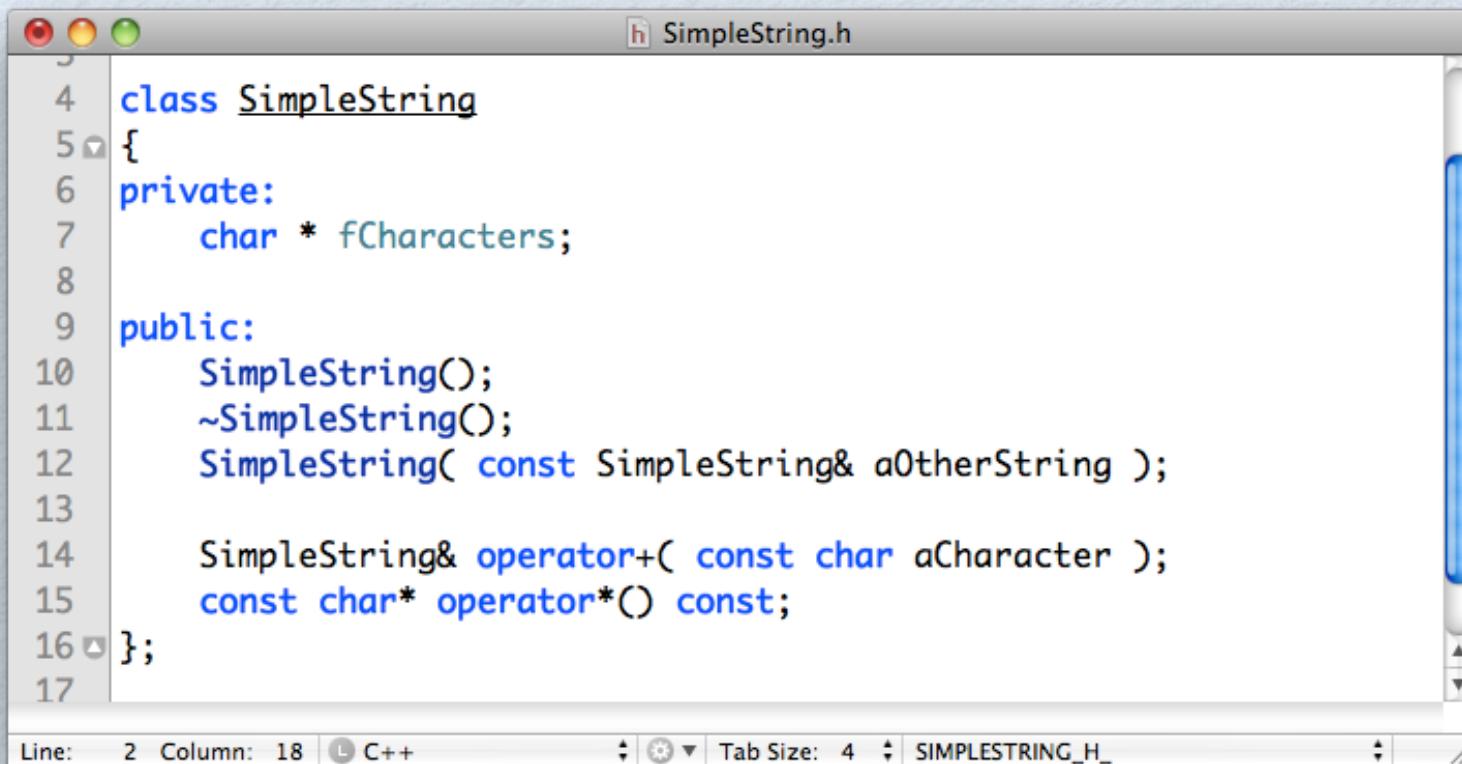
<delete s1;>
<delete s2;>

return 0;
```

**Shallow copy:**  
 $s2.fCharacters == s1.fCharacters$

**Double free:**  
delete  $s2.fCharacters$ , which was  
called in  $s2 + 'B'$ .

# We need an explicit copy constructor!

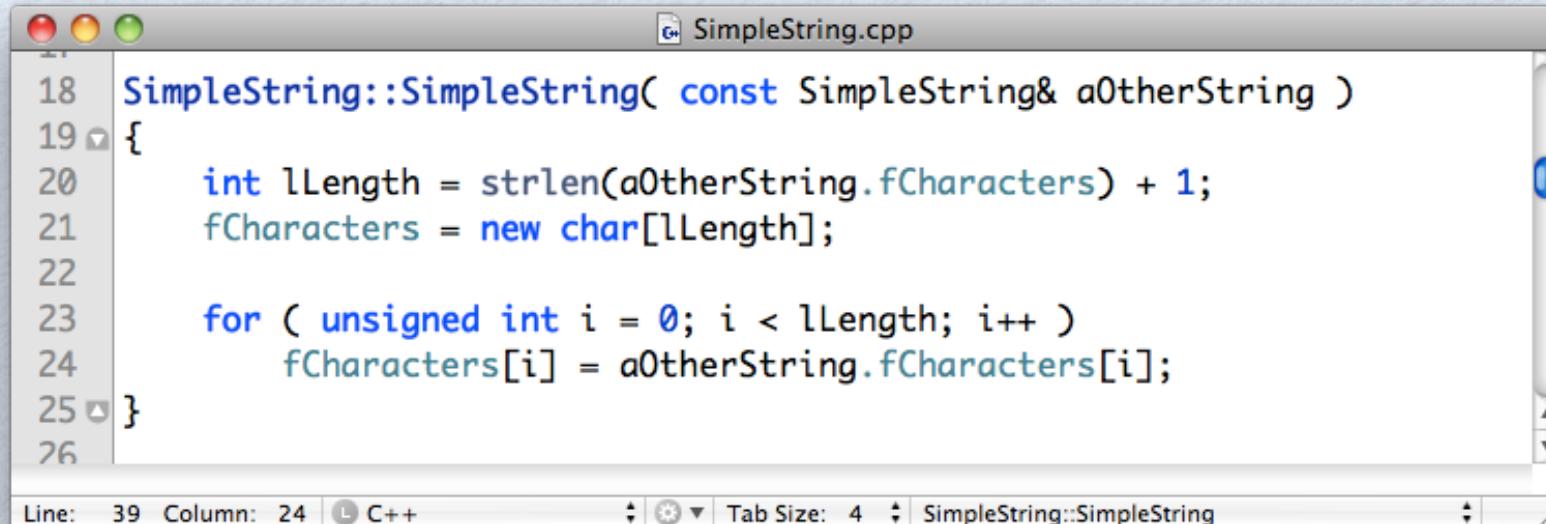


A screenshot of a Mac OS X-style text editor window titled "SimpleString.h". The code defines a class `SimpleString` with private member `fCharacters` and public constructor, destructor, and copy constructor. It also includes operator overloading for addition and conversion to a character pointer.

```
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
```

The status bar at the bottom shows "Line: 2 Column: 18 C++" and "SIMPLESTRING\_H\_".

# The Explicit Copy Constructor



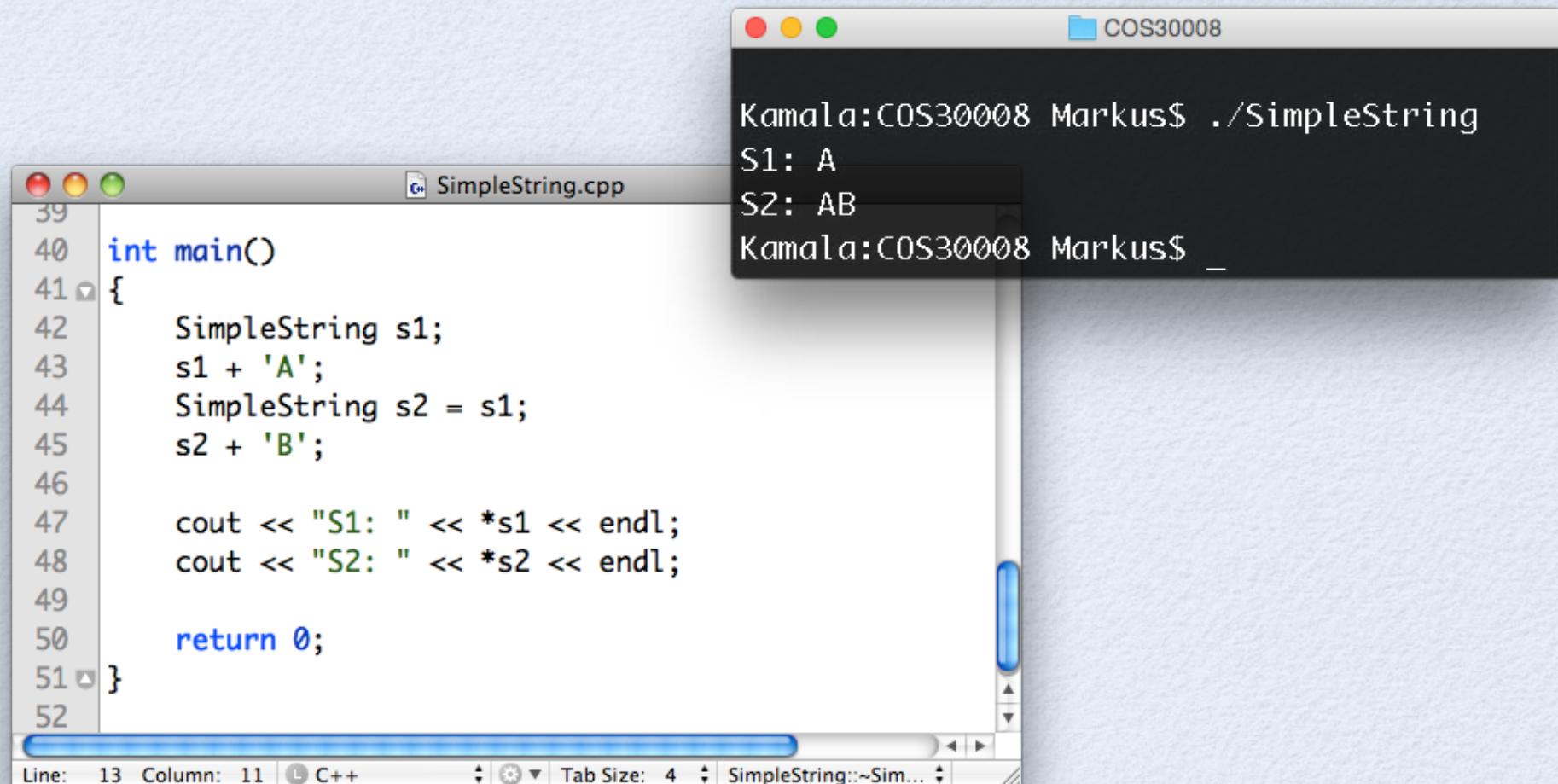
A screenshot of a Mac OS X-style code editor window titled "SimpleString.cpp". The code shown is the implementation of a copy constructor for the `SimpleString` class. It uses `strlen` to determine the length of the source string and `new` to allocate memory for a copy, then copies each character one by one.

```
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



The screenshot shows a Mac OS X desktop environment. On the left is a code editor window titled "SimpleString.cpp" containing C++ code. On the right is a terminal window titled "COS30008" showing the output of a program run.

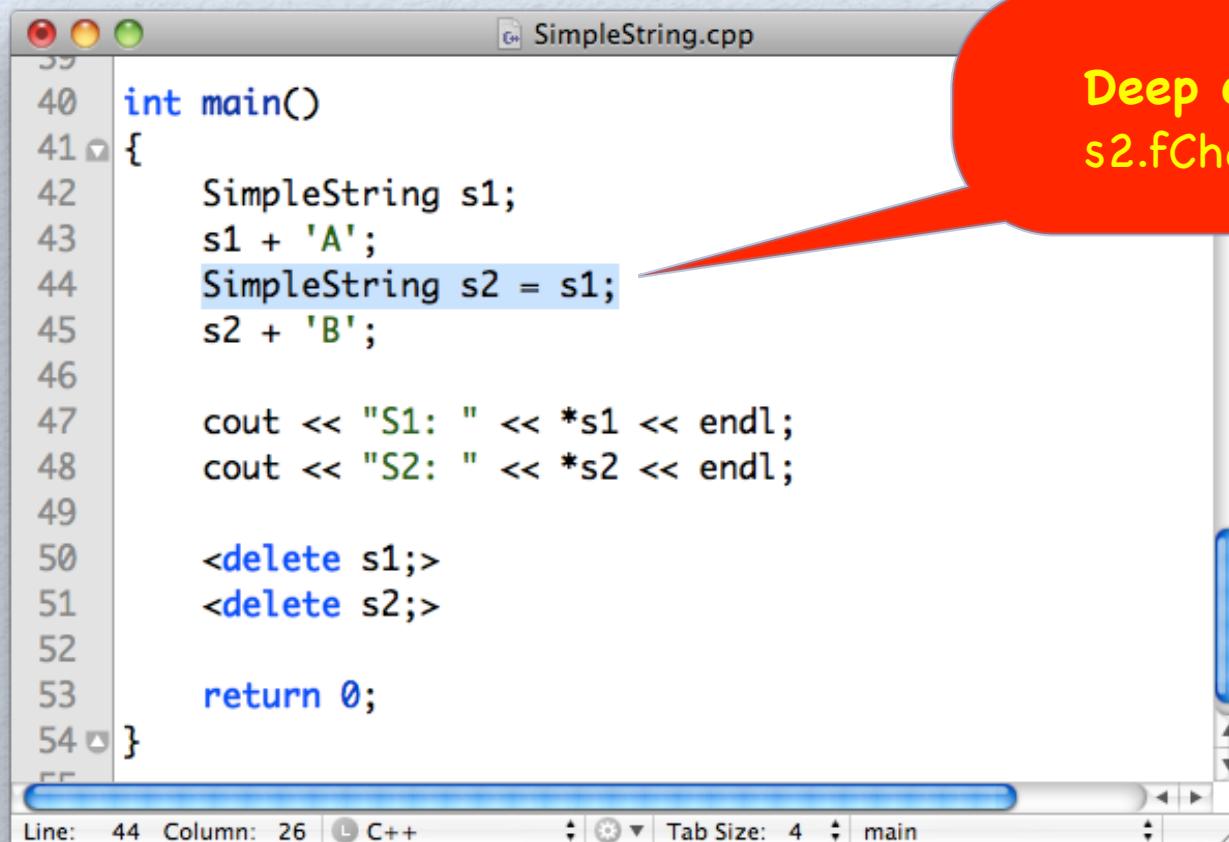
**Code Editor (SimpleString.cpp):**

```
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 }
```

**Terminal Output:**

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

# What Has Happened?



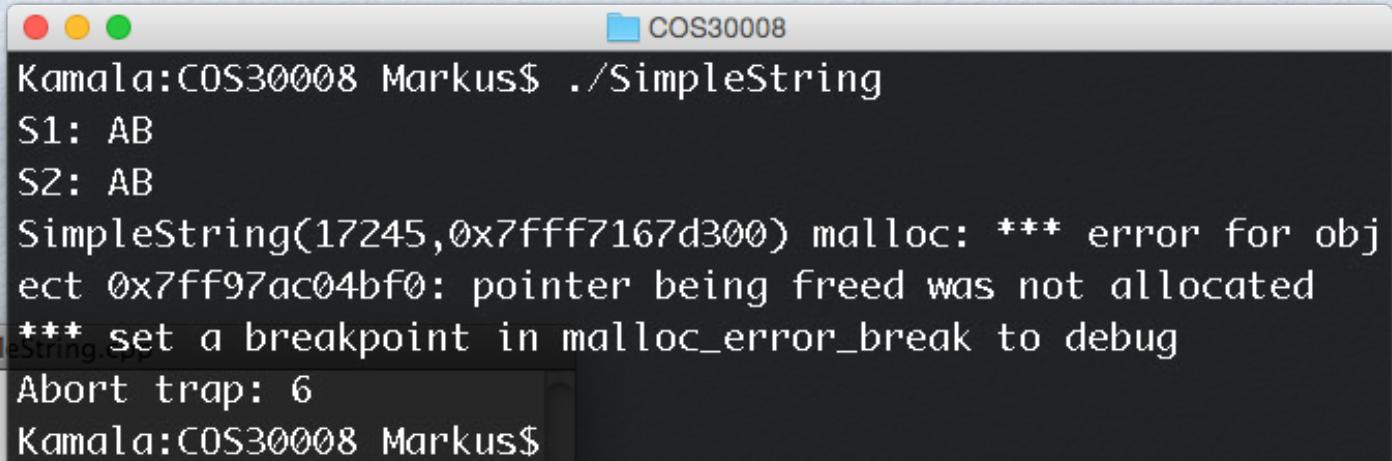
```
SimpleString.cpp
39
40 int main()
41 {
42     SimpleString s1;
43     s1 + 'A';
44     SimpleString s2 = s1;
45     s2 + 'B';
46
47     cout << "S1: " << *s1 << endl;
48     cout << "S2: " << *s2 << endl;
49
50     <delete s1;>
51     <delete s2;>
52
53     return 0;
54 }
```

The screenshot shows a code editor window titled "SimpleString.cpp". The code demonstrates a shallow copy of a string object. Line 44 creates a copy of `s1` and assigns it to `s2`. A red callout bubble points to this assignment line with the text "Deep copy:  
`s2.fCharacters != s1.fCharacters`". The code then prints both strings to the console and deletes both objects.

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

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

# A Simple Assignment



The screenshot shows a Mac OS X desktop environment. In the foreground, a terminal window titled "SimpleString" is open. The command "Kamala:COS30008 Markus\$ ./SimpleString" has been run. The output shows two lines of text: "S1: AB" and "S2: AB". Following this, an error message is displayed: "SimpleString(17245,0x7fff7167d300) malloc: \*\*\* error for object 0x7ff97ac04bf0: pointer being freed was not allocated". Below this, a note says "\*\*\* set a breakpoint in malloc\_error\_break to debug". In the background, a code editor window for "SimpleString.cpp" is visible. It contains C++ code for a main function. The code initializes two SimpleString objects, s1 and s2, and performs assignments and outputs. A cursor is visible at the end of the assignment "s1 = s2;". The status bar at the bottom of the code editor shows "Line: 56 Column: 13 C++ Tab Size: 4 main".

```
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 }
```

# 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 }
```

The screenshot shows a C++ IDE window titled "SimpleString.cpp". The code in the editor is as follows:

```
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 }
```

The line `s1 = s2;` is highlighted in blue. The status bar at the bottom of the IDE shows "Line: 56 Column: 13 C++ Tab Size: 4 main".

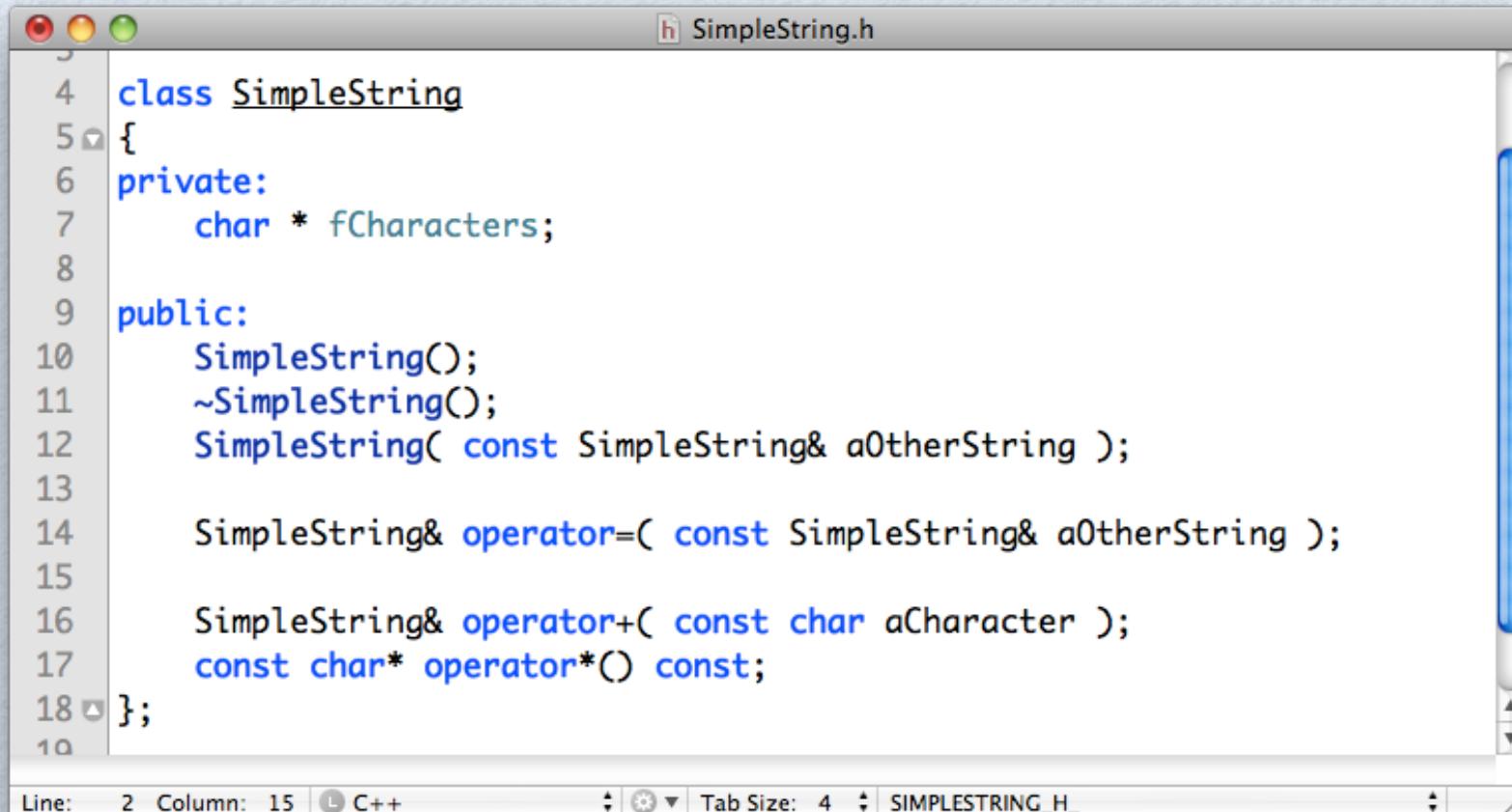
**Shallow copy:**  
`s2.fCharacters == s1.fCharacters`

**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!



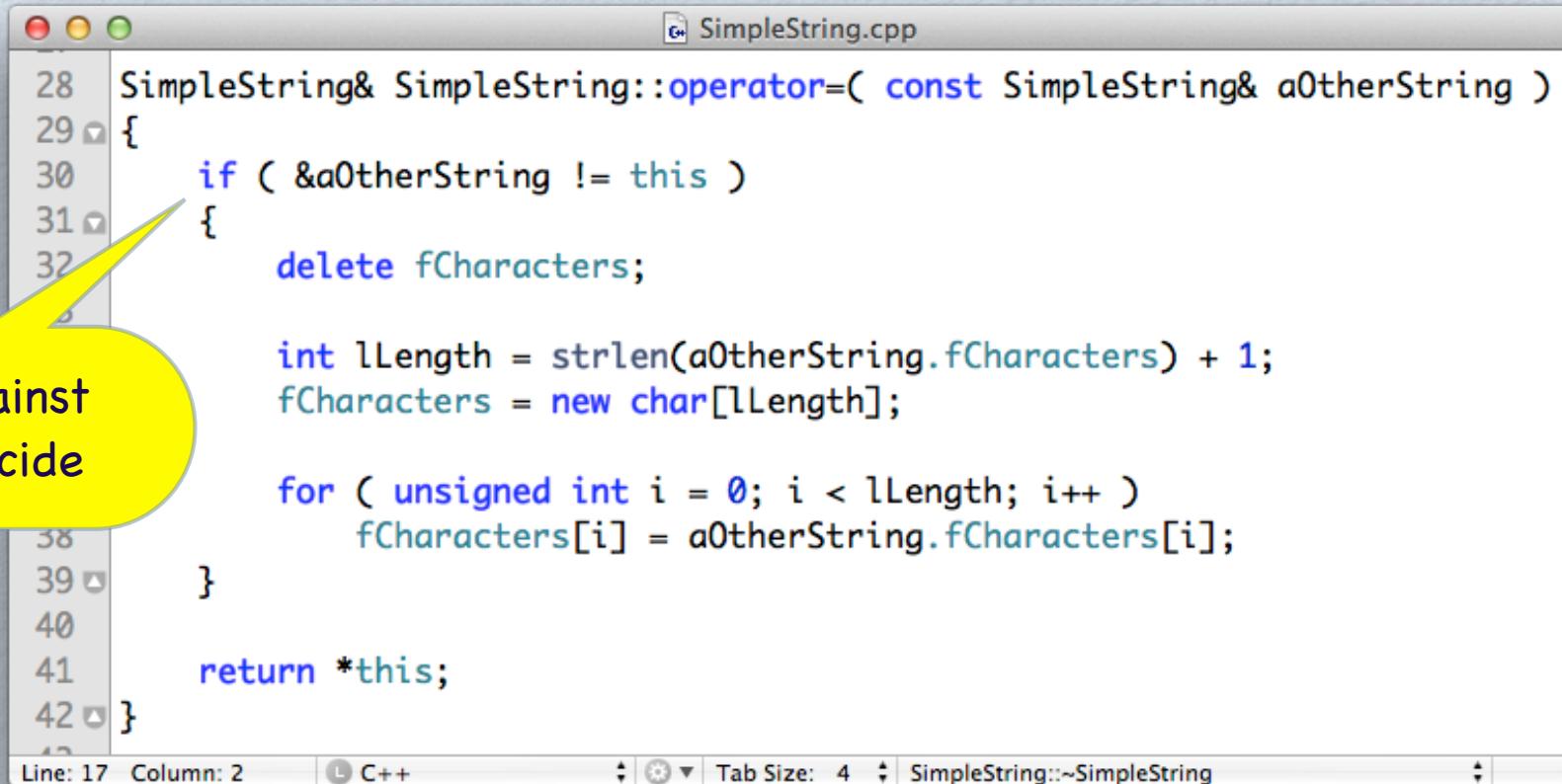
The screenshot shows a code editor window titled "SimpleString.h". The code defines a class `SimpleString` with private member `fCharacters` and public constructors, copy constructor, assignment operator, and conversion operator. The editor interface includes standard Mac OS X window controls (red, yellow, green buttons) and a vertical scroll bar.

```
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}
```

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

# The Explicit Assignment Operator

protection against  
accidental suicide



A screenshot of a C++ IDE showing the implementation of the assignment operator for a class named SimpleString. The code is as follows:

```
SimpleString& SimpleString::operator=( const SimpleString& aOtherString )
{
    if ( &aOtherString != this )
    {
        delete fCharacters;

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

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

    return *this;
}
```

The code includes a check to prevent self-assignment. A yellow callout bubble points to this check with the text "protection against accidental suicide". The IDE interface shows the file name "SimpleString.cpp" in the title bar, and the status bar at the bottom indicates "Line: 17 Column: 2" and "C++".

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

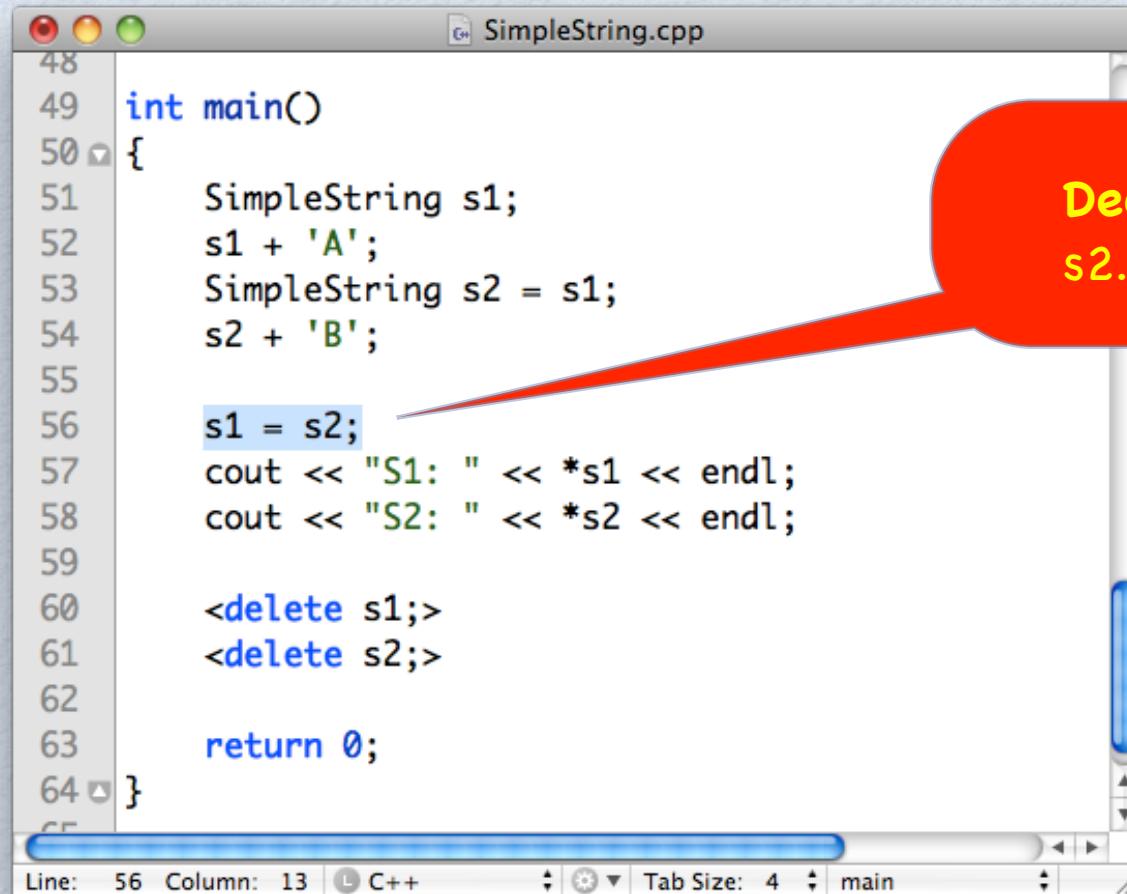
# Explicit Assignment Operator in Use

The image shows a Mac OS X desktop environment. In the foreground, a code editor window titled "SimpleString.cpp" displays C++ code. The code defines a class "SimpleString" and demonstrates its use in a "main" function. A specific line of code, "s1 = s2;", is highlighted in blue. In the background, a terminal window titled "COS30008" shows the output of the program: "S1: AB" and "S2: AB".

```
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 }
```

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;           ← Red arrow points here
57     cout << "S1: " << *s1 << endl;
58     cout << "S2: " << *s2 << endl;
59
60     s1.~SimpleString();
61     s2.~SimpleString();
62
63     return 0;
64 }
```

Deep copy:  
 $s2.fCharacters \neq s1.fCharacters$

# Cloning: Alias Control for References

# Copying Pointers

The image shows a Mac OS X desktop environment. In the foreground, a terminal window titled 'COS30008' is open, displaying the output of a C++ program named 'SimpleString'. The terminal shows the strings 'S1: AB' and 'S2: AB' being printed, followed by a memory error message and an abort trap. In the background, a code editor window titled 'SimpleString.cpp' is visible, showing the source code for the program. The code creates two pointers, ps1 and ps2, both pointing to the same dynamically allocated string. It then attempts to delete both pointers, which leads to a memory error.

```
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++

```
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;

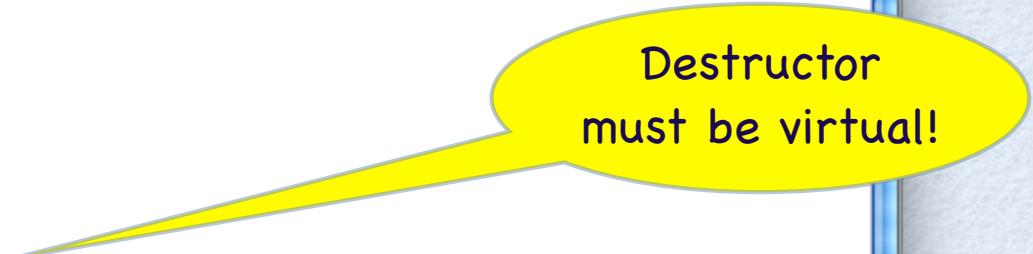
69
70     delete ps1;
71     delete ps2;

72
73     return 0;
74 }
```

Shallow copy:  
ps2. == ps1

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

# Solution: A clone() Method



```
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!

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

# The Use of clone()

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

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

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

```
int main()
{
    SimpleString* ps1 = new SimpleString();
    (*ps1) + 'A';
    SimpleString* ps2 = ps1->clone();
    (*ps2) + 'B';

    cout << "S1: " << **ps1 << endl;
    cout << "S2: " << **ps2 << endl;

    delete ps1;
    delete ps2;

    return 0;
}
```

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

# 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**.

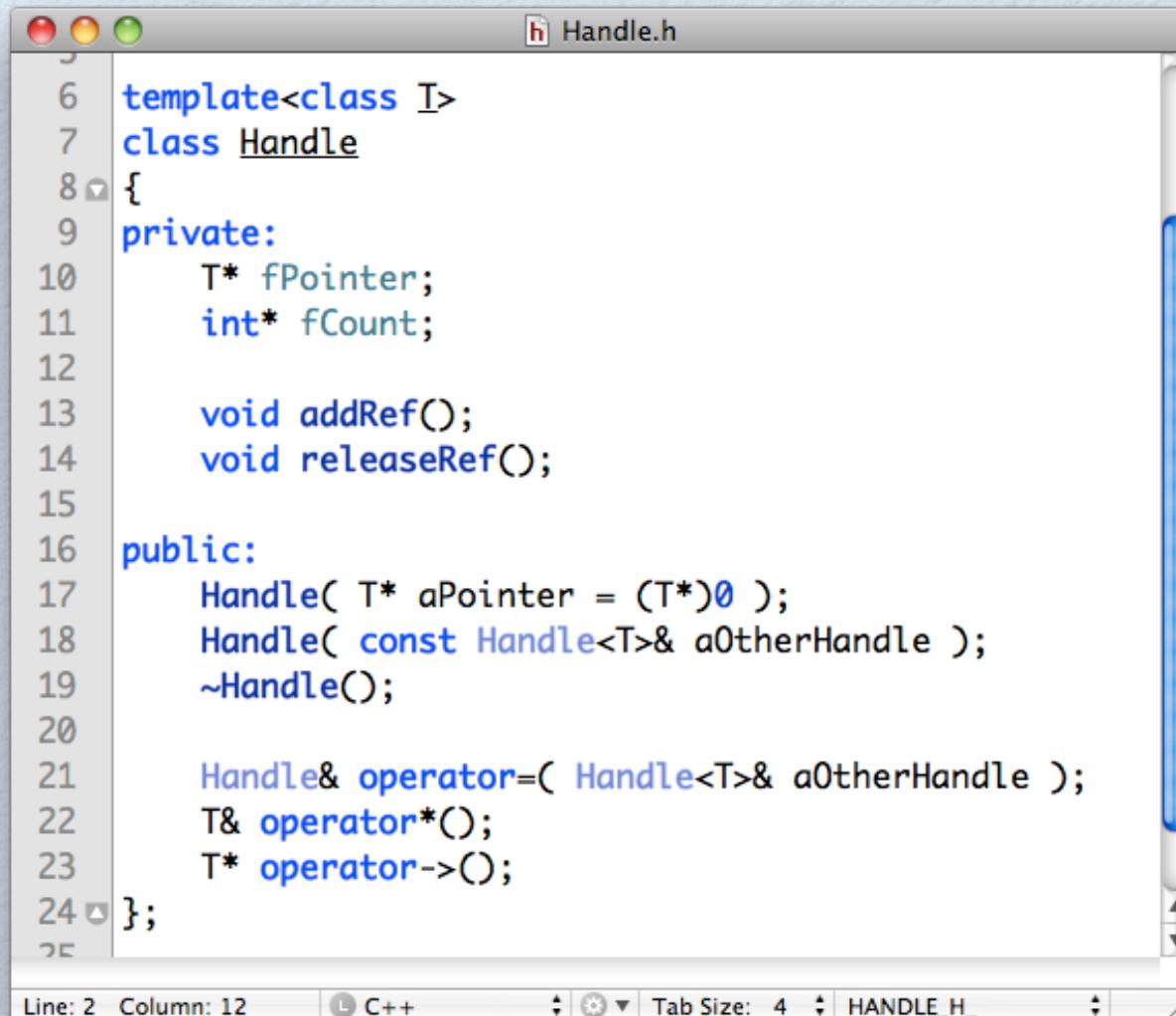
```
SimpleString* pToString = new SubtypeOfSimpleString();
SimpleString* c1 = pToString->clone(); // SimpleString::clone()
```

# 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



A screenshot of a Mac OS X-style code editor window titled "Handle.h". The code defines a template class "Handle" for type T. It contains private members fPointer and fCount, and public constructors, copy constructor, and destructor. It also includes operator=, operator\*, and operator->.

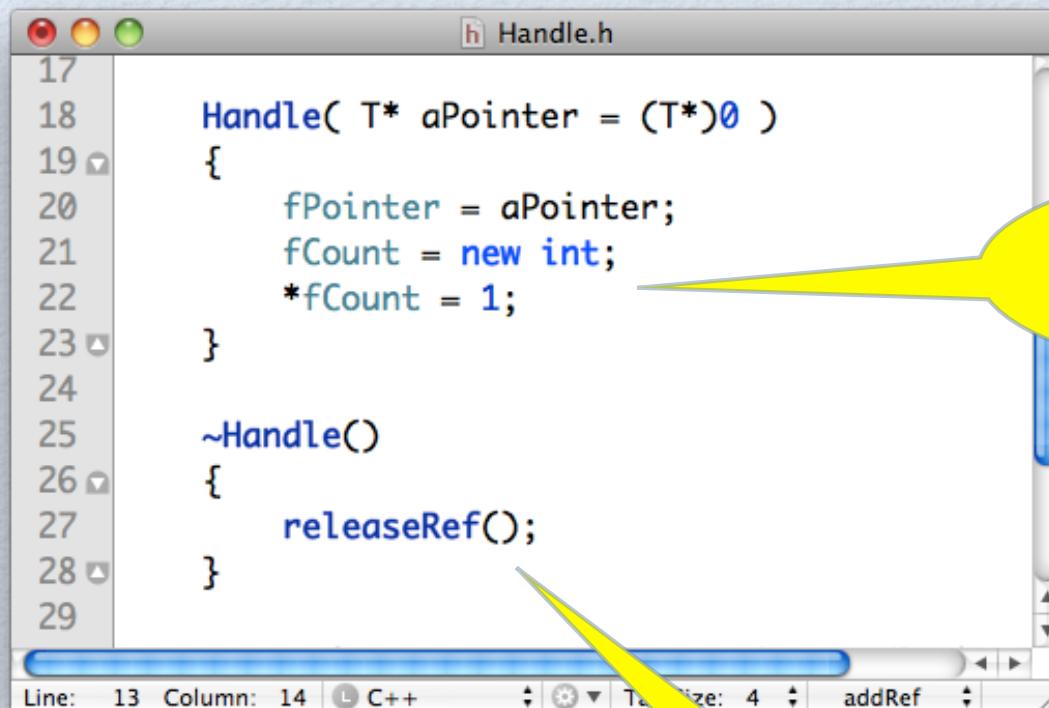
```
6  template<class T>
7  class Handle
8  {
9  private:
10     T* fPointer;
11     int* fCount;
12
13     void addRef();
14     void releaseRef();
15
16 public:
17     Handle( T* aPointer = (T*)0 );
18     Handle( const Handle<T>& aOtherHandle );
19     ~Handle();
20
21     Handle& operator=( Handle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

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



A screenshot of a code editor window titled "Handle.h". The code defines a class Handle with a constructor and a 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
```

The constructor initializes `fPointer` to `aPointer`, creates a shared counter `fCount` using `new int`, and sets its initial value to 1. The destructor calls `releaseRef()`.

