SWINBURNE UNIVERSITY OF TECHNOLOGY

Data Structures and Patterns
Revision 1

SWINBURNE UNIVERSITY OF TECHNOLOGY

Lecture 1

SWINBURNE UNIVERSITY OF TECHNOLOGY

The Structure of a Class

```
class X
{
private:
// private members

protected:
// protected members

public:
// public members
};
```

Access Modifiers

- **private:**
  Private members can be accessed only within the class in which they are declared. Usually class variables are declared here.
- **protected:**
  Protected members can be accessed within the class in which they are declared and within derived classes.
- **public:**
  Public members can be accessed anywhere, including outside of the class itself. Usually methods constructors, accessors, operator overloads and other functions used with the class are declared here.

Constructors

- **Constructors** are executed automatically whenever a new object is created.
- A **constructor** is a kind of member function that initializes an instance of its class.
- A **constructor** has the same name as the class and no return value.
- Constructors may be overloaded.
- The concrete constructor arguments determine which constructor to use.
  - A **constructor** can have any number of parameters and a class may have any number of overloaded **constructors**

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.

```
c:\Users\pzheng\Desktop\DSP.PS3\Mark\initi\Debug\initi.exe
The volume is 10584
The volume is 125
Press any key to continue . . .
```

```
1  #include<iostream>
2
3  using namespace std;
4
5  class Box {
6      int m_width;
7      int m_length;
8      int m_height;
9
10 public:
11     Box(int width, int length, int height)
12         : m_width(width), m_length(length), m_height(height){}
13     Box(): m_width(5), m_length(5), m_height(5){}
14
15     int Volume() {return m_width * m_length * m_height; }
16 };
17
18 int main(){
19     Box b(42, 21, 12), a;
20     cout << "The volume is " << b.Volume()<<endl;
21     cout << "The volume is " << a.Volume()<<endl;
22
23     system("pause");
24     return 0;
25 }
```

Class Book - Constructors

Book.h

```
1  #ifndef BOOK_H_
2  #define BOOK_H_
3
4  #include <iostream>
5
6  class Book
7  {
8   private:
9     std::string fISBN;
10    unsigned fUnitsSold;
11    double fRevenue;
12
13  public:
14    Book() : fUnitsSold(0), fRevenue(0.0) {}
15    Book( const std::string& aBook ) : fISBN(aBook), fUnitsSold(0), fRevenue(0.0) {}
16    Book( std::istream& aIStream ) { aIStream >> *this; }
17
18    Book& operator+=( const Book& aRHS );
19
20    friend bool operator==( const Book& aLeft, const Book& aRight );
21    friend std::istream& operator>>( std::istream& aIStream, Book& aItem );
22    friend std::ostream& operator<<( std::ostream& aOStream, const Book& aItem );
23
24    double getAveragePrice() const;
25    bool hasSameISBN( const Book& aRHS ) const;
26 };
27
28 #endif /* BOOK_H_ */
29
```

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.
- If some data members have built-in or compound types, then the class should not rely on the synthesized default constructor!

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 the class declares those procedures as friends.

Class Book - The Friends

```
6  class Book
7  {
8   private:
9     std::string fISBN;
10    unsigned fUnitsSold;
11    double fRevenue;
12
13  public:
14    Book() : fUnitsSold(0), fRevenue(0.0) {}
15    Book( const std::string& aBook ) : fISBN(aBook), fUnitsSold(0), fRevenue(0.0) {}
16    Book( std::istream& aIStream ) { aIStream >> *this; }
17
18    Book& operator+=( const Book& aRHS );
19
20    friend bool operator==( const Book& aLeft, const Book& aRight );
21    friend std::istream& operator>>( std::istream& aIStream, Book& aItem );
22    friend std::ostream& operator<<( std::ostream& aOStream, const Book& aItem );
23
```

- 'friend' is used to make the overload operators accessible globally. The overload operators shown here are (1) == which returns bool, (2) >> which returns istream, and (3) << which returns ostream.
  - These are defined when required because these operators are not made to be used with the Book type.
  - After we define them we are able to for example, use the == operator like Book1 == Book2;

SWINBURNE UNIVERSITY OF TECHNOLOGY

Lecture 2

SWINBURNE UNIVERSITY OF TECHNOLOGY

Operator Overloading

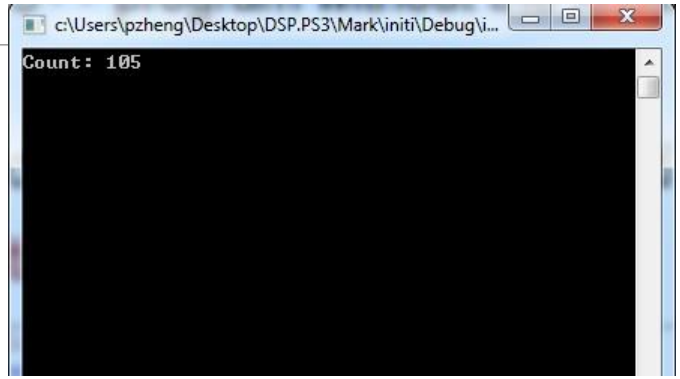
- Operator overloading (less commonly known as ad-hoc polymorphism) is a specific case of polymorphism (part of the OO nature of the language) in which some or all operators like +, = or == are treated as polymorphic functions and as such have different behaviours depending on the types of its arguments.
- Operator overloading is usually only syntactic sugar. It can easily be emulated using function calls.

```
ClassName operator - (ClassName c2)
{
    ... .. ...
    return result;
}

int main()
{
    ClassName c1, c2, result;
    ... .. ....
    result = c1-c2;
    ... .. ...
}
```

Why Operator overloading

- You may write any C++ program without the knowledge of operator overloading.
- However, operator operating are profoundly used by programmers to make program intuitive.

```
1  #include <iostream>
2  using namespace std;
3
4  class Test
5  {
6     private:
7        int count;
8     public:
9         Test(): count(5){}
10        void operator ++()
11        {
12            count = count+100;
13        }
14        void Display() { cout<<"Count: "<<count; }
15 };
16
17 int main()
18 {
19      Test t;
20      // this calls "function void operator ++()" function
21      ++t;
22      t.Display();
23      getchar();
24      return 0;
25 }
```



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.

The Equivalence Operator ==

- The Boolean operator == defines a structural equivalence test for Book objects.
- We use `const` references for the Book arguments to pass Book objects by reference rather than copying their values into the stack frame of the operator ==.

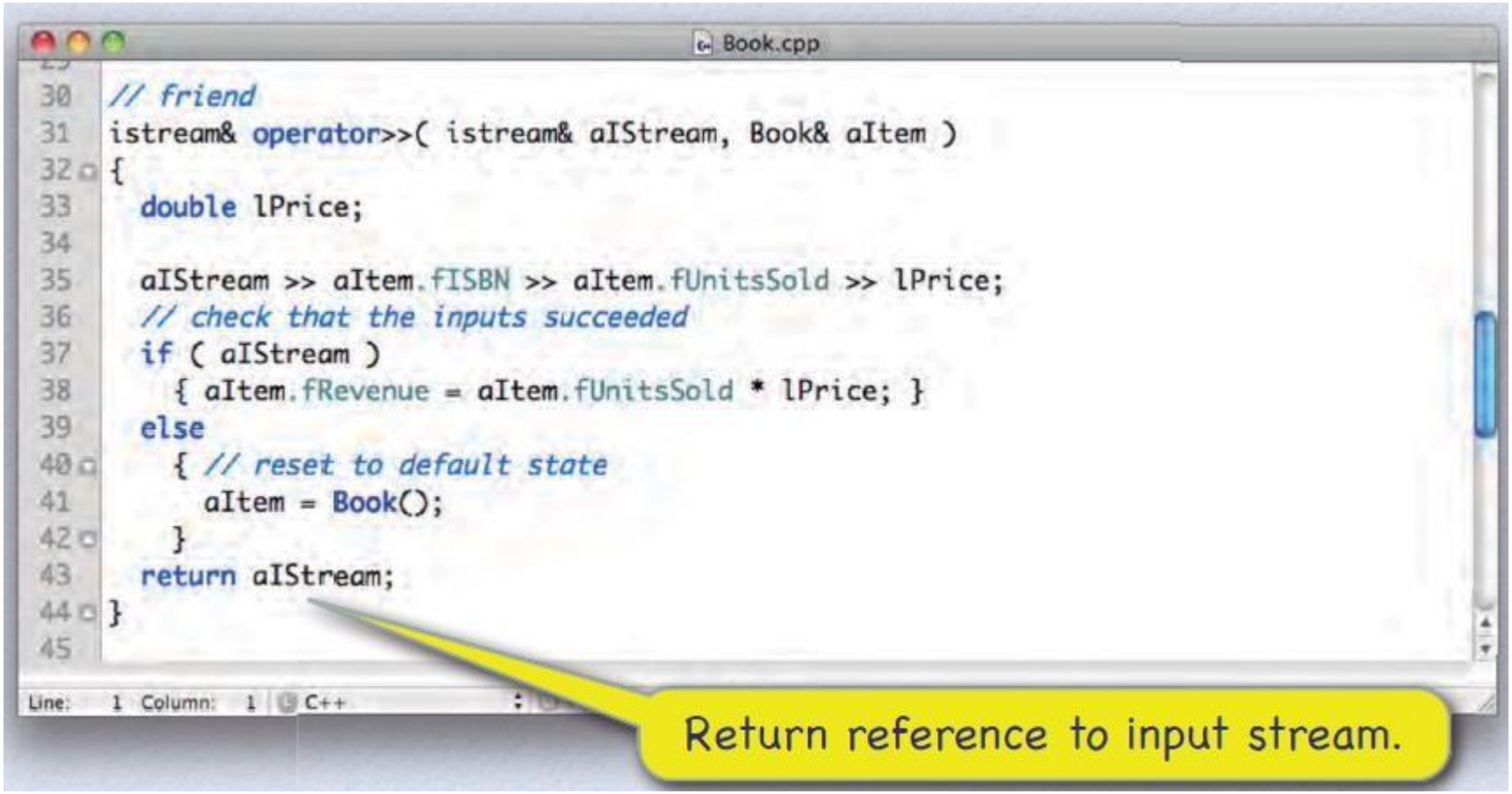
```
// friend
bool operator==( const Book& aLeft, const Book& aRight )
{
  return aLeft.fUnitsSold == aRight.fUnitsSold &&
         aLeft.fRevenue == aRight.fRevenue &&
         aLeft.hasSameISBN( aRight );
}
```

