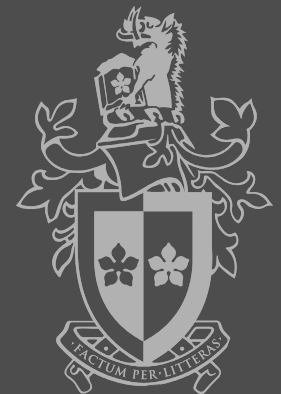


COS30008 Data Structures and Patterns

Lecture 4 - Basic Concepts





Programming Paradigms

- Imperative style:

- program = algorithms + data

- Functional style:

- program = function • function

- Logic programming style:

- program = facts + rules

- Object-oriented style:

- program = objects + messages

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).



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.



C++ Array Examples

```
const unsigned int buf_size = 512, max_files = 20;
int staff_size = 27;

const unsigned int sz = get_size();
// const value not known until runtime

char input_buffer[buf_size];
string fileTable[max_files + 1];
int chess_board[8][8]; // two-dimensional array

double salaries[staff_size]; // error: non const variable
int test_scores[get_size()]; // error: non const expression
int vals[sz]; // error: sz not known until runtime
```



Explicitly Initialized Arrays

- `int arr1[20] = { 1 };`

- Array of 20 integers with `arr1[0] == 1` and `arr1[i] == 0` for $1 < i < 20$.

- `int arr2[] = { 1, 2, 3, 4, 5 };`

- Array of 5 integers initialized 1, 2, 3, 4, and 5.

- `int arr3[10];`

- Array of 10 integers, none initialized.

- Data members of array type are

- Not initialized if the base type is a built-in data type

- Initialized (using the default constructor) if the base type is a class type.

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 ('_')

Arrays in C++



```
main.cpp x
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int bucky[5] = {66, 75, 2, 47, 99};
7
8      cout << bucky[3] << endl;
9  }
```



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 length 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



A C-String Array

```
#define MAX_ID_LENGTH 5
```

```
char keywords[][MAX_ID_LENGTH] =  
{  
    "if",  
    "then",  
    "else"  
};
```

The first dimension
of an array can be
unspecified!



String

- Another way, maybe a better way is using *character pointer* to keep track of a string.

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

int main ()
{
    string name;

    cout << "Please, enter your full name: ";
    cin>>name;

    cout << "Hello, " << name << "!\n";

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

Can this
program be
compiled
correctly and run
correctly?

String



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

int main ()
{
    string name;

    cout << "Please, enter your full name: ";

    getline (cin,name);
    cout << "Hello, " << name << "!\n";

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

Using getline,
we can get the
whole name
including spaces

String



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

int main ()
{
    string name;

    cout << "Please, enter your full name: ";

    getline (cin,name);
    cout << "Hello, " << name << "!\n";
    cout << "The lenght of the name is " << name.length() << "!\n";

    cout << "The second letter of the name is " << name.at(1)<< "!\n";
    cout << "The 3rd letter of the name is " << name[2]<< "!\n";

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

Sum



- The sum of all elements of an array

```
int Sum( int a[], unsigned int n )  
{  
    int Result = 0;  
  
    for ( unsigned int i = 0; i < n; i++ )  
    {  
        Result += a[i];  
    }  
  
    return Result;  
}
```

$$\sum_{i=0}^n a_i$$

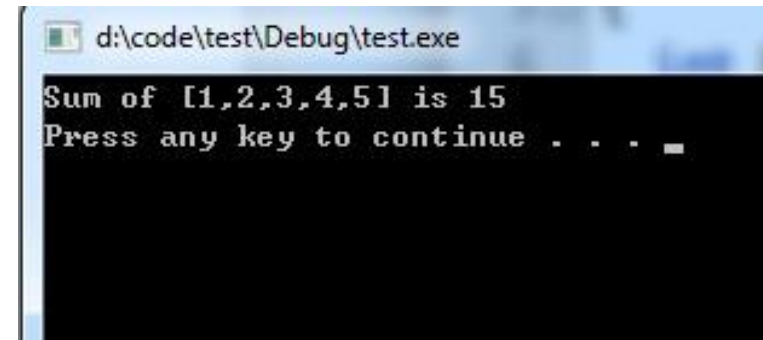


Testing sum

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

int Sum(int *a, unsigned int n)
{
    int Result = 0;
    for (unsigned int i =0; i<n; i++){
        //Result += a [i]; //
        Result+=*(a+i);
    }
    return Result;
}