Create a shared counter

Decrement reference count

# Handle: addRef & releaseRef

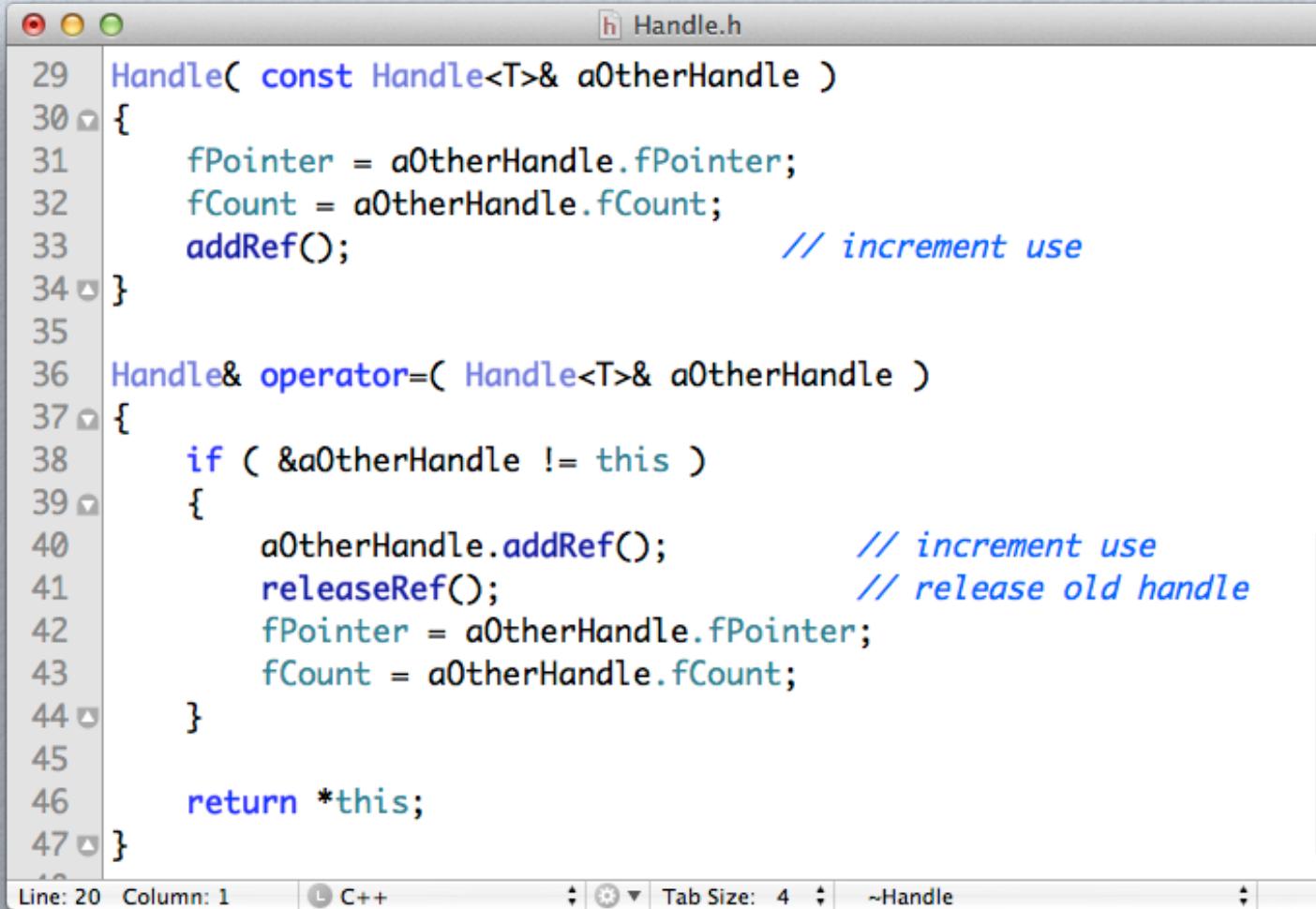
A screenshot of a C++ IDE showing the file `Handle.h`. The code defines two methods: `addRef()` and `releaseRef()`. The `addRef()` method increments the reference count. The `releaseRef()` method decrements the reference count and deletes the object if it is no longer referenced.

```
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

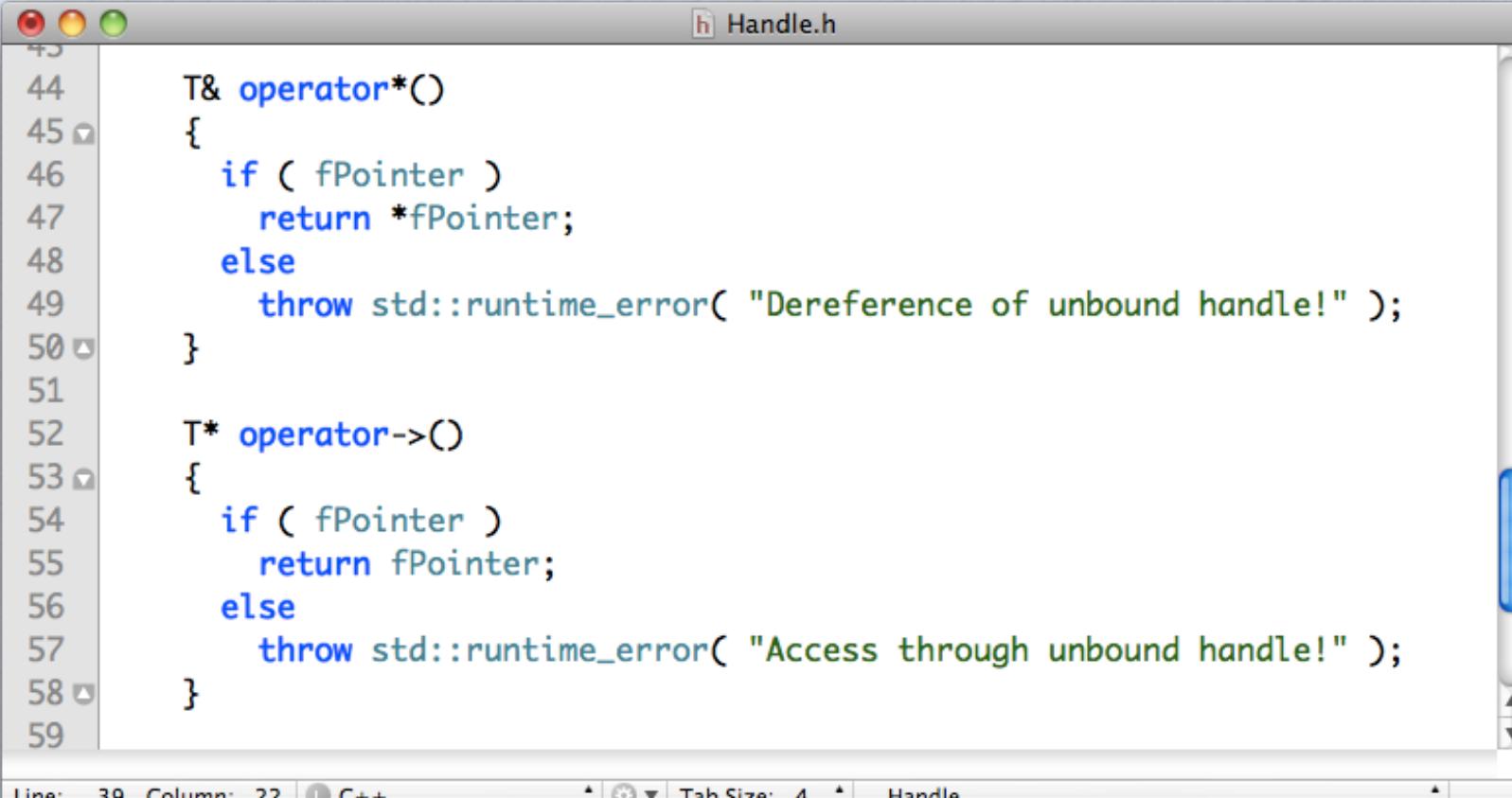


The screenshot shows a code editor window titled "Handle.h". The code implements copy control for a handle class. It includes a copy constructor and an assignment operator that both increment the reference count of the target handle and release the old handle's reference count.

```
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



A screenshot of a code editor window titled "Handle.h". The code implements pointer behavior for a handle class. It contains two operator overloads: one for dereferencing (\*), which returns a reference to the pointed-to object if it exists, or throws a runtime\_error if it doesn't; and another for indirect access (->), which returns a pointer to the object if it exists, or throws a runtime\_error if it doesn't. The code uses an fPointer member variable to store the pointer value.

```
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

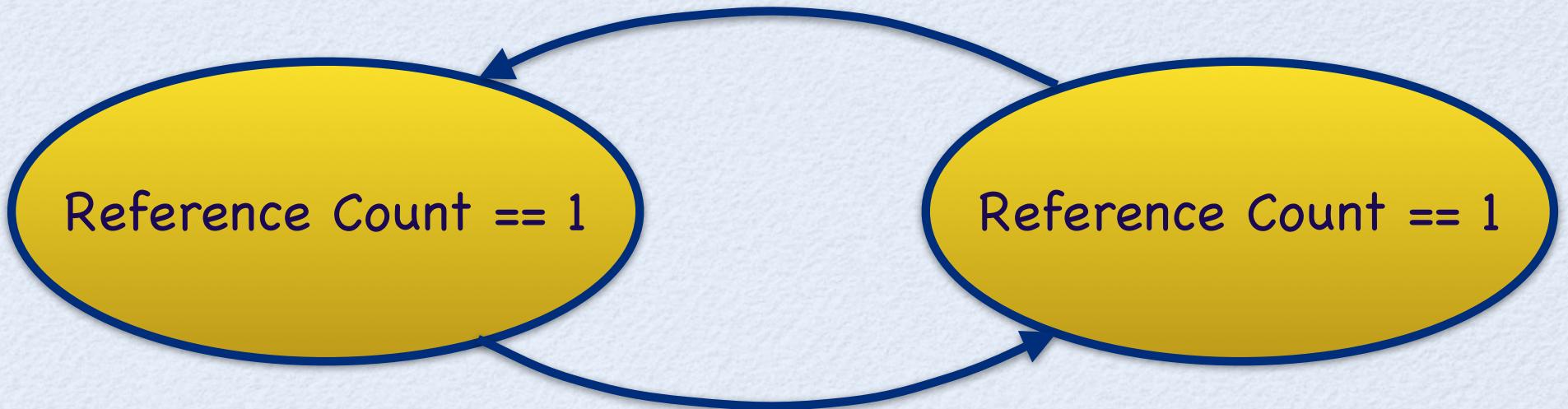
# Using Handle

```
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\$

Line: 60 Column: 25 C++ Tab Size: 4 SimpleString::clone

# 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**

# Container Types, Stacks, and Queues

## Overview

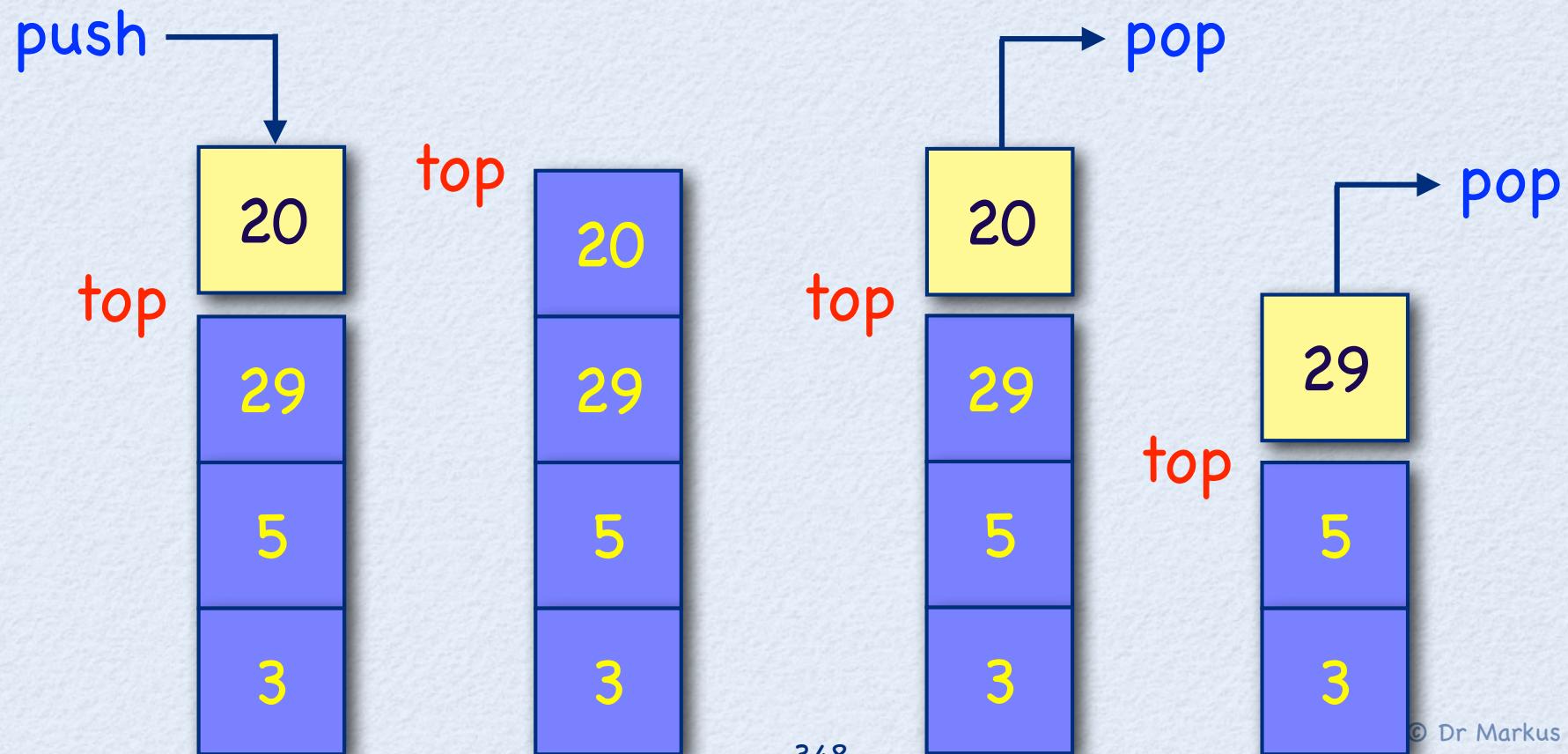
- Stacks
- Container types and references

## References

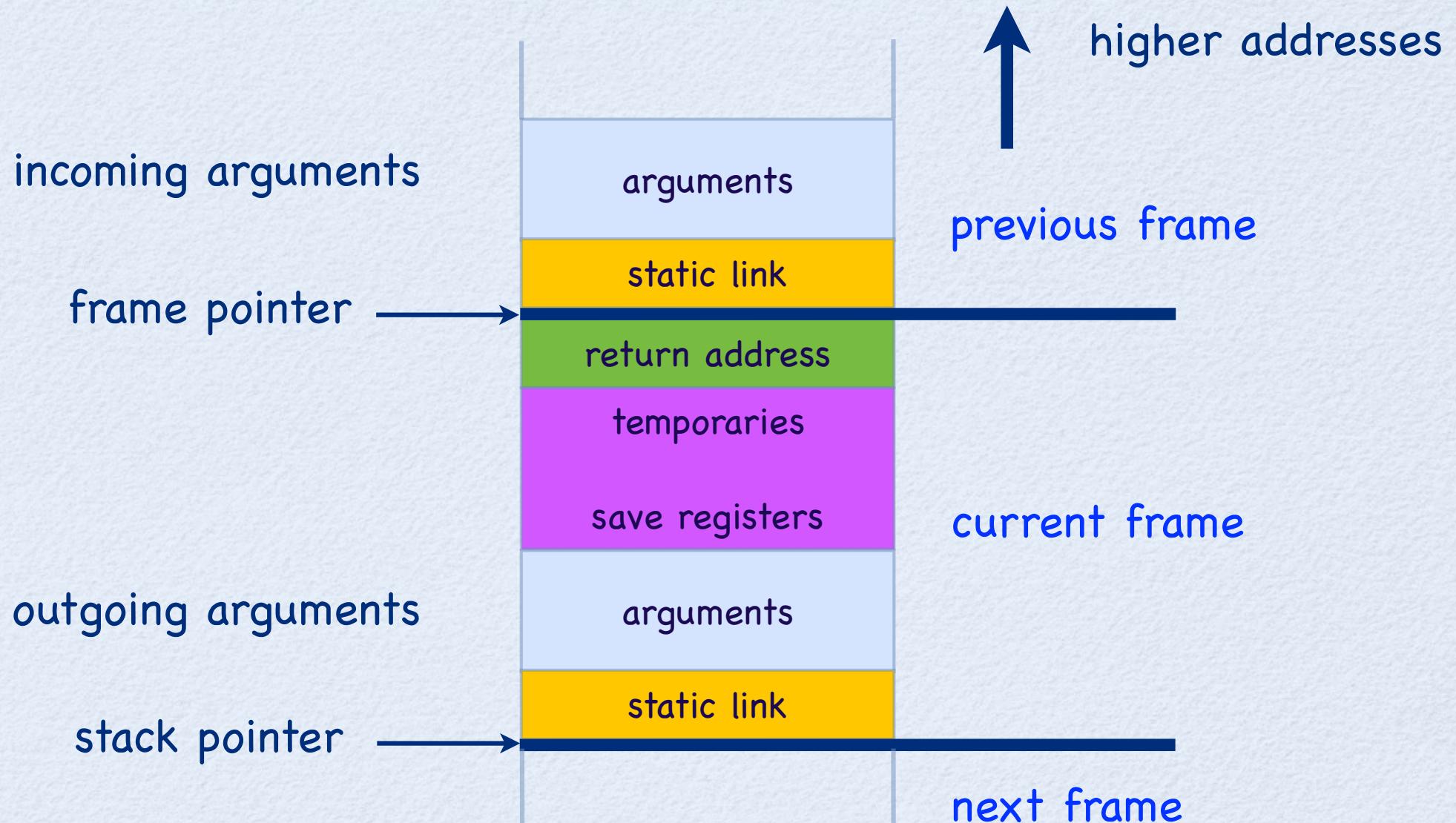
- Bruno R. Preiss: Data Structures and Algorithms with Object-Oriented Design Patterns in C++. John Wiley & Sons, Inc. (1999)
- Richard F. Gilberg and Behrouz A. Forouzan: Data Structures - A Pseudocode Approach with C. 2nd Edition. Thomson (2005)
- Nicolai M. Josuttis: The C++ Standard Library - A Tutorial and Reference. Addison-Wesley (1999)
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)

# Stacks

- A **stack** is a special version of a linear list where items are added and deleted at only one end called the **top**.



# Stack Frames (PASCAL)



# Stack Behavior

- Stacks manage elements in last-in, first-out (LIFO) manner.
- A stack underflow happens when one tries to pop on an empty stack.
- A stack overflow happens when one tries to push onto a full stack.

# Applications of Stacks

- Reversal of input (like `push_front` for `List<T>`)
- Checking for matching parentheses (e.g., stack automata in compiler implementations)
- Backtracking (e.g., Prolog or graph analysis)
- State of program execution (e.g., storage for parameters and local variables of functions)
- Tree traversal

# Reverse Polish Notation

- Reverse Polish Notation (RPN) is a prefix notation wherein operands come before operators.

RPN:    a b \* c +

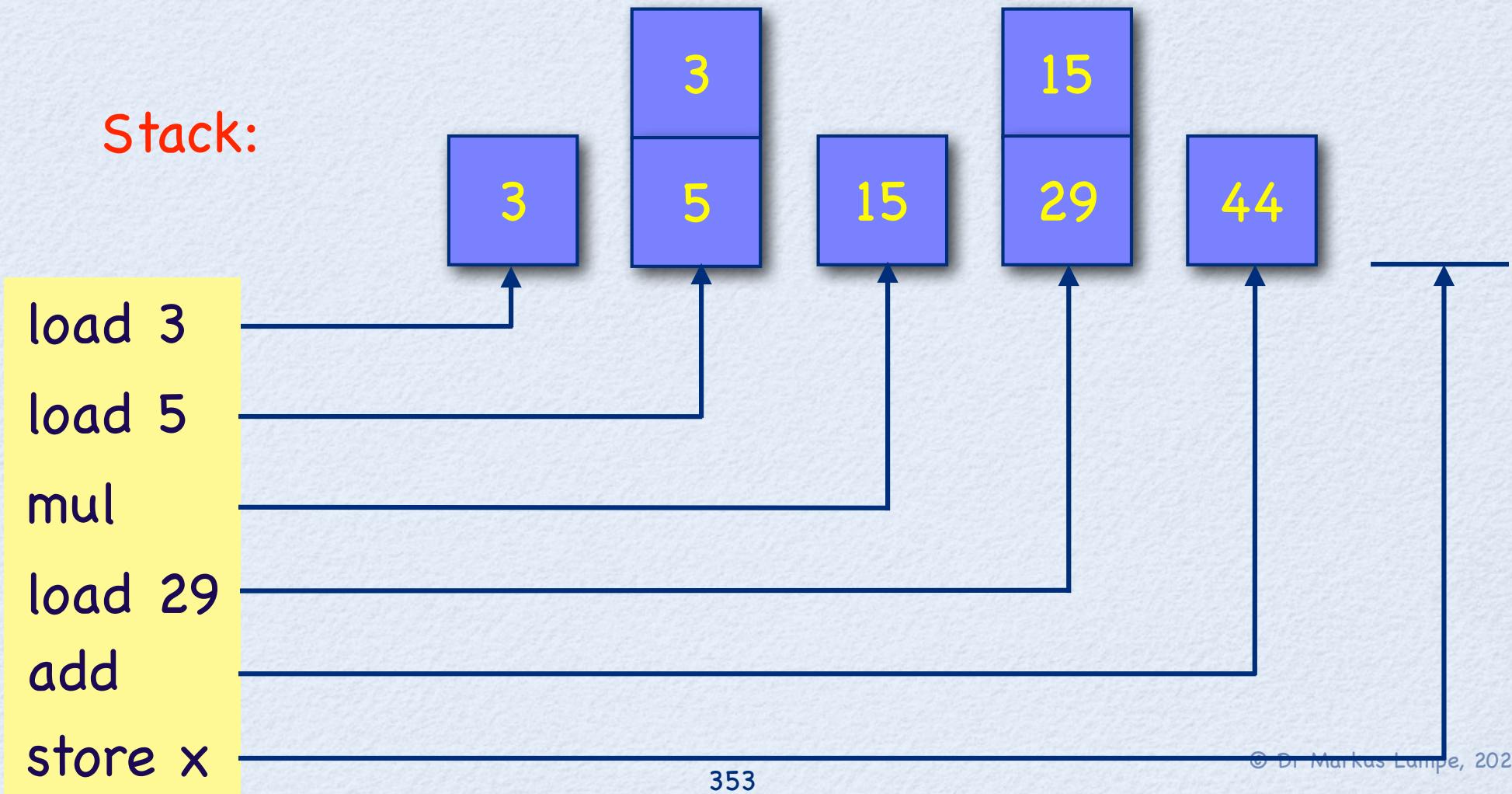


Infix:    a \* b + c

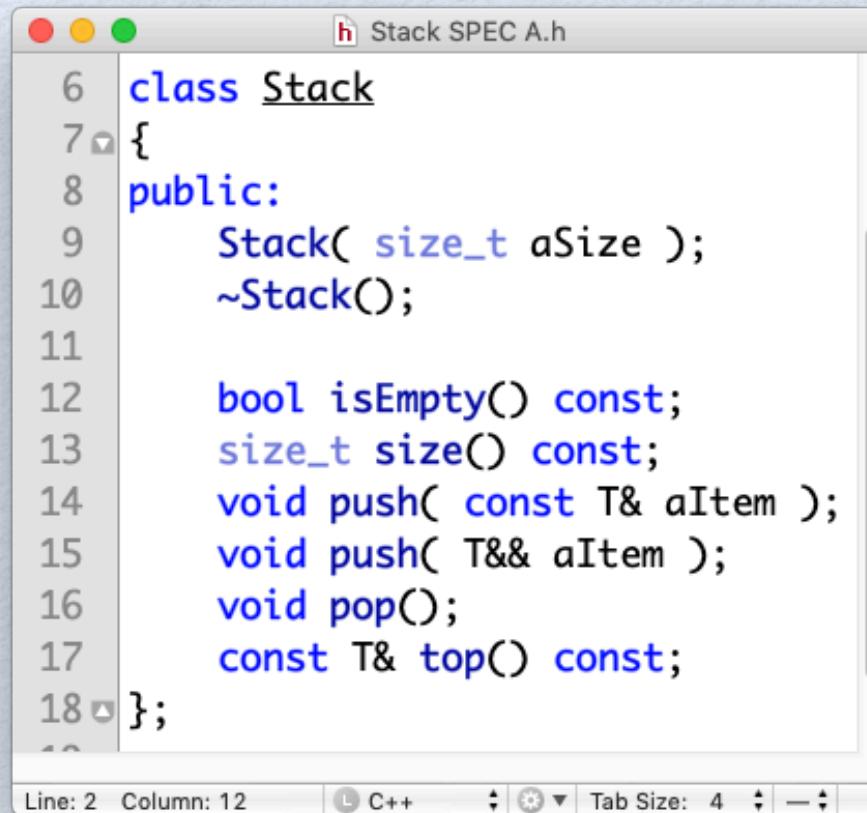
# RPN Calculation

$$\bullet \times = 3 * 5 + 29$$

Stack:



# Stack Interface



A screenshot of a C++ code editor window titled "Stack SPEC A.h". The code defines a class Stack with the following members:

```
6 class Stack
7 {
8 public:
9     Stack( size_t aSize );
10    ~Stack();
11
12    bool isEmpty() const;
13    size_t size() const;
14    void push( const T& aItem );
15    void push( T&& aItem );
16    void pop();
17    const T& top() const;
18};
```

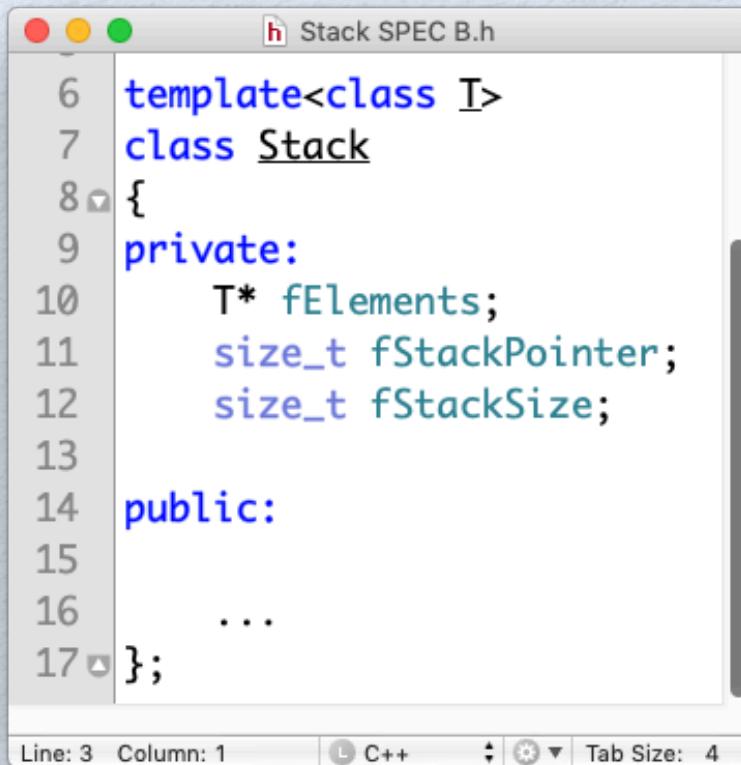
The status bar at the bottom shows "Line: 2 Column: 12" and "C++".

- When defining a container type we wish to minimize the number of value copies required for the objects stored in the container. In order to achieve this, we use references.

# Container Types

- Stacks belong to a special group of data types called container types.
- The de facto standard approach for the definition of container types in C++ is to use value-based semantics.
- Other examples of container types are Lists, Queues, Hash Tables, Maps, Arrays, or Trees.

# The Stack's Private Interface



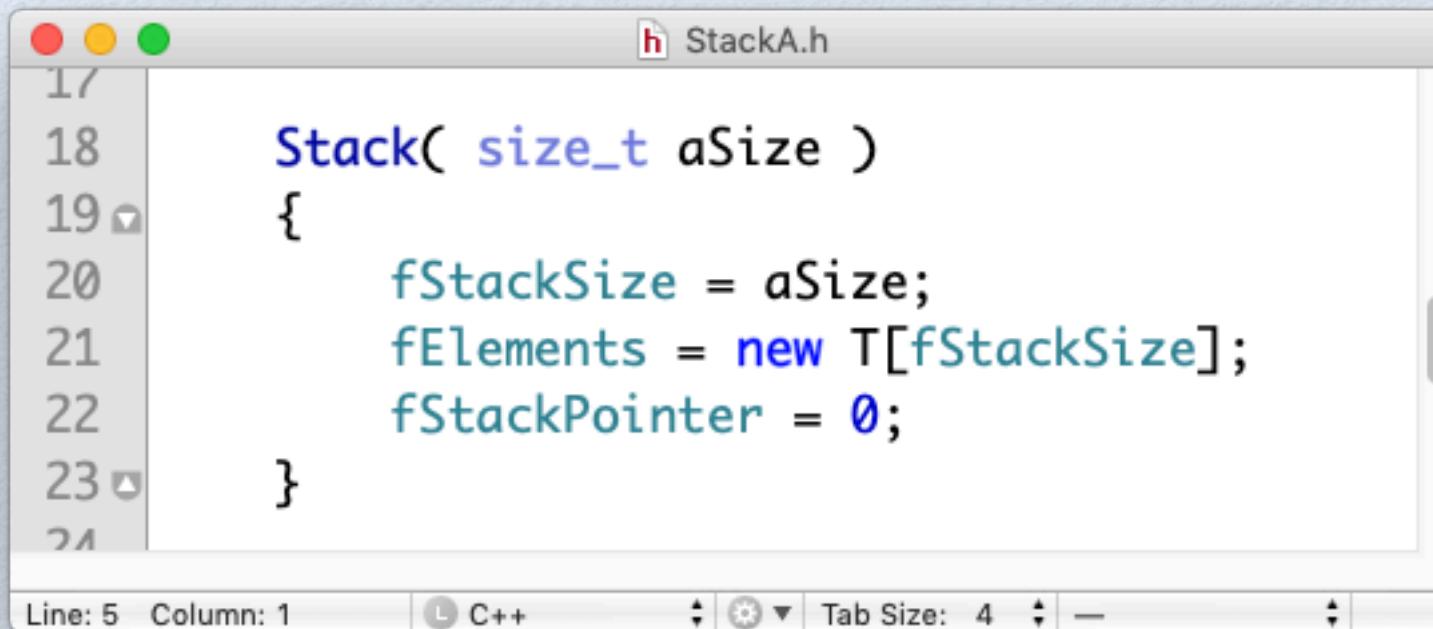
A screenshot of a code editor window titled "Stack SPEC B.h". The code defines a template class `Stack` for type `T`. The class contains private members `fElements`, `fStackPointer`, and `fStackSize`, and a public member `...`. The code is numbered from 6 to 17.

```
6 template<class T>
7 class Stack
8 {
9     private:
10    T* fElements;
11    size_t fStackPointer;
12    size_t fStackSize;
13
14 public:
15
16    ...
17};
```

Line: 3 Column: 1    C++    Tab Size: 4

- Inside `Stack` we need to be able to store objects of type `T`. Hence we need to dynamically allocate memory (i.e, an array of type `T`) and store the address of the first element in a matching pointer variable.

# Stack Constructor

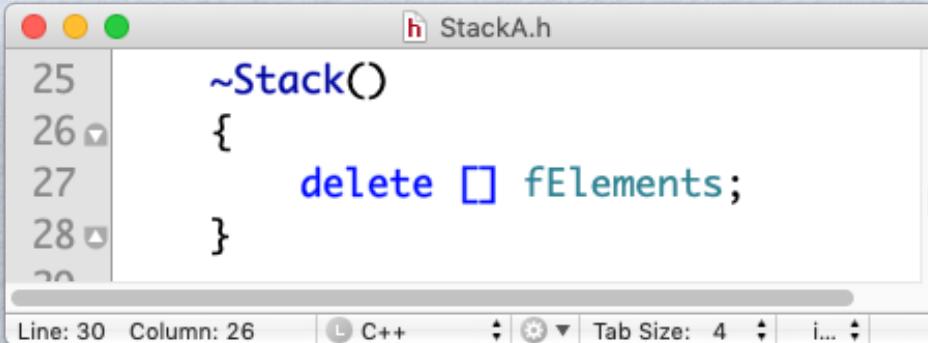


The image shows a screenshot of a code editor window titled "StackA.h". The window has a standard OS X look with red, yellow, and green close buttons at the top left. The file name "StackA.h" is displayed at the top right. The code itself is a constructor definition for a class named "Stack". It initializes three members: "fStackSize" to "aSize", "fElements" to a dynamically allocated array of type "T" of size "fStackSize", and "fStackPointer" to 0. The code is color-coded, with "Stack" in blue, "size\_t" in light blue, "aSize" in black, "fStackSize" in teal, "new" in orange, "T" in light blue, "fElements" in teal, "fStackPointer" in teal, and "0" in black. The editor interface includes a vertical scroll bar on the right, a status bar at the bottom with "Line: 5 Column: 1", and a toolbar below it with icons for C++, settings, and tab size.

```
17
18     Stack( size_t aSize )
19 {
20     fStackSize = aSize;
21     fElements = new T[fStackSize];
22     fStackPointer = 0;
23 }
24
```

Line: 5 Column: 1    C++    Tab Size: 4

# Stack Destructor



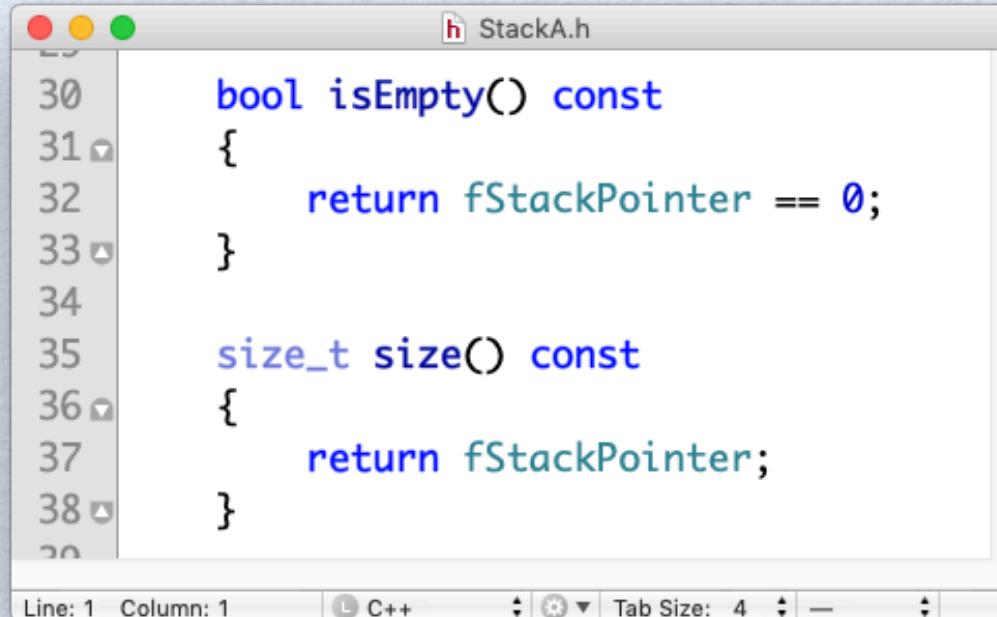
A screenshot of a C++ code editor window titled "StackA.h". The code shown is:

```
25 ~Stack()
26 {
27     delete [] fElements;
28 }
```

The code editor interface includes a status bar at the bottom with "Line: 30 Column: 26", a "C++" tab, and other standard editor controls.

- There are two forms of `delete`:
  - `delete ptr` - release the memory associated with pointer `ptr`.
  - `delete [] ptr` - release the memory associated with all elements of array `ptr` and the array `ptr` itself.
- Whenever you allocate memory for an array of elements of a generic type, say `T* arr = new T[10]`, you must use the array form of `delete`, `delete []`, to guarantee that all array cells are released before the array itself is freed.

# Stack Auxiliaries



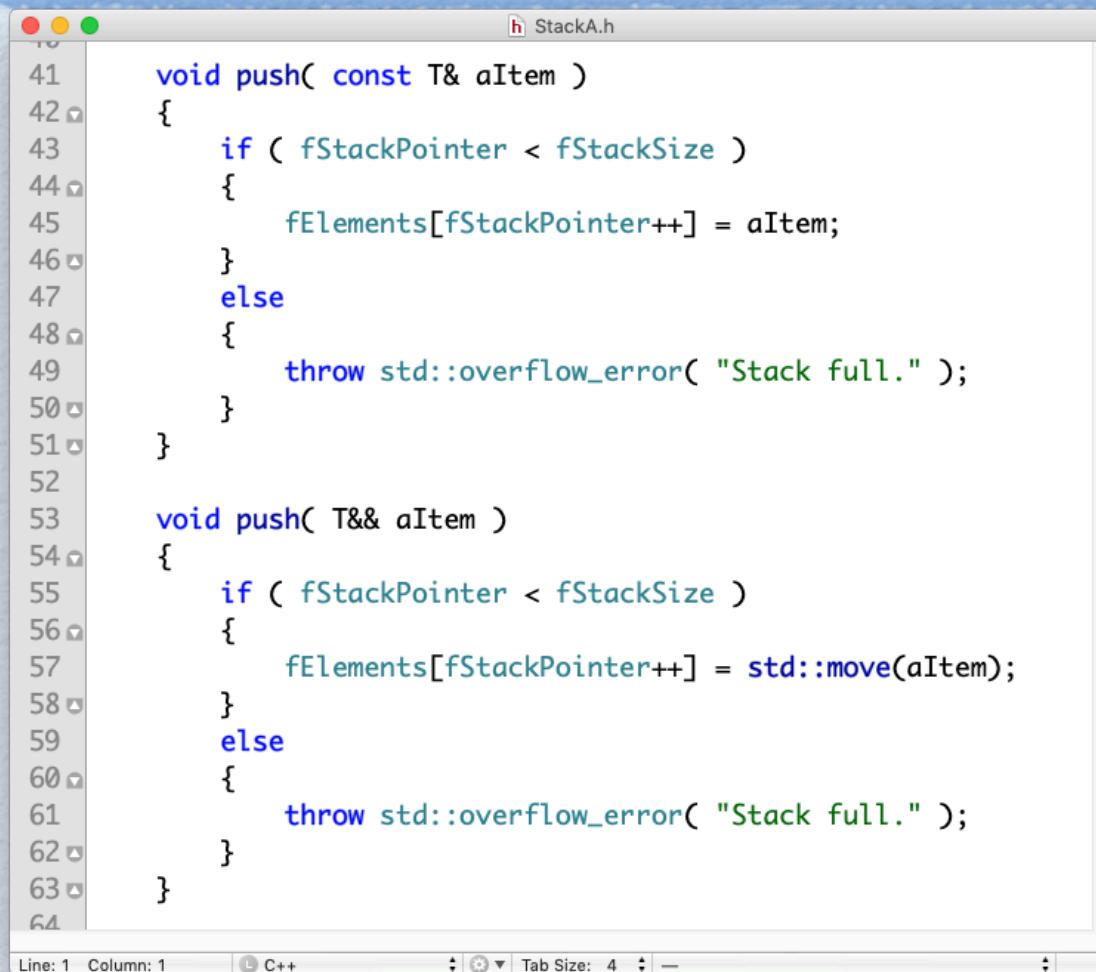
A screenshot of a code editor window titled "StackA.h". The code defines two methods: `isEmpty()` and `size()`. The `isEmpty()` method returns true if the stack pointer is at zero. The `size()` method returns the current stack pointer value.

```
30     bool isEmpty() const
31 {
32     return fStackPointer == 0;
33 }
34
35     size_t size() const
36 {
37     return fStackPointer;
38 }
```

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

- `isEmpty()`: Boolean predicate to indicate whether there are elements on the stack.
- `size()`: returns the actual stack size.