The insertion and extraction operators

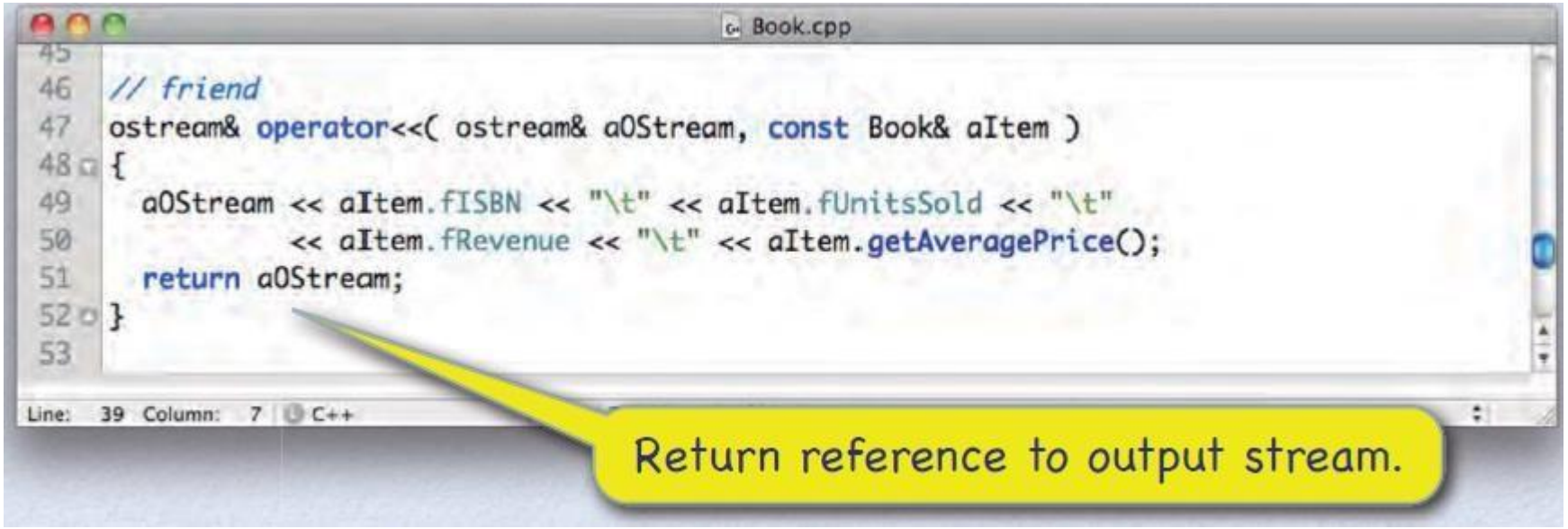
- The **insertion** (<<) **operator**, which is preprogrammed for all standard C++ data types, sends bytes to an output stream object. **Insertion operators** work with predefined "manipulators," which are elements that change the default format of integer arguments.
- The **extraction operator** (>>), which is preprogrammed for all standard C++ data types, is the easiest way to get bytes from an input stream object. Formatted text input **extraction operators** depend on white space to separate incoming data values.
- Reference

  http://faculty.cs.niu.edu/~hutchins/csci241/io-op.htm

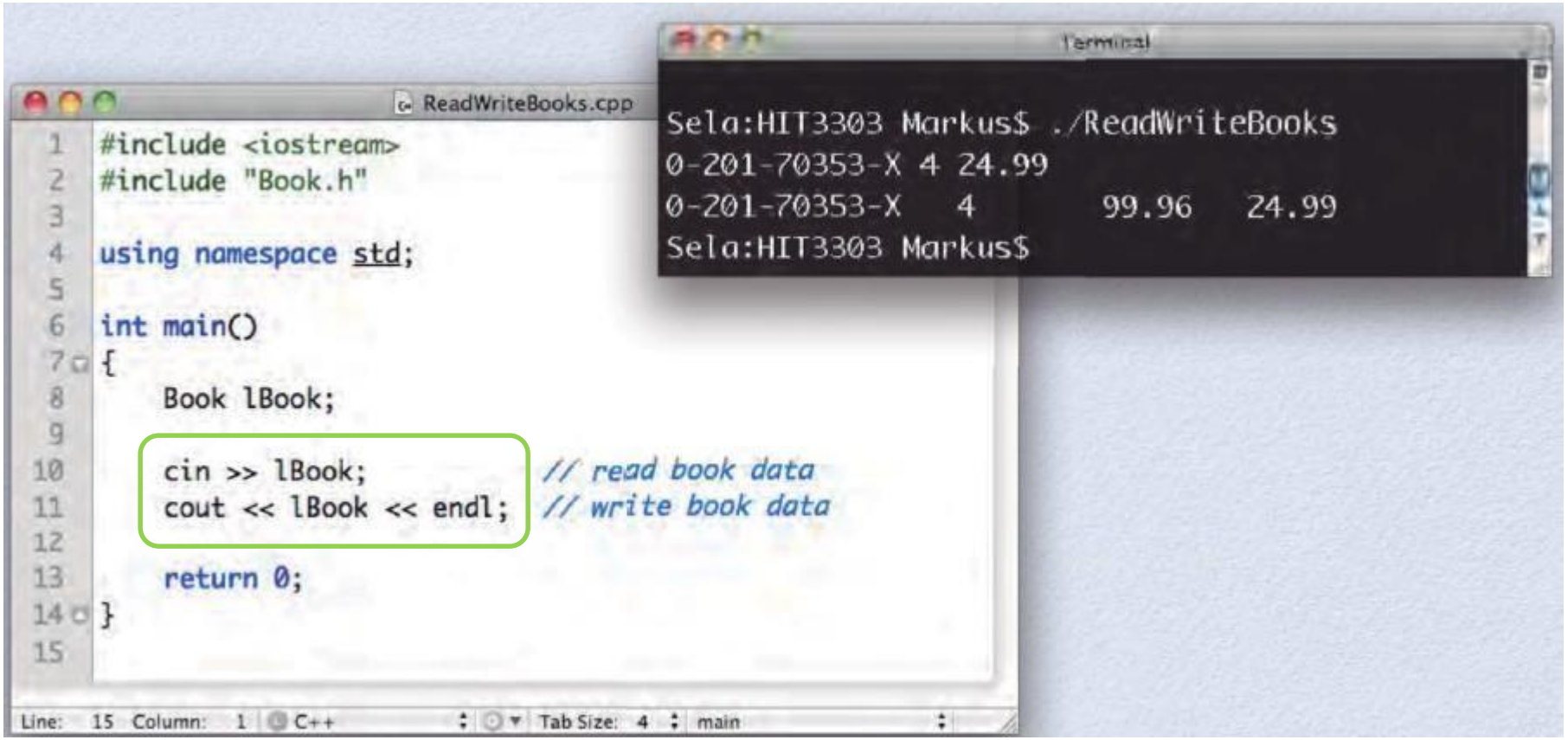
The Input Operator >>



The Output Operator <<



The overloaded >> and << operators used in main()



Overloading stream insertion (<<) and extraction (>>) operators in C++

- In C++, stream insertion operator “<<” is used for output and extraction operator “>>” is used for input.
- We must know the following things before we start overloading these operators.
  1) cout is an object of ostream class
  2) cin is an object of istream class
  3) These operators must be overloaded as a global function. And if we want to allow them to access private data members of class, we must make them a **friend**.

‘this’ pointer in C++

- The ‘this’ pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions.
- ‘this’ pointer is a constant pointer that holds the memory address of the current object. ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).
- Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

A static member is a member of the class which is shared by all objects of the class. No matter how many objects of the class are created, there will only be one copy of the static member.

‘this’ pointer in C++

- There are several situations that we use “this” pointer

1)When local variable’s name is same as member’s name

```
#include<iostream>
using namespace std;

/* local variable is same as a member's name */
class Test
{
private:
   int x;
public:
   void setX (int x)
   {
       // The 'this' pointer is used to retrieve the object's x
       // hidden by the local variable 'x'
       this->x = x;
   }
   void print() { cout << "x = " << x << endl; }
};

int main()
{
   Test obj;
   int x = 20;
   obj.setX(x);
   obj.print();
   return 0;
}
```

