Lecture 4
Useful Features in C++

Towards mastery of C++…

Lecture Overview

- **String** class
  - Declaration and usage
- **Stream** classes
  - Standard input / output stream
  - String stream
- **Template**
- **Standard Template Library** (STL)
  - Container Class
  - Iterator

String

An object-oriented version of string

The "C String" : Limitation

- **Recap:**
  - String in C is **a special case of character array**
  - Use "*string terminator*" ('**`\0`**') to delimit a string
  - Use functions in **`<string.h>`** for string manipulations

- **Limitation:**
  - Fixed size upon declaration
  - The usage of null terminator is error prone
  - No assignment between strings

String in C++

HEADER

```
#include <string>
```

FEATURES

Constructors:

Default constructor
String literal constructor

Operators:

"="	: Assign value to a string object
">", "<", "=="	: Comparison between string objects
"+"	: Defined as string concatenation
"`[index]`"	: Access character at position **`index`**

Methods:

***`size`*`()`**
***`substr`*`(pos, nChar)`**
etc…

You can check online reference to learn more built-in methods of string class

String in C++ : Example 1

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

int main() {
  string str1;
  string str2("xyz");

  str1 = "abc";
  cout << "S1 = " << str1 << endl;
  cout << "S2 = " << str2 << endl;
  cout << "S1 + S2 = " << str1 + str2 << endl;
  cout << "S2 + S1 = " << str2 + str1 << endl;

  if (str1 > str2)
    cout << "S1 > S2" << endl;
  else
    cout << "S1 <= S2" << endl;
}
```

str1 is an empty string

str2 is initialized with the string “xyz”

Use “=“ to assign a string to **str1**

Output:

```
S1 = abc
S2 = xyz
S1 + S2 = abcxyz
S2 + S1 = xyzabc

S1 <= S2
```

String in C++ : Example 2

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

int main() {
  string str1("abcd");
  string str2("efgh");
  string str3;

  str3 = str1 + str2;

  cout << str3 << endl;
  cout << str3.size() << endl;
  cout << str3[4] << endl;
  cout << str3.at(4) << endl;
  cout << str3.substr(2, 5) << endl;
}
```

Addition returns a newly concatenated string

0-based indexing

Output:

```
abcdefgh
8
e
e
cdefg
```

String in C++ : Input/Output

- **Output:**
  - Use the insertion operator **<<** to output string objects
- **Input:**
  - Use the extraction operator **>>** to input string objects
    - Designed to **read a single word only**
  - Use the predefined ***getline()*** function to read in one whole line

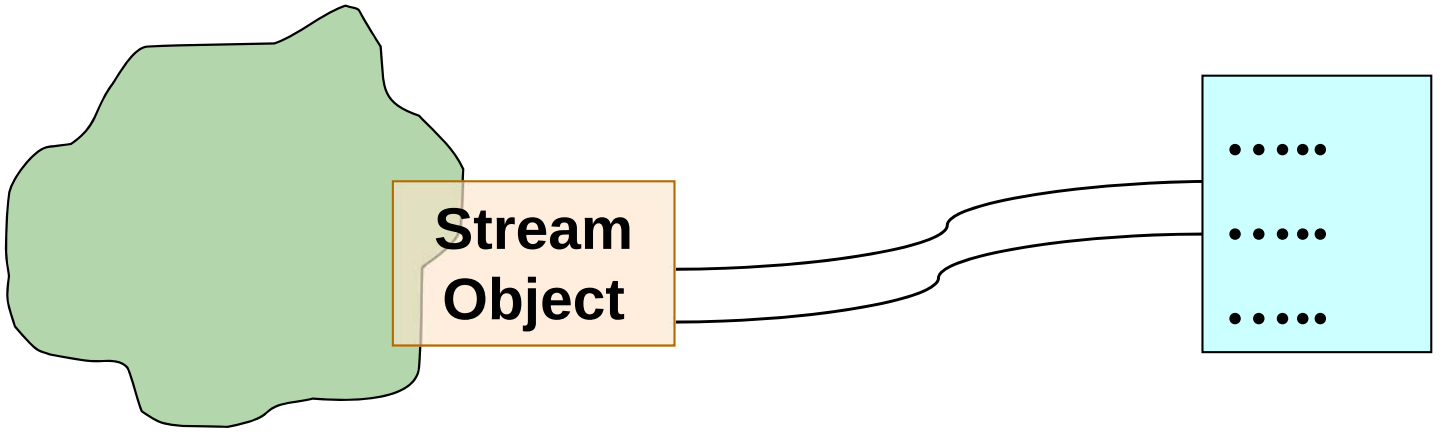
```
string str;

getline(cin, str);
```

Input / Output Stream

Managing input and output in C++

Standard C++ IO Stream

- C++ provides an standardized conceptual model to manage all kinds of input and output "devices"
- A ***stream object*** wraps around any actual device and provide facility for input/output



Executing Process
i.e. your program

Actual Devices
e.g. file, keyboard, monitor

Input Stream objects: Basic Behavior

- Input stream objects:
  - Read values using the ***extractor operator*** (">>")
  - **Syntax:**

```
inputStreamObject >> variable;
```

    - Value will be read from ***inputStreamObject*** and stored in ***variable***
    - Type of variable is determined automatically
  - **Behavior:**
    - Numerical and character values:
      - Skip all white spaces (blank, tab, newline)
    - String:
      - Read one word only