# Push

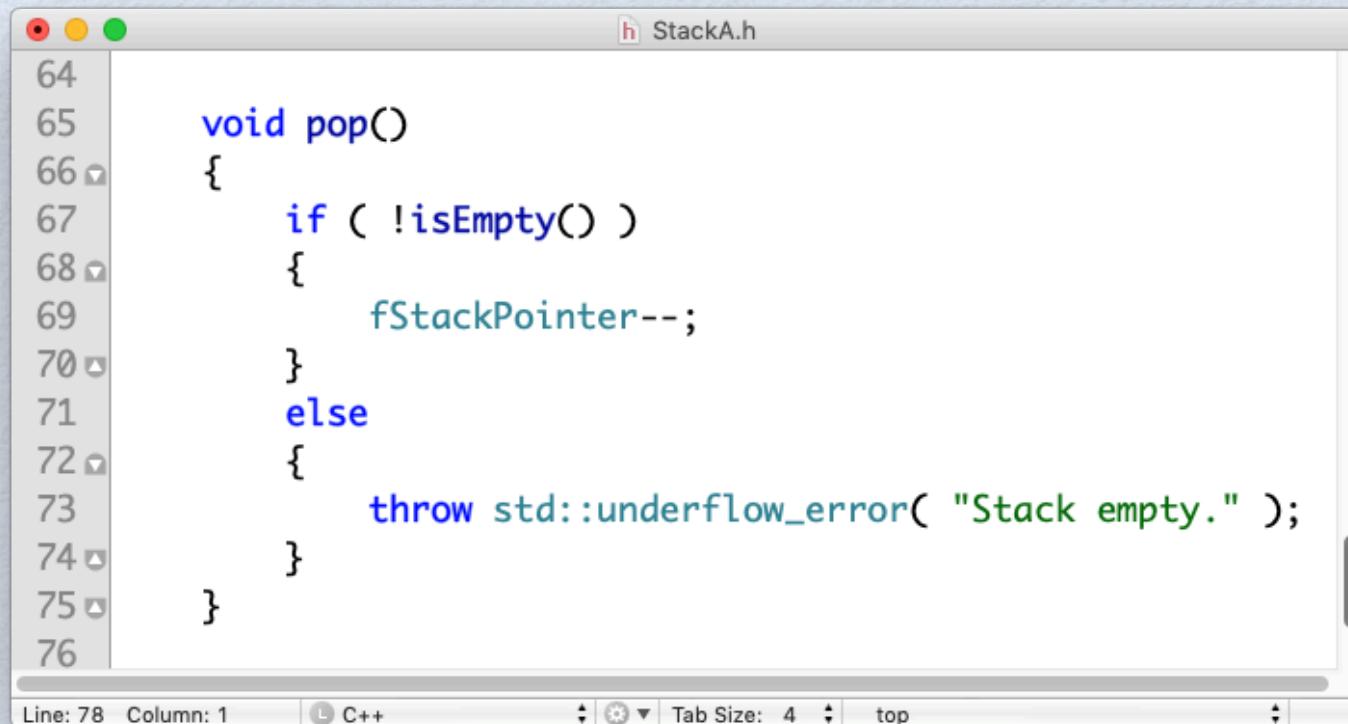


```
StackA.h
41 void push( const T& aItem )
42 {
43     if ( fStackPointer < fStackSize )
44     {
45         fElements[fStackPointer++] = aItem;
46     }
47     else
48     {
49         throw std::overflow_error( "Stack full." );
50     }
51 }
52
53 void push( T&& aItem )
54 {
55     if ( fStackPointer < fStackSize )
56     {
57         fElements[fStackPointer++] = std::move(aItem);
58     }
59     else
60     {
61         throw std::overflow_error( "Stack full." );
62     }
63 }
64
```

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

- The push method stores a item at the next free slot in the stack, if there is room.

# Pop



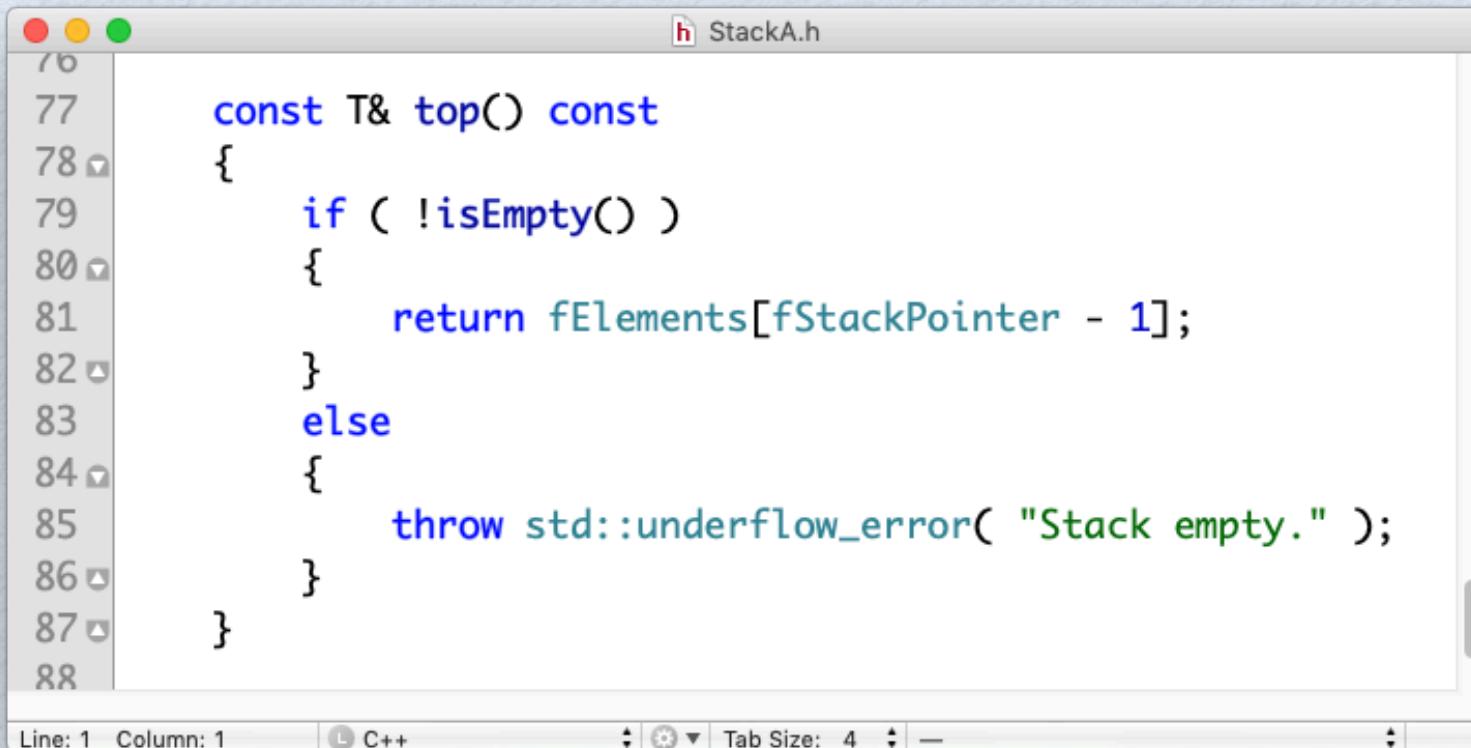
A screenshot of a code editor window titled "StackA.h". The code implements a `pop()` method for a stack. It first checks if the stack is empty using `!isEmpty()`. If not empty, it decrements the `fStackPointer`. If empty, it throws a `std::underflow_error` with the message "Stack empty.".

```
64
65     void pop()
66 {
67     if ( !isEmpty() )
68     {
69         fStackPointer--;
70     }
71     else
72     {
73         throw std::underflow_error( "Stack empty." );
74     }
75 }
76
```

Line: 78 Column: 1 C++ Tab Size: 4 top

- The `pop` method shifts the stack pointer to the previous slot in the stack, if there is such a slot. Note, the element in the current slot itself is not yet destroyed.

# Top



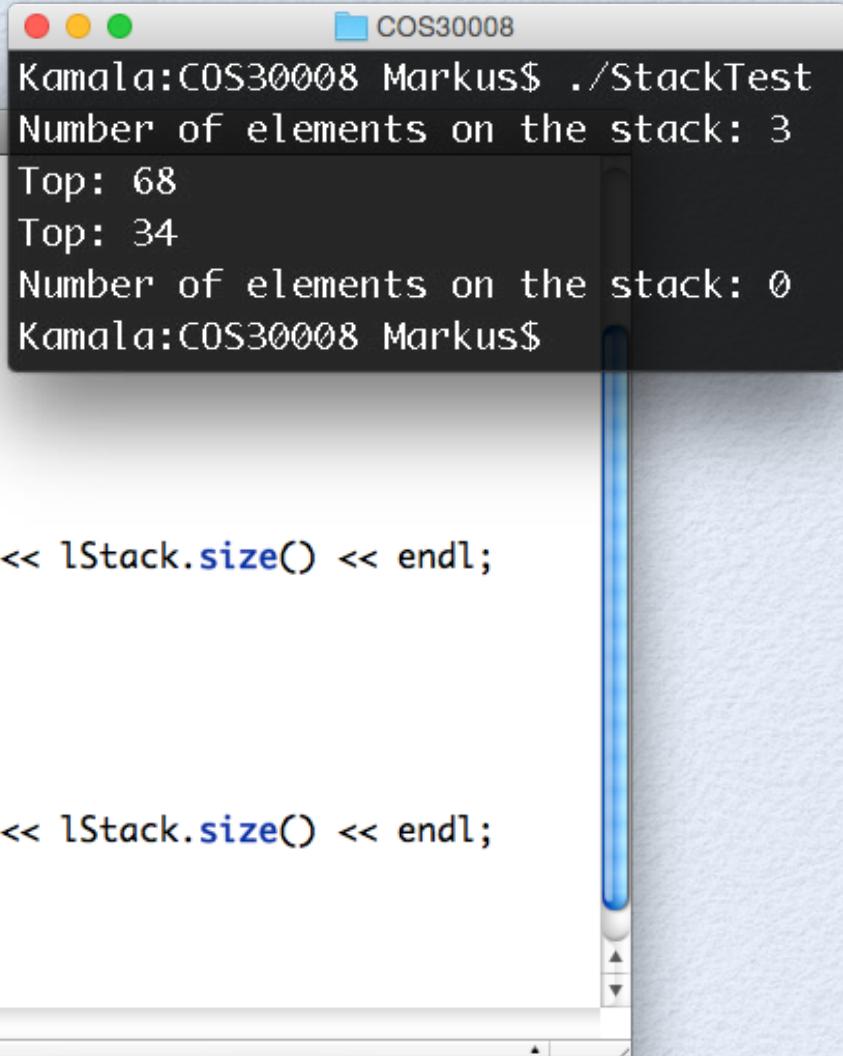
A screenshot of a code editor window titled "StackA.h". The code is written in C++ and defines a const reference member function "top() const". The function checks if the stack is empty using the "isEmpty()" method. If it's not empty, it returns the element at index "fStackPointer - 1". If it is empty, it throws a "std::underflow\_error" exception with the message "Stack empty.".

```
76
77     const T& top() const
78 {
79     if ( !isEmpty() )
80     {
81         return fElements[fStackPointer - 1];
82     }
83     else
84     {
85         throw std::underflow_error( "Stack empty." );
86     }
87 }
88
```

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

- The `top` method returns a `const` reference to the item in the current slot in the stack, if there is such a slot.

# Stack Sample



```
int main()
{
    Stack<int> lStack( 10 );

    lStack.push( 2 );
    lStack.push( 34 );
    lStack.push( 68 );

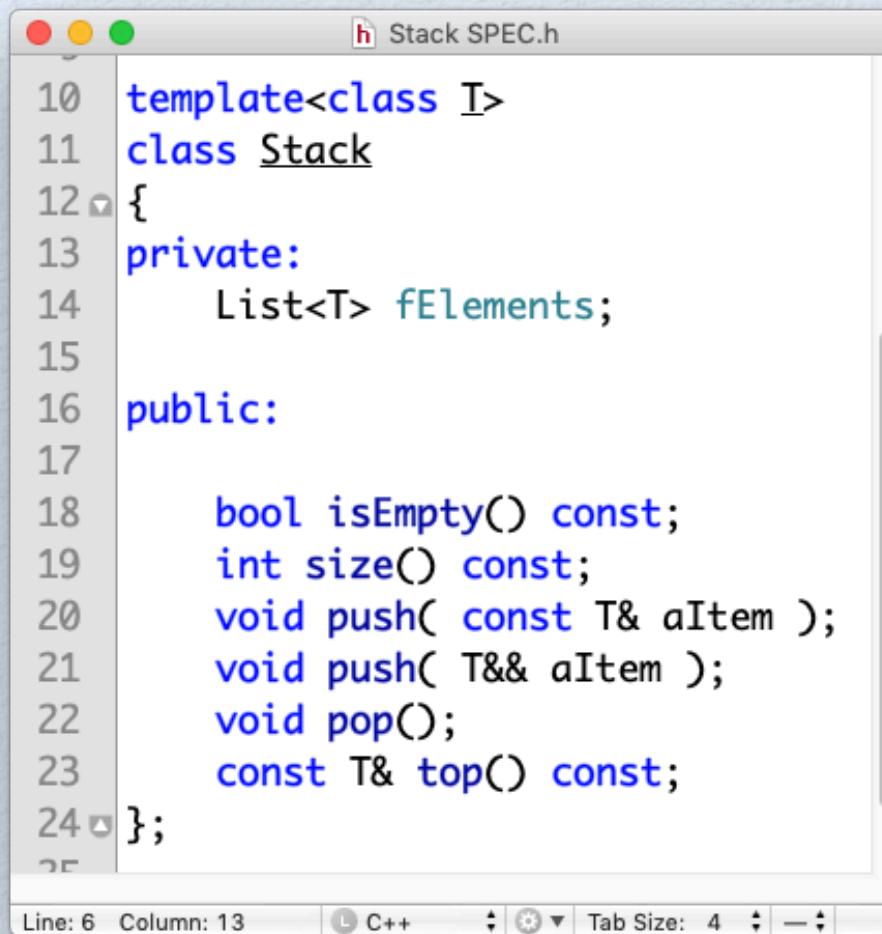
    cout << "Number of elements on the stack: " << lStack.size() << endl;
    cout << "Top: " << lStack.top() << endl;
    lStack.pop();
    cout << "Top: " << lStack.top() << endl;
    lStack.pop();
    lStack.pop();
    cout << "Number of elements on the stack: " << lStack.size() << endl;

    return 0;
}
```

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

# Dynamic Stack

- We can define a dynamic stack that uses a list as underlying data type to host an arbitrary number of elements:



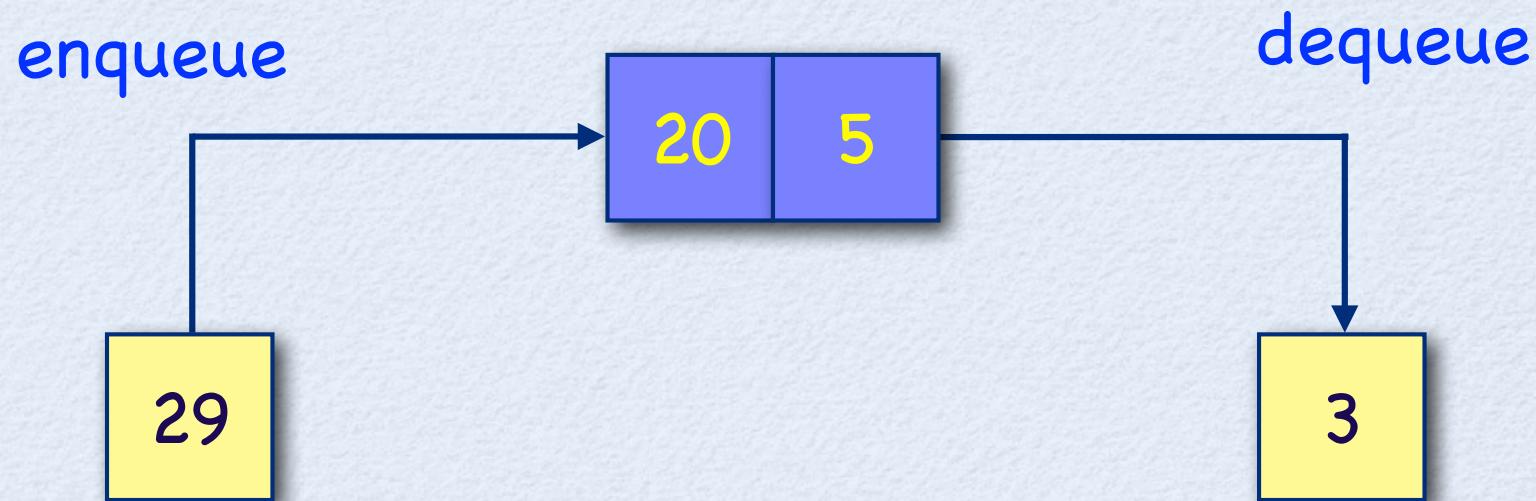
A screenshot of a code editor window titled "Stack SPEC.h". The code defines a template class `Stack` that uses a list to store elements. The class has private members `fElements` and a public interface with methods for checking if it's empty, getting its size, pushing items onto the stack, popping items off the stack, and getting the top item.

```
10 template<class T>
11 class Stack
12 {
13     private:
14         List<T> fElements;
15
16     public:
17
18         bool isEmpty() const;
19         int size() const;
20         void push( const T& aItem );
21         void push( T&& aItem );
22         void pop();
23         const T& top() const;
24 };
25
```

Line: 6 Column: 13 C++ Tab Size: 4 -

# Queues

- A queue is a special version of a linear list where access to items is only possible at its front and end.

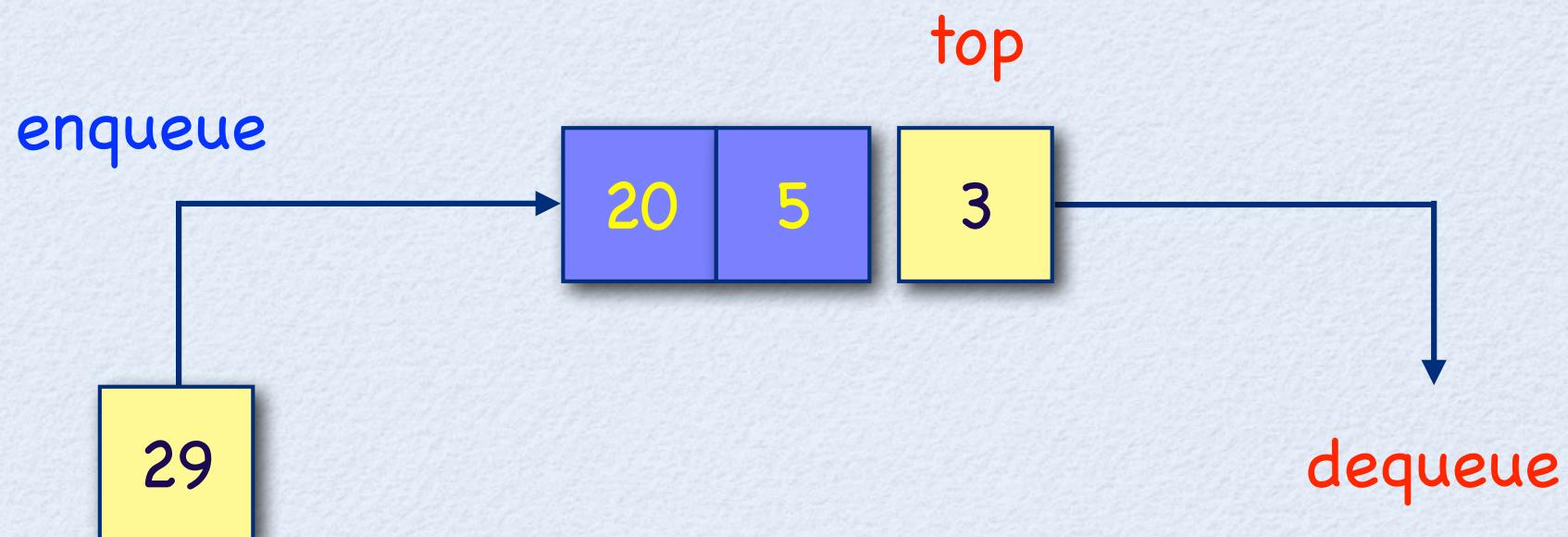


# Queue Behavior

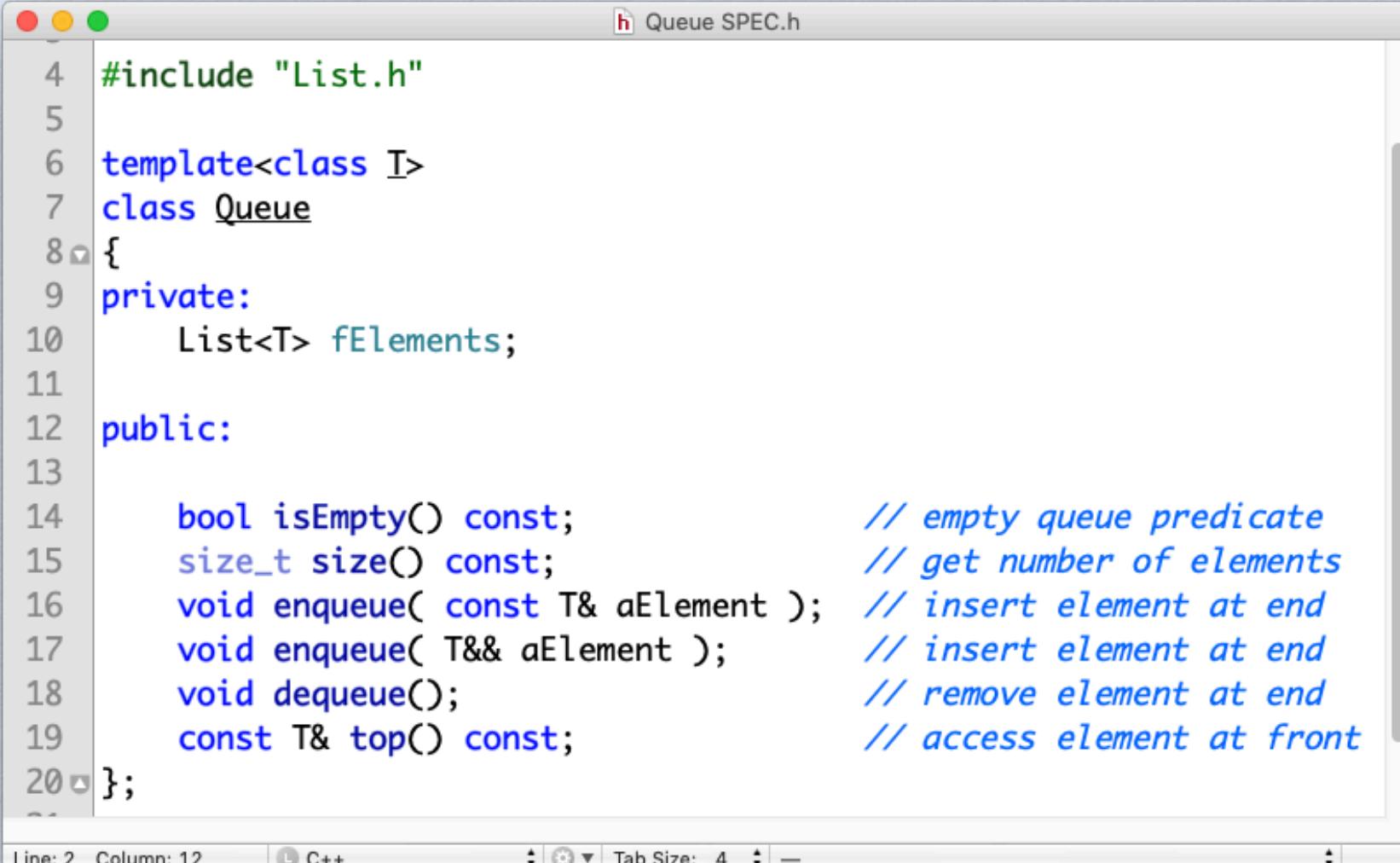
- Queues manage elements in first-in, first-out (FIFO) manner.
- A queue underflow happens when one tries to dequeue on an empty queue.
- A queue overflow happens when one tries to enqueue on a full queue.

# Implementation of Queues

- A concrete queue implementation requires us to split the dequeue operation into two steps: access to the first element (via **top**) and removal of first element (the actual **dequeue**).
- If we were to perform both steps as one (just dequeue), we would create a memory leak in C++ (i.e., we would create a reference to released memory). Hence, we need:



# A Queue Interface

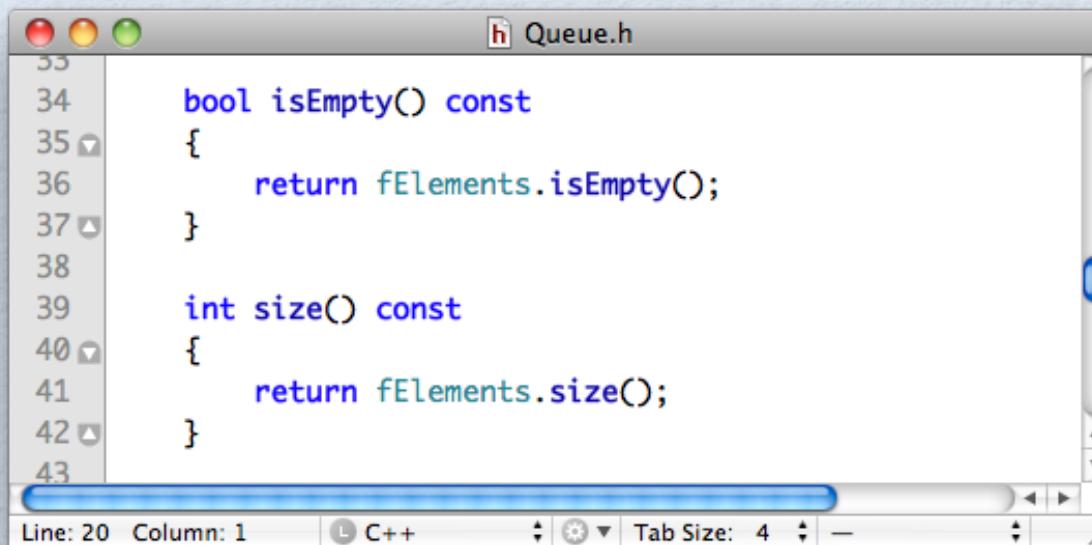


The image shows a screenshot of a code editor window titled "Queue SPEC.h". The code defines a template class `Queue` that inherits from `List.h`. The class has a private member `fElements` of type `List<T>`. The public interface includes methods for checking if the queue is empty, getting its size, enqueueing elements at the end, dequeuing elements from the front, and accessing the element at the front.

```
4 #include "List.h"
5
6 template<class T>
7 class Queue
8 {
9 private:
10     List<T> fElements;
11
12 public:
13
14     bool isEmpty() const;           // empty queue predicate
15     size_t size() const;           // get number of elements
16     void enqueue( const T& aElement ); // insert element at end
17     void enqueue( T&& aElement ); // insert element at end
18     void dequeue();               // remove element at end
19     const T& top() const;          // access element at front
20 }
```

Line: 2 Column: 12    C++    Tab Size: 4

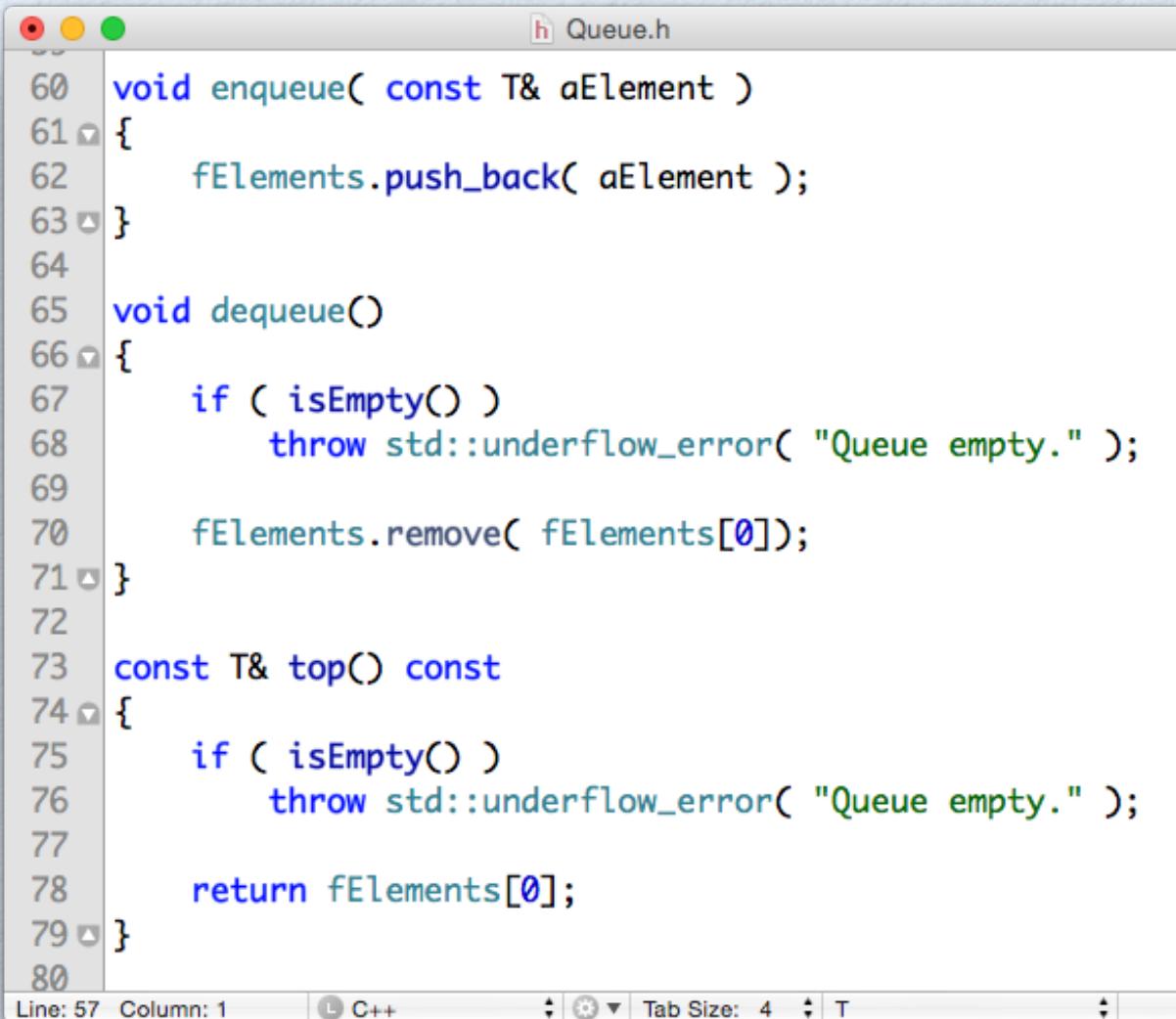
# Queue Service Members



A screenshot of a Mac OS X application window titled "Queue.h". The window contains C++ code for a queue service. The code defines two const member functions: `bool isEmpty() const` and `int size() const`. Both functions return the value of `fElements.isEmpty()` and `fElements.size()` respectively. The code is numbered from 33 to 43. The status bar at the bottom shows "Line: 20 Column: 1", "C++", "Tab Size: 4", and other standard editor controls.

```
33
34     bool isEmpty() const
35     {
36         return fElements.isEmpty();
37     }
38
39     int size() const
40     {
41         return fElements.size();
42     }
43
```

# Queue Semantics



A screenshot of a code editor window titled "Queue.h". The code implements a queue using a vector. It includes methods for enqueueing elements, dequeuing elements, and returning the top element. Error handling is provided for an empty queue.

```
60 void enqueue( const T& aElement )
61 {
62     fElements.push_back( aElement );
63 }
64
65 void dequeue()
66 {
67     if ( isEmpty() )
68         throw std::underflow_error( "Queue empty." );
69
70     fElements.remove( fElements[0] );
71 }
72
73 const T& top() const
74 {
75     if ( isEmpty() )
76         throw std::underflow_error( "Queue empty." );
77
78     return fElements[0];
79 }
80
```

Line: 57 Column: 1 | C++ | Tab Size: 4 | T

# Queue Test

The image shows a Mac OS X desktop environment. On the left is a code editor window titled "QueueTest.cpp" containing C++ code for testing a queue. On the right is a terminal window titled "COS30008" showing the execution of the program and its output.

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

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

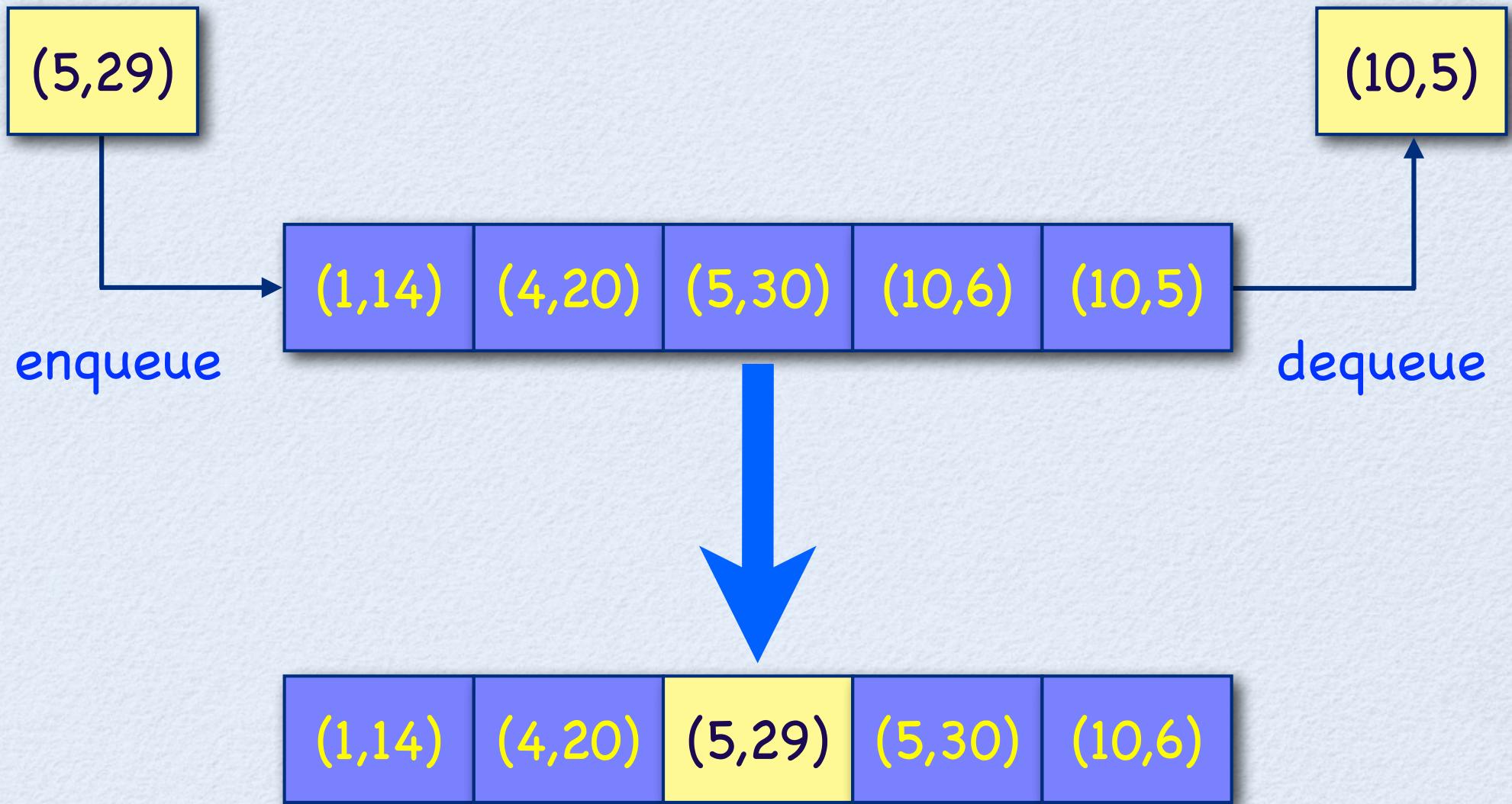
    return 0;
}
```

Kamala:COS30008 Markus\$ ./QueueTest  
Number of elements in the queue: 3  
value: 20  
value: 3  
value: 37  
Number of elements in the queue: 0  
Kamala:COS30008 Markus\$

# Requirements for a Priority Queue

- The underlying data structure for a priority queue must be sorted (e.g., `SortedList<T>`).
- Elements are queued using an integer to specify priority. We 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