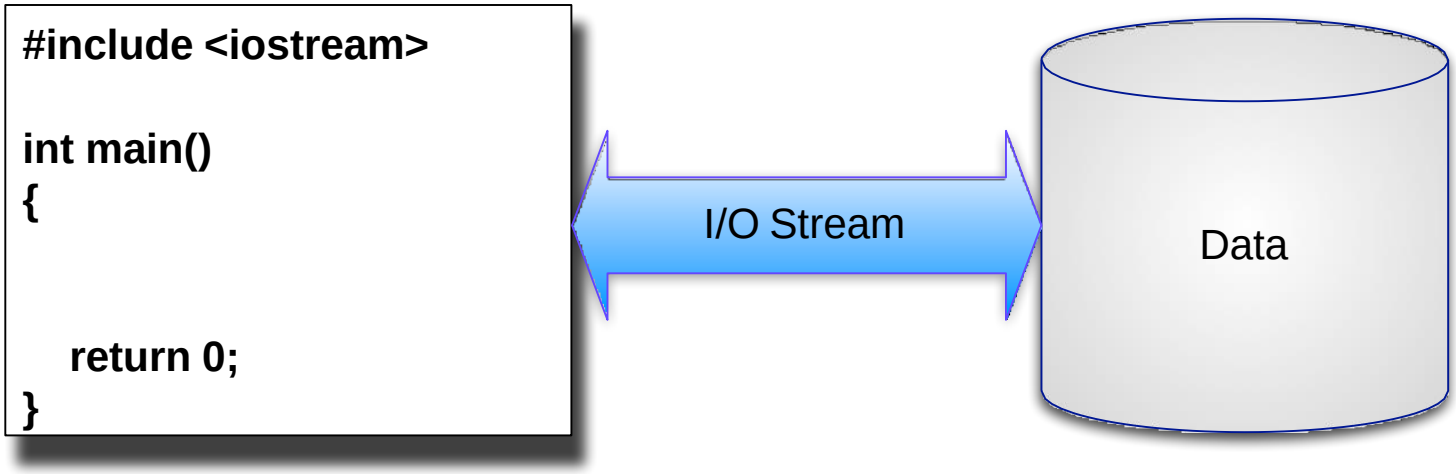
SWINBURNE UNIVERSITY OF TECHNOLOGY

‘this’ pointer in C++

- 2) To return reference to the calling object

```
#include<iostream>
using namespace std;

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test &setX(int a) { x = a; return *this; }
  Test &setY(int b) { y = b; return *this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
  Test obj1(5, 5);

  // Chained function calls.  All calls modify the same object
  // as the same object is returned by reference
  obj1.setX(10).setY(20);

  obj1.print();
  return 0;
}
```

I/O Streams

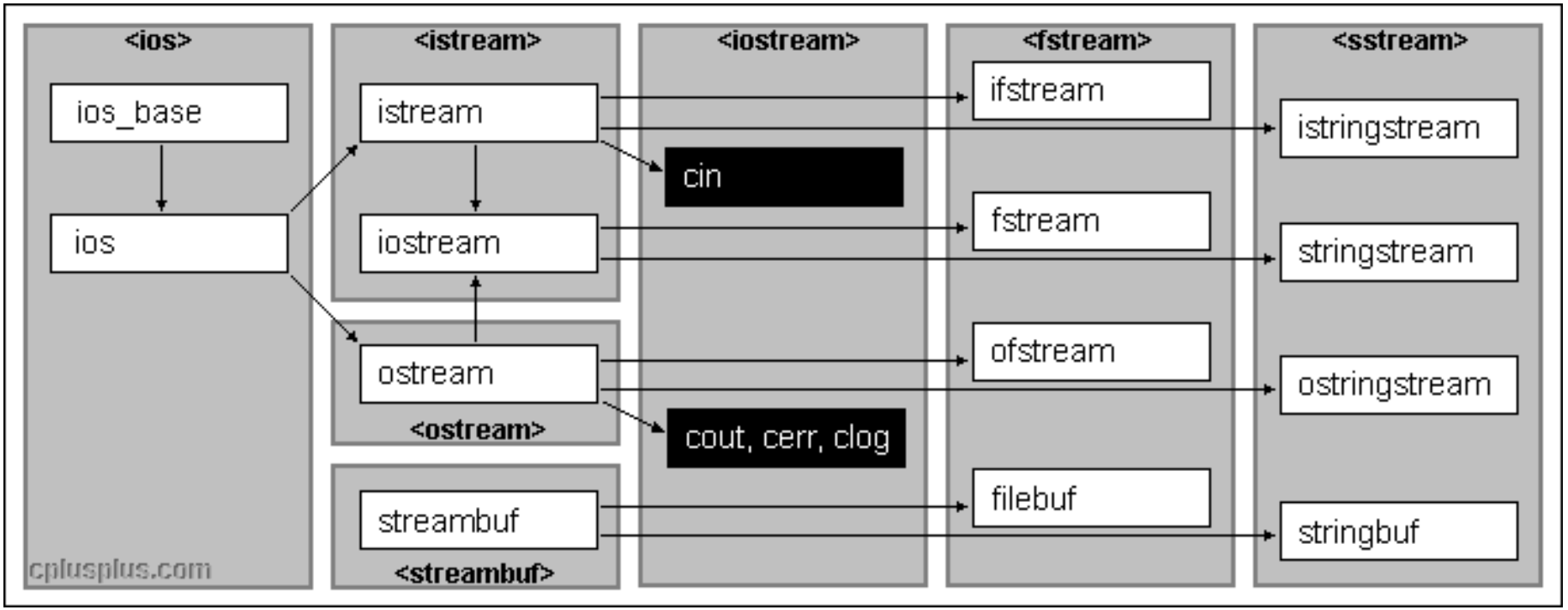


- In C++, input or output, independent of the type of I/O medium, are mapped into logical **data streams** with common properties.
- Two forms of mapping are supported: **text streams** and **binary streams**.

I/O Media

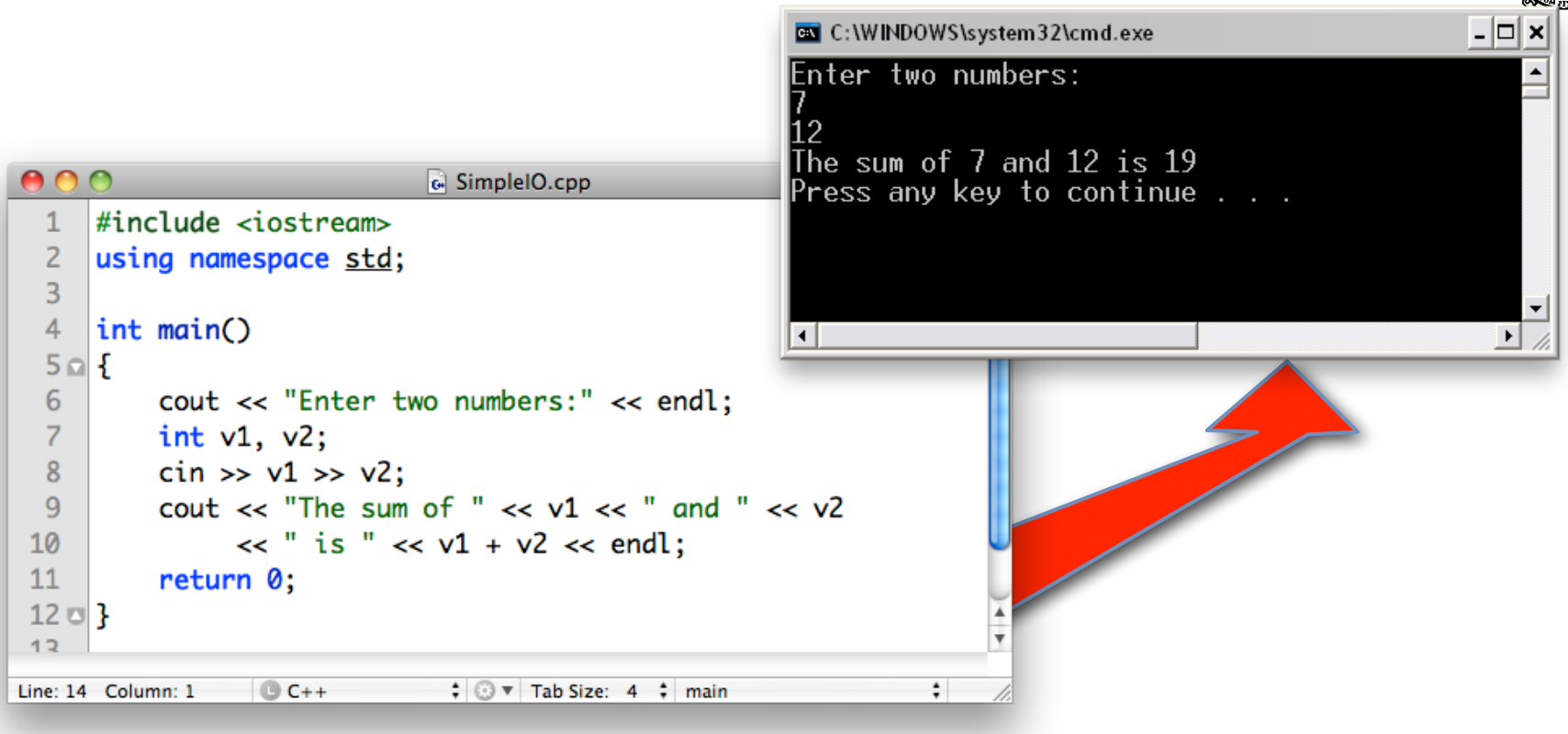
- Streams can be associated with
  - **Physical devices** (e.g., console - cin, cout)
  - **Files** (e.g., coefficients.txt, sales.dbf)
  - **Structured storage** (e.g., int values[10])

Input/Output library



You can read more about the Input/Output library here:
http://www.cplusplus.com/reference/iolibrary/

A Program that uses the C++ I/O library.



- cin and cout represent the standard input stream and the standard output stream in every C++ program.

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 of 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.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Lecture 3

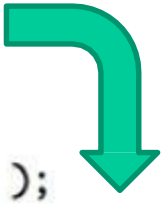
SWINBURNE UNIVERSITY OF TECHNOLOGY

Inheritance

- A mechanism for specialization
- A mechanism for reuse
- Fundamental to supporting polymorphism

Example

```
4  class Account
5  {
6  private:
7    double fBalance;
8
9  public:
10   Account( double aBalance );
11   virtual ~Account() {}
12
13   void deposit( double aAmount );
14   virtual void withdraw( double aAmount );
15   double getBalance();
16 };
```



```
18 class BankAccount: public Account
19 {
20 private:
21   double fInterestRate;
22
23 public:
24   BankAccount( double aRate );
25   ~BankAccount() {}
26
27   virtual void withdraw( double aAmount );
28   void addInterest();
29   void chargeFee( double aAmount );
30 };
```

Access Levels for Class Inheritance

- public:
  - Public members in the base class remain public.
  - Protected members in the base class remain protected.
- protected:
  - Public and protected members in the base class are protected in the derived class.
- private:
  - Public and protected members in the base class become private in the derived class.

https://www.tutorialspoint.com/cplusplus/cpp_class_access_modifiers.htm

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 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

- A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.
- A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.
  - Do not accept arguments.
  - Cannot specify any return type (including void).
  - Cannot return a value using the return statement.