int main()
{
    int a[]={1,2,3,4,5};
    int val=Sum(a,5);
    cout<<"Sum of [1,2,3,4,5] is "<<val<<endl;
    system("PAUSE");
    return 0;
}
```



This is not a good practice and only available for MSVC



Associative Array

- An associate array is a **map** in which elements are indexed by a key rather than by an integer index number.

- Example:

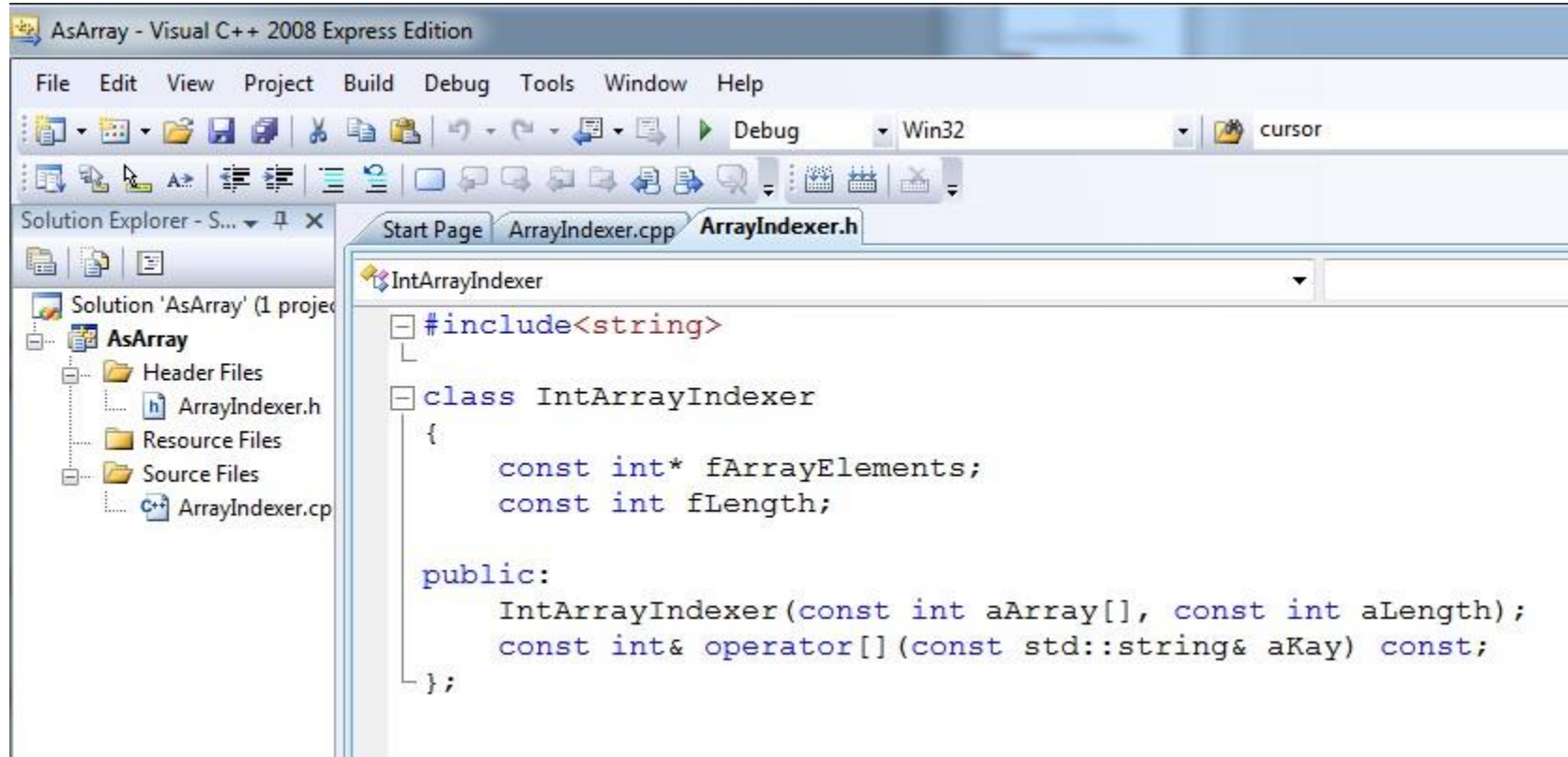
□ $a = \{ \{ "u", 345 \}, \{ "v", 2 \}, \{ "w", 39 \}, \{ "x", 5 \} \}$

□ $a["w"] = 39$



From Indices to Keys

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