```
6 #include "DoublyLinkedList.h"
7 #include "DoublyLinkedListIterator.h"
8
9 #include <stdexcept>
10
11 template<class T>
12 class SortedList
13 {
14 private:
15     // auxiliary definition to simplify node usage
16     using Node = DoublyLinkedList<T>;
17
18     Node* fRoot;      // the first element in the list
19     int fCount;       // number of elements in the list
20
21 public:
22     // auxiliary definition to simplify iterator usage
23     using Iterator = DoublyLinkedListIterator<T>;
24
25     SortedList();
26     SortedList( const SortedList& aOtherList );
27     SortedList( SortedList&& aOtherList );
28     SortedList& operator=( const SortedList& aOtherList );
29     SortedList& operator=( SortedList&& aOtherList );
30     ~SortedList();
31
32     bool isEmpty() const;
33     int size() const;
34
35     void insert( const T& aElement );
36     void insert( T&& aElement );
37     void remove( const T& aElement );
38
39     const T& operator[]( size_t aIndex ) const;
40
41     Iterator begin() const;
42     Iterator end() const;
43     Iterator rbegin() const;
44     Iterator rend() const;
45 };
46
```

// default constructor - creates empty list  
// copy constructor  
// move constructor  
// assignment operator  
// move assignment operator  
// destructor - frees all nodes

// Is list empty?  
// list size

// adds aElement at proper position  
// adds aElement at proper position  
// remove first match from list

// list indexer

// return a forward iterator  
// return a forward end iterator  
// return a backwards iterator  
// return a backwards end iterator

Line: 3 Column: 1 C++ Tab Size: 4

# The Pair Class

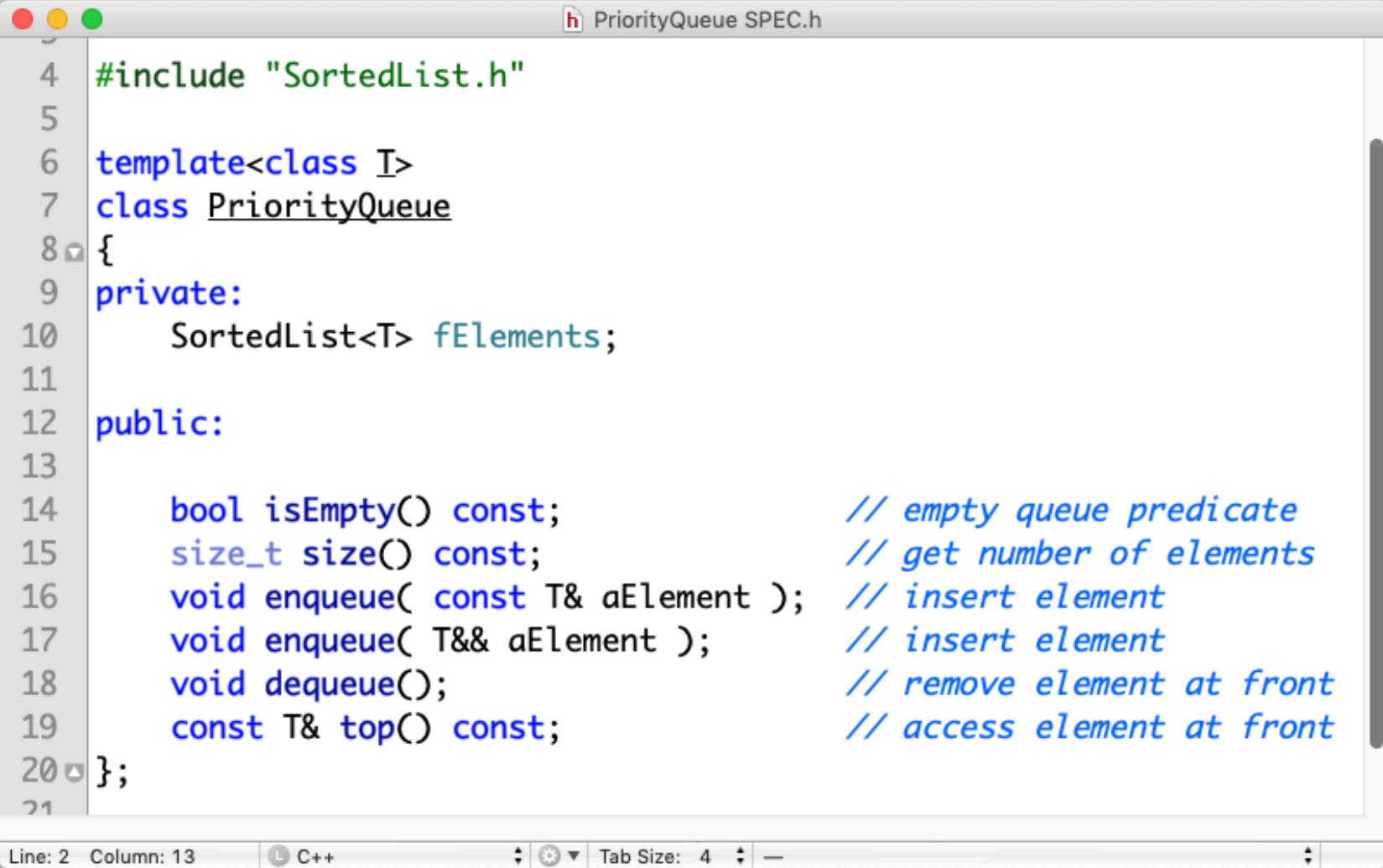
```
4 template<class K, class V>
5 struct Pair
6 {
7     K key;
8     V value;
9
10    Pair( const K& aKey, const V& aValue ) : key(aKey), value(aValue)
11    {}
12
13    bool operator<( const Pair<K,V>& aOther ) const
14    {
15        return key < aOther.key;
16    }
17
18    bool operator==( const Pair<K,V>& aOther ) const
19    {
20        return key == aOther.key && value == aOther.value;
21    }
22};
```

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



SortedList uses an increasing order.

# A Priority Queue



The screenshot shows a Mac OS X application window titled "PriorityQueue SPEC.h". The window contains a code editor with the following C++ code:

```
4 #include "SortedList.h"
5
6 template<class T>
7 class PriorityQueue
8 {
9 private:
10     SortedList<T> fElements;
11
12 public:
13
14     bool isEmpty() const;           // empty queue predicate
15     size_t size() const;           // get number of elements
16     void enqueue( const T& aElement ); // insert element
17     void enqueue( T&& aElement ); // insert element
18     void dequeue();               // remove element at front
19     const T& top() const;          // access element at front
20 };
21
```

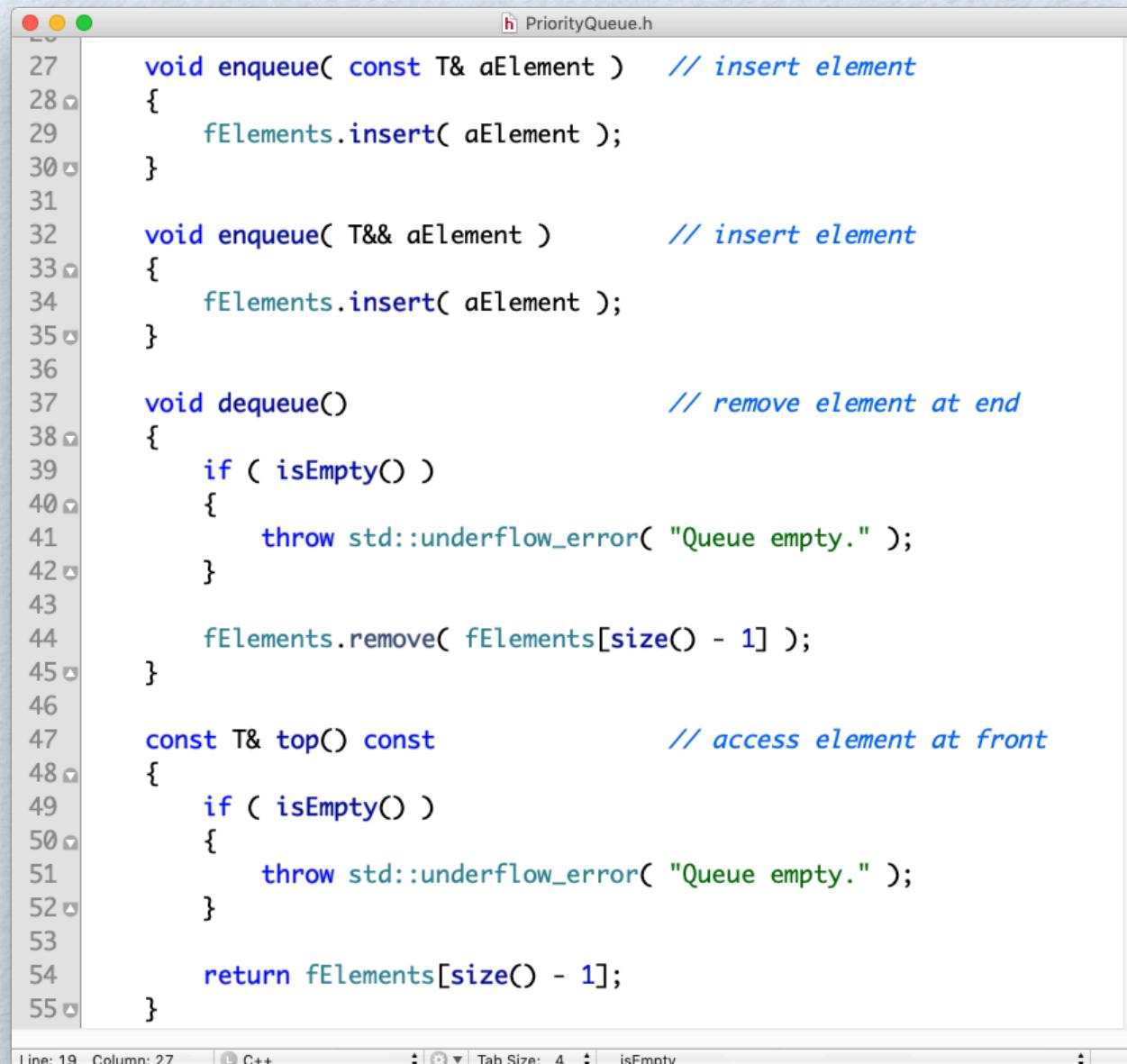
The code defines a template class `PriorityQueue` that uses a `SortedList` to store its elements. It provides methods for checking if the queue is empty, getting its size, enqueuing elements (both by value and by reference), dequeuing elements, and accessing the top element.

Line: 2 Column: 13

C++

Tab Size: 4

# Priority Queue Semantics



A screenshot of a code editor window titled "PriorityQueue.h". The code is written in C++ and defines a priority queue. It includes two enqueue methods (one const, one non-const), a dequeue method that throws an underflow\_error if the queue is empty, and a top() method that also throws an underflow\_error if the queue is empty. The code uses a private member variable fElements to store elements.

```
27     void enqueue( const T& aElement ) // insert element
28     {
29         fElements.insert( aElement );
30     }
31
32     void enqueue( T&& aElement )           // insert element
33     {
34         fElements.insert( aElement );
35     }
36
37     void dequeue()                         // remove element at end
38     {
39         if ( isEmpty() )
40         {
41             throw std::underflow_error( "Queue empty." );
42         }
43
44         fElements.remove( fElements.size() - 1 );
45     }
46
47     const T& top() const                  // access element at front
48     {
49         if ( isEmpty() )
50         {
51             throw std::underflow_error( "Queue empty." );
52         }
53
54         return fElements[ size() - 1 ];
55     }
```

Line: 19 Column: 27    C++    Tab Size: 4    isEmpty

# A PriorityQueue Test

```
PriorityQueueTest.cpp
```

```
7 int main()
8 {
9     PriorityQueue< Pair<int,int> > lQueue;
10
11    Pair<int,int> p1( 4, 20 );
12    Pair<int,int> p2( 5, 30 );
13    Pair<int,int> p3( 5, 29 );
14
15    lQueue.enqueue( p1 );
16    lQueue.enqueue( p2 );
17    lQueue.enqueue( p3 );
18
19    cout << "Number of elements in the queue: " << lQueue.size() << endl;
20
21    cout << "value: " << lQueue.top().value << endl; lQueue.dequeue();
22    cout << "value: " << lQueue.top().value << endl; lQueue.dequeue();
23    cout << "value: " << lQueue.top().value << endl; lQueue.dequeue();
24
25    cout << "Number of elements in the queue: " << lQueue.size() << endl;
26
27    return 0;
28 }
```

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

```
COS30008
```

```
Kamala:COS30008 Markus$ ./PriorityQueueTest
Number of elements in the queue: 3
value: 30
value: 29
value: 20
Number of elements in the queue: 0
Kamala:COS30008 Markus$
```

# Trees

## Overview

- Trees
- Search Trees

## References

- Bruno R. Preiss: Data Structures and Algorithms with Object-Oriented Design Patterns in C++. John Wiley & Sons, Inc. (1999)
- Richard F. Gilberg and Behrouz A. Forouzan: Data Structures - A Pseudocode Approach with C. 2nd Edition. Thomson (2005)
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein: Introduction to Algorithms. 2nd Edition. The MIT Press (2001)

# Basics

- A tree  $T$  is a finite, non-empty set of nodes,

$$T = \{r\} \cup T_1 \cup T_2 \cup \dots \cup T_n,$$

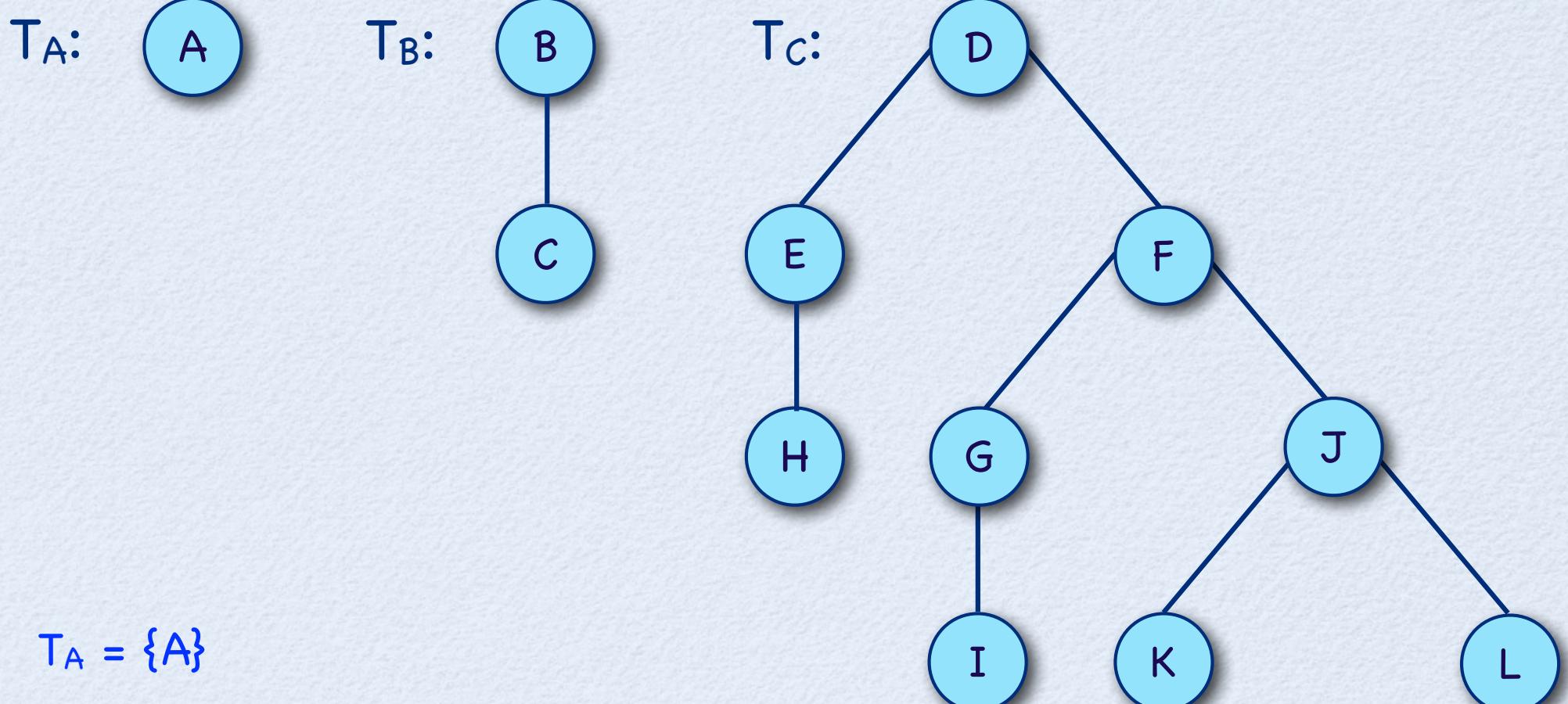
with the following properties:

- A designated node of the set,  $r$ , is called the root of the tree.
- The remaining nodes are partitioned into  $n \geq 0$  subsets  $T_1, T_2, \dots, T_n$ , each of which is a tree.

# Parent, Children, and Leaf

- The root node  $r$  of tree  $T$  is the **parent** of all the roots  $r_i$  of the subtrees  $T_i$ ,  $1 < i \leq n$ .
- Each root  $r_i$  of subtree  $T_i$  of tree  $T$  is called a **child** of  $r$ .
- A **leaf node** is a tree with no subtrees.

# Tree Examples



$$T_A = \{A\}$$

$$T_B = \{B, \{C\}\}$$

$$T_C = \{D, \{E, \{H\}\}, \{F, \{G, \{I\}\}, \{J, \{K\}, \{L\}\}\}\}$$

# Degree

- The **degree** of a node is the **number of subtrees** associated with that node. For example, the degree of  $T_C = \{D, \{E, \{H\}\}, \{F, \{G, \{I\}\}, \{J, \{K\}, \{L\}\}\}\}$  is 2.
- A node of degree zero has no subtrees. Such a node is called a **leaf**. For example, the leaves of  $T_C$  are {H, I, K, L}.
- Two roots  $r_i$  and  $r_j$  of distinct subtrees  $T_i$  and  $T_j$  with the same parent in tree  $T$  are called **siblings**. For example,  $T_i = \{G, \{I\}\}$  and  $T_j = \{J, \{K\}, \{L\}\}$  are **siblings** in  $T_C$ .

# Path and Path Length

- Given a tree  $T$  containing the set of nodes  $R$ , a path in  $T$  is defined as a non-empty sequence of nodes

$$P = \{r_1, r_2, \dots, r_k\}$$

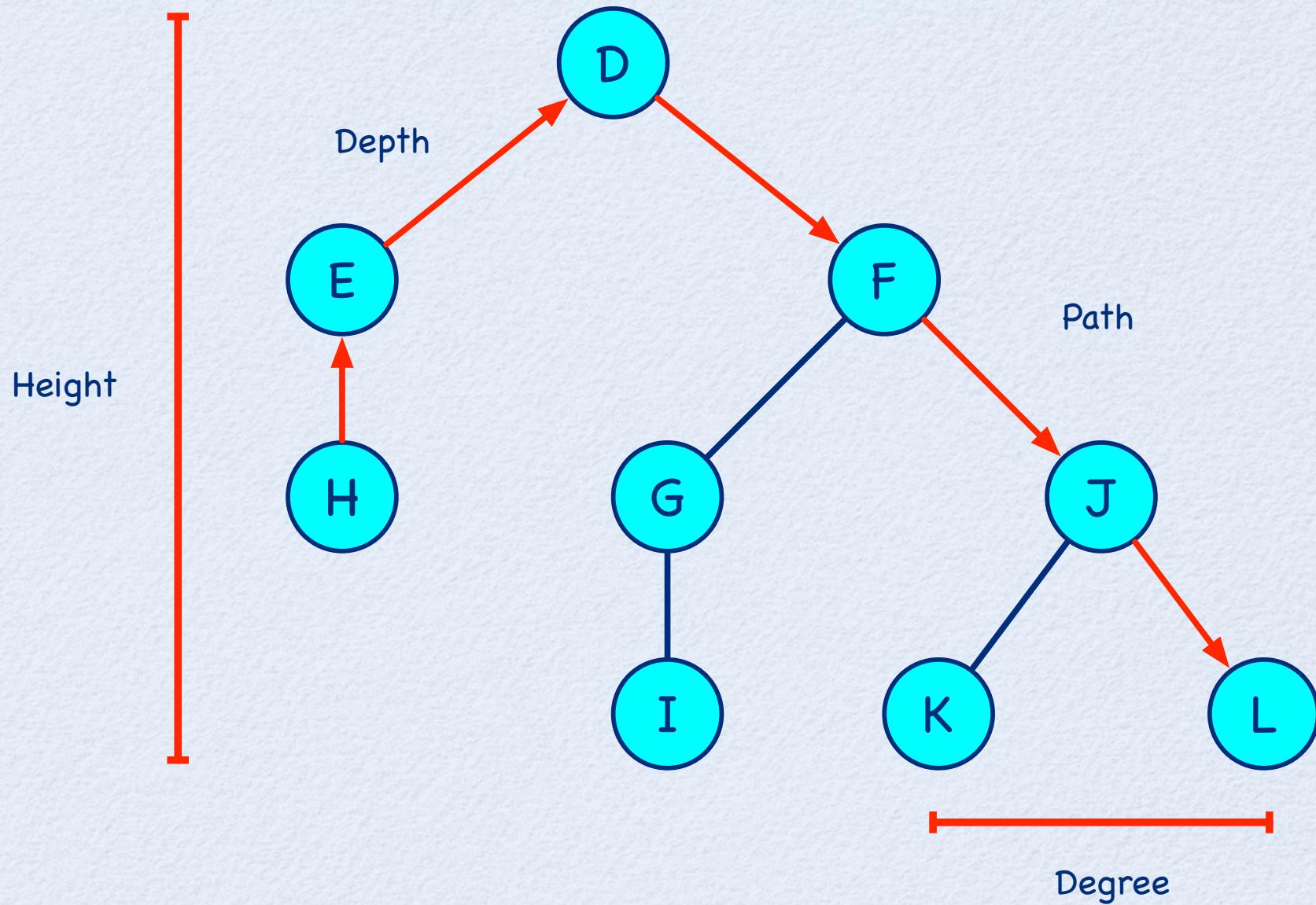
where  $r_i \in R$ , for  $1 \leq i \leq k$  such that the  $i$ th node in the sequence,  $r_i$ , is the parent of the  $(i+1)$ th node in the sequence  $r_{i+1}$ .

- The length of path  $P$  is  $k-1$ , which corresponds to the distance from the root  $r_1$  to the leaf  $r_k$ .

# Depth and Height

- The **depth** of a node  $r_i \in R$  in a tree  $T$  is the length of the unique path in  $T$  from its root to the node  $r_i$ .
- The **height** of a node  $r_i \in R$  in a tree  $T$  is the length of the longest path from node  $r_i$  to a leaf. Therefore, the leaves are all at height zero.
- The **height** of a tree  $T$  is the height of its root node  $r$ .

# Path, Depth, and Height



# Nodes With the Same Degree

- The general case allows each node in a tree to have a different degree. We now consider a variant of trees in which each node has the same degree.
- Unfortunately, it is not possible to construct a tree that has a finite number of nodes which all have the same degree  $N$ , except the trivial case  $N = 0$ .
- We need a special notation, called empty tree, to realize trees in which all nodes have the same degree.

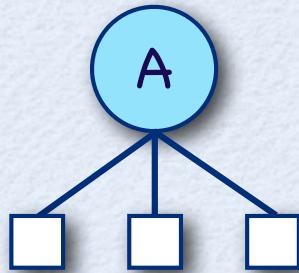
# N-ary Trees

- An N-ary tree  $T$ ,  $N \geq 1$ , is a finite set of nodes with one of the following properties:
  - Either the set is empty,  $T = \emptyset$ , or
  - The set consists of a root,  $R$ , and exactly  $N$  distinct N-ary trees, That is, the remaining nodes are partitioned into  $N \geq 1$  subsets,  $T_1, T_2, \dots, T_N$ , each of which is an N-ary tree such that

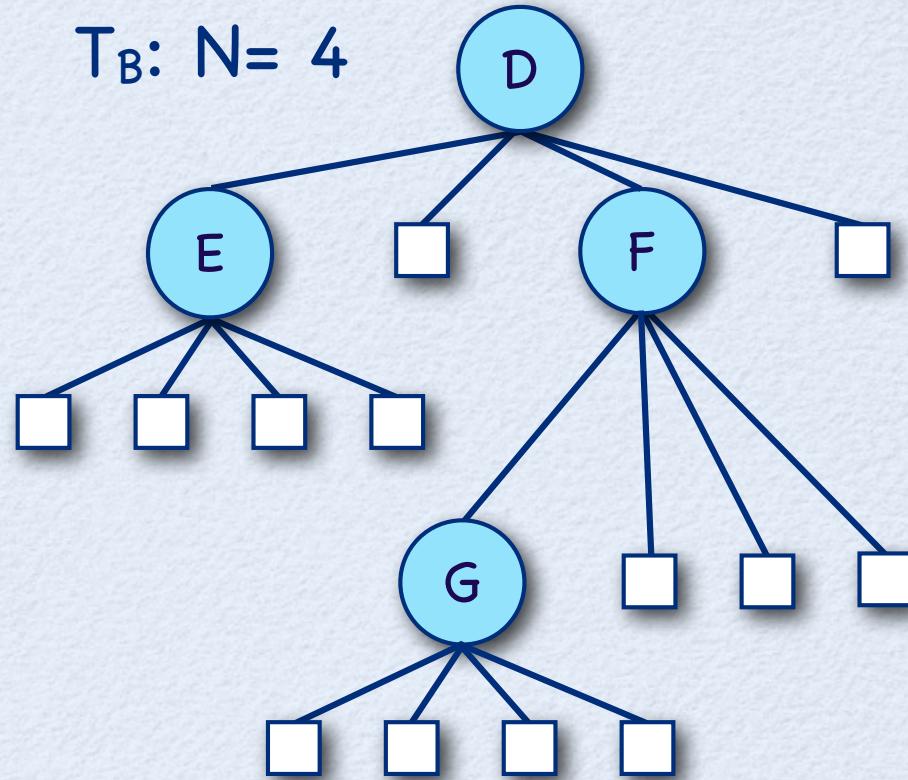
$$T = \{R, T_1, T_2, \dots, T_N\}.$$

# N-ary Tree Examples

$T_A: N = 3$



$T_B: N= 4$



$$T_A = \{A, \emptyset, \emptyset, \emptyset\}$$

$$T_B = \{D, \{E, \emptyset, \emptyset, \emptyset, \emptyset\}, \emptyset, \{F, \{G, \emptyset, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset, \emptyset\}, \emptyset\}$$

# The Empty Tree

- The empty tree,  $T = \emptyset$ , is a tree.
- From the modeling point of view an empty N-ary tree has no key and has to have the same type as a non-empty N-ary tree.
- To use null (i.e., `nullptr`) to denote an empty N-ary tree is inappropriate, as null refers to nothing at all!

# Sentinel Node: NIL

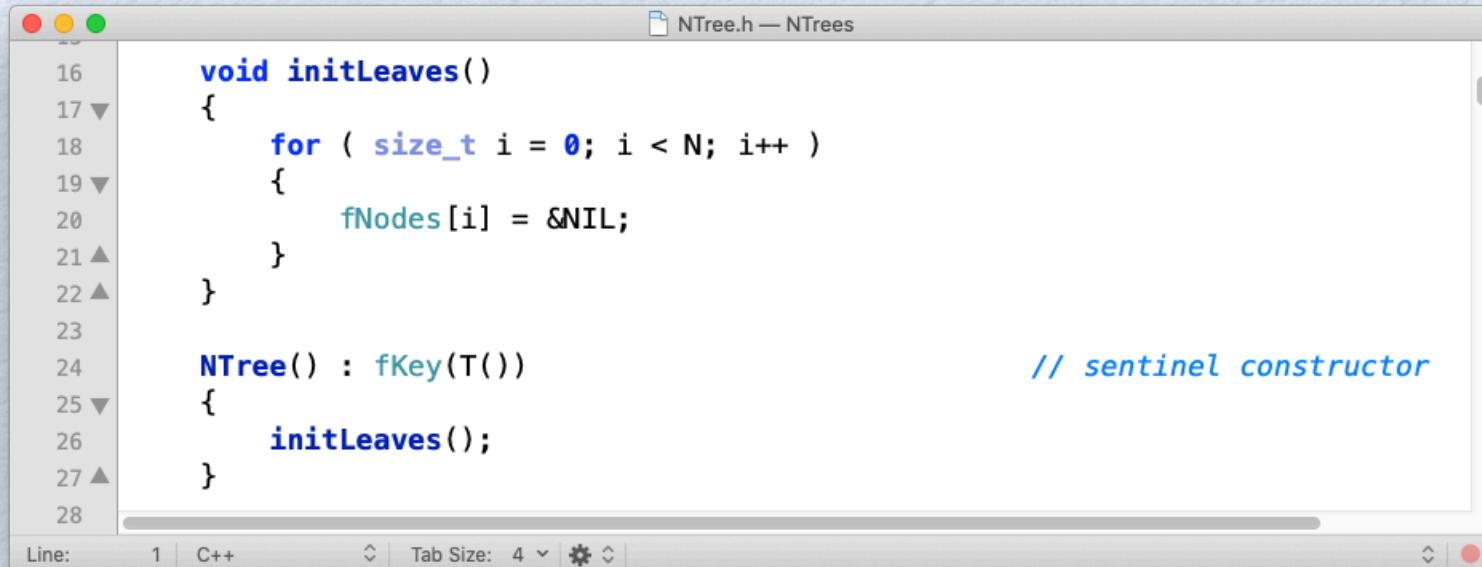
- A sentinel node is a programming idiom used to facilitate tree-based operations.
- A sentinel node in tree structures indicates a node with no children.
- Sentinel nodes behave like null-pointers. However, unlike null-pointers, which refer to nothing, sentinel nodes denote proper, yet empty, subtrees.

# Class Template NTree<T,N>

```
6  template<typename T, size_t N>
7  class NTree
8  {
9  private:
10     T fKey;                                     // T() for empty NTree
11     NTree<T,N>* fNodes[N];                     // N subtrees of degree N
12
13     void initLeaves();                         // initialize subtree nodes
14
15     NTree();                                    // sentinel constructor
16
17 public:
18     static NTree<T,N> NIL;                     // Empty NTree
19
20     NTree( const T& aKey );                   // NTree leaf
21     NTree( T&& aKey );                      // NTree leaf
22
23     NTree( const NTree& aOtherNTree );        // copy constructor
24     NTree( NTree&& aOtherNTree );            // move constructor
25
26     virtual ~NTree();                          // destructor
27
28     NTree& operator=( const NTree& aOtherNTree ); // copy assignment operator
29     NTree& operator=( NTree&& aOtherNTree );    // move assignment operator
30
31     virtual NTree* clone();                   // clone a tree
32
33     bool empty() const;                       // is tree empty
34     const T& operator*() const;                // get key (node value)
35
36     const NTree& operator[]( size_t aIndex ) const; // indexer
37
38     // tree manipulators
39     void attach( size_t aIndex, const NTree<T,N>& aNTree );
40     const NTree& detach( size_t aIndex );
41 }
```

We do not wish to allow clients to create empty NTrees.

# The Private NTree<T,N> Constructor

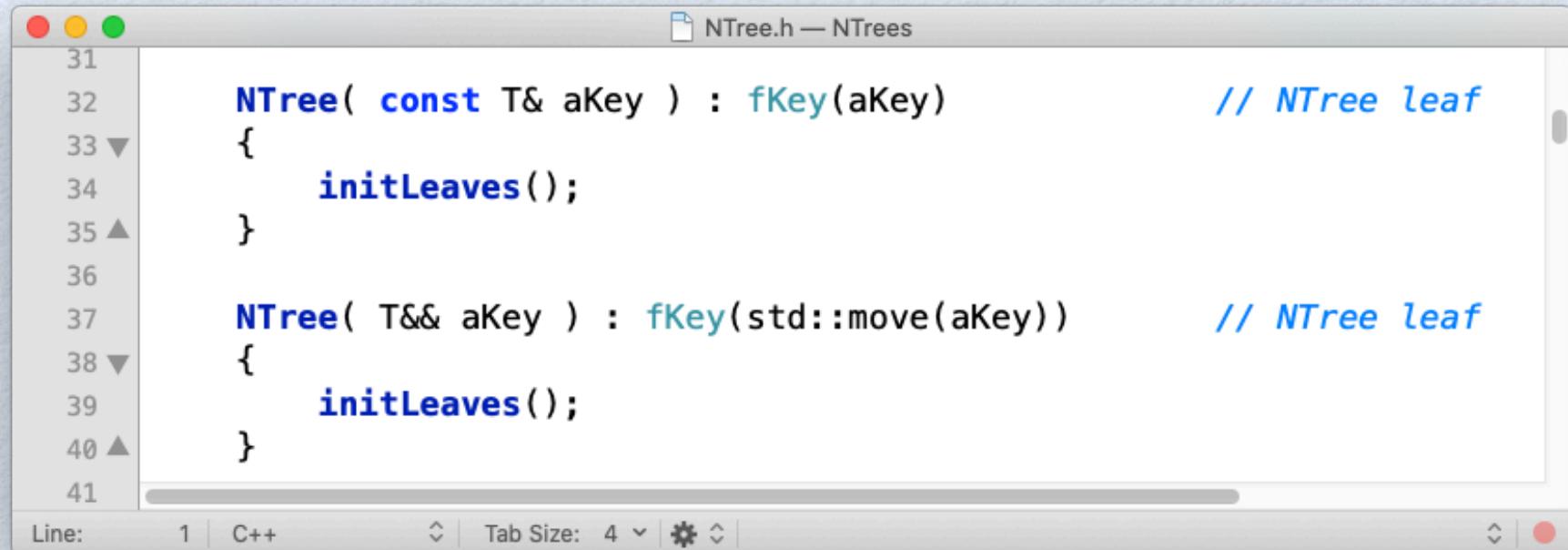


A screenshot of a code editor window titled "NTree.h — NTrees". The code is written in C++ and defines a private constructor for the class NTree. The constructor initializes the fKey member variable using the default constructor for type T, and it also calls the initLeaves() method to initialize all subtree-nodes to the location of NIL.

```
16 void initLeaves()
17 {
18     for ( size_t i = 0; i < N; i++ )
19     {
20         fNodes[i] = &NIL;
21     }
22 }
23
24 NTree() : fKey(T())
25 // sentinel constructor
26 {
27     initLeaves();
28 }
```

- We use `T()`, the default constructor for type `T`, to initialize the `fKey`.
- Each subtree-node is set to to the location of `NIL`, the sentinel node for `NTree<T,N>` using `initLeaves()`.
- This constructor is *solely* being used to set up the sentinel for `NTree<T,N>`. Clients should and cannot use the default constructor.

# The Public NTree<T,N> Constructors



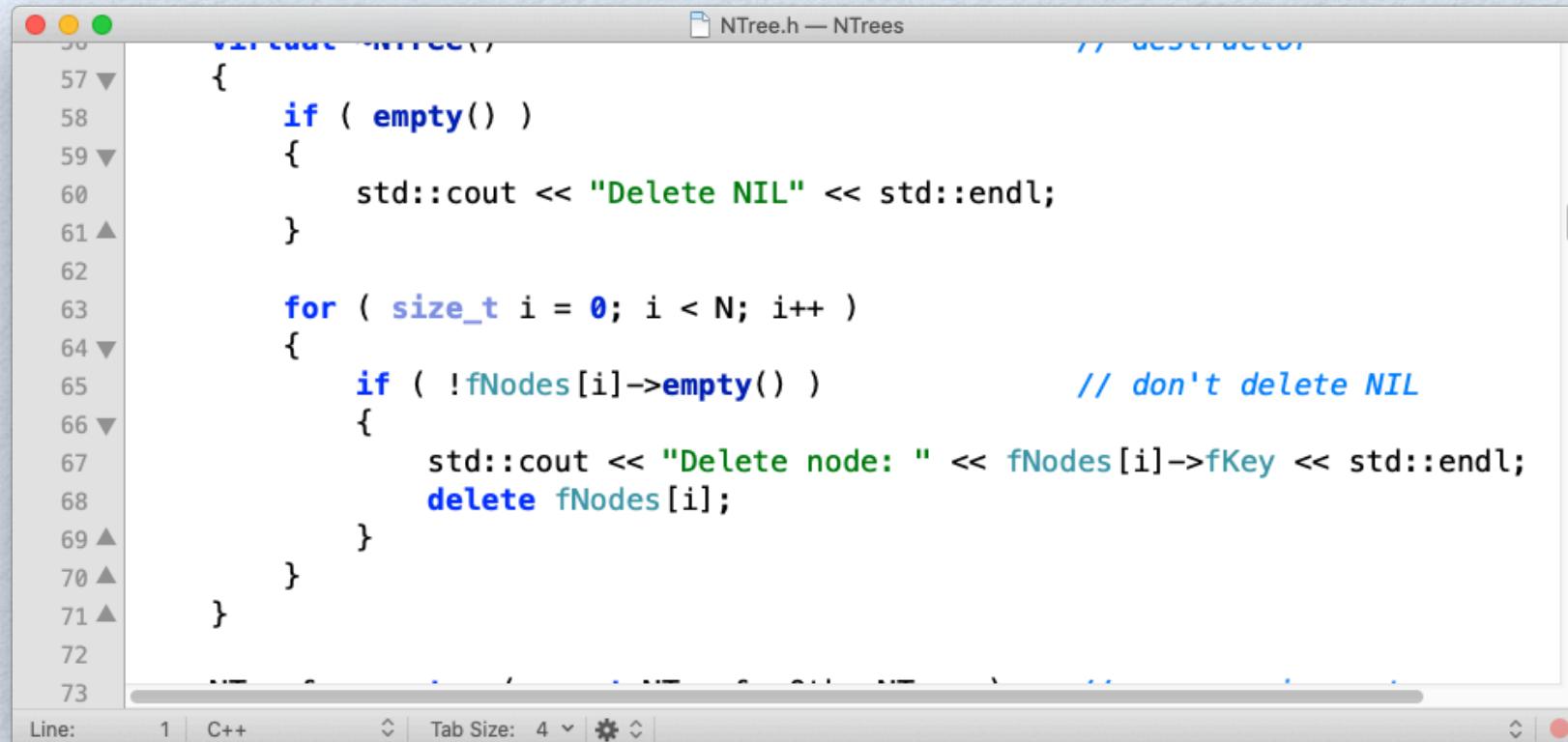
```
NTree( const T& aKey ) : fKey(aKey)           // NTree leaf
{
    initLeaves();
}

NTree( T&& aKey ) : fKey(std::move(aKey))      // NTree leaf
{
    initLeaves();
}
```

The screenshot shows a code editor window titled "NTree.h — NTrees". The code displays two constructor definitions for the class NTree. Both constructors initialize a member variable fKey. The first constructor takes a const reference to a key and moves it into fKey. The second constructor takes a moveable reference to a key and moves it into fKey. Both constructors call a private method initLeaves() to initialize child nodes.

- We copy (or move) aKey into fKey.
- Each child node in a non-empty NTree<T,N> leaf node is set to the location of NIL, the sentinel node for NTree<T,N> using initLeaves().

# The NTree<T,N> Destructor



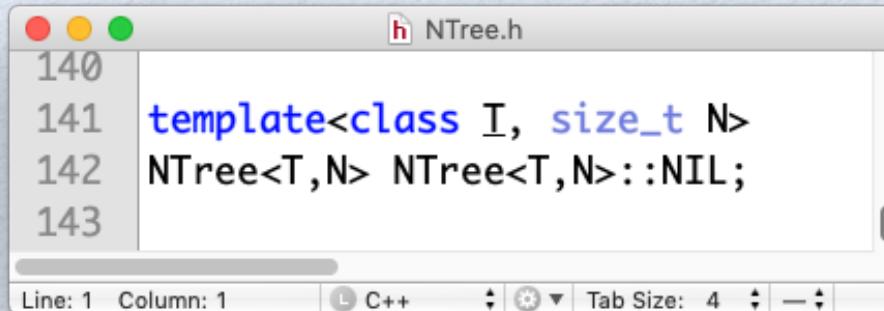
A screenshot of a C++ code editor window titled "NTree.h — NTrees". The code shows the implementation of a virtual destructor for the NTree class. The code iterates through an array of pointers to nodes, fNodes, and prints a message for each node if it is not empty. It then checks if the node is not NIL (empty) and if not, it prints a message and deletes the node using the delete operator. The code is annotated with a comment // don't delete NIL.