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.

Virtual Destructor

```
class Base
{
 public:
 ~Base() {cout << "Base Destructor\t"; }
};

class Derived:public Base
{
 public:
 ~Derived() { cout<< "Derived Destructor"; }
};

int main()
{
 Base* b = new Derived;      //Upcasting
 delete b;
}
```

Output :

Base Destructor

- **delete b** will only call the Base class destructor, which is undesirable because, then the object of Derived class **remains undestructed**, because its destructor is never called. This results in memory leak.

Virtual Destructor

```
1  class Base
2  {
3   public:
4   virtual ~Base() {cout << "Base Destructor\t"; }
5  };
6
7  class Derived:public Base
8  {
9   public:
10  ~Derived() { cout<< "Derived Destructor"; }
11 };
12
13 int main()
14 {
15  Base* b = new Derived;      //Upcasting
16  delete b;
17 }
```

Output :

Derived Destructor
Base Destructor

- When we have Virtual destructor inside the base class, then first Derived class's destructor is called and then Base class's destructor is called, which is the desired behaviour.

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 *Aptr;
BAptr = new BankAccount( 2.25 );

Aptr = BAptr; … // upper casting
…
delete Aptr;
```

Polymorphism

- The word **polymorphism** means having many forms. Typically, **polymorphism** occurs when there is a hierarchy of classes and they are related by inheritance.
- **C++ polymorphism** means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.
- Polymorphism allows derived classes to use the same functions but have different outcomes. This is done by **function overriding**.

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.
- When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class' version of the function.

The difference between using virtual method and just overriding the methods

- A method can be defined in a base class and overridden in the derived class whether it is a virtual method or not.
- If the overridden method is called directly from the derived class, the derived class' method will be used.
- However, when a **base class pointer points to a derived class object**, it will call the **base class' method** if the it is **not a virtual method**, but will use the **derived class' method** if it is **virtual**.
- Declaring a method 'virtual' means that C++ will use mechanisms to support polymorphism and check to see if there is a more derived version of the method when you call via a base class pointer.

More details here:
https://stackoverflow.com/questions/11067975/overriding-non-virtual-methods

About pure virtual function

- A pure virtual function is a virtual function whose declaration ends in “=0”, it's just syntax!

```
class Base {
   // ...
   virtual void f() = 0;
   // ...
}
```

- A pure virtual function implicitly makes the class it is defined as *abstract*

What is an abstract class?

- The purpose of an **abstract class** (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit.
- Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error.
- Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC.
- Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.

Read more:
https://www.tutorialspoint.com/cplusplus/cpp_interfaces.htm

SWINBURNE UNIVERSITY OF TECHNOLOGY

Lecture 4

SWINBURNE UNIVERSITY OF TECHNOLOGY

Values

- In computer science we classify as a value everything that maybe 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”

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

Composite Data Type

- Composite types are built up using primitive values and composite types. The layout of composite types is in general implementation dependent.
- E.g.
  - struct,
  - array
  - enumerations
  - files
  - etc

```
typedef struct Account_ {
   int    account_number;
   char   *first_name;
   char   *last_name;
   float  balance;
} Account;
```

```
enum Color
{
    Red,
    Blue,
    Green
};
```

- https://en.wikipedia.org/wiki/Composite_data_type

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).

http://stackoverflow.com/questions/8080617/does-java-really-have-pointers-or-not

Arrays

- An array is a compound data type that consists of
  - a type specifier,
  - an identifier, and
  - a dimension.
- Arrays define an ordered, homogeneous sequence of data of a specific length.
- Arrays define a number-based position association among the elements.
- The compiler can reserve the required amount of space when the program is compiled.

Multi-Dimensional Array Initialization

- int board[3][3] = { {1, 2, 3} };
  - only the first row is initialized, the remaining elements are set to integer 0
- char tic-tac-toe[][3] = { {‘_’, ‘_’, ‘_’},

  {‘_’, ‘_’, ‘_’},

  {‘_’, ‘_’, ‘_’} };
- fully-initialized array of 3×3 characters (‘_’)

String handling in C++

- For C++, there are two ways to handle string.
  - using string.h
  - using c style string

Can this program be compiled correctly and run correctly?

```
#include <iostream>


using namespace std;

int main () {
   char str1[10] = "Hello";
   char str2[10] = "World";
   char str3[10];
   int  len;

   // copy str1 into str3
   strcpy( str3, str1);
   cout << "strcpy( str3, str1) : " << str3 << endl;

   // concatenates str1 and str2
   strcat( str1, str2);
   cout << "strcat( str1, str2): " << str1 << endl;

   // total lenghth of str1 after concatenation
   len = strlen(str1);
   cout << "strlen(str1) : " << len << endl;

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

C-Strings

- A C-String is an array of characters: i.e. C-style string
  - char a_string[] = “Hello World!”;
- The length of a C-String is the number of characters of the C-String plus one:

  char a_string[] = { ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘ ’, ‘W’, ‘o’, ‘r’, ‘l’, ‘d’, ‘!’, ‘\0’ };
- advantage: array element accessing using array index
- disadvantage: specify the size needed in advance
  - static allocation: the size of the array is determined at compile-time

The indexer - the operator[]

```
const int& IntArrayIndexer::operator[](const string& aKey) const
{
    int lIndex=0;

    for(unsigned int i=0; i<aKey.length();i++){
        lIndex = lIndex*10+(aKey[i]-'0');
    }

    if(lIndex<fLength)
        return fArrayElements[lIndex];
    else
        throw out_of_range("Index is out of bounds");

}
```

- 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.

Iterators

- **Iterator**: a pointer-like object that can be incremented with ++, dereferenced with *, and compared against another iterator with !=.
- Iterators are generated by STL container member functions, such as begin() and end(). Some containers return iterators that support only the above operations, while others return iterators that can move forward and backward, be compared with <, and so on.

Dereferencing a pointer means getting the value that is stored in the memory location pointed by the pointer.

Iterators

- The generic algorithms use iterators just as you use pointers in C to get elements from and store elements to various containers. Passing and returning iterators makes the algorithms
  - more generic, because the algorithms will work for any containers, including ones you invent, as long as you define iterators for them
  - more efficient (as discussed here)
    - Some algorithms can work with the minimal iterators, others may require the extra features. So a certain algorithm may require certain containers because only those containers can return the necessary kind of iterators.

types of Iterators

- Input Iterator
- Output Iterator
- Forward Iterator
- Bidirectional Iterator
- Random Access Iterator

The Dereference Operator

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

- The deference 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.

Prefix Increment

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

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

Postfix Increment

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

```
24  IntArrayIterator IntArrayIterator::operator ++(int)
25  {
26      IntArrayIterator  temp = *this;
27      fIndex++;
28      return temp;
29  }
```

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

Iterator Equivalence

```
bool IntArrayIterator::operator ==(const IntArrayIterator& aOther) const
{
    return(fIndex==aOther.fIndex)&&(fArrayElements == aOther.fArrayElements);
}
```

- 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 that is constant throughout runtime.

Iterator Inequality

```
bool IntArrayIterator::operator !=(const IntArrayIterator &aOther) const
{
    return !(*this == aOther);
}
```

- We implement != in terms of ==.

Auxiliary Methods

We use the default value 0 for aStart here.

```
41  IntArrayIterator IntArrayIterator::begin() const
42  {
43      return IntArrayIterator(fArrayElements, fLength);
44  }
45
46  IntArrayIterator IntArrayIterator::end() const
47  {
48      return IntArrayIterator(fArrayElements, fLength, fLength);
49  }
50
```

- 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.

Input and output stream

- The standard library defines a handful of stream objects that can be used to access what are considered the standard sources and destinations of characters by the environment where the program runs

- Two main operators are stream insertion << and stream extraction >> operators.

stream	description
cin	standard input stream
cout	standard output stream

- The << operator inserts the data that follows it into the stream that precedes it. e.g. *the standard output stream cout*
- The extraction operation on cin uses the type of the variable after the >> operator to determine how it interprets the characters read from the input

I/O: Working With Files

- We can pass the name of the files our program needs to work with through the command line arguments.

```
#include <iostream>      // include standard IO library
#include <fstream>       // include file IO library

using namespace std;

int main( int argc, char* argv[] )
{
    if ( argc < 3 )
    {
        cerr << "Arguments missing" << endl;
        cerr << "Usage: euclid infile outfile" << endl;
        return 1;        // program failed
    }

    ...

    return 0;
}
```

File I/O

- You need extra library to handle file input and output rather than standard library.
- include <fstream> in your C++ code
- Writing to a file …
- http://www.cplusplus.com/reference/fstream/

```
1  #include<iostream>
2  #include<fstream>
3
4  using namespace std;
5
6  int main(int argc, char* argv[])
7  {
8
9      ofstream oFile;
10
11     oFile.open("myfile.txt");
12
13     oFile<<"Hello world!"<<endl;
14
15     oFile.close();
16     system("pause");
17
18     return 0;
19 }
```

SWINBURNE UNIVERSITY OF TECHNOLOGY

Lecture 5

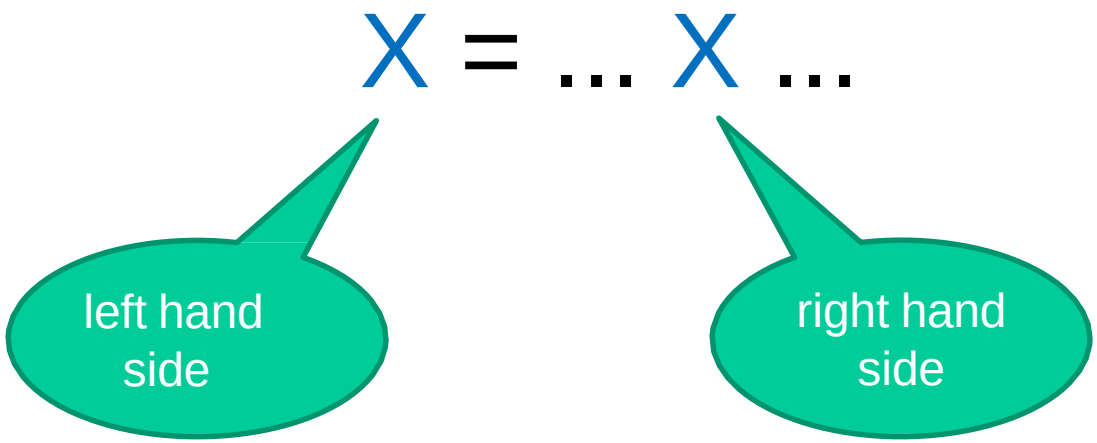
SWINBURNE UNIVERSITY OF TECHNOLOGY

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 the 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. This condition that is used to stop the recursion is called the **base case**.