```
AsArray - Visual C++ 2008 Express Edition

File Edit View Project Build Debug Tools Window Help

Solution Explorer - S...
Solution 'AsArray' (1 project)
  AsArray
    Header Files
    ArrayIndexer.h
    Resource Files
    Source Files
    ArrayIndexer.cpp

Start Page ArrayIndexer.cpp ArrayIndexer.h

IntArrayIndexer

#include<string>

class IntArrayIndexer
{
    const int* fArrayElements;
    const int fLength;

public:
    IntArrayIndexer(const int aArray[], const int aLength);
    const int& operator[](const std::string& aKey) const;
};
```



From Indices to Keys

```
#include<iostream>
#include<stdexcept>
#include"ArrayIndexer.h"
```

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

```
using namespace std;
```

```
IntArrayIndexer::IntArrayIndexer(const int aArray[], const int aLength):
fArrayElements(aArray), fLength(aLength){}
```

We must use member initializer to initialize const instance variables!



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.



Testing the Indexer

```
int main(){

    int a[]={1,2,3,4,5};
    IntArrayIndexer indexer(a,5);

    string keys[]{"0","1","2","3","4"};

    int Sum=0;

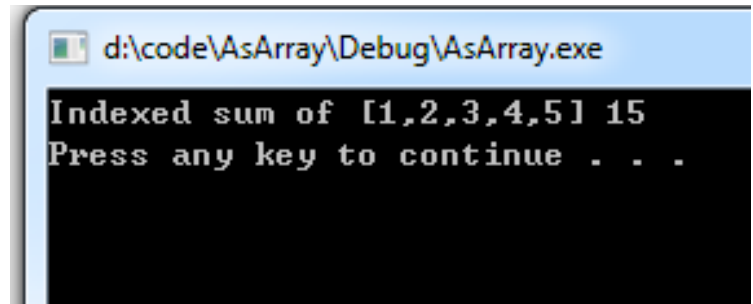
    for(int i=0; i<5; i++){

        Sum+=indexer[keys[i]];

    }
    cout<<"Indexed sum of [1,2,3,4,5] "<<Sum<<endl;
    //cout<<"Indexed "<<indexer[keys[0]]<<endl;
    //cout<<"Indexed key "<<keys[0]<<endl;

    system("PAUSE");
    return 0;

}
```



d:\code\AsArray\Debug\AsArray.exe

Indexed sum of [1,2,3,4,5] 15
Press any key to continue . . .



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

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	

**template
containers**

Input Iterator



Expression	Effect
<code>*iter</code>	Provides read access to the actual element
<code>iter->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)

Forward Iterator



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

Random Access Iterator



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

A Read-Only Forward Iterator



```
age ArrayIterator.cpp ArrayIterator.h
ayIterator
1 class IntArrayIterator
2 {
3     const int* fArrayElements;
4     const int fLength;
5     int fIndex;
6
7 public:
8     IntArrayIterator(const int aArray[], const int aLength, int aStart=0);
9
10    const int& operator*() const;
11    IntArrayIterator& operator++();
12    IntArrayIterator operator++(int);
13
14    bool operator==(const IntArrayIterator& aOther) const;
15    bool operator!=(const IntArrayIterator& aOther) const;
16
17    IntArrayIterator begin() const;
18    IntArrayIterator end() const;
19 };
```



Forward Iterator Constructor

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