```
virtual ~NTree()
{
    if ( empty() )
    {
        std::cout << "Delete NIL" << std::endl;
    }

    for ( size_t i = 0; i < N; i++ )
    {
        if ( !fNodes[i]->empty() )           // don't delete NIL
        {
            std::cout << "Delete node: " << fNodes[i]->fKey << std::endl;
            delete fNodes[i];
        }
    }
}
```

- In the destructor of NTree<T,N> only non-sentinel nodes are destroyed.
- The output is for debugging purposes only.

# The NTree<T,N> Sentinel



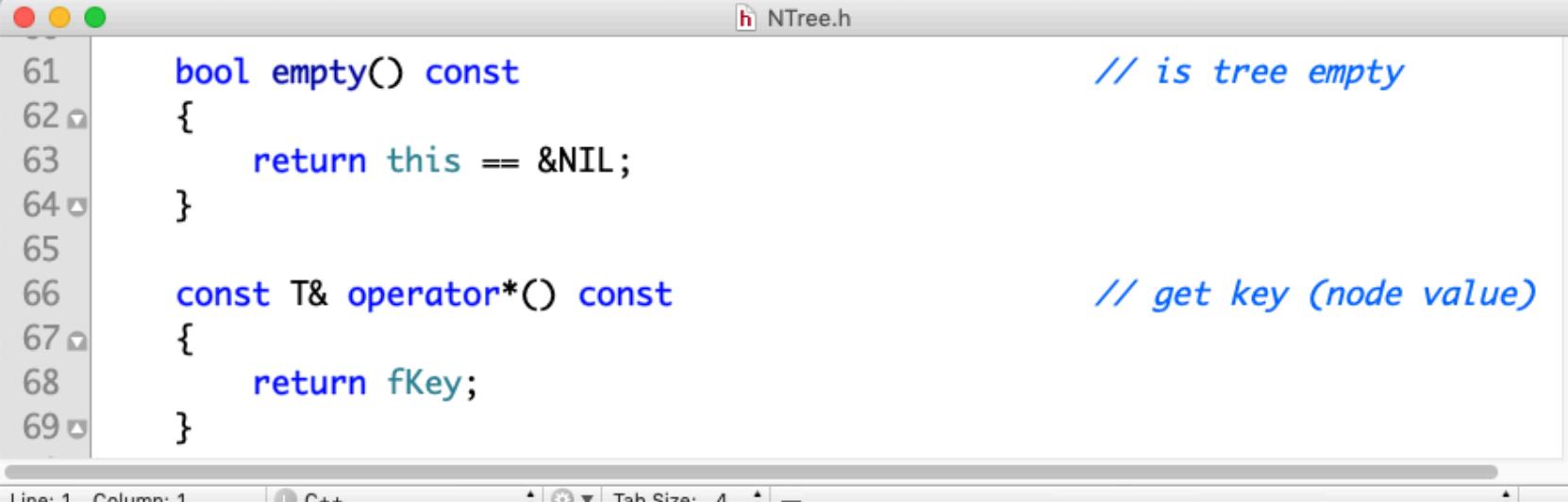
A screenshot of a C++ code editor window titled "NTree.h". The code shown is:

```
140
141 template<class T, size_t N>
142 NTree<T,N> NTree<T,N>::NIL;
143
```

The editor interface includes standard window controls (red, yellow, green buttons), a file icon, and the file name "NTree.h". At the bottom, it shows "Line: 1 Column: 1", "C++", "Tab Size: 4", and other typical IDE controls.

- Static instance variables, like the `NTree<T,N>` sentinel `NIL`, need to be initialized outside the class definition.
- Here, `NIL` is initialized using the private default constructor.
- The scope of `NIL` is `NTree<T,N>`, which means that all members of `NTree<T,N>` are available, including the private constructor to initialize `NIL`.

# The NTree<T,N> Auxiliaries



A screenshot of a C++ code editor window titled "NTree.h". The code displays two member functions of the NTree class:

```
61     bool empty() const           // is tree empty
62 {
63     return this == &NIL;
64 }
65
66     const T& operator*() const  // get key (node value)
67 {
68     return fKey;
69 }
```

The editor interface includes a status bar at the bottom showing "Line: 1 Column: 1", a tab labeled "C++", and a "Tab Size: 4" setting.

- A tree of type `NTree<T,N>` is empty if it is equal to the sentinel `NIL`.
- The dereference operator returns the payload (i.e., the root) of a `NTree<T,N>` tree. (We can use `NIL` as temporary storage.)

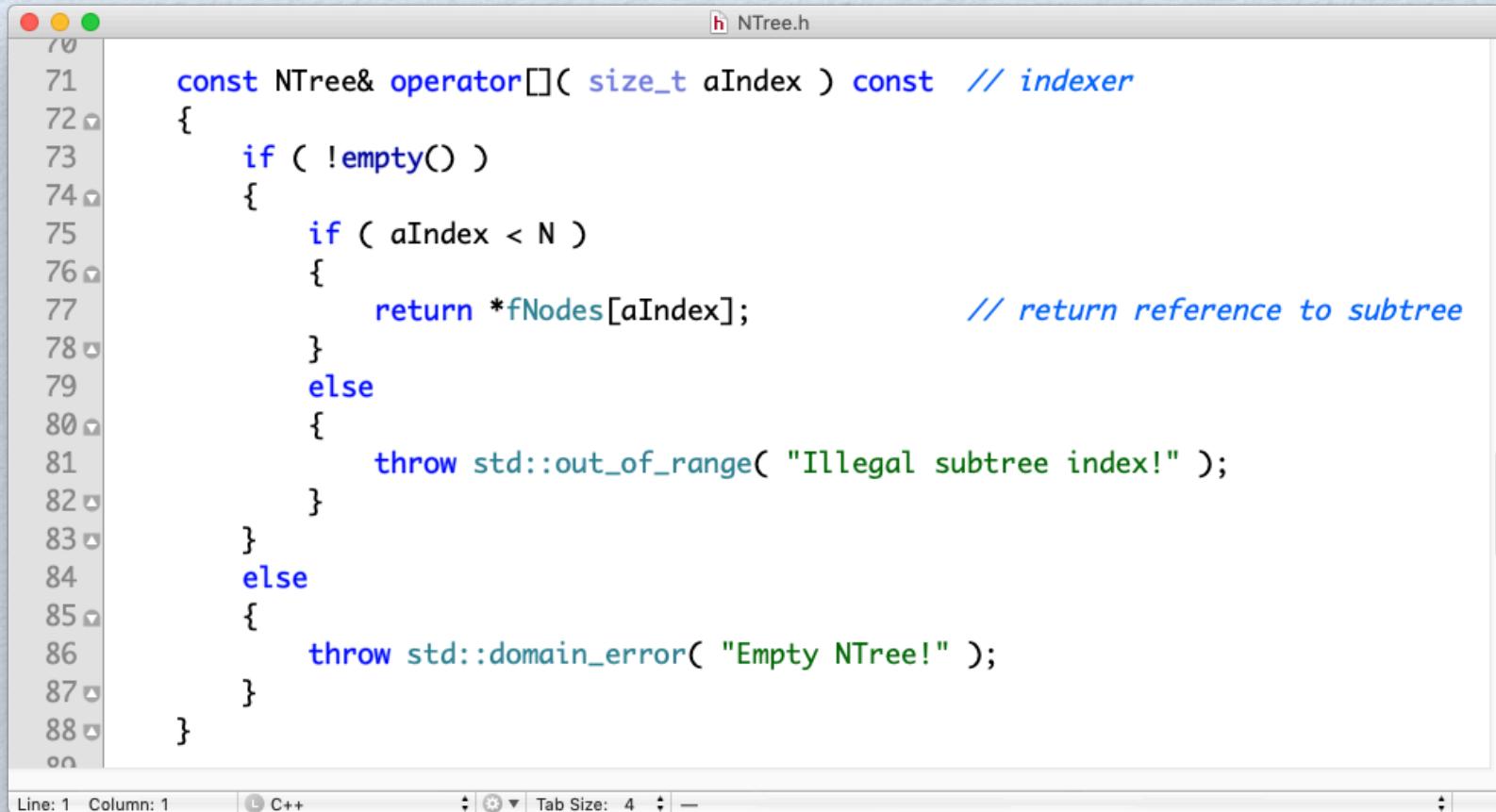
# Attaching a New Subtree



The screenshot shows a code editor window with the file `NTree.h` open. The code implements the `attach` method for a class `NTree<T,N>`. The method takes an index and another tree as parameters and checks if the target node is empty before attaching the new tree.

```
91
92     void attach( size_t aIndex, const NTree<T,N>& aNTree )
93     {
94         if ( !empty() )
95         {
96             if ( aIndex < N )
97             {
98                 if ( fNodes[aIndex]->empty() )
99                 {
100                     fNodes[aIndex] = const_cast<NTree<T,N>*>(&aNTree);
101                 }
102                 else
103                 {
104                     throw std::domain_error( "Non-empty subtree present!" );
105                 }
106             }
107             else
108             {
109                 throw std::out_of_range( "Illegal subtree index!" );
110             }
111         }
112         else
113         {
114             throw std::domain_error( "Empty NTree!" );
115         }
116     }
```

# Accessing a Subtree



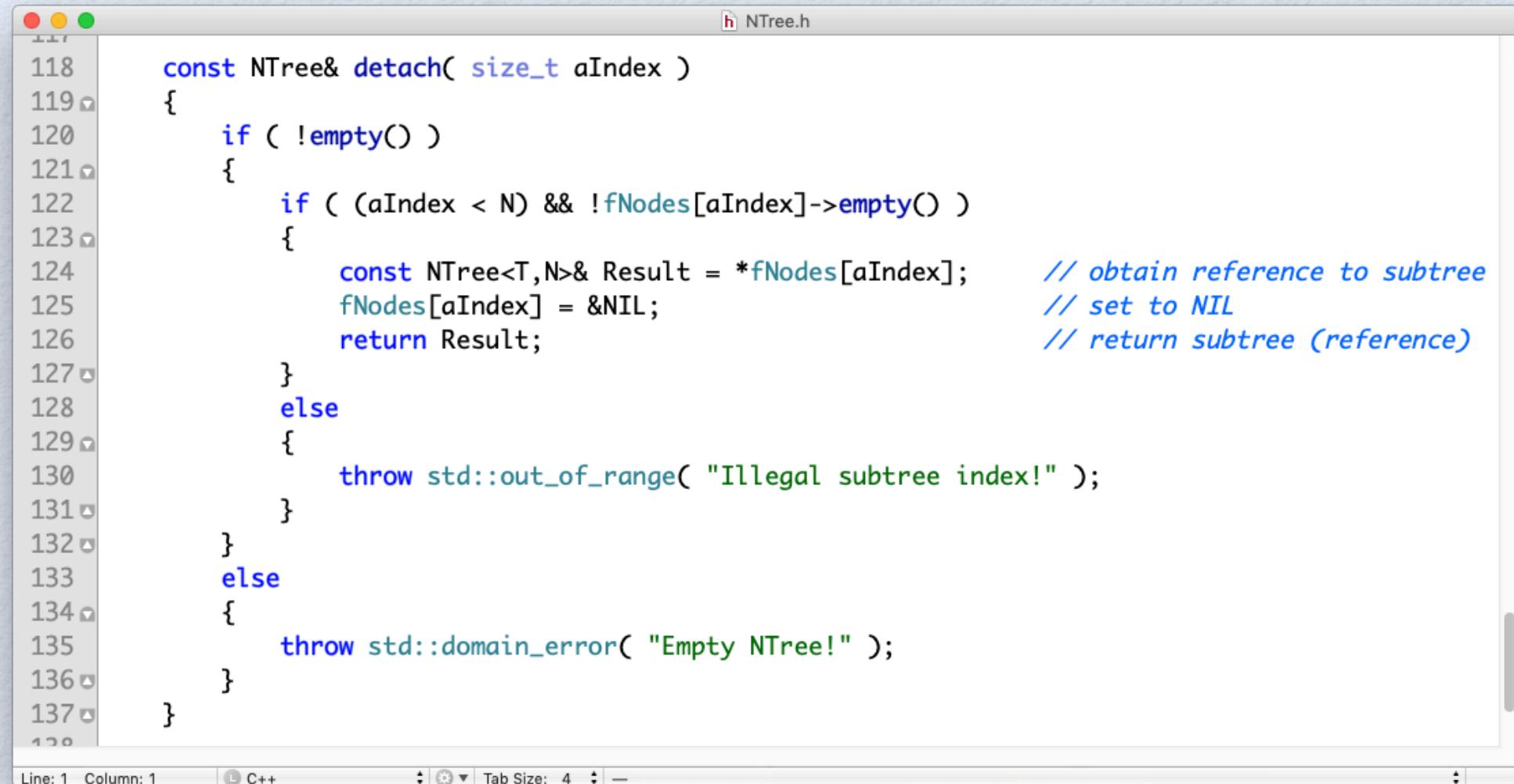
A screenshot of a code editor window titled "NTree.h". The code is a C++ implementation of a class named NTree. It contains a const operator[] method that takes an index and returns a reference to a subtree. The code includes error handling for illegal indices and empty trees.

```
70
71     const NTree& operator[]( size_t aIndex ) const // indexer
72     {
73         if ( !empty() )
74         {
75             if ( aIndex < N )
76             {
77                 return *fNodes[aIndex];           // return reference to subtree
78             }
79             else
80             {
81                 throw std::out_of_range( "Illegal subtree index!" );
82             }
83         }
84         else
85         {
86             throw std::domain_error( "Empty NTree!" );
87         }
88     }
89 
```

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

- We return a reference to the subtree rather than a pointer. This way, we prevent accidental manipulations outside the tree structure.

# Removing a Subtree

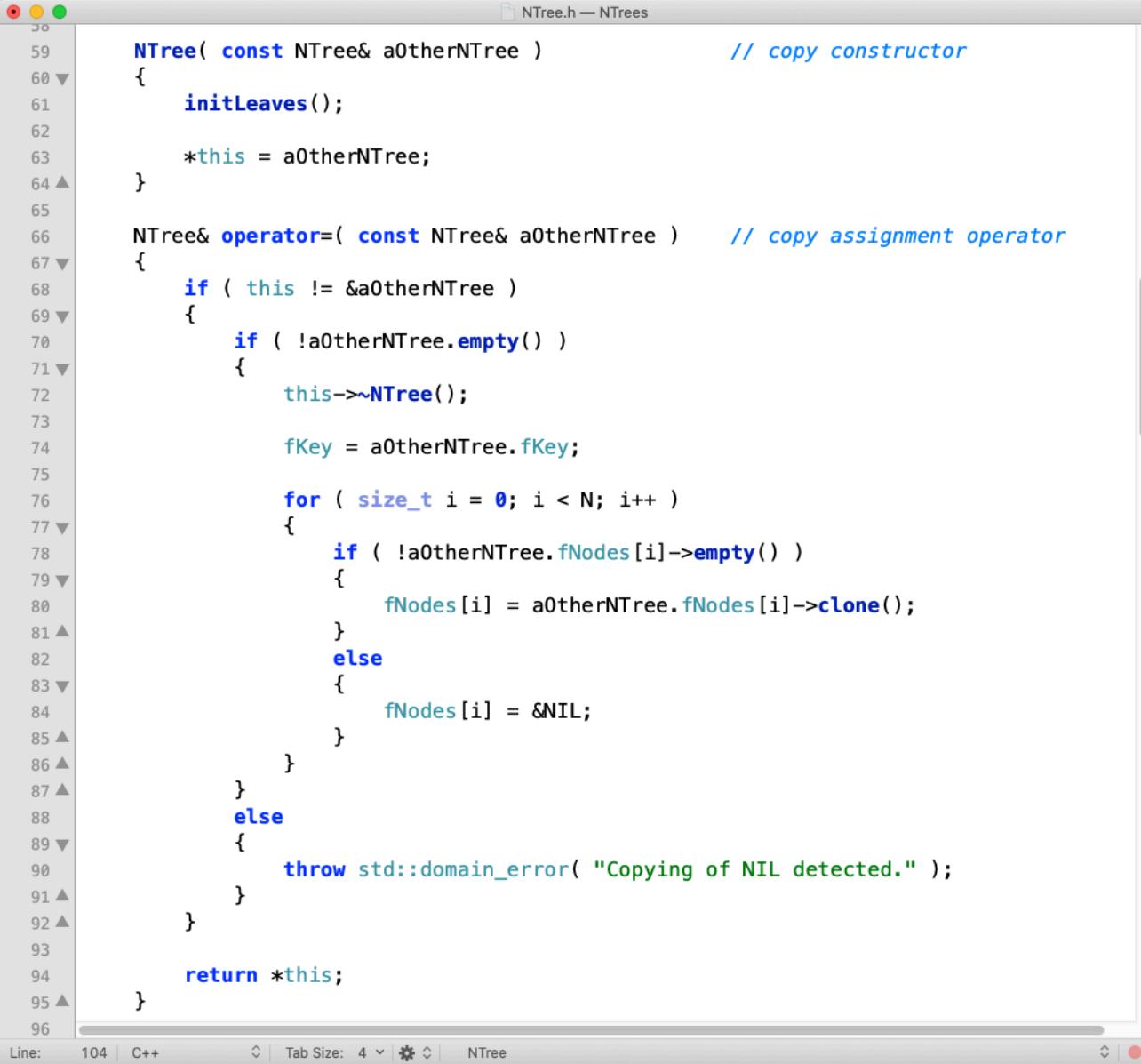


A screenshot of a code editor window titled "NTree.h". The code implements a function `detach` that removes a subtree from a binary tree structure. The code uses a vector of pointers to nodes (`fNodes`) and checks if a node is empty before attempting to remove its subtree. It throws exceptions for illegal index or empty tree cases.

```
118     const NTree& detach( size_t aIndex )
119 {
120     if ( !empty() )
121     {
122         if ( (aIndex < N) && !fNodes[aIndex]->empty() )
123         {
124             const NTree<T,N>& Result = *fNodes[aIndex];           // obtain reference to subtree
125             fNodes[aIndex] = &NIL;                                // set to NIL
126             return Result;                                     // return subtree (reference)
127         }
128         else
129         {
130             throw std::out_of_range( "Illegal subtree index!" );
131         }
132     }
133     else
134     {
135         throw std::domain_error( "Empty NTree!" );
136     }
137 }
```

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

# Copy Semantics



```
NTree( const NTree& aOtherNTree )           // copy constructor
{
    initLeaves();

    *this = aOtherNTree;
}

NTree& operator=( const NTree& aOtherNTree ) // copy assignment operator
{
    if ( this != &aOtherNTree )
    {
        if ( !aOtherNTree.empty() )
        {
            this->~NTree();

            fKey = aOtherNTree.fKey;

            for ( size_t i = 0; i < N; i++ )
            {
                if ( !aOtherNTree.fNodes[i]->empty() )
                {
                    fNodes[i] = aOtherNTree.fNodes[i]->clone();
                }
                else
                {
                    fNodes[i] = &NIL;
                }
            }
        }
        else
        {
            throw std::domain_error( "Copying of NIL detected." );
        }
    }
    return *this;
}
```

# Move Semantics

```
NTree( NTree&& aOtherNTree ) // move constructor
{
    initLeaves();

    *this = std::move(aOtherNTree);
}

NTree& operator=( NTree&& aOtherNTree ) // move assignment operator
{
    if ( this != &aOtherNTree )
    {
        if ( !aOtherNTree.empty() )
        {
            this->~NTree();

            fKey = std::move(aOtherNTree.fKey);

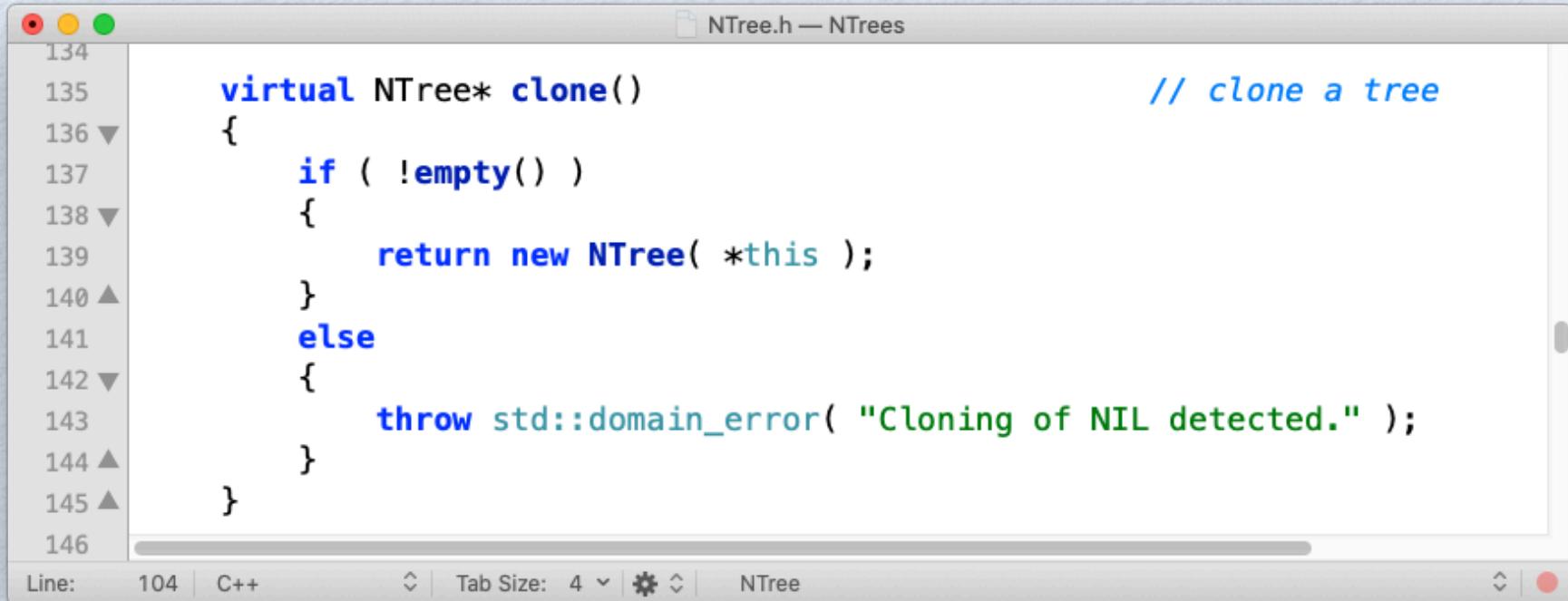
            for ( size_t i = 0; i < N; i++ )
            {
                if ( !aOtherNTree.fNodes[i]->empty() )
                {
                    fNodes[i] = const_cast<NTree<T,N>*>(&aOtherNTree.detach( i ));
                }
                else
                {
                    fNodes[i] = &NIL;
                }
            }
        }
        else
        {
            throw std::domain_error( "Moving of NIL detected." );
        }
    }

    return *this;
}
```



Steal memory

# Clone



The screenshot shows a code editor window titled "NTree.h — NTrees". The code is written in C++ and defines a virtual clone method for the NTree class. The method checks if the tree is empty; if it is, it throws a domain error indicating that cloning NIL is detected. If the tree is not empty, it returns a new instance of NTree with a copy of the current object.

```
134
135     virtual NTree* clone() // clone a tree
136 {
137     if ( !empty() )
138     {
139         return new NTree( *this );
140     }
141     else
142     {
143         throw std::domain_error( "Cloning of NIL detected." );
144     }
145 }
146
```

Line: 104 | C++ | Tab Size: 4 | NTree

- The method `clone()` must not duplicate NIL.
- The sentinel NIL is a unique instance of `NTree<N,T>` for every instantiation of N and T.

# A NTree<T,N> Example

```
17 void testBasicOperations()
18 {
19     using NS3Tree = NTree<string,3>;
20
21     string s1( "A" );
22     string s2( "B" );
23     string s3( "C" );
24
25     NS3Tree root( "Hello World!" );
26     NS3Tree nodeA( s1 );
27     NS3Tree nodeB( s2 );
28     NS3Tree nodeC( s3 );
29     NS3Tree nodeAB( "AB" );
30
31     root.attach( 0, nodeA );
32     root.attach( 1, nodeB );
33     root.attach( 2, nodeC );
34     const_cast<NS3Tree&>(root[1]).attach( 1, nodeAB );
35
36     cout << "root:      " << *root << endl;
37     cout << "root[0]:    " << *root[0] << endl;
38     cout << "root[1]:    " << *root[1] << endl;
39     cout << "root[2]:    " << *root[2] << endl;
40     cout << "root[1][1]: " << *root[1][1] << endl;
41
42     const_cast<NS3Tree&>(root[1]).detach( 1 );
43     root.detach( 0 );
44     root.detach( 1 );
45     root.detach( 2 );
46 }
```

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

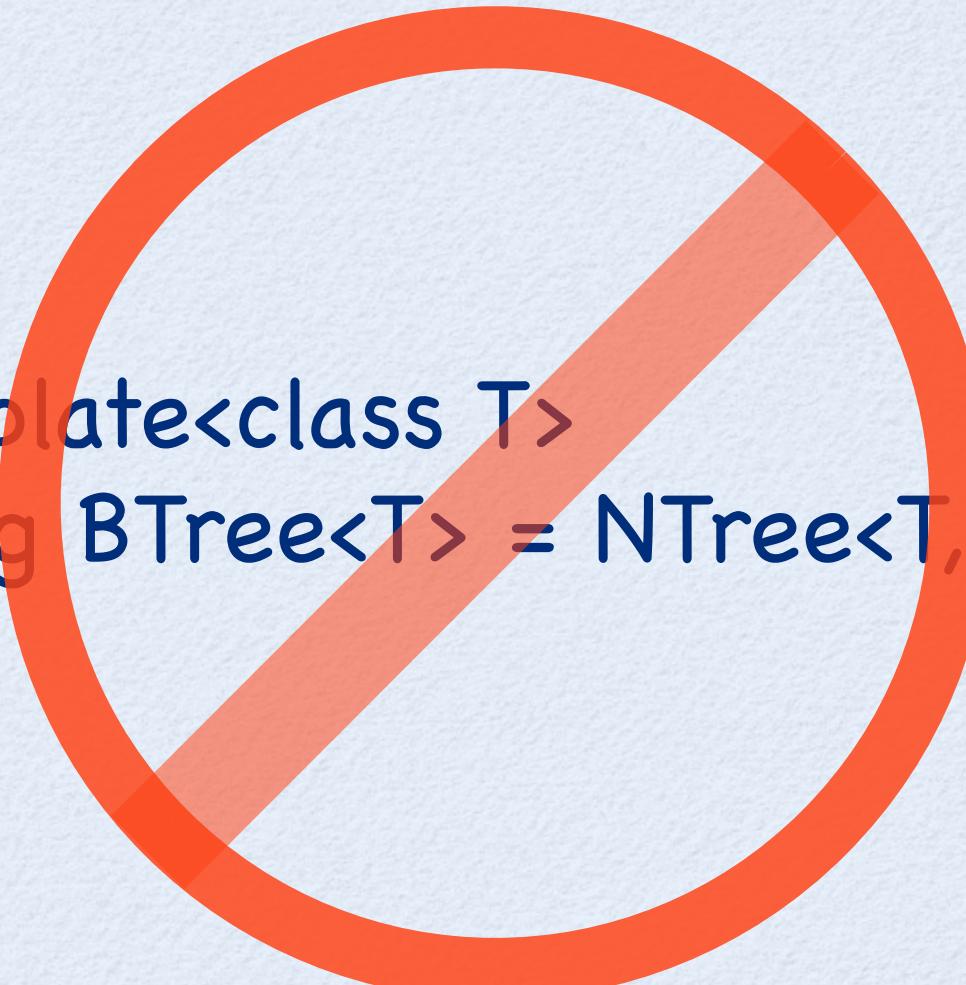
```
Kamala:NTrees Markus$ ./NTreeTest
root:      Hello World!
root[0]:    A
root[1]:    B
root[2]:    C
root[1][1]: AB
Kamala:NTrees Markus$
```

See full test on Canvas

# 2-ary Trees: Binary Trees

- A binary tree  $T$  is a finite set of nodes with one of the following properties:
  - Either the set is empty,  $T = \emptyset$ , or
  - The set consists of a root,  $r$ , and exactly 2 distinct binary trees  $T_L$  and  $T_R$ ,  $T = \{r, T_L, T_R\}$ .
- The tree  $T_L$  is called the left subtree of  $T$  and the tree  $T_R$  is called the right subtree of  $T$ .

# We cannot just create a top-level type alias!



```
template<class T>
using BTree<T> = NTree<T,2>;
```

# BTree<T>

```
8  template<typename T>
9  class BTree
10 {
11     private:
12         T fKey;                                     // TC) for empty BTree
13         BTree<T>* fLeft;
14         BTree<T>* fRight;
15
16     BTree();                                       // sentinel constructor
17
18     public:
19         static BTree<T> NIL;                      // Empty BTree
20
21         BTree( const T& aKey );                  // BTREE leaf
22         BTree( T&& aKey );                      // BTREE leaf
23
24         BTree( const BTree& aOtherBTree );       // copy constructor
25         BTree( BTree&& aOtherBTree );           // move constructor
26
27         virtual ~BTree();                         // destructor
28
29         BTree& operator=( const BTree& aOtherBTree ); // copy assignment operator
30         BTree& operator=( BTree&& aOtherBTree );   // move assignment operator
31
32         virtual BTree* clone();                   // clone a tree
33
34         bool empty() const;                      // is tree empty
35         const T& operator*() const;              // get key (node value)
36
37         const BTree& left() const;
38         const BTree& right() const;
39
40         // tree manipulators
41         void attachLeft( const BTree<T>& aBTree );
42         void attachRight( const BTree<T>& aBTree );
43         const BTree& detachLeft();
44         const BTree& detachRight();
45 };
```

Line: 2 Column: 11 C++ Tab Size: 4

# Tree Traversal

- Many different algorithms for manipulating trees exist, but these algorithms have in common that they systematically visit all the nodes in the tree.
- There are essentially two methods for visiting all nodes in a tree:
  - Depth-first traversal,
  - Breadth-first traversal.

# Depth-first Traversal

- Pre-order traversal:
  - Visit the root first; and then
  - Do a preorder traversal of each of the subtrees of the root one-by-one in the order given (from left to right).
- Post-order traversal:
  - Do a postorder traversal of each of the subtrees of the root one-by-one in the order given (from left to right); and then
  - Visit the root.
- In-order traversal:
  - Traverse the left subtree; and then
  - Visit the root; and then
  - Traverse the right subtree.

# Breadth-first Traversal

- Breadth-first traversal visits the nodes of a tree in the order of their depth (from left to right).

# Pre-order Traversal Example

Depth = 0:

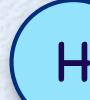


D-E-H-F-G-I-J-K-L

Depth = 1:



Depth = 2:



Depth = 3:



# Post-order Traversal Example

Depth = 0:

H-E-I-G-K-L-J-F-D

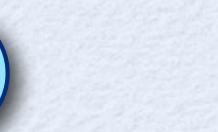
Depth = 1:



Depth = 2:



Depth = 3:



# In-order Traversal Example

Depth = 0:



H-E-D-I-G-F-K-J-L

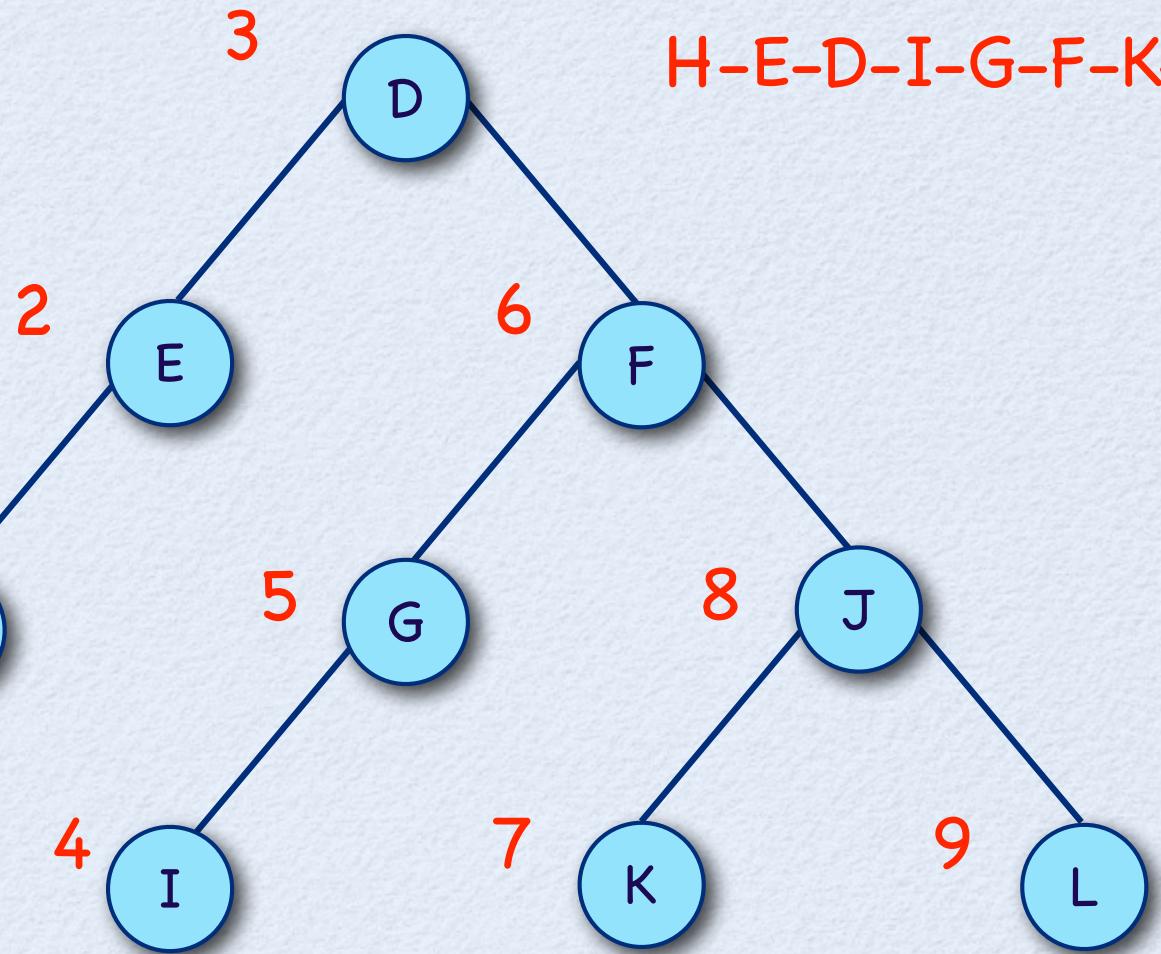
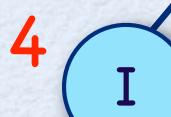
Depth = 1:



Depth = 2:



Depth = 3:



# Breadth-first Traversal Example

Depth = 0:



D-E-F-H-G-J-I-K-L

Depth = 1:



Depth = 2:



Depth = 3:



1

2

3

4

5

6

7

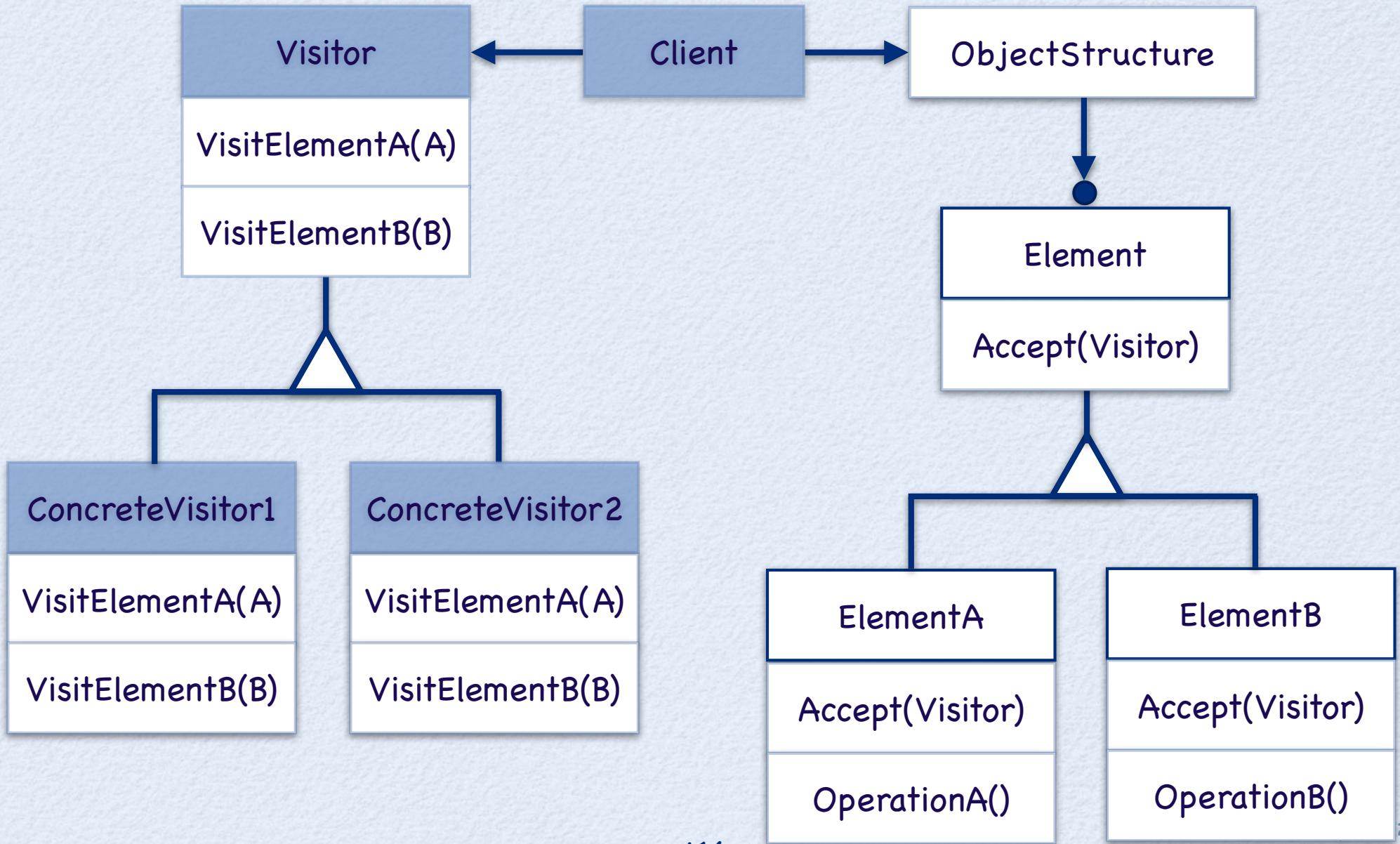
8

9

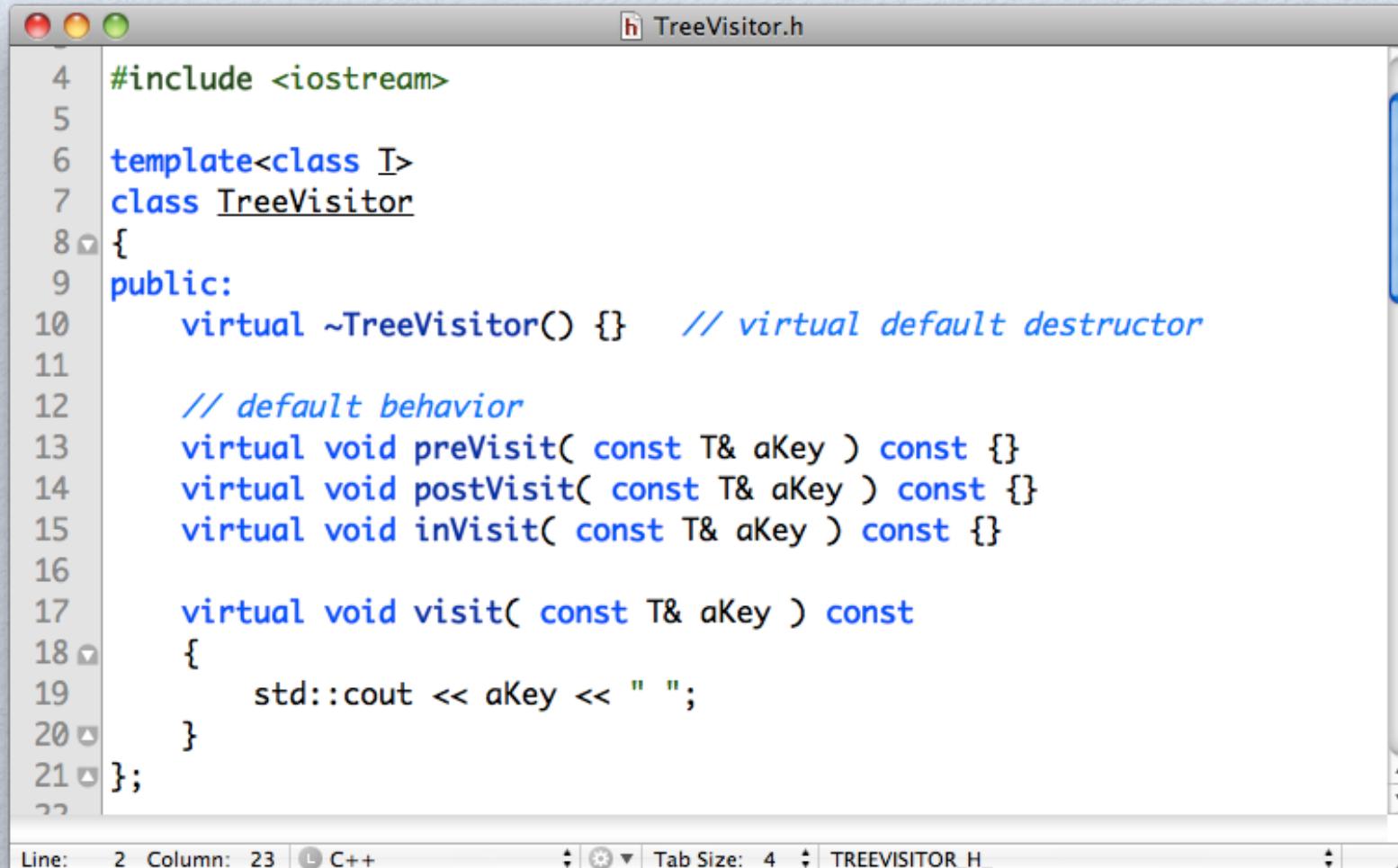
# The Visitor Pattern

- Intent:
  - Represent an operation to be performed **on** the elements of an object structure. Visitor lets one define a new operation without changing the classes of the elements on which it operates.
- Collaborations:
  - A client that uses the Visitor pattern must create a **ConcreteVisitor** object and then traverse the object structure, visiting each element with the visitor.
  - When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary.

# Structure of Visitor



# A Tree Visitor

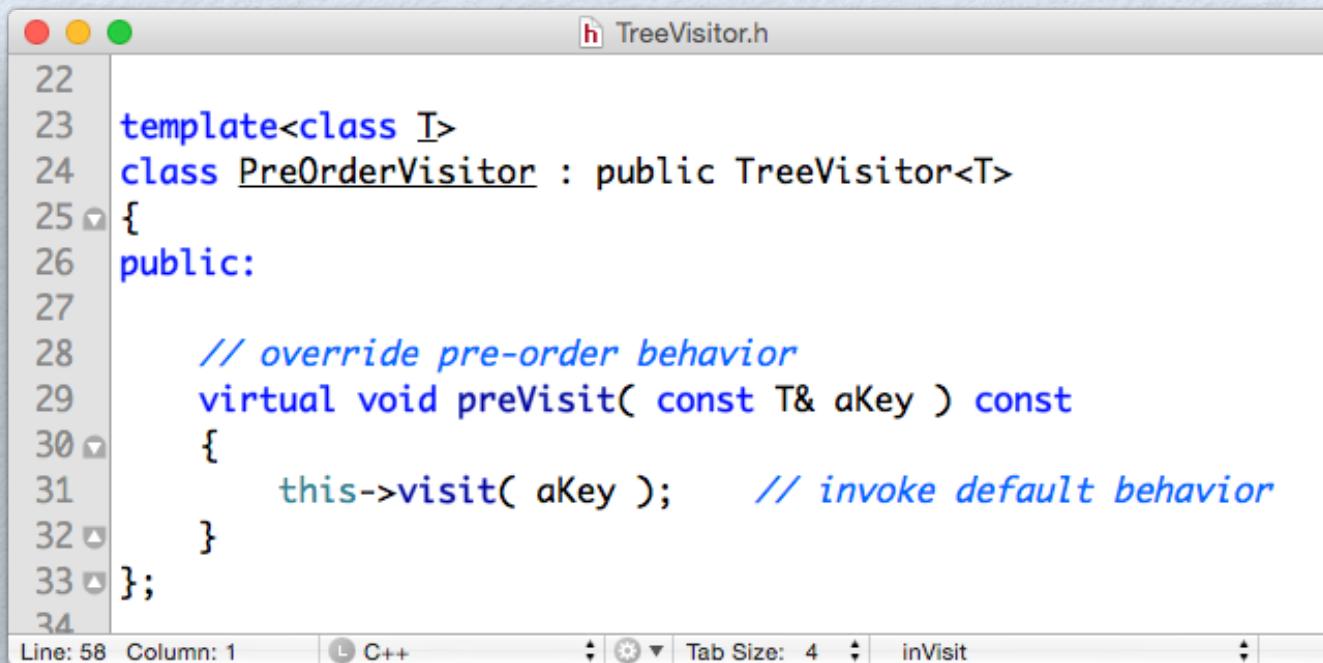


The screenshot shows a code editor window titled "TreeVisitor.h". The code defines a template class `TreeVisitor` with a virtual default destructor and three virtual methods: `preVisit`, `postVisit`, and `inVisit`. It also includes a concrete implementation for the `visit` method that outputs the key to `std::cout`.