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



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

List Nodes

- A list manages a collection of elements.
- The class ListNode defines a value-based sequence container for values of type int.

```
2  class ListNode{
3
4  public:
5      int item;
6      ListNode* next;
7
8
9      ListNode(){
10         item =0;
11         next = NULL;
12     }
13
14     ListNode(int n){
15         item =n;
16         next = NULL;
17
18     }
19
20     ListNode(int n, ListNode *p){
21         item = n;
22         next = p;
23     }
24 };
```

Templates

- Templates are blueprints from which classes and/or functions automatically generated by the compiler based on a set of parameters.
- Each time a template is used with different parameters is used, a new version of the class or function is generated.
- A new version of a class or function is called specialization of the template.

Function template

- "Type substitutes"

```
template <class T>

T max(T a, T b)

{ return a<b ? b:a;}
```

- The symbol T is called a type parameter. It is simply a place holder that is replaced by an actual type or class when the function is invoked.

Template header

- A function template is declared in the same way as an ordinary function, except that it is preceded by the specification
  - template <class T>
- The type parameter T may be used in place of ordinary types within the function definition
- The word class is used to mean a class or primitive type.
  - A template may have several type parameters, specified like this:
    - template<class T, class U, class V>

Class template

- Class template work the same way as a function template except that it generates classes instead of function. The general syntax is

```
template<class T, ...> class X{...}
```

- As with function templates, a class template may have several template parameters. Some of them can be primitive types parameters

```
template<class T, int n, class U> class X{...}
```

Node Class template

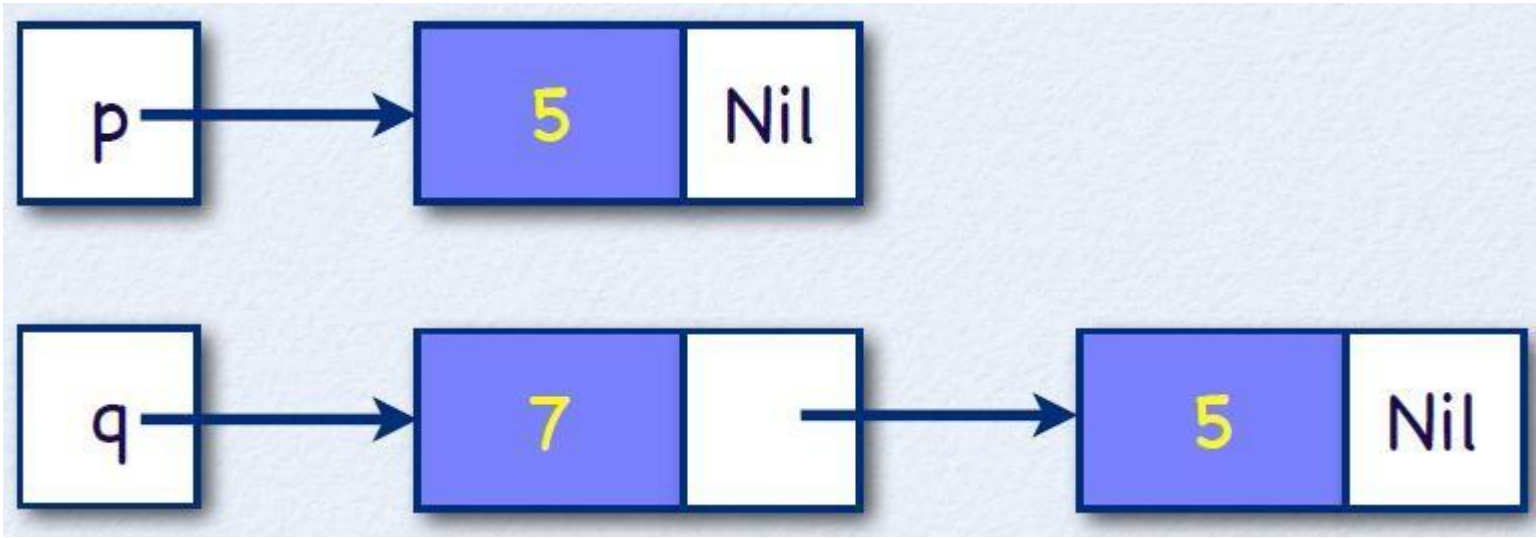
```
4   template <class DataType>
5   class ListNode
6   {
7   public:
8       DataType fData;
9       ListNode* fNext;
10
11  public:
12      ListNode( const DataType& aData, ListNode* aNext = (ListNode*)0 )
13      {
14          fData = aData;
15          fNext = aNext;
16      }
17  };
```

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:
  - ListNode<int>* pListOfInteger = (ListNode<int>*)0;
  - Here (ListNode<int>*)0 stands for Nil.

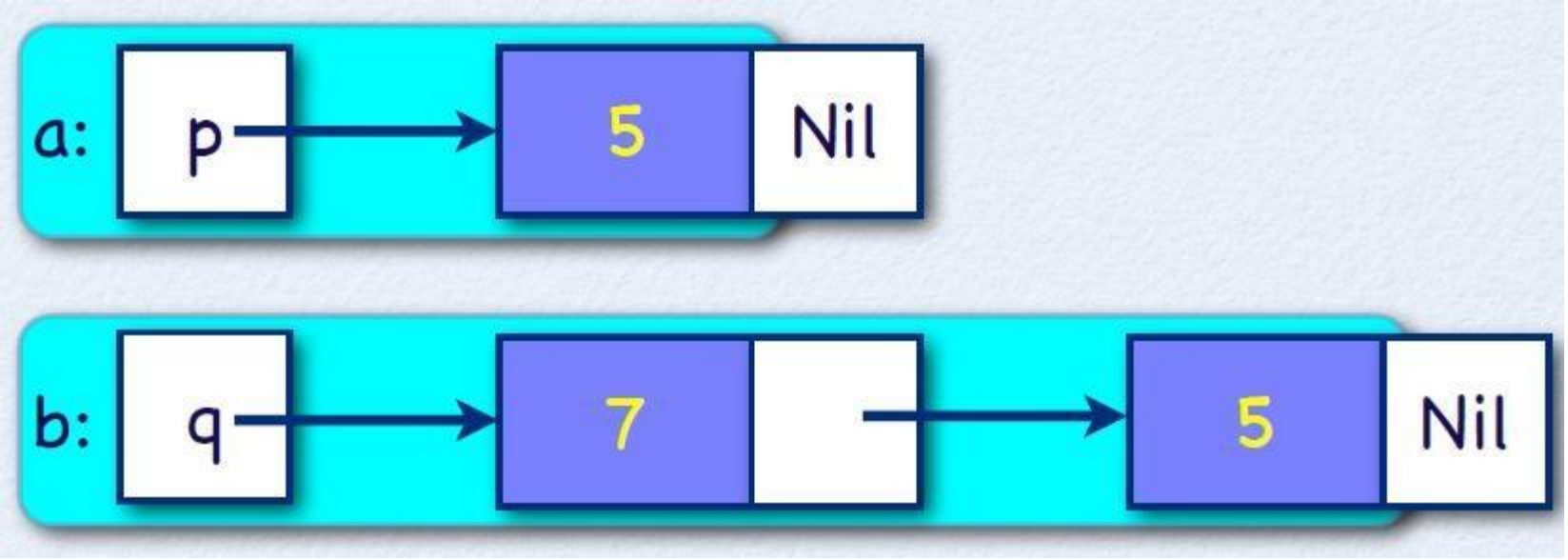
Node Construction

- `IntegerNode *p, *q;`
- `p = new IntegerNode( 5 );`
- `q = new IntegerNode( 7, p );`