```
Page ArrayIterator.cpp ArrayIterator.h  
ArrayIterator  
1 #include<iostream>  
2 #include"ArrayIterator.h"  
3  
4  
5 using namespace std;  
6  
7 IntArrayIterator::IntArrayIterator(const int aArray[], const int aLength, int aStart):  
8     fArrayElements(aArray), fLength(aLength)  
9 {  
10     fIndex = aStart;  
11 }  
12
```

use member initializer to initialize const instance variables!

Not repeating the default value

The Dereference Operator



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

- 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.



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.

Putting Everything Together



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

Iterators in C++





What is a Design Pattern?

- Christopher Alexander (architect and design theorist) 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.”
- Software design patterns are **abstractions** that help structure system designs.
- A design pattern is neither a static solution, nor is it an algorithm.



Why Design Pattern

- A **pattern** is a way to describe and address by
 - name (mostly a simplistic description of its goal),
 - a repeatable solution or approach to a common design problem, that is, a common way to solve a generic problem (how generic or complex, depends on how restricted the target goal is).
- Patterns can emerge on their own or by design.
 - This is why design patterns are **useful** as an abstraction over the implementation and a help at design stage.
 - With this concept, an easier way to facilitate communication over a design choice as normalization technique is given so that every person can share the design concept.



Design Pattern Elements

■ Problem/requirement

- To use a design pattern, we need to go through a mini analysis design that may be coded to test out the solution. This section states **the requirements of the problem** we want to solve. This is usually a common problem that will occur in more than one application.

■ Forces

- This section states the **technological boundaries**, that helps and guides the creation of the solution.

■ Solution

- This section describes how to write the code to solve the above problem. This is the design part of the design pattern. It may contain class diagrams, sequence diagrams, and or whatever is needed to **describe how to code the solution.**

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.

Categories of Design Pattern



- Depending on the **design problem** they address, design patterns can be classified in different categories, of which the main categories are:
 - Creational Patterns
 - Structural Patterns
 - Behavioural Patterns



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.



Creational Patterns

- Builder: Used to separate the construction of a complex object from its representation so that the same construction process can create different objects representations.
 - Factory: A utility class that creates an instance of a class from a family of derived classes
 - Prototype: Used in software development when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects.
- Other Creational Pattern: Singleton

Structural Patterns



- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy
- Curiously Recurring Template
- Interface-based Programming (IBP)



Behavioural Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor
- Model-View-Controller (MVC)

https://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns

File Input/Output



Input and output stream

- C++ uses a **convenient abstraction** called *streams* to perform input and output operations in sequential media such as the screen, the keyboard or a file.
- A *stream* is an entity where a program can either insert or extract characters to/from.
- Streams are a source/destination of characters, and that these characters are provided/accepted sequentially (i.e., one after another).



Input and output stream

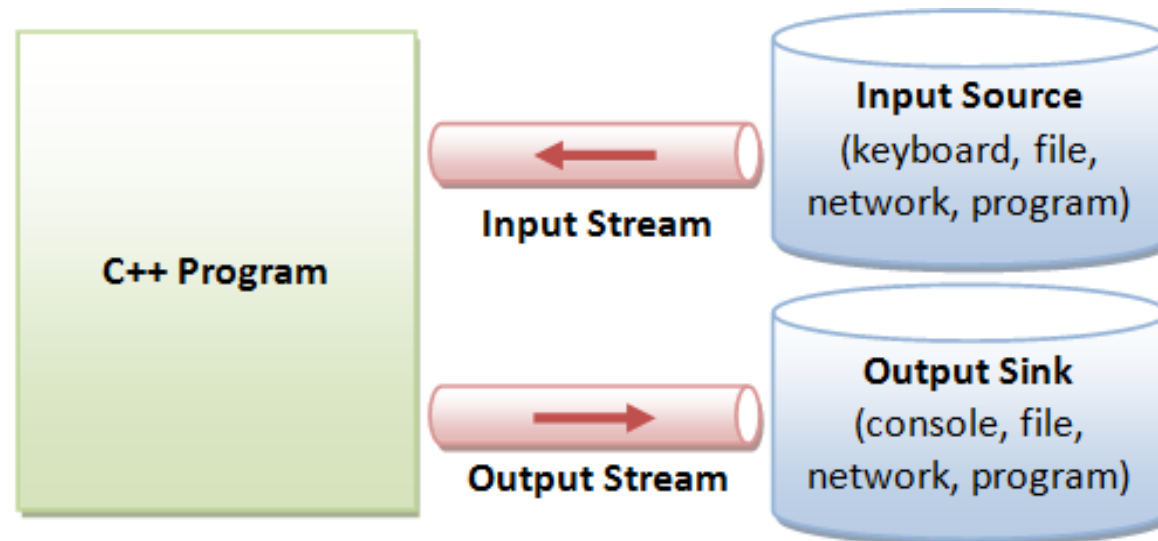
- The **standard library** defines a handful of stream objects that can be used to access what are considered the standard sources and destinations of characters by the environment where the program runs
- Two main operators are stream insertion << and stream extraction >> operators.
 - The << operator 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

stream	description
cin	standard input stream
cout	standard output stream



Input and output stream

- **cin** extraction always considers spaces (whitespaces, tabs, new-line...) as **terminating** the value being extracted, and thus extracting a string means to always extract a single word, not a phrase or an entire sentence.



Internal Data Formats:

- Text: `char`, `wchar_t`
- `int`, `float`, `double`, etc.

External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

Example



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



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 I/O library
#include <fstream>             // include file I/O 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 }
```