```
4 #include <iostream>
5
6 template<class T>
7 class TreeVisitor
8 {
9 public:
10     virtual ~TreeVisitor() {} // virtual default destructor
11
12     // default behavior
13     virtual void preVisit( const T& aKey ) const {}
14     virtual void postVisit( const T& aKey ) const {}
15     virtual void inVisit( const T& aKey ) const {}
16
17     virtual void visit( const T& aKey ) const
18     {
19         std::cout << aKey << " ";
20     }
21 };
22
```

Line: 2 Column: 23 C++ Tab Size: 4 TREEVISITOR\_H\_

# PreOrderVisitor

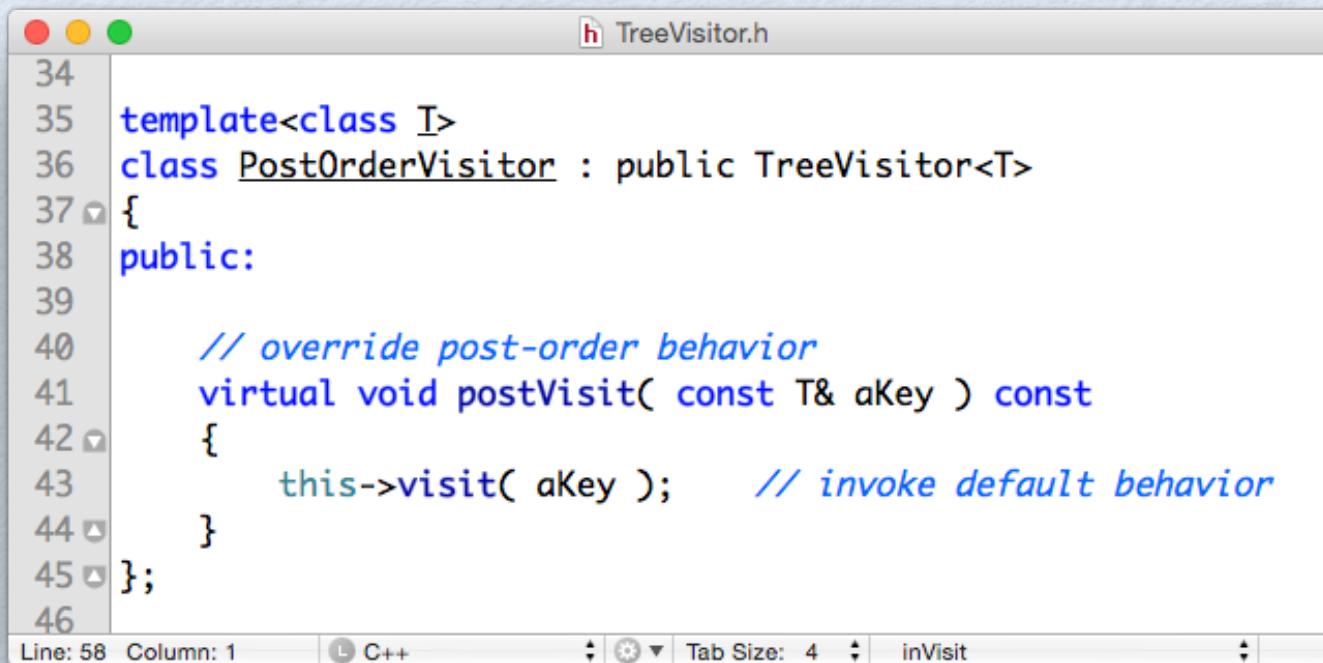


The screenshot shows a code editor window titled "TreeVisitor.h". The code implements a template class for a pre-order visitor. The class inherits from "TreeVisitor<T>" and overrides the "preVisit" method to invoke the default behavior.

```
22
23 template<class T>
24 class PreOrderVisitor : public TreeVisitor<T>
25 {
26     public:
27
28     // override pre-order behavior
29     virtual void preVisit( const T& aKey ) const
30     {
31         this->visit( aKey ); // invoke default behavior
32     }
33 };
34
```

Line: 58 Column: 1    C++    Tab Size: 4    inVisit

# PostOrderVisitor

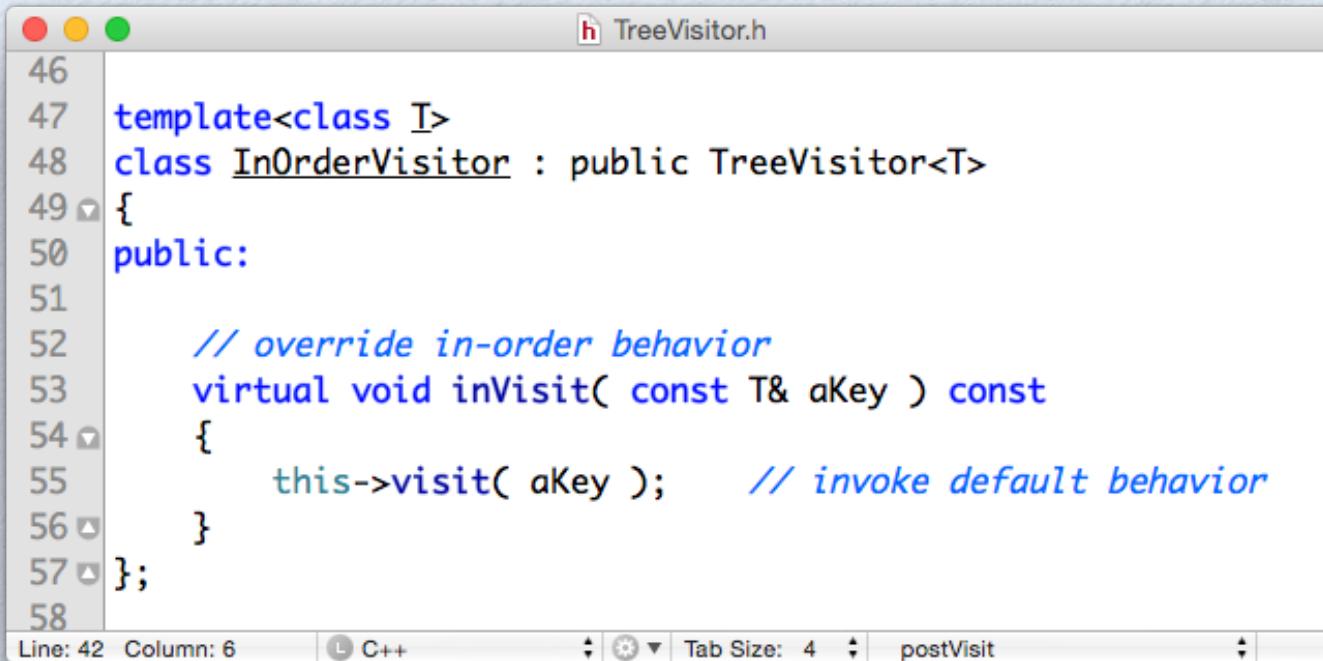


A screenshot of a code editor window titled "TreeVisitor.h". The code implements a post-order visitor pattern for a tree structure. It defines a template class `PostOrderVisitor` that inherits from `TreeVisitor<T>`. The class has a public section containing a virtual member function `postVisit` that delegates to the `visit` function of the base class.

```
34
35 template<class T>
36 class PostOrderVisitor : public TreeVisitor<T>
37 {
38     public:
39
40     // override post-order behavior
41     virtual void postVisit( const T& aKey ) const
42     {
43         this->visit( aKey ); // invoke default behavior
44     }
45 }
46
```

The status bar at the bottom shows "Line: 58 Column: 1", "C++", "Tab Size: 4", and "inVisit".

# InOrderVisitor

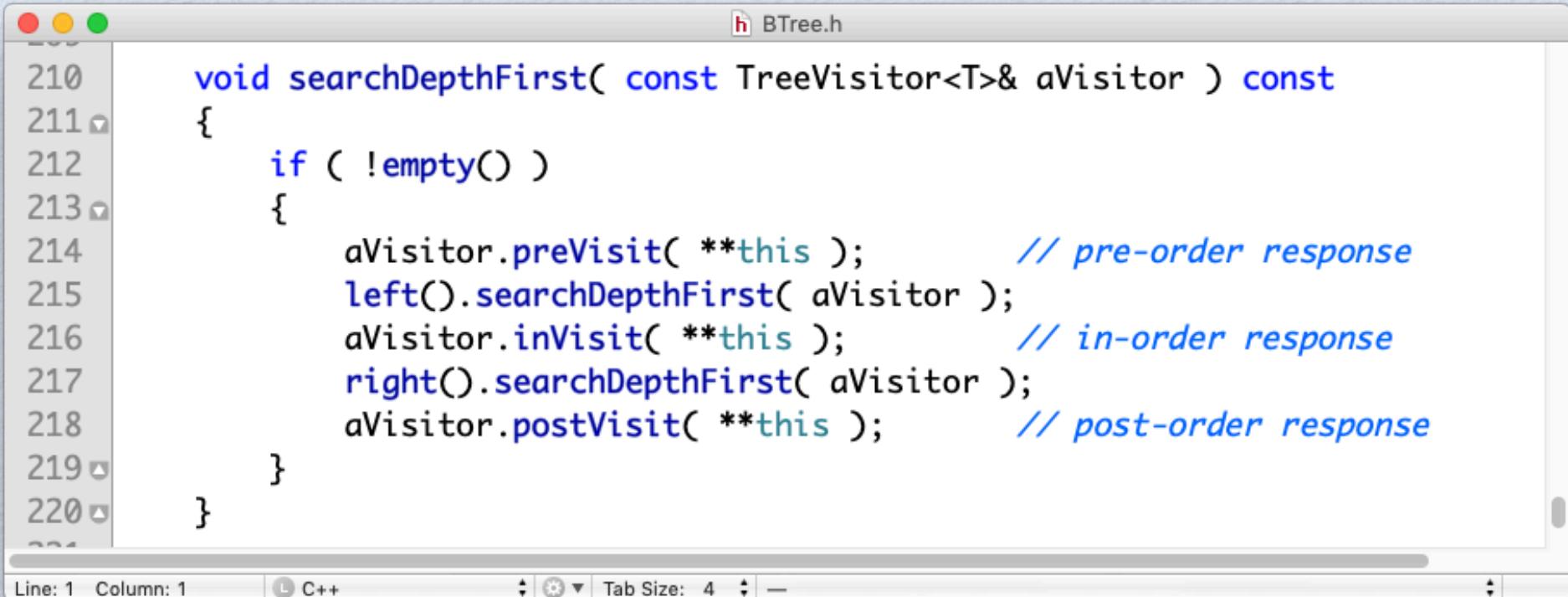


A screenshot of a code editor window titled "TreeVisitor.h". The code implements an `InOrderVisitor` class template. The code is as follows:

```
46
47 template<class T>
48 class InOrderVisitor : public TreeVisitor<T>
49 {
50 public:
51
52     // override in-order behavior
53     virtual void inVisit( const T& aKey ) const
54     {
55         this->visit( aKey );    // invoke default behavior
56     }
57 };
58
```

The code editor interface includes standard window controls (red, yellow, green buttons), a file icon, and the filename "TreeVisitor.h". At the bottom, it shows "Line: 42 Column: 6", "C++", "Tab Size: 4", and "postVisit".

# Depth-first Traversal for BTree



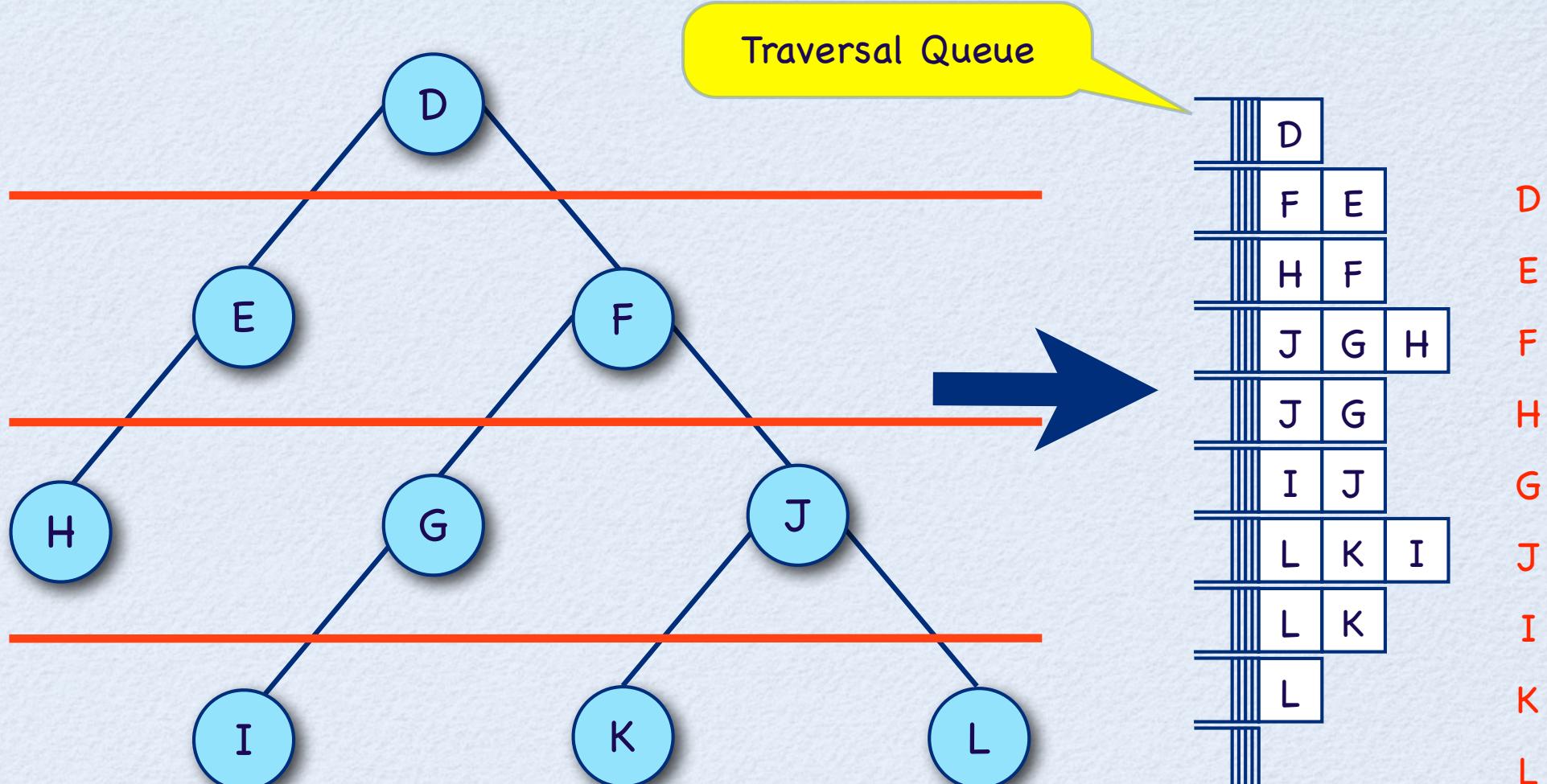
The screenshot shows a code editor window titled "BTree.h". The code is a C++ implementation of a depth-first search algorithm for a BTree. The function `searchDepthFirst` takes a reference to a `TreeVisitor<T>` object and performs a pre-order traversal. It first calls `preVisit` on the visitor, then recursively calls `searchDepthFirst` on the left child, then `inVisit` on the visitor, then recursively on the right child, and finally `postVisit` on the visitor. The code uses double pointers (`\*\*this`) to allow modification of the tree nodes during traversal.

```
210 void searchDepthFirst( const TreeVisitor<T>& aVisitor ) const
211 {
212     if ( !empty() )
213     {
214         aVisitor.preVisit( **this );           // pre-order response
215         left().searchDepthFirst( aVisitor );
216         aVisitor.inVisit( **this );          // in-order response
217         right().searchDepthFirst( aVisitor );
218         aVisitor.postVisit( **this );        // post-order response
219     }
220 }
```

```
219 #include "BTree.h"
220
221 void testDFS()
222 {
223     cout << "Test DFS." << endl;
224
225     using StringBTree = BTree<string>;
226
227     string s1( "A" );
228     string s2( "B" );
229     string s3( "C" );
230
231     StringBTree root( "Hello World!" );
232     StringBTree nodeA( s1 );
233     StringBTree nodeB( s2 );
234     StringBTree nodeAA( s3 );
235     StringBTree nodeBB( "D" );
236
237     root.attachLeft( nodeA );
238     root.attachRight( nodeB );
239     const_cast<StringBTree&>(root.left()).attachLeft( nodeAA );
240     const_cast<StringBTree&>(root.right()).attachRight( nodeBB );
241
242     cout << "root:      " << *root << endl;
243     cout << "root->L:    " << *root.left() << endl;
244     cout << "root->R:    " << *root.right() << endl;
245     cout << "root->L->L: " << *root.left().left() << endl;
246     cout << "root->R->R: " << *root.right().right() << endl;
247
248     root.searchDepthFirst( PreOrderVisitor<string>() );
249     cout << endl;
250
251     const_cast<StringBTree&>(root.right()).detachRight();
252     const_cast<StringBTree&>(root.left()).detachLeft();
253     root.detachRight();
254     root.detachLeft();
255
256     cout << "All trees are going to be deleted now!" << endl;
257 }
```

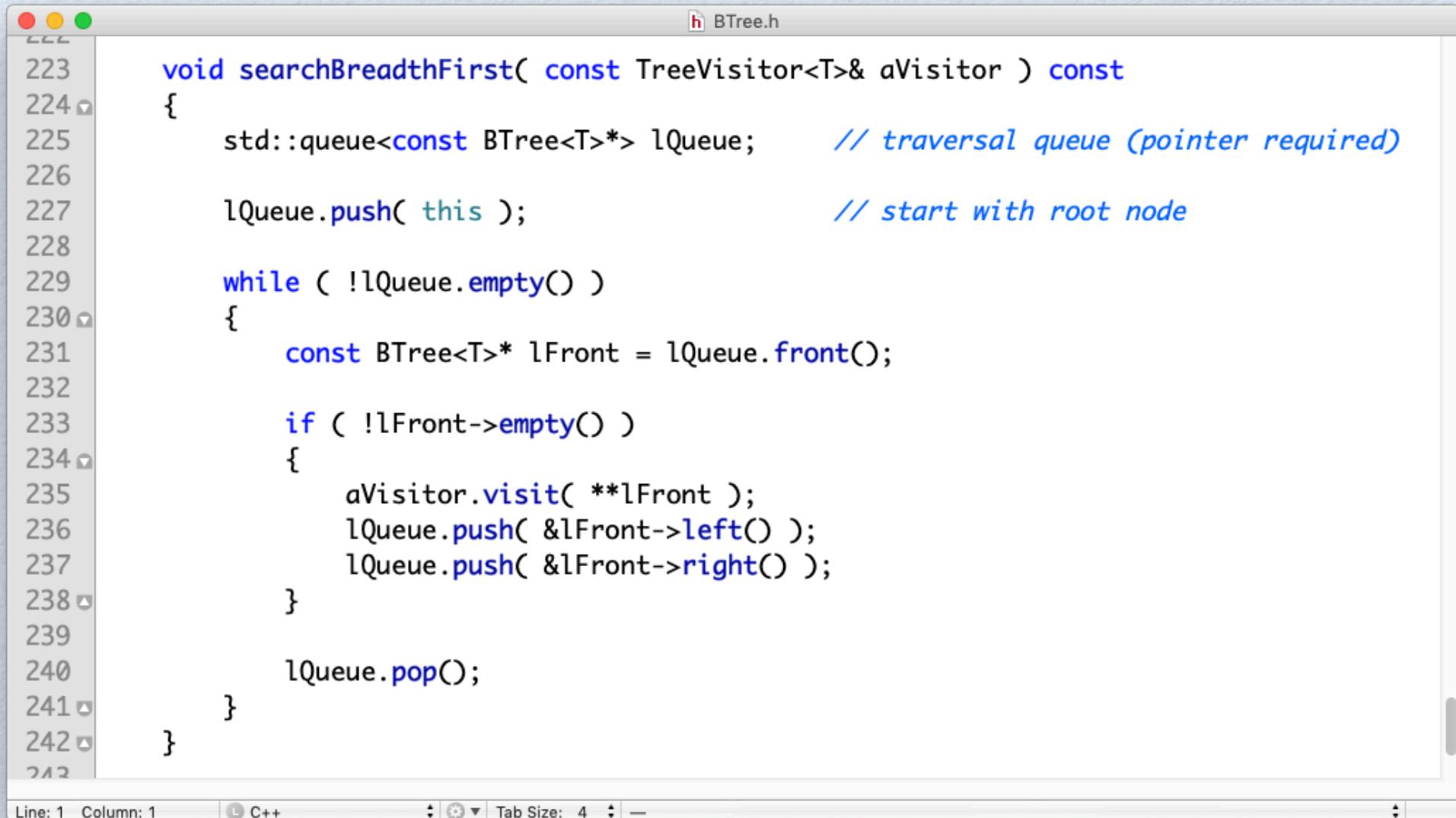
```
Kamala:BTree Markus$ ./BTreeTest
Test DFS.
root:      Hello World!
root->L:    A
root->R:    B
root->L->L: C
root->R->R: D
Hello World! A C B D
All trees are going to be deleted now!
```

# Breadth-first Traversal Implementation



D-E-F-H-G-J-I-K-L

# Breadth-first Traversal for BTree



The screenshot shows a code editor window titled "BTree.h". The code implements a breadth-first search (BFS) traversal for a BTree. The traversal starts at the root node and processes all nodes at the current level before moving to the next level. The code uses a std::queue to store pointers to the nodes at each level. The visitor pattern is used to allow external code to visit each node.

```
223 void searchBreadthFirst( const TreeVisitor<T>& aVisitor ) const
224 {
225     std::queue<const BTree<T>*> lQueue;           // traversal queue (pointer required)
226
227     lQueue.push( this );                           // start with root node
228
229     while ( !lQueue.empty() )
230     {
231         const BTree<T>* lFront = lQueue.front();
232
233         if ( !lFront->empty() )
234         {
235             aVisitor.visit( **lFront );
236             lQueue.push( &lFront->left() );
237             lQueue.push( &lFront->right() );
238         }
239
240         lQueue.pop();
241     }
242 }
```

Line: 1 Column: 1 | C++ | Tab Size: 4 | —

```
265 #include "BTree.h"
266
267 void testBFS()
268 {
269     cout << "Test BFS." << endl;
270
271     using StringBTree = BTree<string>;
272
273     string s1( "A" );
274     string s2( "B" );
275     string s3( "C" );
276
277     StringBTree root( "Hello World!" );
278     StringBTree nodeA( s1 );
279     StringBTree nodeB( s2 );
280     StringBTree nodeAA( s3 );
281     StringBTree nodeBB( "D" );
282
283     root.attachLeft( nodeA );
284     root.attachRight( nodeB );
285     const_cast<StringBTree&>(root.left()).attachLeft( nodeAA );
286     const_cast<StringBTree&>(root.right()).attachRight( nodeBB );
287
288     cout << "root:      " << *root << endl;
289     cout << "root->L:    " << *root.left() << endl;
290     cout << "root->R:    " << *root.right() << endl;
291     cout << "root->L->L: " << *root.left().left() << endl;
292     cout << "root->R->R: " << *root.right().right() << endl;
293
294     root.searchBreadthFirst( TreeVisitor<string>() );
295     cout << endl;
296
297     const_cast<StringBTree&>(root.right()).detachRight();
298     const_cast<StringBTree&>(root.left()).detachLeft();
299     root.detachRight();
300     root.detachLeft();
301
302     cout << "All trees are going to be deleted now!" << endl;
303 }
```

```
Kamala:BTree Markus$ ./BTreeTest
Test BFS.
root:      Hello World!
root->L:    A
root->R:    B
root->L->L: C
root->R->R: D
Hello World! A B C D
All trees are going to be deleted now!
```

# M-way Search Tree

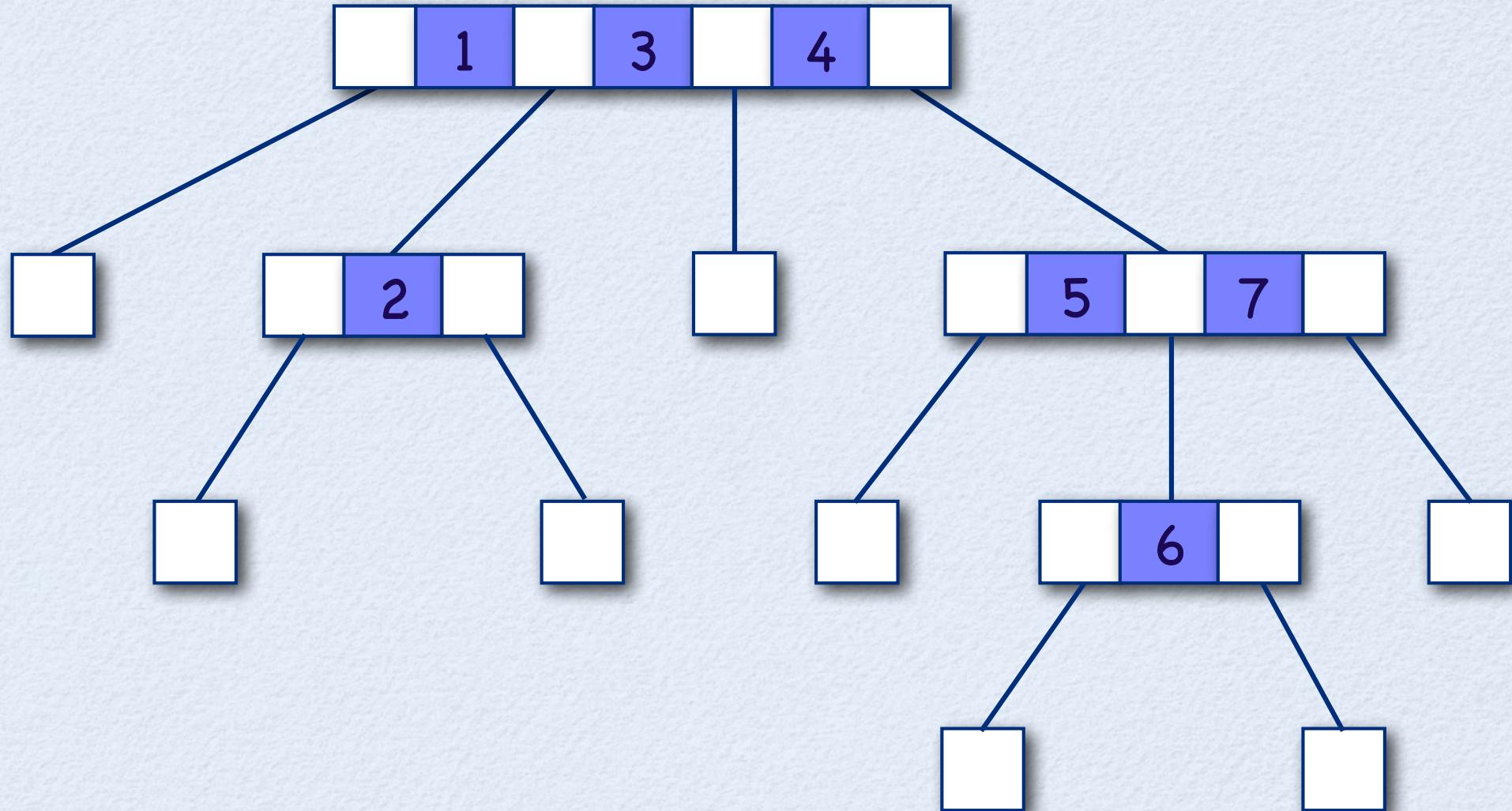
- An M-ary search tree  $T$  is a finite set of nodes with one of the following properties:

- either the set is empty,  $T = \emptyset$ , or
- for  $2 \leq n \leq M$ , the set consists of  $n$  M-ary subtrees  $T_1, T_2, \dots, T_{n-1}, T_n$  and  $n-1$  keys  $k_1, k_2, \dots, k_{n-1}$ .

and the keys and nodes satisfy the data ordering properties:

- The keys in each node are distinct and ordered, i.e.,  $k_i < k_{i+1}$ , for  $1 \leq i \leq n-1$ .
- All the keys contained in subtree  $T_{i-1}$  are less than  $k_i$ , i.e.,  $\forall k \in T_{i-1}: k < k_i$ , for  $1 \leq i \leq n-1$ . The tree  $T_{i-1}$  is called left subtree with respect the key  $k_i$ .
- All the keys contained in subtree  $T_i$  are greater than  $k_i$ , i.e.,  $\forall k \in T_i: k > k_i$ , for  $1 \leq i \leq n-1$ . The tree  $T_i$  is called right subtree with respect the key  $k_i$ .

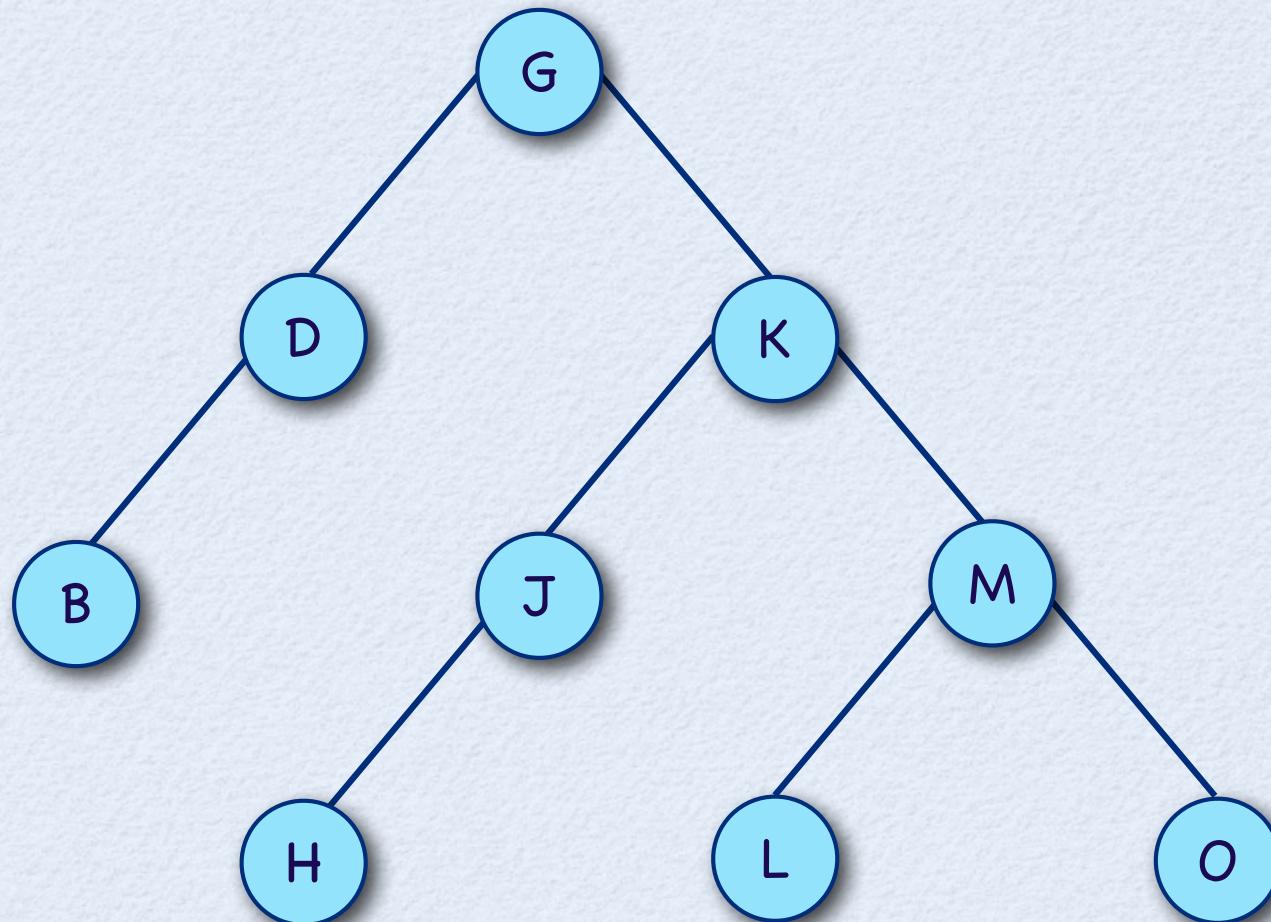
# A 4-way Search Tree



# 2-way Search Tree

- A 2-ary (binary) search tree  $T$  is a finite set of nodes with one of the following properties:
  - either the set is empty,  $T = \emptyset$ , or
  - the set consists of one key,  $r$ , and exactly 2 binary subtrees  $T_L$  and  $T_R$  such that following properties are satisfied:
    - All the keys in the left subtree,  $T_L$ , are less than  $r$ , i.e.,  $\forall k \in T_L: k < r$ .
    - All the keys contained in the right subtree,  $T_R$ , are greater than  $r$ , i.e.,  $\forall k \in T_R: k > r$ .

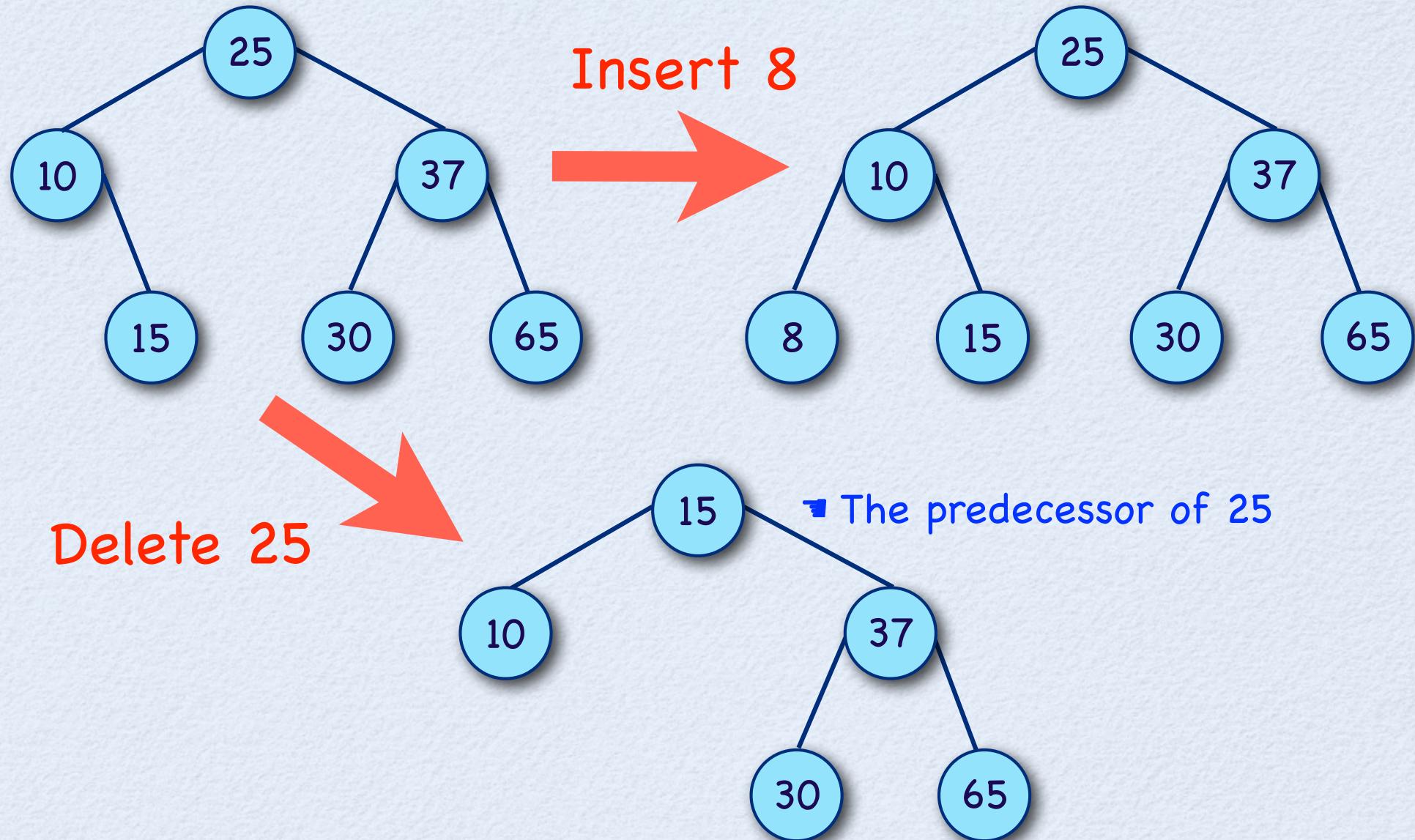
# A Binary Search Tree Example



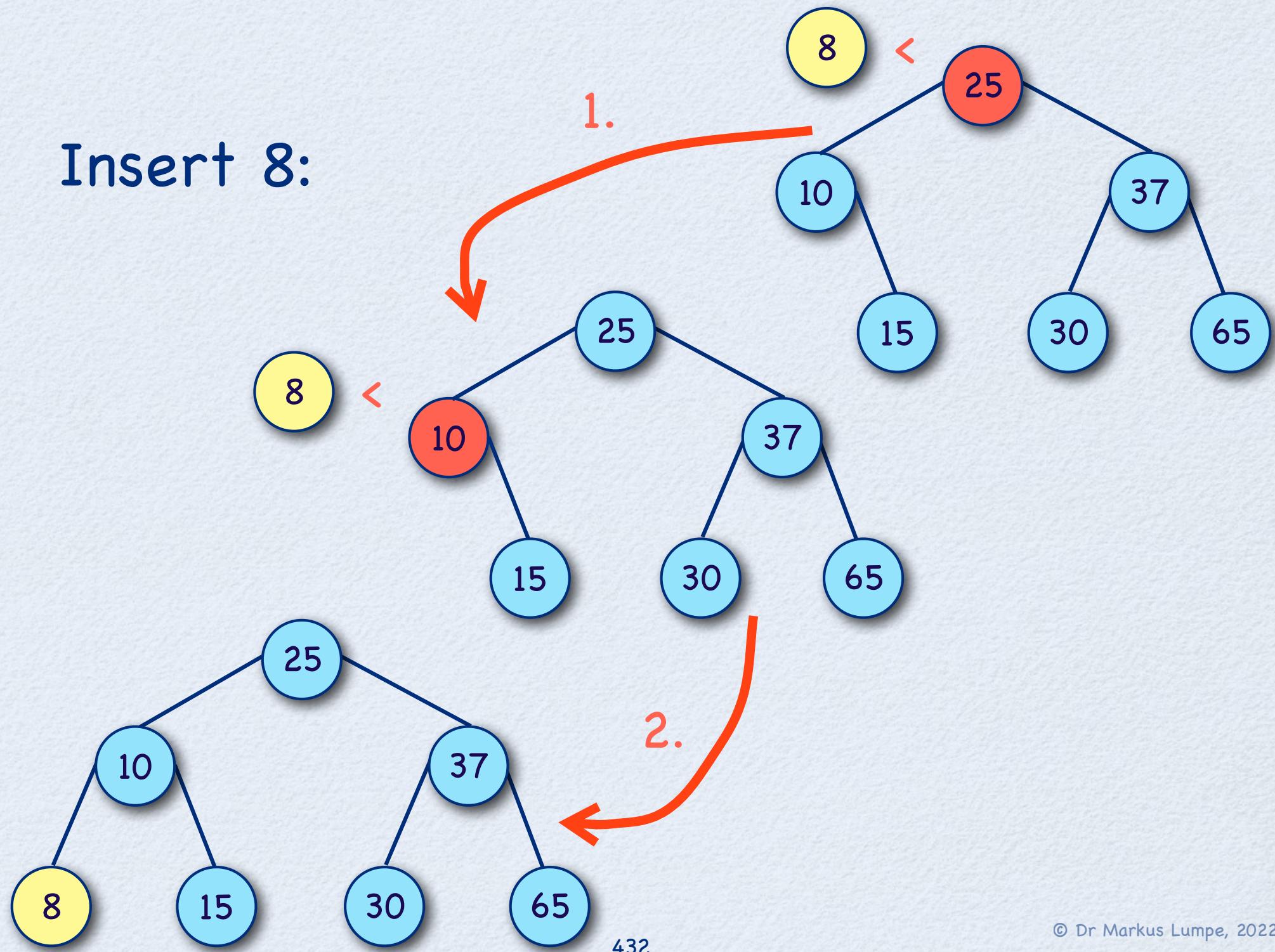
# Traversing a Binary Search Tree

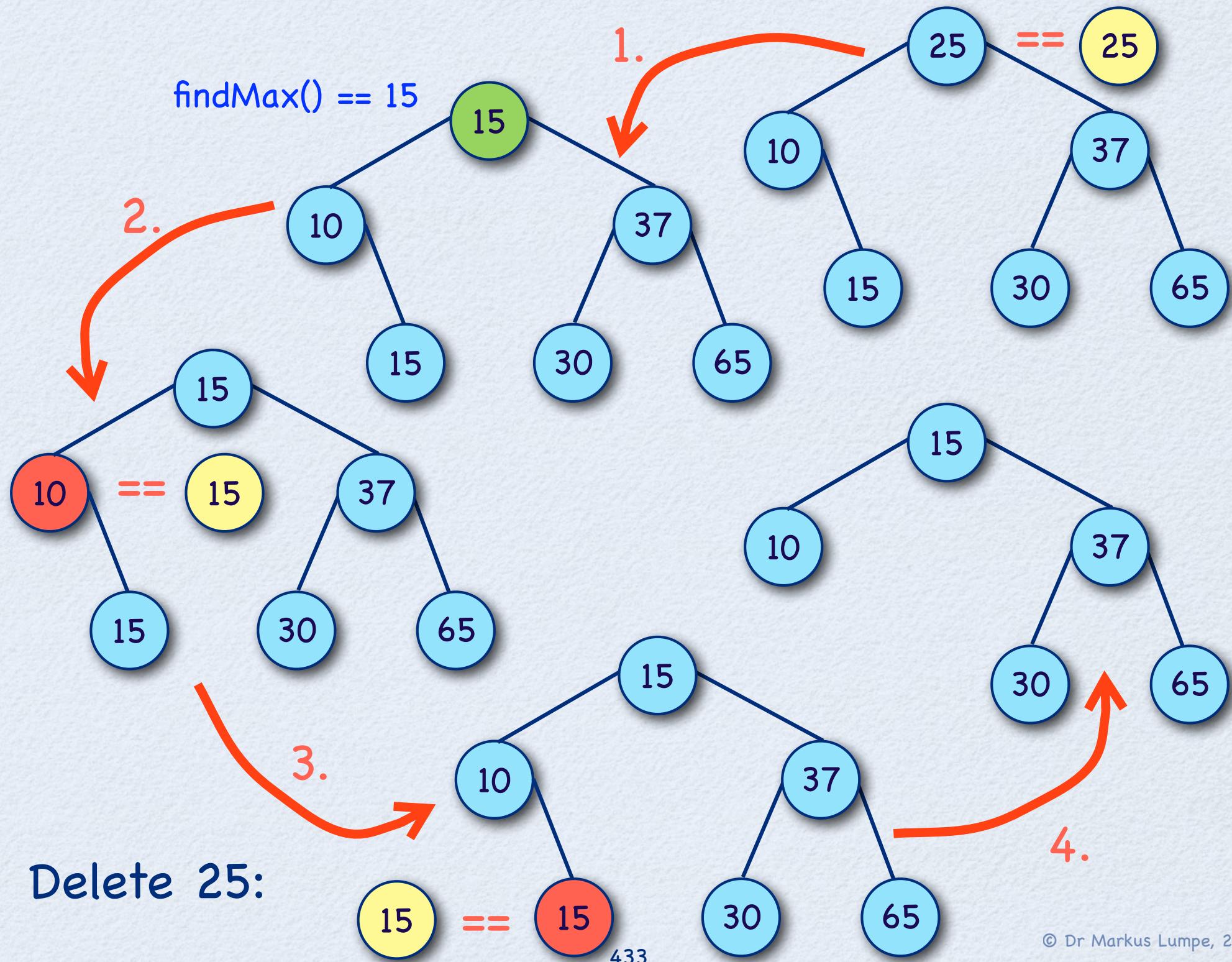
- Binary Tree Search:
  - Traverse the left subtree, and then
  - Visit the root, and then
  - Traverse the right subtree.
- We use in-order traversal to search for a given key in an M-ary search tree.

# Binary Search Tree Operations



## Insert 8:





We define the representation of a binary search tree in class BNode.