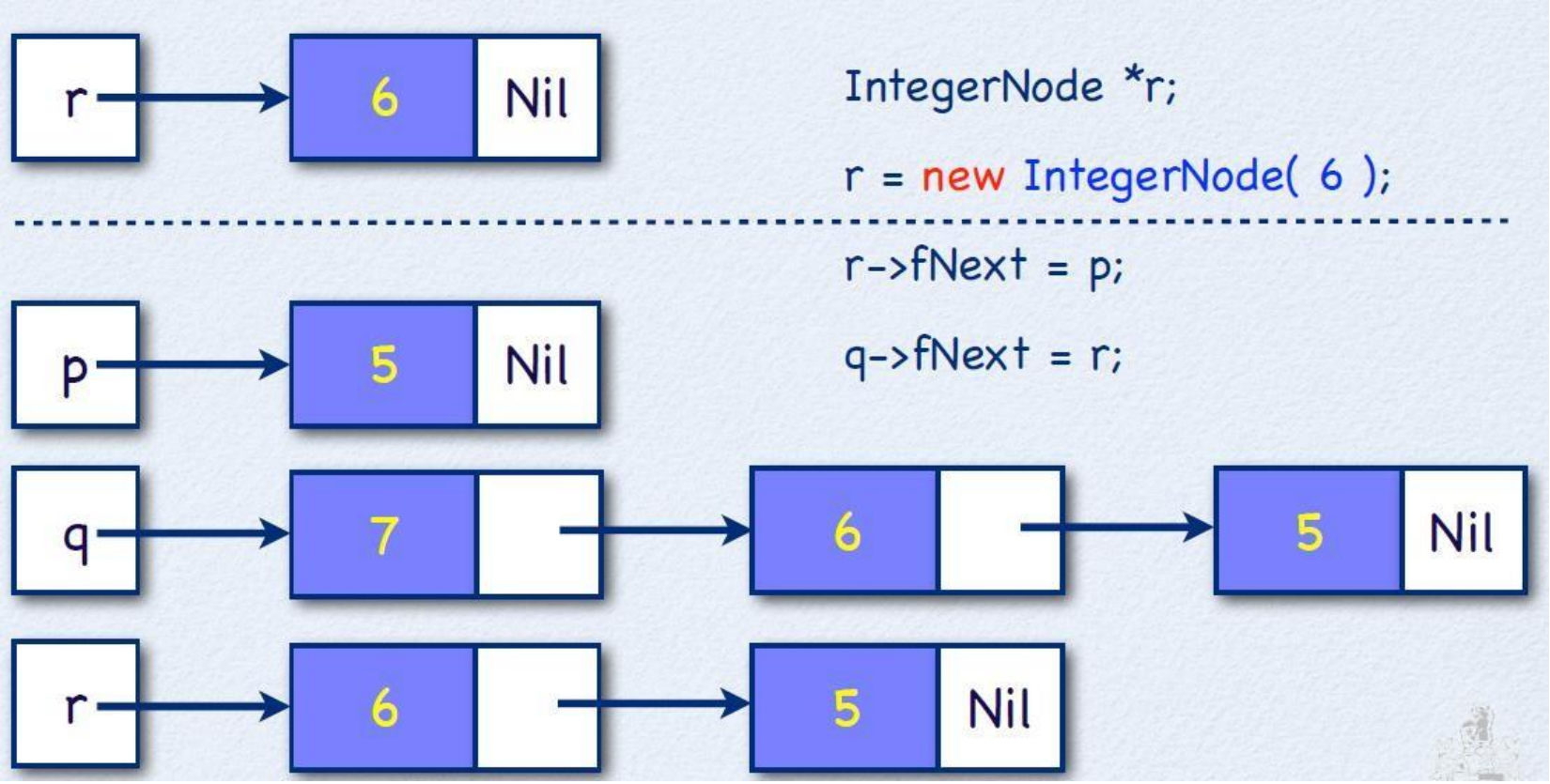


Node Content Access

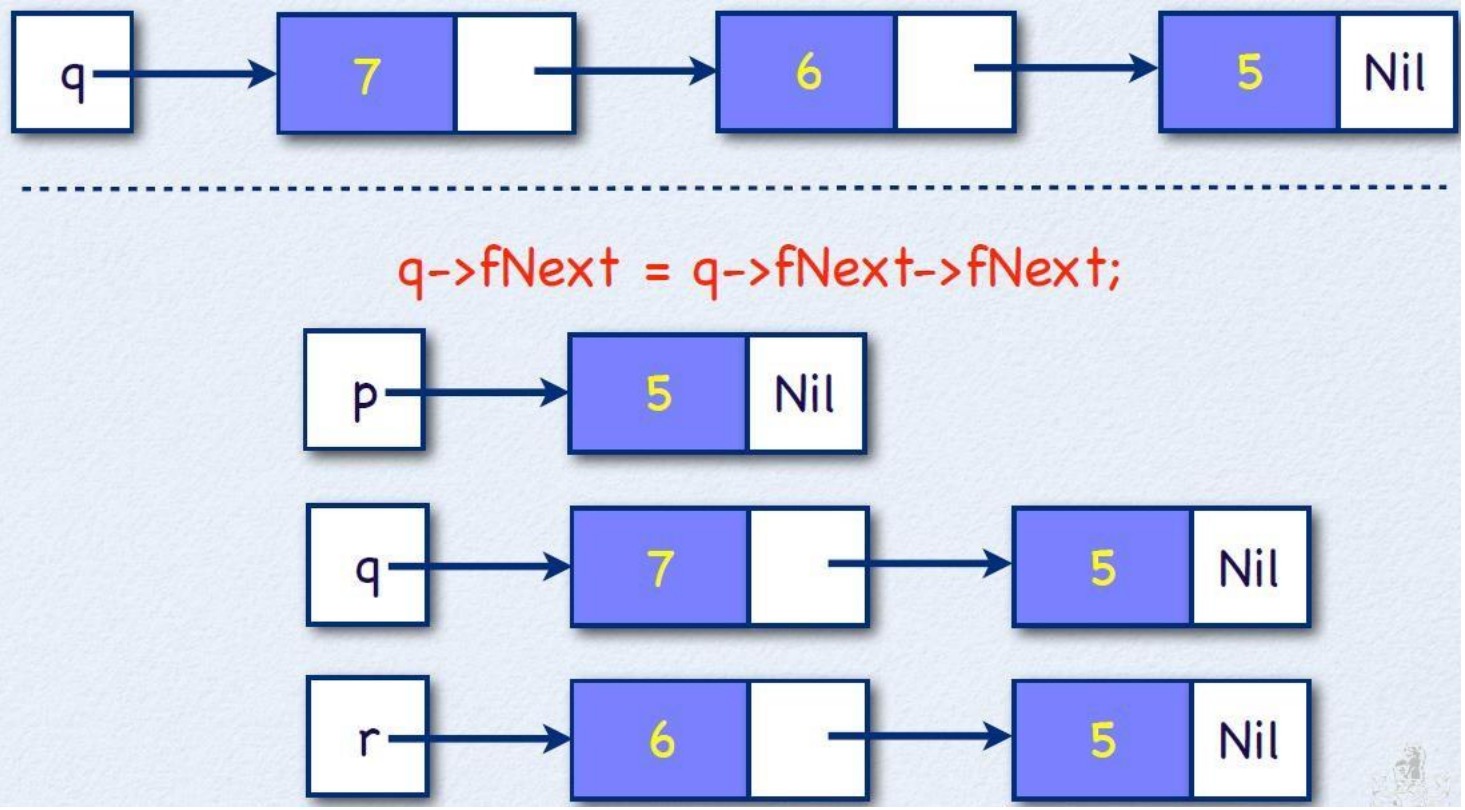
- `int a = p->fData;`
- `int b = q->fNext->fData;`



Inserting a Node



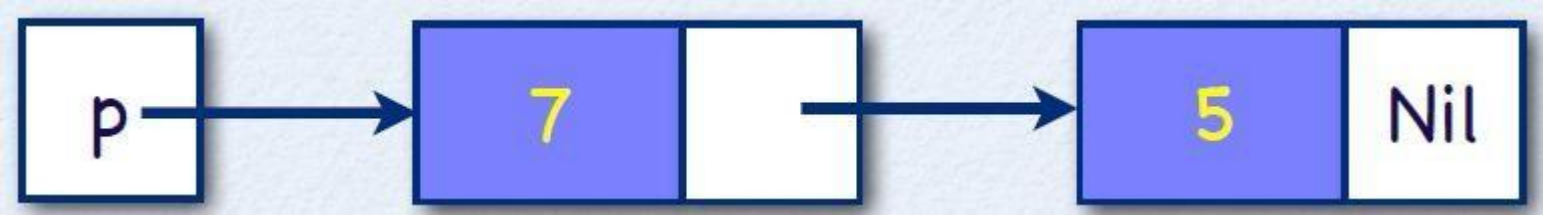
Deleting a Node



Insert at the Top(at the end)

- IntegerNode *p = (IntegerNode*)0;
- p = new IntegerNode(5, p);

- p = new IntegerNode(7, p);