File I/O

■ Read from a file.

```
1 #include<iostream>
2 #include<fstream>
3 #include<string>
4
5 using namespace std;
6
7 int main(int argc, char* argv[])
8 {
9     string s;
10
11     ifstream rFile;
12     rFile.open("myfile2.txt");
13
14     if(rFile.fail())
15     {
16         cout<<"Warning: can't open the file!"<<endl;
17
18         system("pause");
19         return 2; //terminate the program
20     }
21
22     while(rFile>>s)
23     //while(getline(rFile,s))
24     {
25         cout<<s<<endl;
26     }
27
28     rFile.close();
29     system("pause");
30     return 0;
31 }
```





File I/O

- Read from a file as an argument.

```
1 #include<iostream>
2 #include<fstream>
3
4 using namespace std;
5
6 int main(int argc, char* argv[])
7 {
8
9     //...
10    //...
11    //set up input file,
12    ifstream iFile; //declare an input file instance/object.
13
14
15    iFile.open(argv[1], ifstream::in); //open an input text file
16
17    if(!iFile.good())
18    {
19        //file can't be opened properly
20        cout<<"Warning: can't open input file" << argv[1]<<endl;
21        return 2; //program failed
22    }
23
24    //...
25    //...
26
27
28    iFile.close();
29    system("pause");
30
31    return 0;
32 }
```



File-based Character Counter

```
40     CharacterCounter lCounter;  
41  
42     unsigned char lChar;  
43  
44     while ( lInput >> lChar )  
45     {  
46         lCounter.count( lChar );  
47     }  
48  
49     lOutput << lCounter;
```

- We read input through input file stream lInput and write results to output file stream lOutput.

I/O Manipulator Tests



```
1 #include<iostream>
2 #include<iomanip>
3
4 using namespace std;
5
6 int main(){
7
8     cout<<"the number 12345"<<endl;
9     cout<<"in HEX "<<hex<<12345<<" and "<<dec <<12345<<endl;
10
11     cout<<"The number 100 as 8 digit hexadecimal number:\t"
12         <<setw(8)<<setfill('0')<<hex<<100<<endl;
13     //Sets the field width to be used on output operations.
14
15     cout<<"The number 28 as 8 digit hexadecimal number:\t"
16         <<setw(7)<<setfill('0')<<hex<<28<<endl;
17     cout<<"The number 28 as 8 digit hexadecimal number:\t"
18         <<setw(6)<<setfill('0')<<hex<<28<<endl;
19     cout<<"The number 28 as 8 digit hexadecimal number:\t"
20         <<setw(5)<<setfill('-')<<hex<<28<<endl;
21     cout<<"The number 28 as 8 digit hexadecimal number:\t"
22         <<setw(4)<<setfill(' ')<<hex<<28<<endl;
23
24     system("pause");
25     return 0;
26 }
```




Using I/O manipulators

■ Formatting an integer value:

```
int lIValue = 12;
cout << lIValue << endl;
cout << hex << lIValue << endl;
cout << dec << lIValue << endl;
cout << setw( 3 ) << lIValue << endl;
cout << setw( 3 ) << setfill( '0' ) << lIValue << endl;
```