```
9 #include "BNode.h"
10 #include "TreeVisitor.h"
11
12 template<typename T>
13 class BinarySearchTree
14 {
15 private:
16     BNode<T>* fRoot;
17
18 public:
19
20     BinarySearchTree();
21     ~BinarySearchTree();
22
23     bool empty() const;
24     size_t height() const;
25
26     bool insert( const T& aKey );
27     bool remove( const T& aKey );
28
29     void traverseDepthFirst( const TreeVisitor<T>& aVisitor ) const;
30 }
```

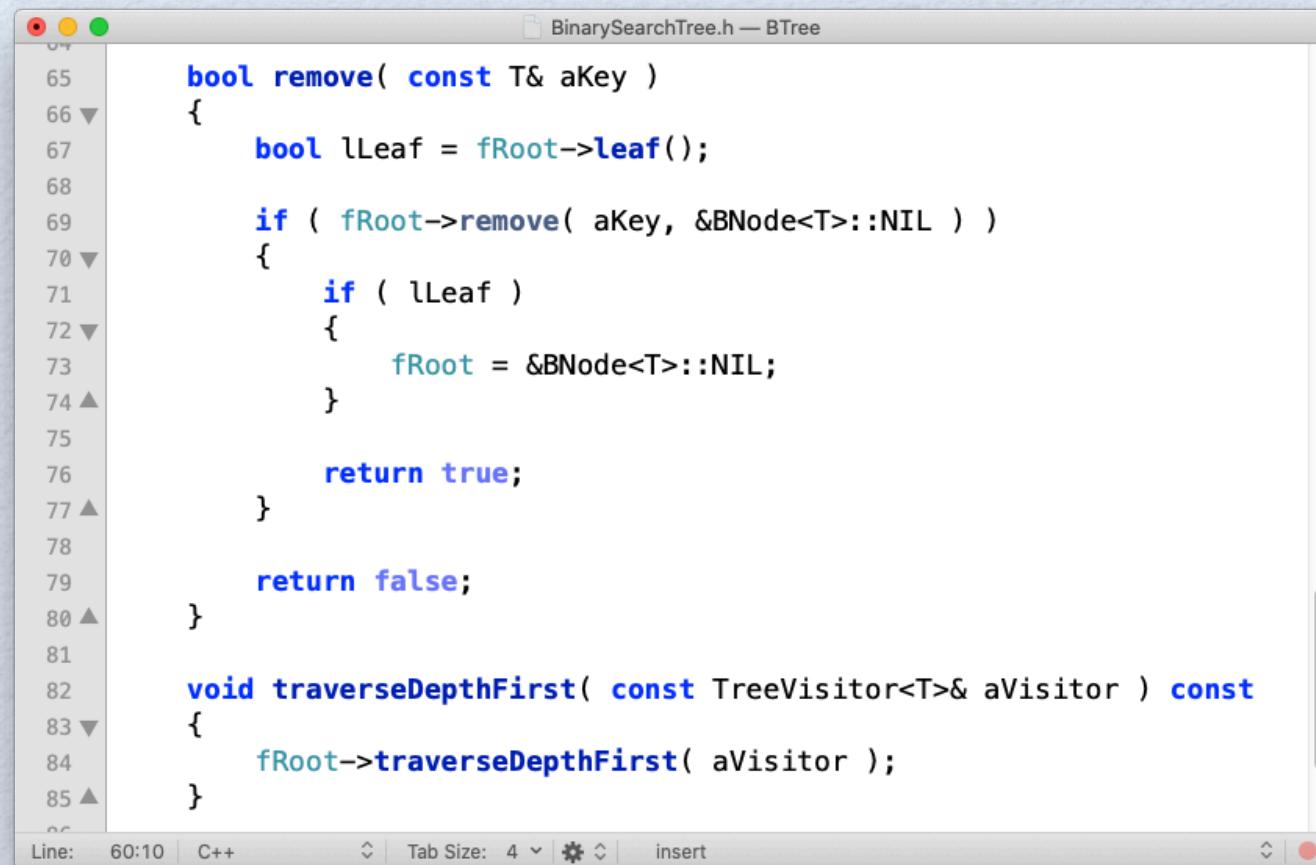
```
8 #include <stdexcept>
9
10 template<typename S>
11 struct BNode
12 {
13     S key;
14     BNode<S>* left;
15     BNode<S>* right;
16
17     static BNode<S> NIL;
18
19     ...
20 };
21
22 template<typename S>
23 BNode<S> BNode<S>::NIL;
24
```

The screenshot shows a window titled "BNode.h" containing C++ code. The code defines a function `bool insert( const S& aKey )` for a class `BNode<S>`. The function inserts a key into a binary search tree. It first checks if the current node `x` is empty. If it is, it creates a new node `z` and returns false because insertion failed at NIL. If `x` is not empty, it compares the key to be inserted with the current node's key. If they are equal, it returns false indicating a duplicate key error. Otherwise, it sets the left child of `x` to `z` if the key is less than `x`'s key, or the right child if it is greater. Finally, it returns true indicating the insertion was successful.

```
63     bool insert( const S& aKey )
64 {
65     BNode<S>* x = this;
66     BNode<S>* y = &BNode<S>::NIL;
67
68     while ( !x->empty() )
69     {
70         y = x;
71
72         if ( aKey == x->key )
73         {
74             return false;           // duplicate key - error
75         }
76
77         x = aKey < x->key ? x->left : x->right;
78     }
79
80     BNode<S>* z = new BNode<S>( aKey );
81
82     if ( y->empty() )
83     {
84         return false;           // insert failed (NIL)
85     }
86     else
87     {
88         if ( aKey < y->key )
89         {
90             y->left = z;
91         }
92         else
93         {
94             y->right = z;
95         }
96     }
97
98     return true;                // insert succeeded
99 }
100
```

Line: 129 Column: 1 C++ Tab Size: 4 remove

# Remove and Depth-first Traversal



The screenshot shows a code editor window titled "BinarySearchTree.h — BTTree". The code implements a binary search tree with templated nodes. It includes methods for removing a key from the tree and traversing it depth-first.

```
65     bool remove( const T& aKey )
66 {
67     bool lLeaf = fRoot->leaf();
68
69     if ( fRoot->remove( aKey, &BNode<T>::NIL ) )
70     {
71         if ( lLeaf )
72         {
73             fRoot = &BNode<T>::NIL;
74         }
75
76         return true;
77     }
78
79     return false;
80 }
81
82 void traverseDepthFirst( const TreeVisitor<T>& aVisitor ) const
83 {
84     fRoot->traverseDepthFirst( aVisitor );
85 }
```

Line: 60:10 | C++ | Tab Size: 4 | insert

- The class `BSTree` defines an adapter for `BSTNode`.
- The operation `insert` can be defined as a simple while-loop over `BSTNodes` from the root node.
- The operations `remove` and `traverseDepthFirst` use recursion to explore `BSTNodes`. In the beginning, both start with the root node.

```
121     bool remove( const S& aKey, BNode<S>* aParent )  
122     {  
123         BNode<S>* x = this;  
124         BNode<S>* y = aParent;  
125  
126         while ( !x->empty() )  
127         {  
128             if ( aKey == x->key )  
129                 break;  
130  
131             y = x;  
132             x = aKey < x->key ? x->left : x->right;  
133         }  
134  
135         if ( x->empty() )  
136             return false; // delete failed  
137  
138         if ( !x->left->empty() )  
139         {  
140             const S& lKey = x->left->findMax(); // find max to left  
141             x->key = lKey;  
142             x->left->remove( lKey, x );  
143         }  
144         else  
145         {  
146             if ( !x->right->empty() )  
147             {  
148                 const S& lKey = x->right->findMin(); // find min to right  
149                 x->key = lKey;  
150                 x->right->remove( lKey, x );  
151             }  
152             else  
153             {  
154                 if ( y->left == x )  
155                     y->left = &NIL;  
156                 else  
157                     y->right = &NIL;  
158  
159                 delete x; // free deleted node  
160             }  
161         }  
162  
163         return true;  
164     }
```

Line: 129 Column: 1

C++

Tab Size: 4

remove

# Binary Search Tree Test

```
>Main.cpp
138 #include "BinarySearchTree.h"
139
140 void testVisitor()
141 {
142     BinarySearchTree<int> lTree;
143
144     lTree.insert( 25 );
145     lTree.insert( 10 );
146     lTree.insert( 15 );
147     lTree.insert( 37 );
148     lTree.insert( 30 );
149     lTree.insert( 65 );
150
151     lTree.traverseDepthFirst( PreOrderVisitor<int>() );
152     cout << endl;
153
154     lTree.insert( 8 );
155
156     lTree.traverseDepthFirst( PreOrderVisitor<int>() );
157     cout << endl;
158
159     lTree.remove( 25 );
160
161     lTree.traverseDepthFirst( PreOrderVisitor<int>() );
162     cout << endl;
163 }
```

```
COS30008
Kamala:COS30008 Markus$ ./BSTreeTest
25 10 15 37 30 65
25 10 8 15 37 30 65
15 10 8 37 30 65
Kamala:COS30008 Markus$
```

# Other Tree Variants

- Rose Trees (directories)
- Expression Trees (internal program representation)
- Multi-rooted trees (C++: multiple inheritance)
- Red-Black Trees (directories in compound documents, `java.util.TreeMap`)
- AVL Trees (Adelson-Velskii & Landis balanced BTrees)

# AVL vs. Red-Black Trees

- Both AVL trees and Red-Black trees are self-balancing binary search trees. However, the operations to balance the trees are different.
- AVL and Red-Black trees have different height limits.  
For a tree of size  $n$ :
  - An AVL tree's height is limited to  $1.44\log_2(n)$ .
  - A Red-Black tree's height is limited to  $2\log_2(n)$ .
- The AVL tree is more rigidly balanced than Red-Black trees, leading to slower insertion and removal but faster retrieval.

# Algorithmic Patterns

## Overview

- Algorithm Efficiency
- Solution Strategies

## References

- Bruno R. Preiss: Data Structures and Algorithms with Object-Oriented Design Patterns in C++. John Wiley & Sons, Inc. (1999)
- Russ Miller and Laurence Boxer: Algorithms Sequential & Parallel - A Unified Approach. 2nd Edition. Charles River Media (2005)
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein: Introduction to Algorithms. 3rd Edition. The MIT Press (2009)

# The “Best” Algorithm

- There are usually multiple algorithms to solve any particular problem.
- The notion of the “best” algorithm may depend on many different criteria:
  - Structure, composition, and readability
  - Time required to implement
  - Extensibility
  - Space requirements
  - Time requirements

# Time Analysis

- Example:
  - Algorithm A runs 2 minutes and algorithm B takes 1 minutes and 45 second to complete for the same input.
- Is B “better” than A?  $\Rightarrow$  Not necessarily!:
  - We have tested A and B only on one (fixed) input set. Another input set might result in a different runtime behavior.
  - Algorithm A might have been interrupted by another process.
  - Algorithm B might have been run on a different computer.
- A reasonable time and space approximation should be machine-independent.

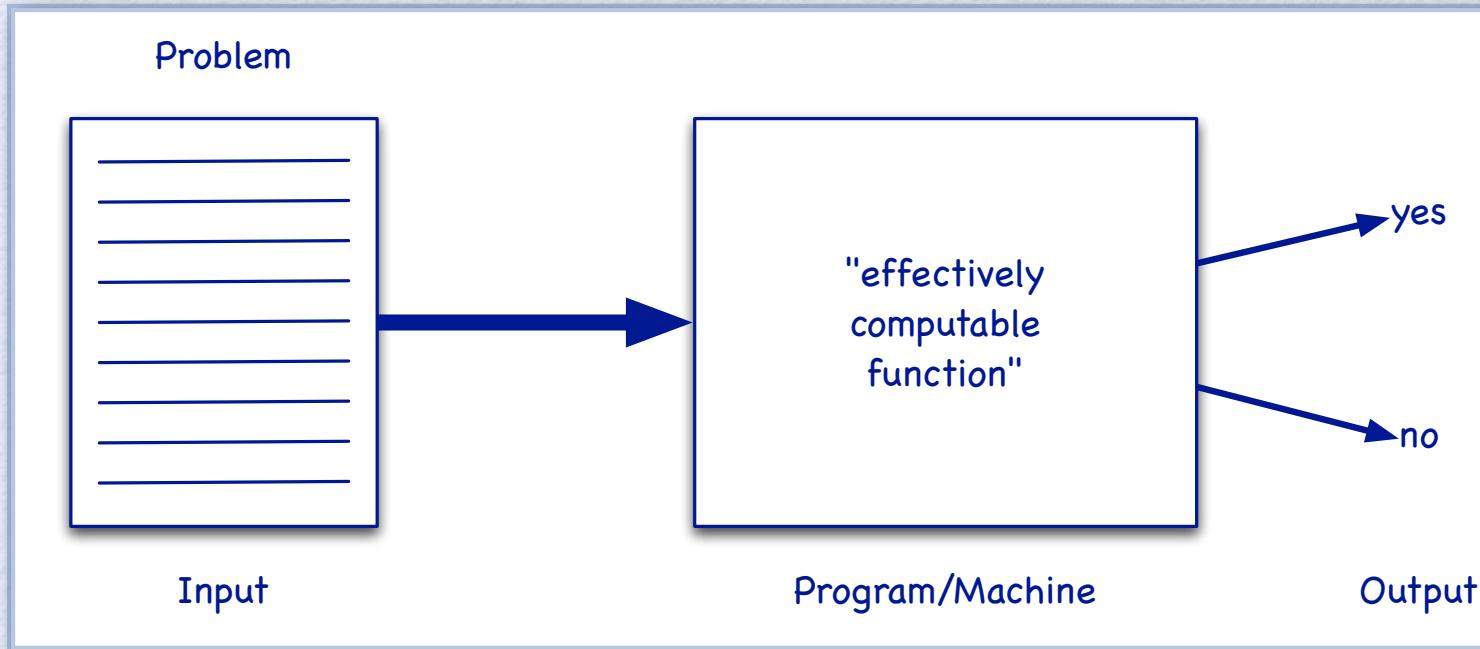
# Running Time in Terms of Input Size

- What is the one of most interesting aspect about algorithms?
  - How many seconds does it take to complete for a particular input size  $n$ ?
  - How does an increase of size  $n$  effect the running time?
- An algorithm requires
  - Constant time if the running time remains the same as the size  $n$  changes,
  - Linear time if the running time increases proportionally to size  $n$ .
  - Exponential time if the running time increases exponentially with respect to size  $n$ .

# What is computable?

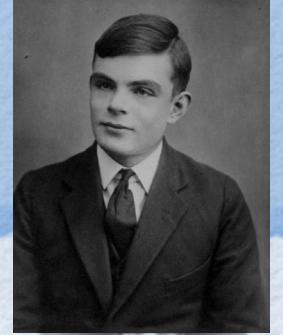
- Computation is usually modeled as a mapping from inputs to outputs, carried out by a “formal machine”, or program, which processes its input in a sequence of steps.
- An “effectively computable” function is one that can be computed in a finite amount of time using finite resources.

# Abstract Machine Model

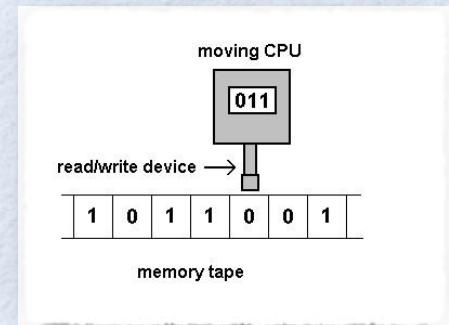


- Church's Thesis: It is not possible to build a machine that is more powerful than a Turing machine.

# Turing Machine



- A Turing machine is an abstract representation of a computing device. It consists of a read/write head that scans a (possibly infinite) one-dimensional (bi-directional) tape divided into squares, each of which is inscribed with a 0 or 1 (possibly more symbols).
- Computation begins with the machine, in a given "state", scanning a square. It erases what it finds there, prints a 0 or 1, moves to an adjacent square, and goes into a new state until it moves into HALT.
- This behavior is completely determined by three parameters:
  - the state the machine is in,
  - the number on the square it is scanning, and
  - a table of instructions.



# Formal Definition of a Turing Machine

- A Turing machine is a septuple  $(Q, \Gamma, \gamma, \Sigma, q_0, F, \sigma)$  with:
  - a set  $Q = \{q_0, q_1, \dots\}$  of states,
  - a set  $\Gamma$  is a finite, non-empty set of tape symbols,
  - a blank symbol  $\gamma \in \Gamma$  (the only symbol to occur infinitely often)
  - a set  $\Sigma \subseteq \Gamma \setminus \{\gamma\}$  of input symbols (to appear as initial tape contents),
  - a designated start state  $q_0$ .
  - a subset  $F \subseteq Q$  called the accepting states,
  - a partial function  $\sigma : (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times \{R, L\}$  called the transition function.
- A Turing machine halts if it enters a state in  $F$  or if  $\sigma$  is undefined for the current state and tape symbol.

# Addition on a Turing Machine

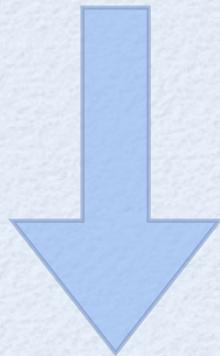
- $\Sigma = \{"1", "+", "\ "\}$  - the tape symbols

$Q \setminus \Gamma$	" "	"1"	"+"
1	1/" ",R	2/"1",R	
2		2/"1",R	3/"+",R
3		4/"+",L	
4			5/"1",R
5	6/" ",L	4/"+",L	5/"+",R
6			HALT/" ",R

# 5+3 on a Turing Machine



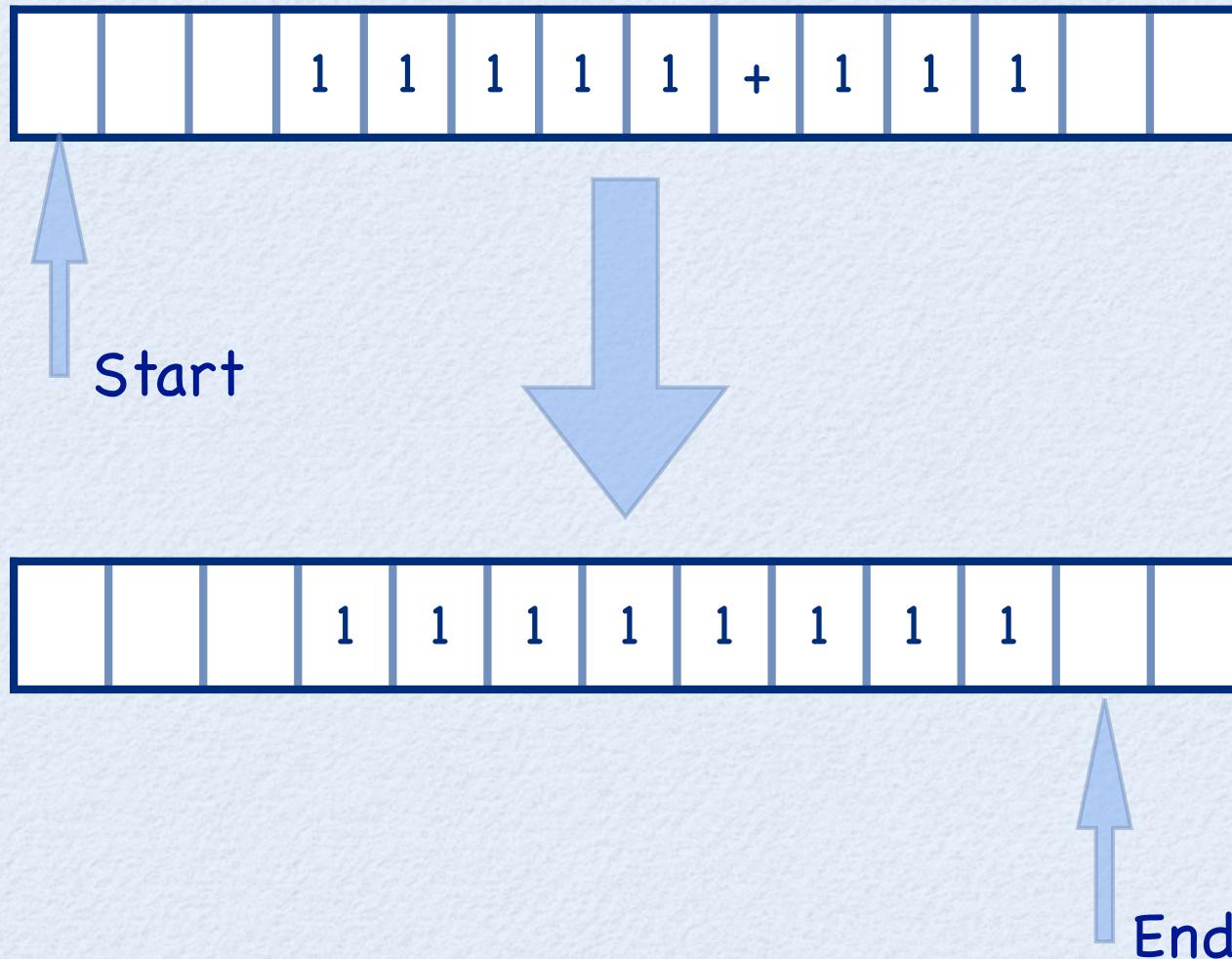
Start



End

$Q \setminus \Gamma$	" "	"1"	"+"
1	1/" ",R	2/"1",R	
2		2/"1",R	3/"+",R
3			4/"+",L
4			5/"1",R
5	6/" ",L	4/"+",L	5/"+",R
6			HALT/" ",R

# 5+3 on a Turing Machine (No Move Allowed)



State/Input	" "	"1"	"0"	"+"
0	-/R/-	-/R/1		
1		-/R/-		"0"/R/2
2	-/L/3	-/R/4		
3			" "/-/stop	
4	-/L/5	-/R/4		
5		-/L/5	" "/-/stop	

# The Ackermann Function

- The Ackermann function is a simple example of a computable function that grows much faster than polynomials or exponentials even for small inputs.
- The Ackermann function is defined recursively for non-negative integers  $m$  and  $n$  as follows:

$$A(0, n) = n + 1$$

$$A(m+1, 0) = A(m, 1)$$

$$A(m+1, n+1) = A(m, A(m+1, n))$$

# Ackermann Function Value Table

$A(m,n)$	$n = 0$	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$
$m = 0$	1	2	3	4	5	6
$m = 1$	2	3	4	5	6	7
$m = 2$	3	5	7	9	11	13
$m = 3$	5	13	29	61	125	253
$m = 4$	13	65533	$2^{65536}-3$	$2^{2^{65536}-3}$	$A(3,A(4,3))$	$A(3,A(4,4))$
$m = 5$	65533	$A(4,65533)$	$A(4,A(5,1))$	$A(4,A(5,2))$	$A(4,A(5,3))$	$A(4,A(5,4))$
$m = 6$	$A(4,65533)$	$A(5,A(6,0))$	$A(5,A(6,1))$	$A(5,A(6,2))$	$A(5,A(6,3))$	$A(5,A(6,4))$

# A(4,2) – Number with 19,729 Digits

A(4,2) =

200352993040684646497907235156025575044782547556975141926501697371089405955631145  
3089506130880933348101038234342907263181822949382118812668869506364761547029165041  
871916351587966347219442930927982084309104855990570159318959639524863372367203002  
9169695921561087649488892540908059114570376752085002066715637023661263597471448071  
117748158809141357427209671901518362825606180914588526998261414250301233911082736038  
437678764490432059603791244909057075603140350761625624760318637931264847037437829  
549756137709816046144133086921181024859591523801953310302921628001605686701056516467  
505680387415294638422448452925373614425336143737290883037946012747249584148649159  
30647252015155693922628180691650796381064132275307267143998158508811292628901134237  
7827055674210800700652839633221550778312142885516755540733451072131124273995629827  
1976915005488390522380435704584819795639315785351001899200002414196370681355984046  
4039472194016069517690156119726982337890017

...

13366337713784344161940531214452918551801365755586676150193730296919320761200092550  
6508158327550849934076879725236998702356793102680413674571895664143185267905471716  
996299036301554564509004480278905570196832831363071899769915316667920895876857229  
06009154729196363816735966739599757103260155719202373485805211281174586100651525988  
8384311451189488055212914577569914657753004138471712457796504817585639507289533753  
9755822087777506072339445587895905719156733

# Halting Problem

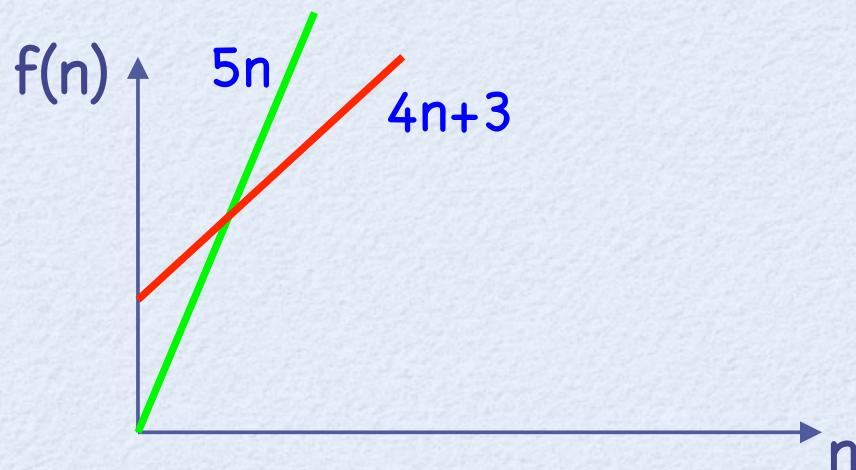
- A problem that cannot be solved by any machine in finite time (or any equivalent formalism) is called uncomputable.
  - An uncomputable problem cannot be solved by any real computer.

## The Halting Problem:

- Given an arbitrary machine and its input, will the machine eventually halt?
- The Halting Problem is provably uncomputable – which means that it cannot be solved in practice.

# The Big-Oh

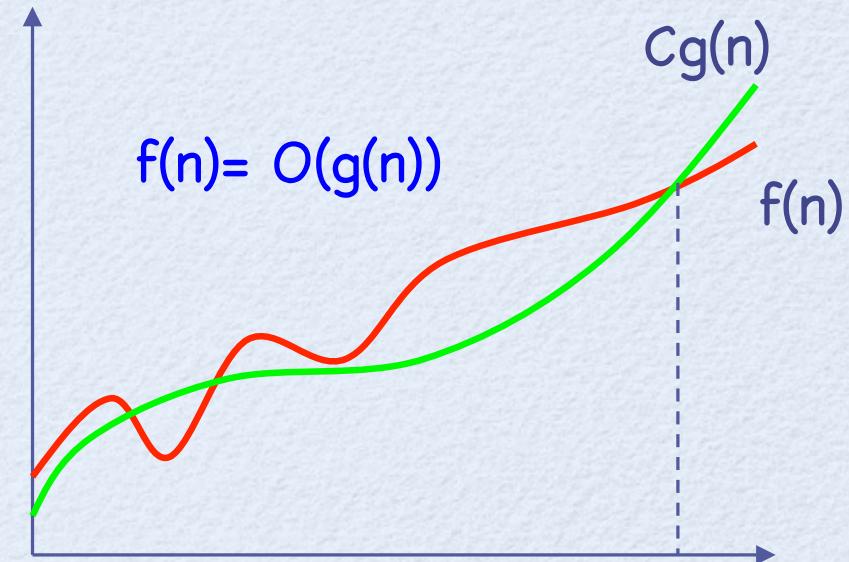
- An algorithm  $f(n)$  is  $O(g(n))$ , read “has order  $g(n)$ ”, if there exist constants  $C > 0$  and integer  $n_0$  such that the algorithm  $f(n)$  requires at most  $C*g(n)$  steps for all input sizes  $n \geq n_0$ .
- Example:
  - Algorithm A takes  $4n + 3$  steps, that is, it is  $O(n)$ .
  - Choose  $C = 5$  and  $n_0 = 4$ , then  $4n + 3 < 5n$  for all  $n \geq 4$ .



# Facts About Big-Oh

- Big-Oh focuses on growth rate as running time of input size approaches infinity ( $n \rightarrow \infty$ ).
- Big-Oh does not say anything about the running time on small input.
- The function  $g(n)$  in  $O(g(n))$  is a simple function for comparison of different algorithms:

- $1, n, \log n, n \log n, n^2, 2^n, \dots$



# On Running Time Estimation

- Big-Oh ignores lower-order terms:
  - Lower order terms in the computation steps count functions that are concerned with initialization, secondary arguments, etc.
- Big-Oh does not consider the multiplier in higher order terms:
  - These terms are machine-dependent.

# Performance Analysis

- Best-Case (Lower Bound):
  - The search for a given element A in an array of size n can be O(1), if the element is the first. (Also applies to binary search trees.)
- Worst-Case (Upper Bound):
  - The search for a given element A in an array of size n is O(n), if the element is the last in the array. (For binary search trees, it is also O(n), if the element is the last in a totally unbalanced binary search tree, but O(log n) for a balanced search tree.)
- Average-Case:
  - The search for a given element A in an array of size n takes on average  $n/2$ , whereas the lookup in a binary search tree is O(log n).

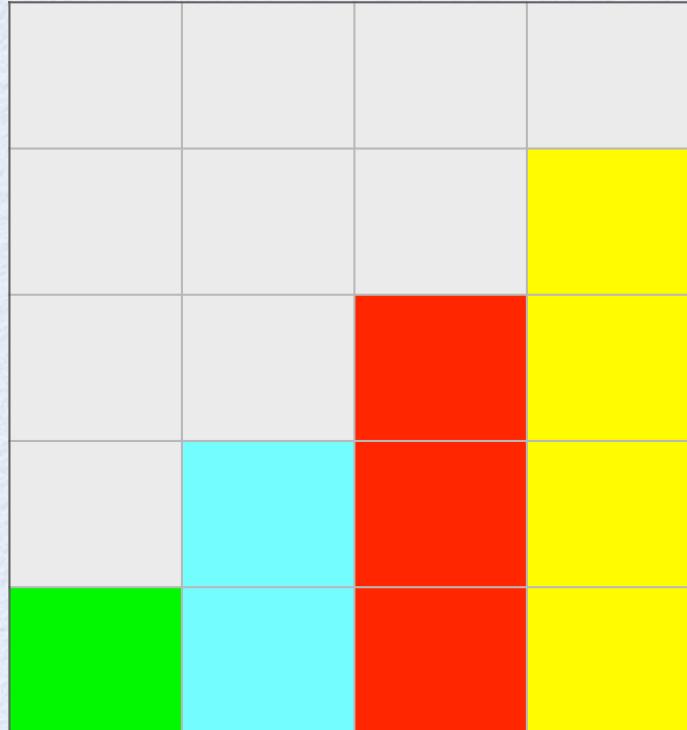
# Constant Time

- Algorithm A requires 2,000,000 steps:  $O(1)$
- As a young boy, the later mathematician Carl Friedrich Gauss was asked by his teacher to add up the first hundred numbers, in order to keep him quiet for a while. As we know today, this did not work out, since:

$$\text{sum}(n) = n(n+1)/2$$

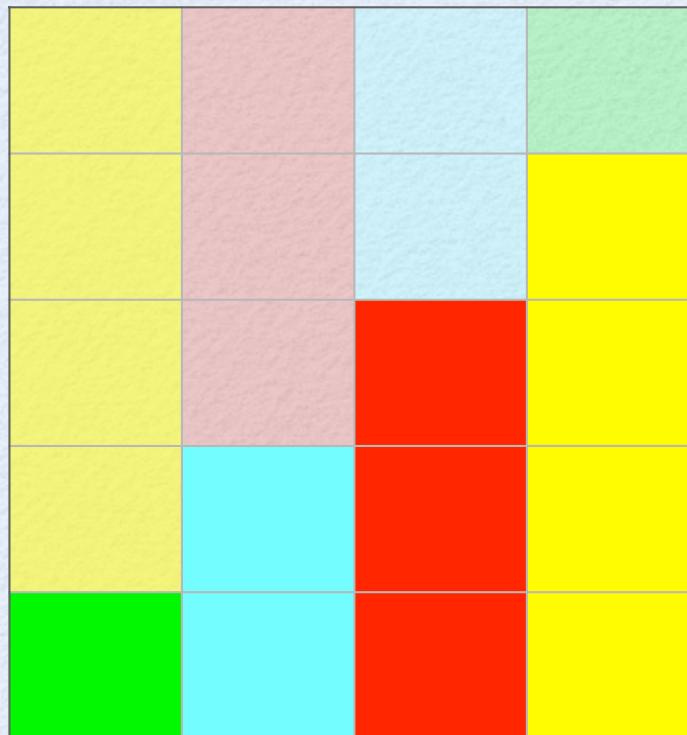
which is  $O(1)$ .

# How does Gauss's formula work?



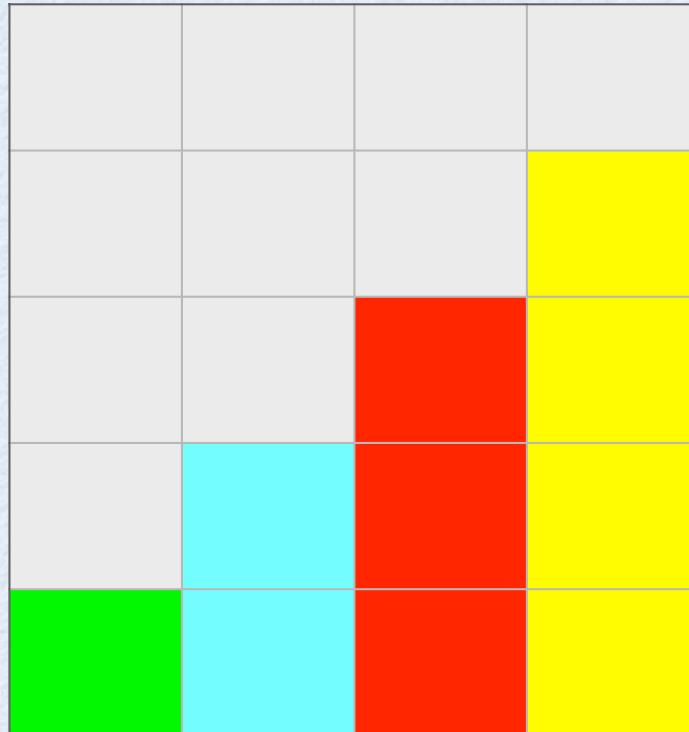
$$1 + 2 + 3 + 4 = ?$$

# Let's look at the whole rectangle.



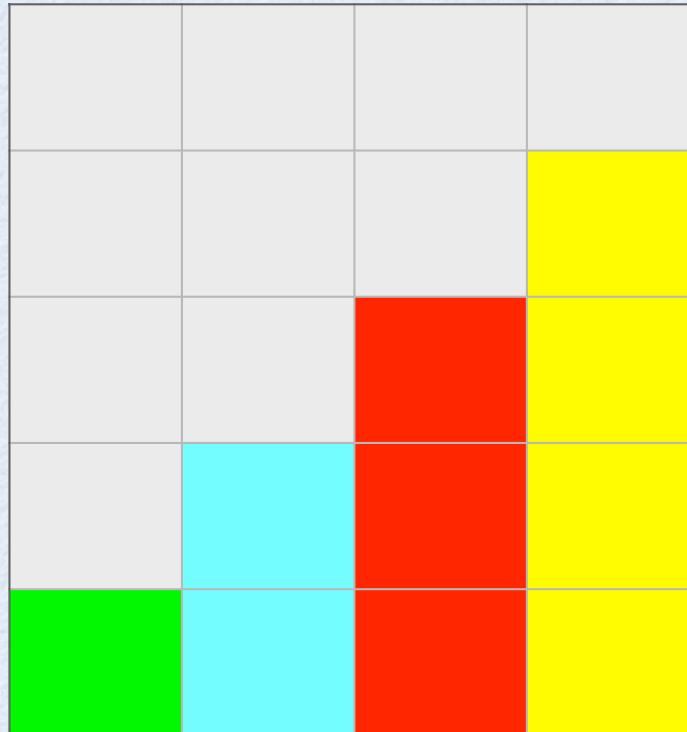
$$2 \times (1 + 2 + 3 + 4) = 4 \times 5$$

# So, we have ...



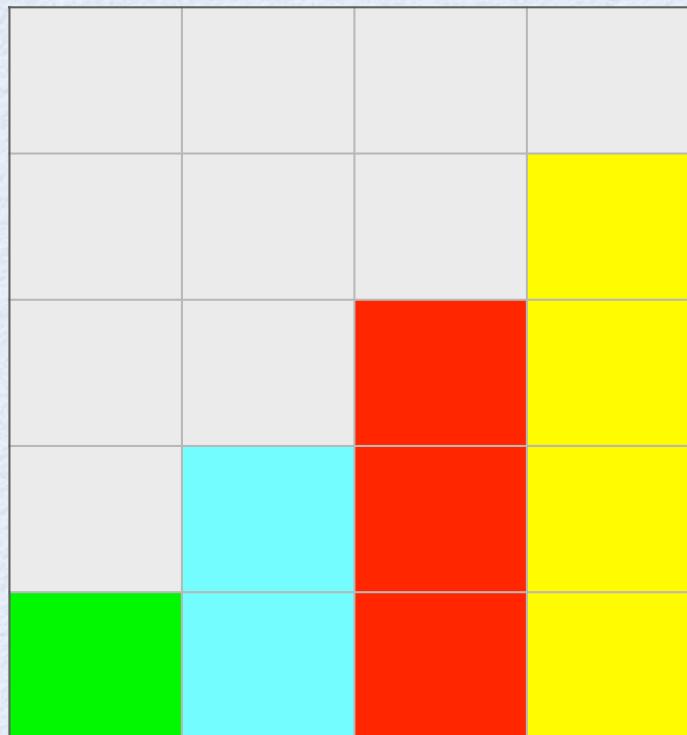
$$1 + 2 + 3 + 4 = (4 \times 5) / 2$$

# We generalize ...



$$1 + 2 + \dots + n = n \times (n + 1) / 2$$

# We focus on the growth rate $n \rightarrow \infty$



$n \times (n + 1) / 2$  is  $O(1)$

# Polynomial Time

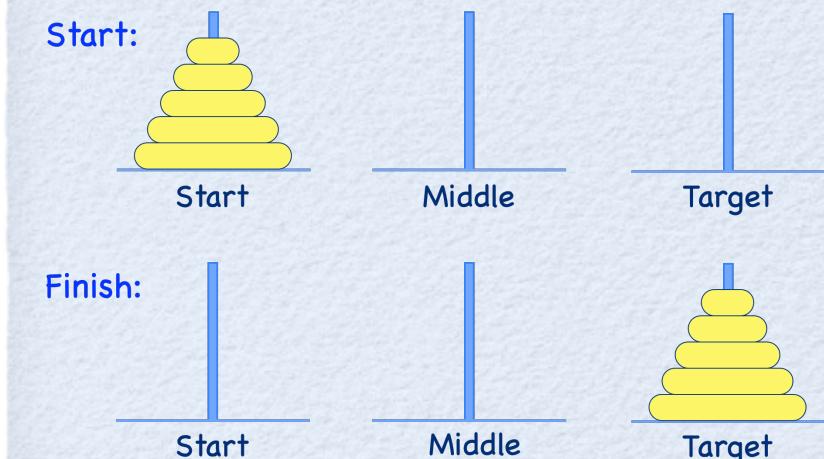
- $4n^2 + 3n + 1$ :
  - Ignore lower-order terms:  $4n^2$
  - Ignore constant coefficients:  $O(n^2)$
- $a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 = O(n^k)$

# Logarithmic & Exponential Functions

- $\log_{10} n = \log_2 n / \log_2 10 = O(\log n)$ :
  - Ignore base:  $\log_{10} n$
- $345n^{4536} + n(\log n) + 2^n = O(2^n)$ :
  - Ignore lower-order terms  $345n^{4536}$  and  $n(\log n)$

# Towers of Hanoi: The Recursive Procedure