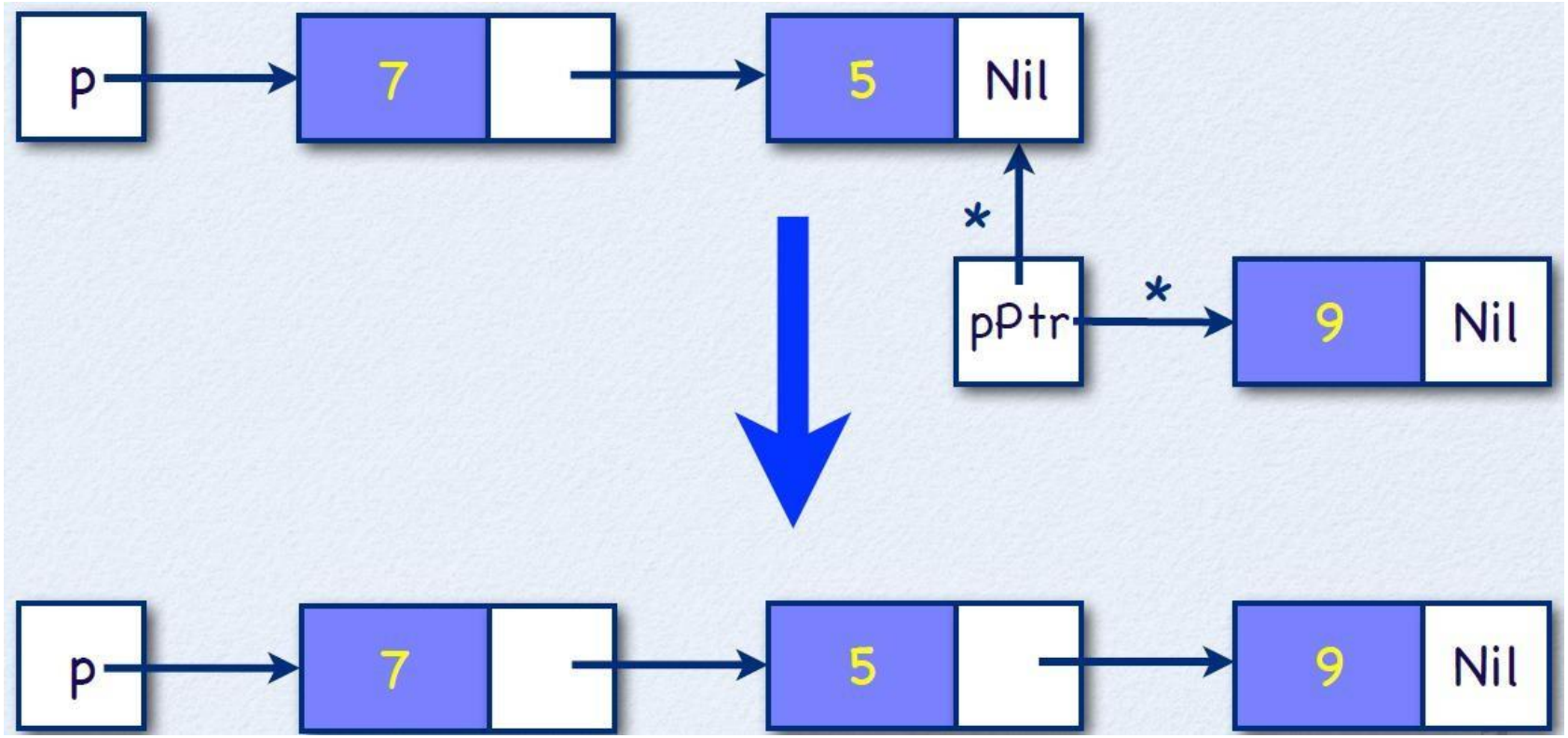


Insert at the Beginning

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

```
IntegerNode *p = (IntegerNode*)0;
IntegerNode **pPtr = &p;

while ( *pPtr != (IntegerNode*)0 )
    pPtr = &((*pPtr)->fNext);   // pointer to next

*pPtr = new IntegerNode( 9 );
```

Insert at the Beginning



SWINBURNE UNIVERSITY OF TECHNOLOGY

Lecture 6

SWINBURNE UNIVERSITY OF TECHNOLOGY

Abstract Data Type

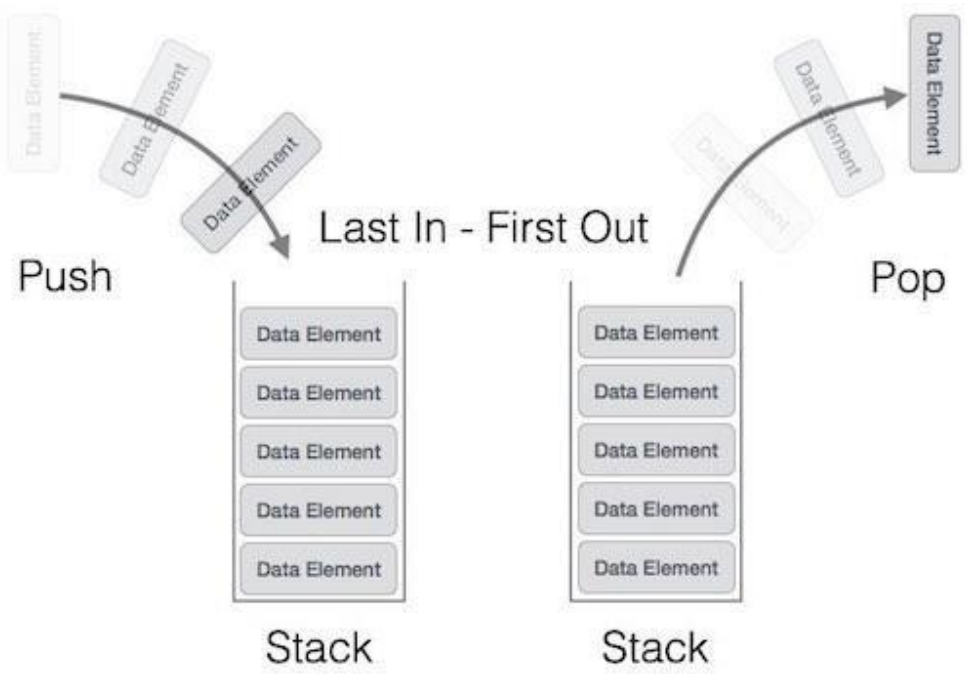
- Abstract
  - Existing in thought or as an idea but not having a physical or concrete existence.
  - Consider something theoretically
    - i.e., A summary of the contents of a book, article, or speech.
- The technique used to define new data types independently of their actual representation is called data abstraction.
- Data abstraction provides only essential information to the outside world and hiding their background details,
  - i.e., to represent the needed information in program without presenting the details.

Data Structure: Stack

- A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack
- A real-world stack allows operations at one end only.
  - For example, we can place or remove a card or plate from the top of the stack only.
- Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

Stack

- It is a LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first.
- In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.



Stack operations

- We have seen the function `push_back()`.
  - It appends an element to the end of the controlled sequence.
- There is a counterpart function, `pop_back()`,
  - It removes the last element in the controlled sequence.

```
std::vector<int> v;
v.push_back(999);
v.pop_back();
```

Applications of Stacks

- Reversal of input
- Checking for matching parentheses
- Backtracking (e.g., Prolog or graph analysis)
- State of program execution (e.g., storage for parameters and local variables of functions)
- Tree traversal
- ...

Container

- A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.
- The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

Stack Interface

- 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 const references (e.g. const T&).

```
4   template<class T>
5   class Stack
6   {
7   public:
8       Stack( int aSize );
9       ~Stack();
10
11      bool isEmpty() const;
12      int size() const;
13      void push( const T& aItem );
14      void pop();
15      const T& top() const;
16  };
```

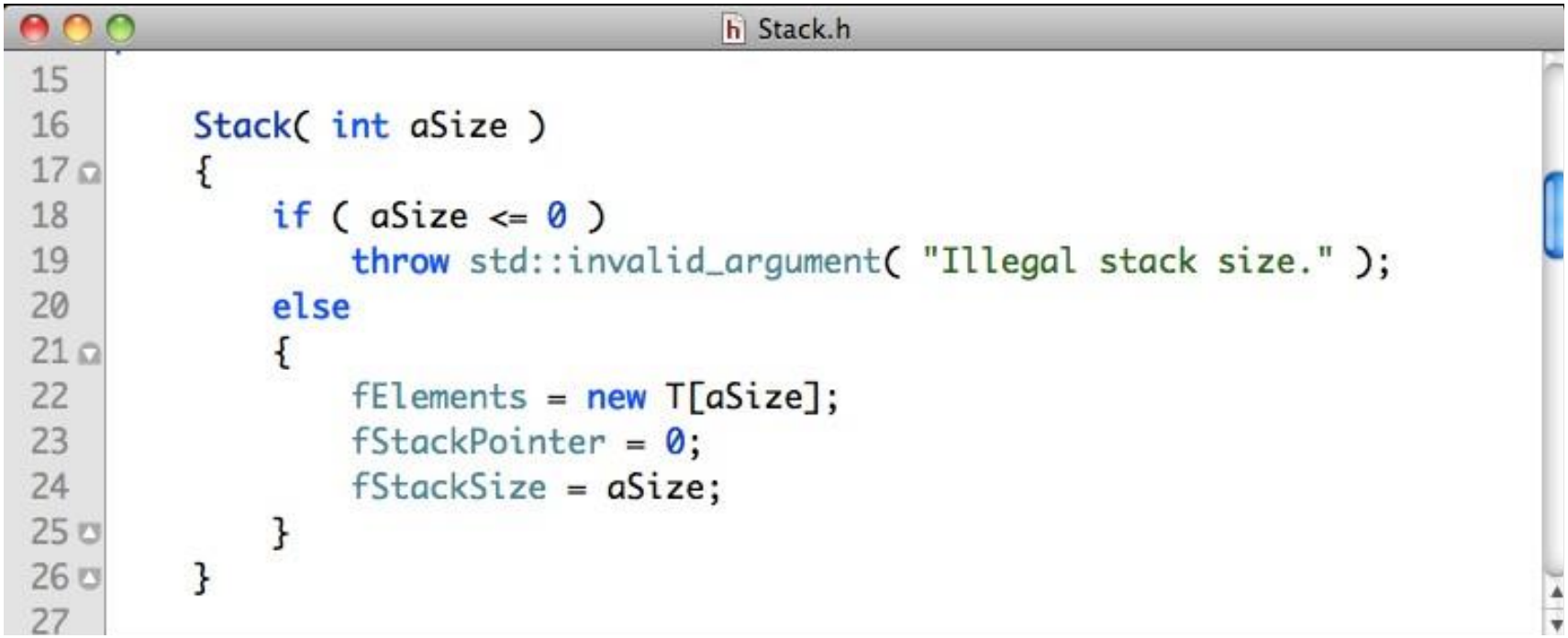
  - const references will allow us to refer to the actual object or value rather than a mere copy of it.

The Stack's Private abstraction

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

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

Stack Constructor



Stack Destructor

```
~Stack()
{
    delete [] fElements;
}
```

- There are two forms of delete:
  - delete ptr - release the memory associated with pointer ptr.
  - delete [] arr - release the memory associated with all elements of array arr and the array arr itself.
- Whenever one allocates memory for an array, for example char* arr = new char[10], one must use the array form of delete to guarantee that all array cells are released.

Stack utilities

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

```
bool isEmpty() const
{
    return fStackPointer == 0;
}

int size() const
{
    return fStackPointer;
}
```

Push and pop

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

```
void push( const T& aItem )
{
    if ( fStackPointer < fStackSize )
        fElements[fStackPointer++] = aItem;
    else
        throw std::overflow_error( "Stack full." );
}
```

- 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.