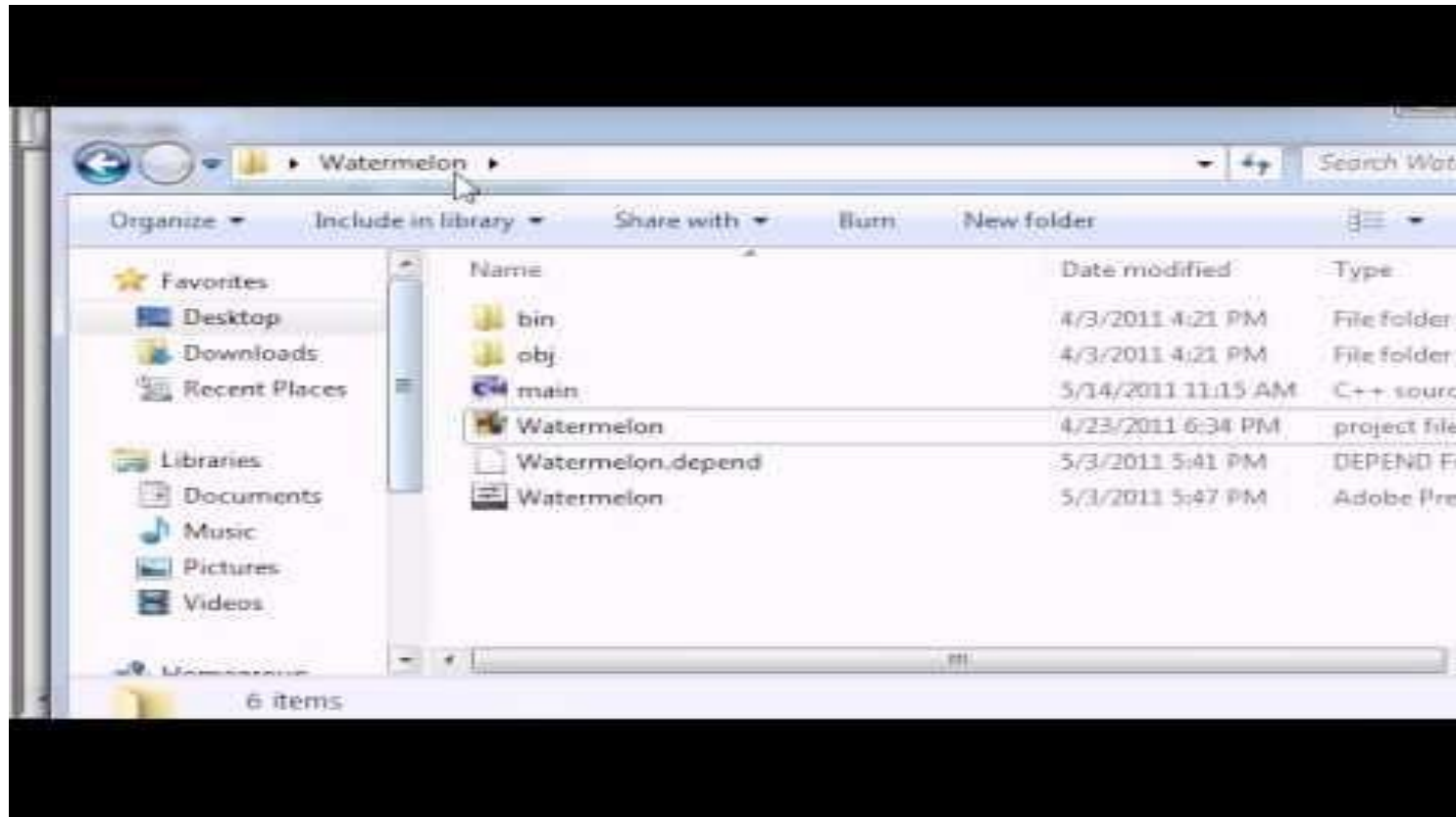
```
12
c
12
 12
012
```

■ Formatting a floating-point value:

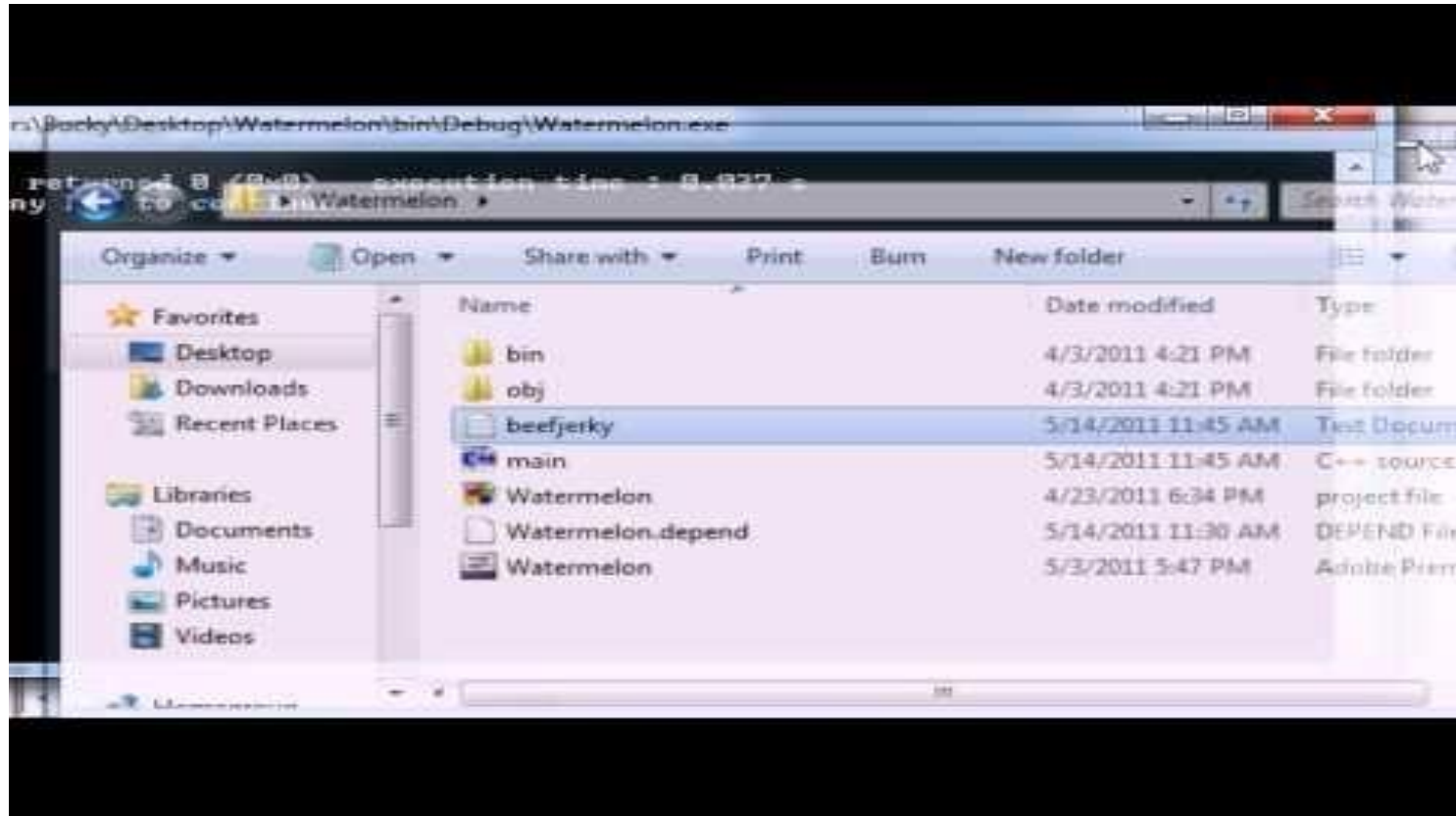
```
double lDValue = 3.141592654;
cout << lDValue << endl;
cout << setprecision( 3 ) << lDValue << endl;
cout << setprecision( 9 ) << lDValue << endl;
cout << fixed << setprecision( 4 ) << lDValue << endl;
cout << scientific << lDValue << endl;
cout << scientific << setprecision( 15 ) << lDValue << endl;
```

```
3.14159
3.14
3.14159265
3.1416
3.1416e+00
3.141592654000000e+00
```