```
procedure TON( n, S, T, M )
begin
  if n > 0 then
    TON( n-1, S, M, T )
    d(T) ← d(S)
    TON( n-1, M, T, S )
  end
```



# Towers of Hanoi Complexity

- The body of TON requires  $T(n) = 2T(n-1) + 1$  operations, for an input size  $n$ .
- We have
  - $T(n-1) = 2T(n-2) + 1 = 2*(2T(n-3) + 1) + 1 = 2^2T(n-3) + 2^1 + 1$
  - $T(n-2) = 2T(n-3) + 1 = 2*(2T(n-4) + 1) + 1$
  - ...
  - $T(2) = 2T(1) + 1 = 2(2^1 + 1) + 1 = 2^2 + 2^1 + 1$
  - $T(1) = 2T(0) + 1 = 2^1 + 1$
  - $T(0) = 1$
- Solving the recursive equations, we obtain
  - $T(n) = 2^n + 2^{(n-1)} + \dots + 2^1 + 1 = 2^{(n+1)} - 1 = O(2^n)$

# A Simple Example

$$\sum_{i=1}^n i^3$$

```
Sum13.cpp
2 int sum( int n )
3 {
4     int Result = 0;
5
6     for ( int i = 1; i <= n; i++ )
7         Result += i * i * i;
8
9     return Result;
10}
```

Line: 13 Column: 1    C++    Tab Size: 4    sum

$$\Rightarrow 2n + 8 = O(n)$$

- We count the computation units in each line:

- line 2: Zero, the declaration requires no time.
- line 4 & 9: One time unit each.
- line 6: Hidden costs for initializing i, testing  $i \leq n$ , and incrementing i; 1 time unit initialization,  $n+1$  time units for all tests, and  $n$  time units for all increments:  $2n + 2$ .
- line 7: 4 time units, two multiplications, one addition, and one assignment.

# Ranking of Big-Oh

- Fastest:

$O(1)$

$O(\log n)$

$O(n)$

$O(n \log n)$

$O(n^2)$

$O(n^2 \log n)$

$O(n^3)$

$O(2^n)$

- Slowest:

$O(n!)$



# General Rules

# For Loops

```
for ( initializer; condition; expression )  
    statement
```

- The running time for a for-loop is at most the running time of the statement inside the for loop times the number of iterations.
- Let  $C$  be the running time of statement. Then a for-loop has a running time of at most  $Cn$  or  $O(n)$ .
- For loops have a linear running time.

# Nested For Loops

```
for ( initializer1; condition1; expression1 )  
  for ( initializer2; condition2; expression2 )  
    statement
```

- The running time of a nested for-loop is at most the running time of statement multiplied by the product of the sizes of all the for-loops.
- Let  $C$  be the running time of statement. Then a nested for-loop has a running time of at most  $Cn^*n$  or  $O(n^2)$ .
- Nested for-loops have quadratic running time. Any additional nesting level adds one factor. A  $k$ -nested for-loop requires polynomial time or  $O(n^k)$ .

# Consecutive Statements

```
statement1;  
statement2;  
...  
statementn;
```

- The running time for consecutive statements is the sum of each statement.
- Let  $m_i$  be the running time for each statement. Then consecutive statements have a running time of at most  $m_1 + m_2 + \dots + m_n$  or  $O(m)$  where  $m = \max(m_1, m_2, \dots, m_n)$ .

# If-Then-Else Branching

```
if ( condition )
    true-statement
else
    false-statement
```

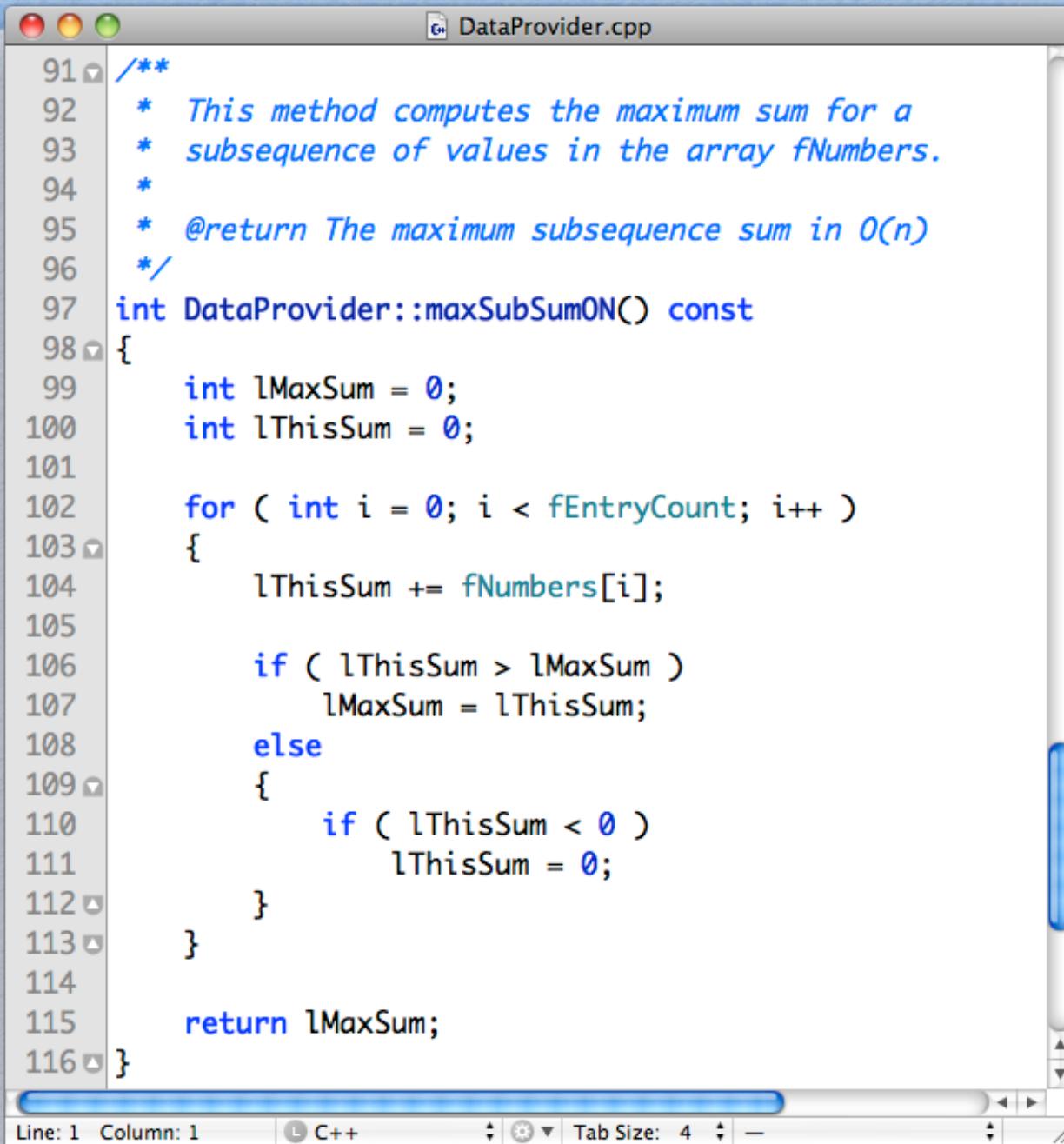
- The running time for an if-then-else statement is at most the running time of the condition plus the larger of the running times of the true-statement and the false-statement.
- This can be an overestimate in some cases, but it is never an underestimate.

# Maximum Subsequence Sum Problem in $O(n^3)$

```
DataProvider.cpp
66 /**
67 * This method computes the maximum sum for a
68 * subsequence of values in the array fNumbers.
69 *
70 * @return The maximum subsequence sum in O(n^3)
71 */
72 int DataProvider::maxSubSumON3() const
73 {
74     int lMaxSum = 0;
75
76     for (int i = 0; i < fEntryCount; i++)
77         for (int j = i; j < fEntryCount; j++)
78     {
79         int lThisSum = 0;
80
81         for (int k = i; k <= j; k++)
82             lThisSum += fNumbers[k];
83
84         if (lThisSum > lMaxSum)
85             lMaxSum = lThisSum;
86     }
87
88     return lMaxSum;
89 }
```

$$\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i}^j 3$$

# Maximum Subsequence Sum Problem in O(n)

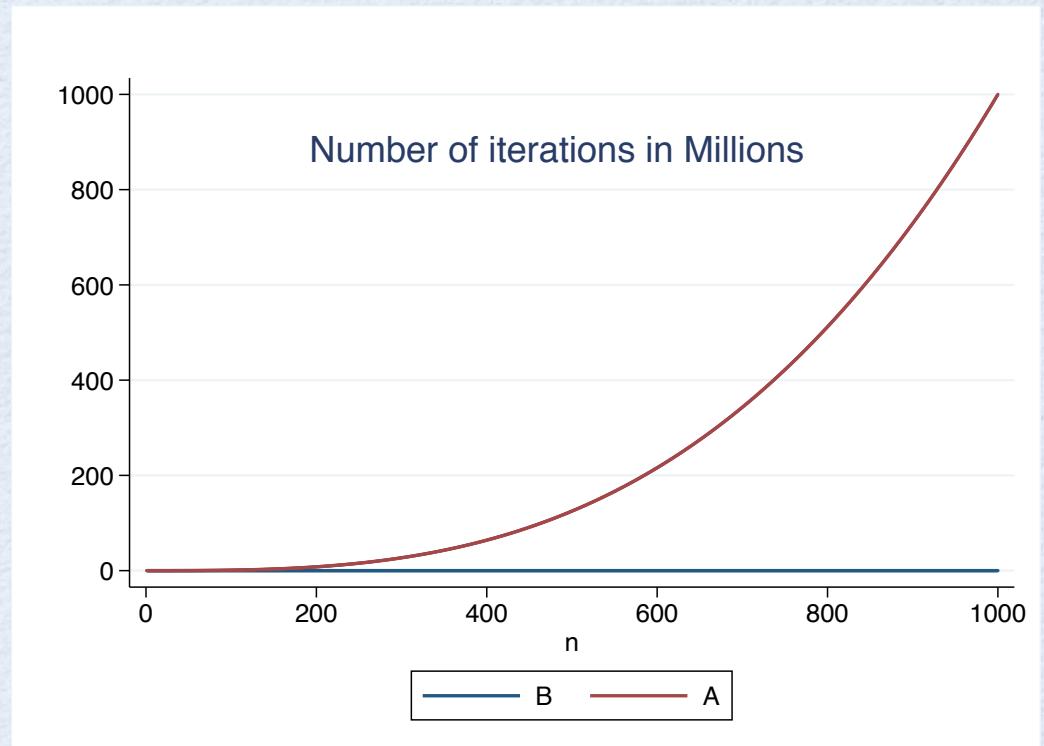


```
DataProvider.cpp
91 /**
92 * This method computes the maximum sum for a
93 * subsequence of values in the array fNumbers.
94 *
95 * @return The maximum subsequence sum in O(n)
96 */
97 int DataProvider::maxSubSumON() const
98 {
99     int lMaxSum = 0;
100    int lThisSum = 0;
101
102    for (int i = 0; i < fEntryCount; i++ )
103    {
104        lThisSum += fNumbers[i];
105
106        if (lThisSum > lMaxSum )
107            lMaxSum = lThisSum;
108        else
109        {
110            if (lThisSum < 0 )
111                lThisSum = 0;
112        }
113    }
114
115    return lMaxSum;
116 }
```

$$\sum_{i=0}^{n-1} 3$$

# Running Time $T$ (Algorithm)

- Test environment: MacPro, 2.66 GHz Quad-Core Intel Xeon
- Algorithm A –  $O(n^3)$ :
  - $n = 2,000$ :  $T(A) = 5\text{s}$
  - $n = 5,000$ :  $T(A) = 72\text{s}$
  - $n = 10,000$ :  $T(A) = 579\text{s}$
- Algorithm B –  $O(n)$ :
  - $n = 2,000$ :  $T(B) < 1\text{s}$
  - $n = 5,000$ :  $T(B) < 1\text{s}$
  - $n = 10,000$ :  $T(B) < 1\text{s}$



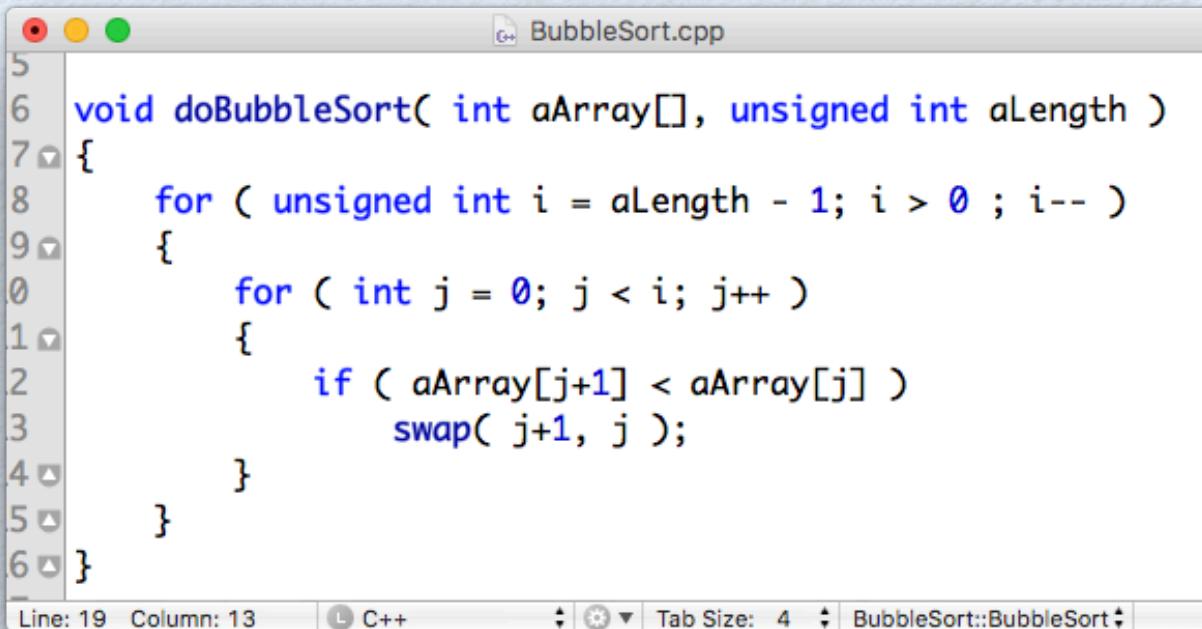
# Algorithmic Patterns

- Direct solution strategies:
  - Brute force and greedy algorithms
- Backtracking strategies:
  - Simple backtracking and branch-and-bound algorithms
- Top-down solution strategies:
  - Divide-and-conquer algorithms
- Bottom-up solution strategies:
  - Dynamic programming
- Randomized strategies:
  - Monte Carlo algorithms

# Brute-force Algorithms

- Brute-force algorithms are not distinguished by their structure.
- Brute-force algorithms are separated by their way of solving problems.
- A problem is viewed as a sequence of decisions to be made. Typically, brute-force algorithms solve problems by exhaustively enumerating all the possibilities.

# Bubble Sort



A screenshot of a C++ code editor window titled "BubbleSort.cpp". The code implements the bubble sort algorithm. It features two nested loops: an outer loop from line 7 to 6 that iterates over the array elements, and an inner loop from line 10 to 11 that compares adjacent elements and swaps them if they are in the wrong order. The code uses standard C++ syntax with curly braces for blocks and colons for conditionals.

```
5
6 void doBubbleSort( int aArray[], unsigned int aLength )
7 {
8     for ( unsigned int i = aLength - 1; i > 0 ; i-- )
9     {
10        for ( int j = 0; j < i; j++ )
11        {
12            if ( aArray[j+1] < aArray[j] )
13                swap( j+1, j );
14        }
15    }
16 }
```

Line: 19 Column: 13    C++    Tab Size: 4    BubbleSort::BubbleSort

- Bubble sort uses a nested for-loop to sort an array in increasing order.
- Bubble sort is  $O(n^2)$ . It works fine on small arrays, but it is unsuitable on larger data sets.

# Greedy Algorithms

- Greedy algorithms do not really explore all possibilities.  
They are optimized for a specific attribute.
- Example: Knapsack Problem
  - Profit - Maximal value of items
  - Weight - Maximal weight stored first
  - Density - Maximal profit per weight
- Greedy algorithms produce a feasible solution, but do not guarantee an optimal solution.

# Sudoku Solver: Greedy

1. Find  $M[i,j]$  with the minimal number of choices.
2. Let  $C$  be the possibilities for  $M[i,j]$ .
3. Set  $M[i,j]$  to first element of  $C$ .
4. Solve puzzle recursively with new configuration.
5. If 4. succeeds, then report result and exit.
6. If 4. fails, then set  $M[i,j]$  to next element if  $C$ . If there is no more element, then report error.
7. Continue with 4.

# Sudoku

A hard puzzle:

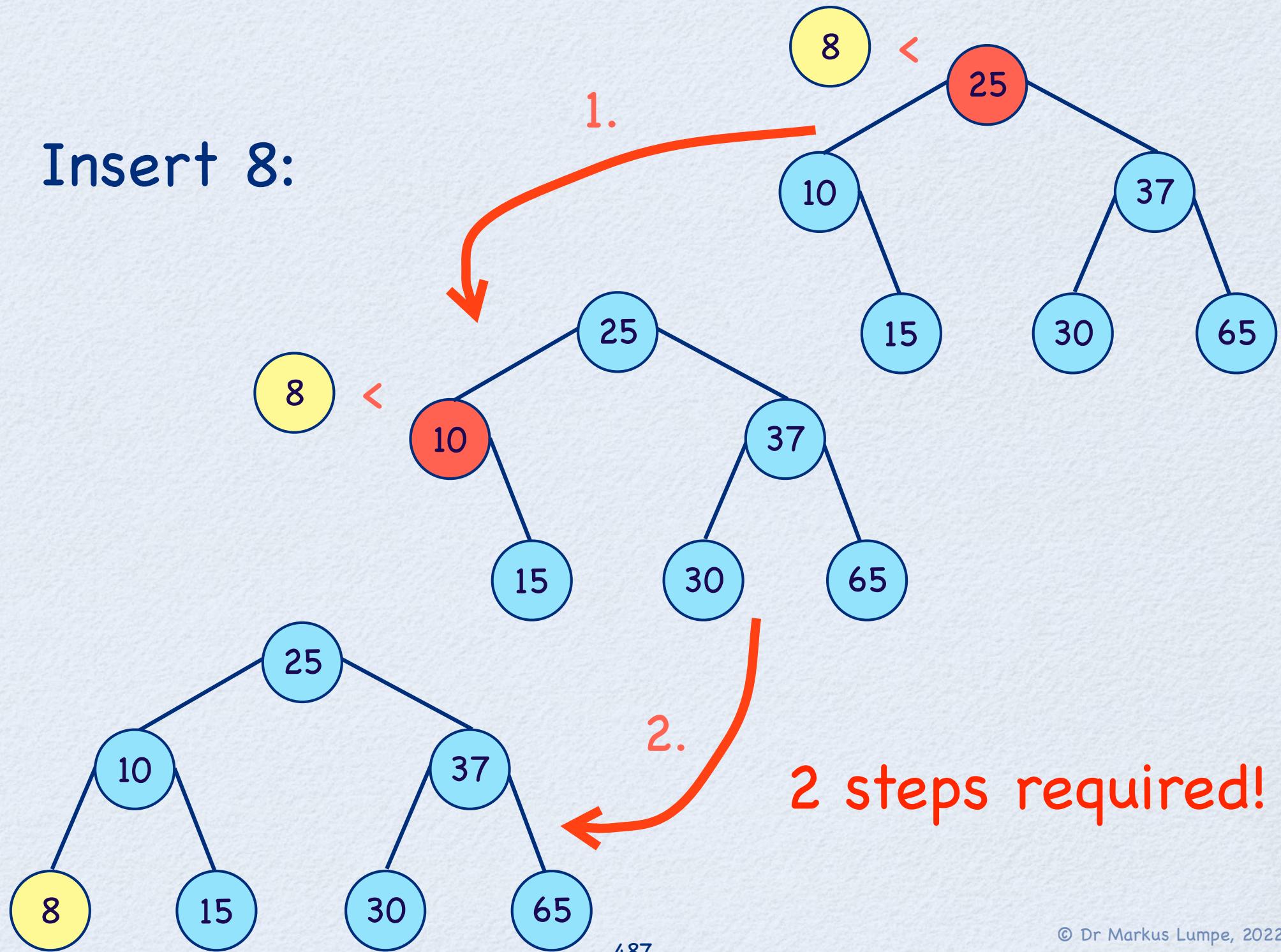
3,5,7,8

5			8			7	9	
			1				3	
	6	9				8		
	4		6	3				
1		2				3		6
			5	1			7	
		1				9	2	
2				4				
9	6			3				8

# Divide-and-Conquer

- Top-down algorithms use recursion to divide-and-conquer the problem space.
- This class of algorithms has the advantage that not all possibilities have to be explored.
- Example: Binary Search, Merge Sort, Quick Sort

## Insert 8:



# Binary Search: $O(\log n)$

```
BinarySearch.cpp
40 int doBinarySearch( int aSortedArray[], int aLength, int aSearchValue )
41 {
42     int lLowIndex = 0;                      // leftmost array index
43     int lHighIndex = aLength - 1;            // rightmost array index
44
45     while ( lLowIndex <= lHighIndex )        // at least one more element
46     {
47         int lMidIndex = (lLowIndex + lHighIndex) / 2;
48
49         if ( aSortedArray[lMidIndex] == aSearchValue )
50             return lMidIndex;                  // element found
51
52         if ( aSortedArray[lMidIndex] > aSearchValue )
53             lHighIndex = lMidIndex - 1;        // new rightmost array index
54         else
55             lLowIndex = lMidIndex + 1;        // new leftmost array index
56     }
57     return -1; // element not found
58 }
```

Line: 38 Column: 1 C++ Tab Size: 4 printArrayLH

$$\log(10) = 2.3$$

```
Terminal
Search 7 in [1, 2, 3, 3, 4, 5, 5, 6, 7, 9]
Search 7 in [5, 5, 6, 7, 9]
Search 7 in [7, 9]
8
Search 20 in [1, 2, 3, 3, 4, 5, 5, 6, 7, 9]
Search 20 in [5, 5, 6, 7, 9]
Search 20 in [7, 9]
Search 20 in [9]
```

# Backtracking Algorithms

- A backtracking algorithm systematically considers all possible outcomes for each decision.
- Backtracking algorithms are distinguished by the way in which the space of possible solutions is explored. Sometimes a backtracking algorithm can detect that an exhaustive search is unnecessary.
- Example: Knapsack Problem

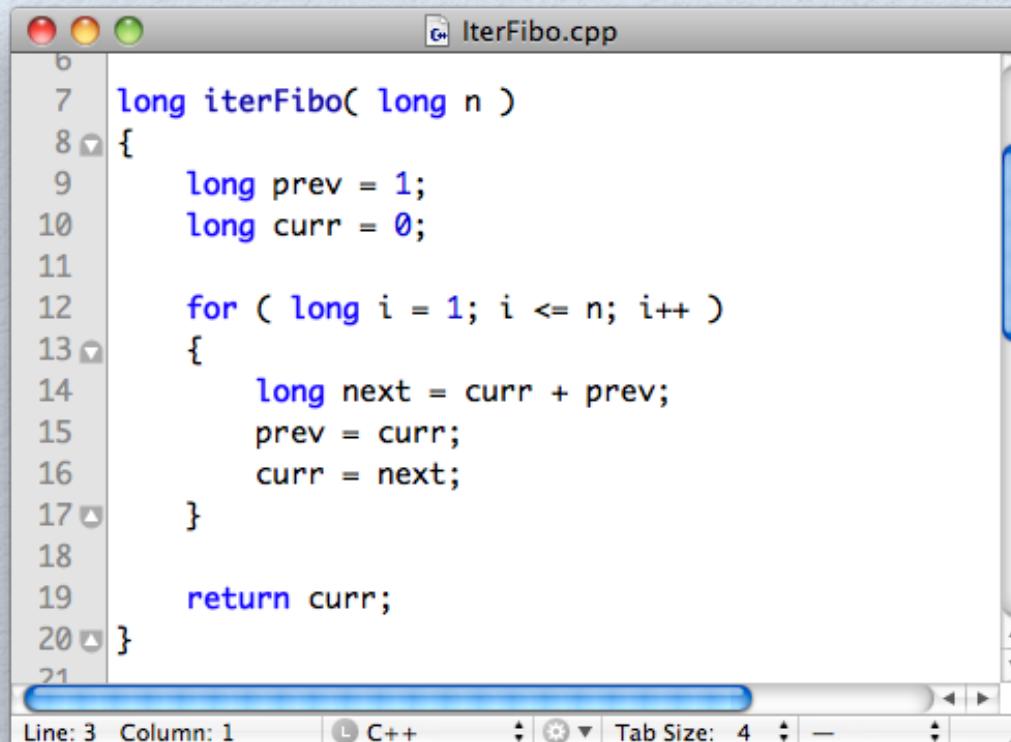
# Sudoku Solver: Backtracking

1. Find  $M[i,j]$  with the minimal number of choices.
2. Let  $C$  be the possibilities for  $M[i,j]$ .
3. Set  $M[i,j]$  to first element of  $C$ .
4. Solve puzzle recursively with new configuration.
5. If 4. succeeds, then report result and exit.
6. If 4. fails, then set  $M[i,j]$  to next element if  $C$ . If there is no more element, then report error.
7. Continue with 4.

# Bottom-up

- Bottom-up algorithms employ dynamic programming.
- Bottom-up algorithms solve a problem by solving a series of subproblems.
- These subproblems are carefully devised in such a way that each subsequent solution is obtained by combining the solutions to one or more of the subproblems that have already been solved.
- Example: Parsing, Pretty-Printing

# Fast Fibonacci



The screenshot shows a Mac OS X application window titled "IterFibo.cpp". The window contains the following C++ code:

```
6
7 long iterFibo( long n )
8 {
9     long prev = 1;
10    long curr = 0;
11
12    for ( long i = 1; i <= n; i++ )
13    {
14        long next = curr + prev;
15        prev = curr;
16        curr = next;
17    }
18
19    return curr;
20}
21
```

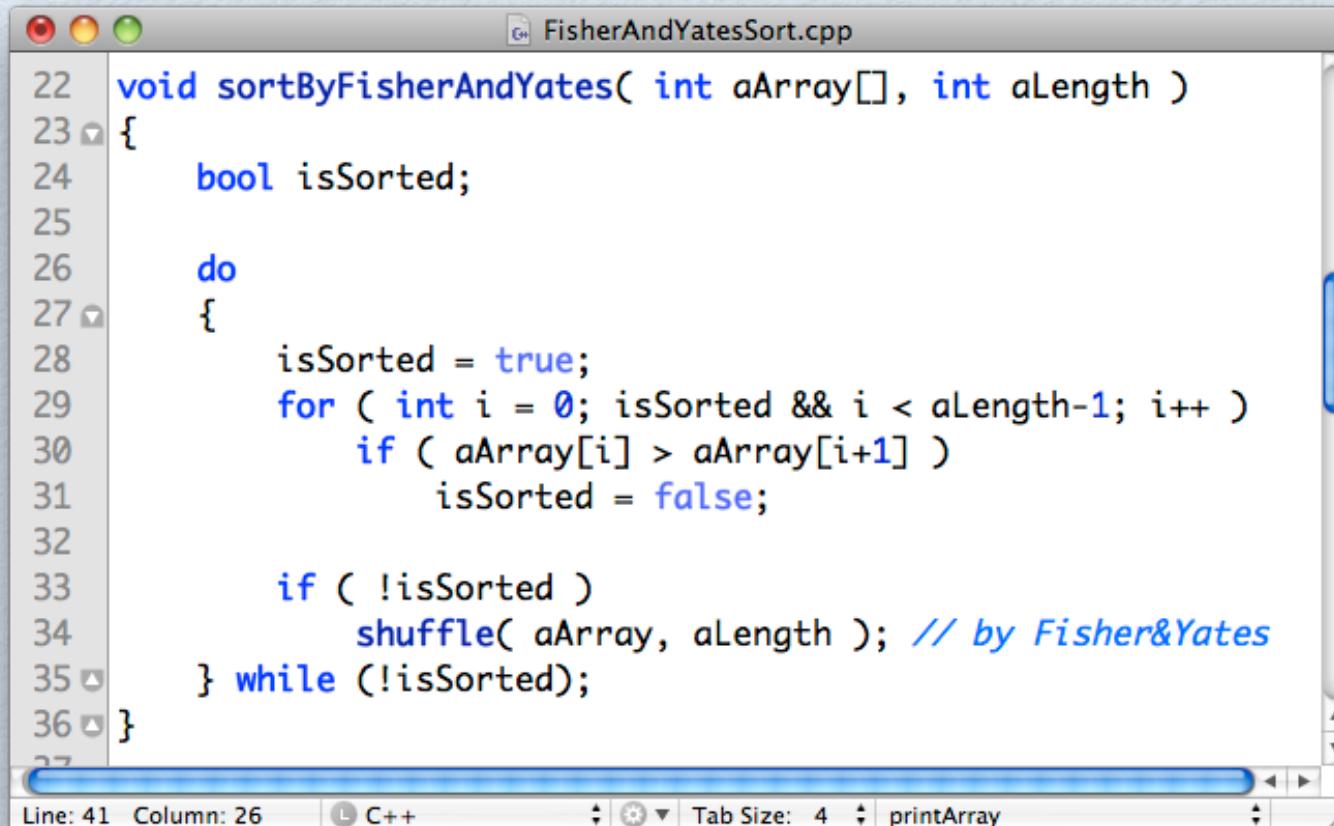
The status bar at the bottom of the window displays "Line: 3 Column: 1" and "C++".

- The iterative solution of the Fibonacci function has its origin in dynamic programming.
- We define the Fibonacci sequence as a table with two elements: the previous value and the current value. In each step we compute the next value by adding the table entries.

# Randomized Algorithms

- Randomized algorithms behave randomly.
- Randomized algorithms select elements in a random order to solve a given problem.
- Eventually, all possibilities are explored, but different runs can produce results faster or slower, if a solution exists.
- Example: Monte Carlo Methods, Simulation

# Sorting by Fisher&Yates



```
22 void sortByFisherAndYates( int aArray[], int aLength )
23 {
24     bool isSorted;
25
26     do
27     {
28         isSorted = true;
29         for ( int i = 0; isSorted && i < aLength-1; i++ )
30             if ( aArray[i] > aArray[i+1] )
31                 isSorted = false;
32
33         if ( !isSorted )
34             shuffle( aArray, aLength ); // by Fisher&Yates
35     } while ( !isSorted );
36 }
```

- There is at least one configuration that satisfies the sorting criterion, but the use of the Fisher&Yates shuffling process makes this algorithm  $O(n*n!)$ . It is called “Bogosort.”