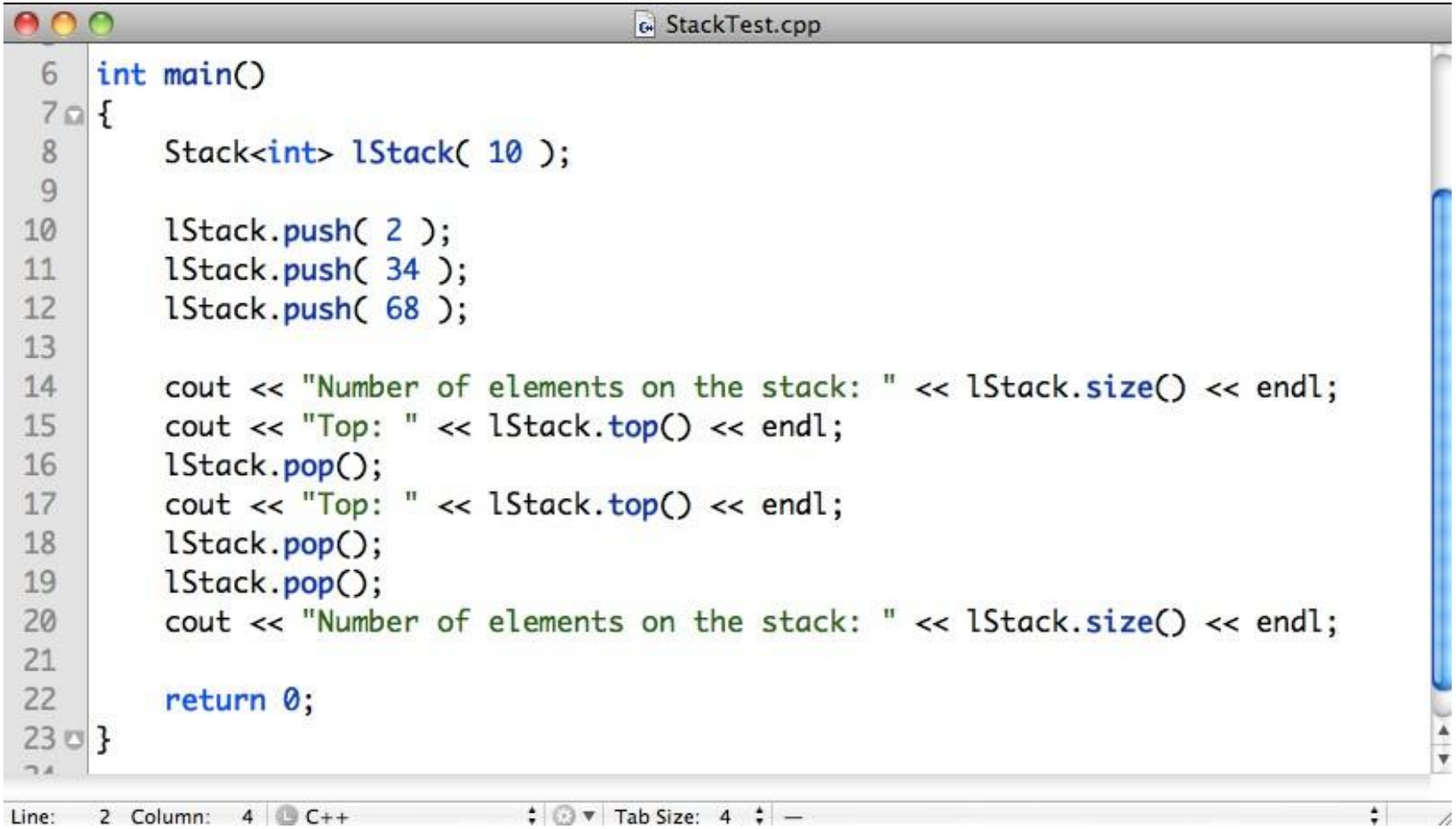
```
void pop()
{
    if ( !isEmpty() )
        fStackPointer--;
    else
        throw std::underflow_error( "Stack empty." );
}
```

Top

```
const T& top() const
{
    if ( !isEmpty() )
        return fElements[fStackPointer-1];
    else
        throw std::underflow_error( "Stack empty." );
}
```

- 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



Dynamic Stack

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

```
template<class T>
class DynamicStack
{
private:
    List<T> fElements;

public:

    bool isEmpty() const;
    int size() const;
    void push( const T& aItem );
    void pop();
    const T& top() const;
};
```

Containers

- The STL makes seven basic containers available, and three derived.
- There are two categories of containers in the STL
  - sequence: vector, list and deque
  - associative: set, multiset, map and multimap
  - specialised derived container: stack, queue and priority queue

Vector

- Vectors are sequence containers representing arrays that can change in size. (dynamic arrays)
  - contiguous storage locations
  - efficiently direct element access as arrays
  - (unlike array) size can change dynamically
  - internally vectors do not reallocate each time an element is added/removed to the container.
  - allocate some extra storage to accommodate for possible growth
- Compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.

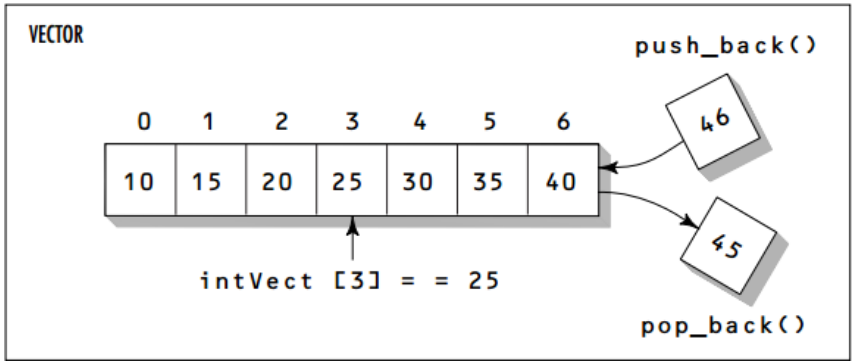
List

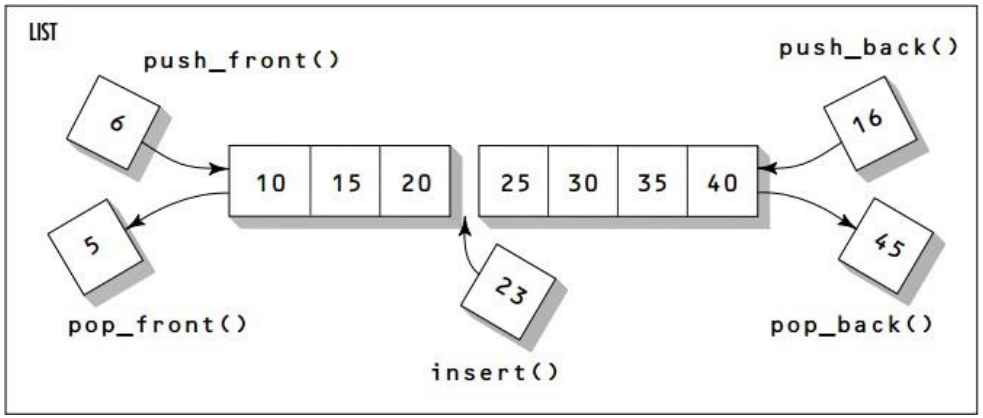
- each element contains a pointer not only to the next element but also to the preceding one.
- The container stores the address of both the front and also the back elements
  - the front: the first element
  - the back: the last element
- This ensures the fast access of both ends of the list.

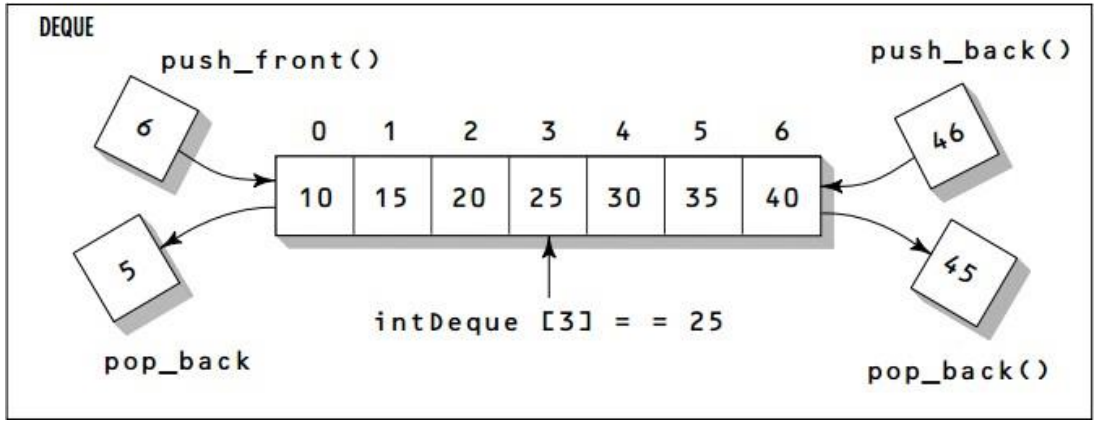
Deque

- Deque is short for Double-Ended Queue
- A deque is like
  - a vector : it supports random access using [] operator
  - a linked list: can be access from front and end
- a double ended vector and supports push_front(),pop_front () and front().
- Deque stores data in several different non-contiguous memory location.(segmented)

Vector, List and Deque

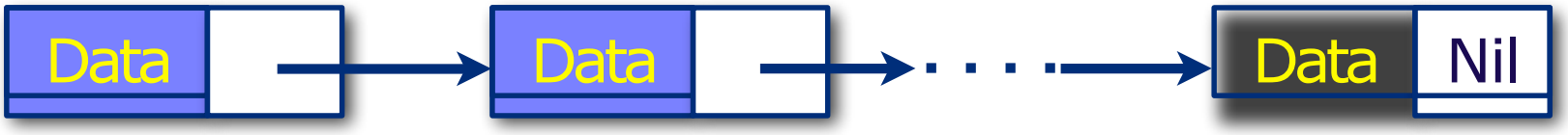






What is a singly-linked list?

- 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.

What is doubly-linked list?

- A doubly-linked list is a sequence of data items, each connected by two links called next and previous.
- A data item may be a primitive value, a composite value, or even another pointer.
- Traversal in a double-linked list is bidirectional.
- Deleting of a node at either end of a doubly-linked list is straight forward.

Sentinel Node: NIL

- A sentinel node is a programming idiom used to replace null-pointers with a special token denoting "no value" or nil.
- Sentinel nodes behave like null-pointers. However, unlike null-pointers, which refer to nothing, sentinels denote proper, yet empty, values.

End of Revision 1