Working with Files in C++



C++ File Handling Tips

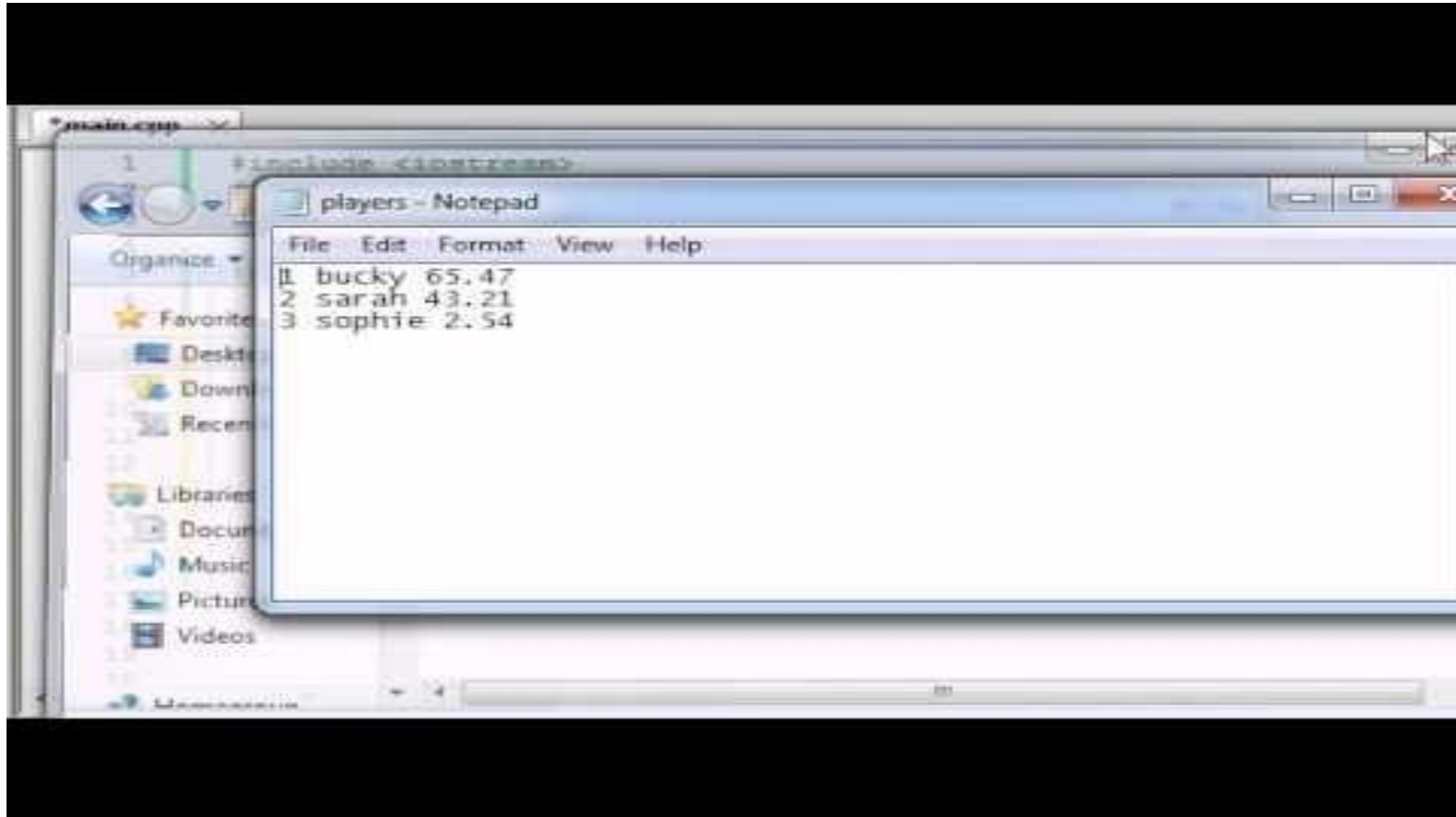


C++ Writing Custom File Structures

A screenshot of a code editor window titled "main.cpp". The code is as follows:

```
3 using namespace std;
4
5 int main() {
6     ofstream theFile("players.txt");
7
8     cout << "Enters playrs ID, Name, and Money" << endl;
9     cout << "press Ctrl+Z to quit\n" << endl;
10
11     int idNumber;
12     string name;
13     double money;
14
15
16
17
18 }
19
20
21
```

C++ Reading Custom File Structures



End of Basic Concepts