Output Stream objects: Basic Behavior

- Output stream objects:
  - Print values using the ***insertor operator*** ("<<")
  - **Syntax:**

    ```
    outputStreamObject << varOrValue;
    ```

    - Values stored in variable or literals will be converted to string automatically
  - User defined data type can provide their own string conversion through operator overloading (not covered)

Output Stream : Manipulator

- **IO Manipulator** modifies the behavior of the output stream

HEADER

```
#include <iomanip>
```

Usage

```
cout << manipulator;
cout << data;
```

The output format of **data** is modified according to the ***manipulator*** supplied

- Common manipulators:
  - **`endl`** : Flush the output
  - ***`setprecision`*`(n)`** : Set number of significant digits to **`n`**
  - ***`setw`*`(n)`** : Set field width to **`n`**
  - **`boolalpha`** : prints boolean value as "**`true`**"/"**`false`**"
  - etc

Output Stream Manipulator: Example

```
double d = 3.141592;
bool b = true;

cout << d << endl;

cout << setprecision(3);
cout << d << endl;

cout << setw(10) << 1234 << endl;
cout << 1234 << endl;

cout << b << endl;
cout << boolalpha << b << endl;
```

Output:

```
3.14159
3.14
      1234
1234
1
true
```

String Stream

Reading and writing to string

String Stream

- Stream objects have the useful ability to convert data to the required type automatically

- Is it possible to provide that functionality for string objects as well? e.g.
  - Convert string into other data type
  - Convert other data type into string

- C++ provides **String Stream** for the above functionalities

String Stream : Input String Stream

- **Header File:**
  - `#include <sstream>`
- `istringstream` (Input String Stream)
  - Use String as input stream
  - Can be constructed from raw string e.g. “Hello” or string object

```
istringstream instr("Now 2.14 30");

string s;
double d;
int x;

instr >> s;
instr >> d;
instr >> x;
```

```
s = "Now"
d = 2.14
x = 30
```

Input String Stream : Example

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

int main() {
    int sentenceCount = 1, wordCount;
    string word, sentence;

    while (getline(cin, sentence)) {
        istringstream iss(sentence);
        wordCount = 0;
        while (iss >> word)
            wordCount++;
        cout << "Sentence No." << sentenceCount;
        cout << " has " << wordCount << " words.\n";
        sentenceCount++;
    }
    return 0;
}
```

Header File

A program to count the number of words in each input sentence

An input string stream object can be constructed from a string object / literals

We can extract input from input string stream just like any other input stream object

String Stream : Output String Stream

- **Header File:**
  - `#include <sstream>`
- `ostringstream` (Output String Stream)
  - String stream for output
  - Use **<<** insertor to place output into a string output stream object
  - Use ***str***() method to get a string object back from the string output stream object

```
string s = "Now";
double d = 3.14;
int x = 30;
ostringstream outstr;

outstr << s << " " << d << " " << x << endl;
string t = outstr.str();

cout << t;
```

Output:

```
Now 3.14 30
```

Output String Stream : Example

Example usage:
Concatenate a number of values of different data types as a string

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

class BankAcct {
    // other code not shown
public:
   string toString() {
       ostringstream os;

       os << "Acct No: " << _acctNum;
       os << " Balance: " << _balance;

       return os.str();
   }
};

int main() {
    BankAcct ba1(1234, 300.50);

    cout << ba1.toString() + "**" << endl;
}
```

An output string stream object

These output are captured in the output string stream object

Extract the output captured so far as a string

`toString()` is more flexible than **`print()`** as we can further manipulate the output string before printing

Templates

One definition, many data types

Generic Programming

- There are programming solutions that are applicable to a wide range of different data types
  - The code is exactly the same other than the data type declarations

- **In C**: No easy way to exploit the similarity
  - One implementation for each data type

- **In C++**: Use the **template code** mechanism
  - Code is write once only
  - Data types can be specified later during actual usage

Example : A pair of integers

```
class IntPair {
private:
  int _first;  // first value
  int _second; // second value

public:
  IntPair(int a, int b) : _first(a), _second(b) {}

  int getFirst() { return _first; }
  int getSecond() { return _second; }
};

int main() {
  IntPair point(4, 5);

  cout << "x=" << point.getFirst() << endl;
  cout << "y=" << point.getSecond() << endl;

  return 0;
}
```

IntPair can be used to store one pair of integers

Question:

What if we need to store a pair of strings?

Example : A pair of strings

```
class StringPair {
private:
  string _first;              // first value
  string _second;             // second value

public:
  IntPair(string a, string b)
  : _first(a), _second(b) {}

  string getFirst() { return _first; }
  string getSecond() { return _second; }
};

int main() {
  StringPair name("Skywalker", "Luke");

  .......
}
```

Difference in data type declarations only!

Template : Class Pair

```
template <typename T>
class Pair {

private:
  T _first;  // first value
  T _second;  // second value

public:
  Pair(T a, T b) : _first(a), _second(b) {}

  T getFirst()  { return _first;  }
  T getSecond() { return _second; }

};
```

Keyword to indicate template implementation

Indicates that the user will specify a **typename** (data type), **T**

These will be substituted with actual data type later

- **T** is a **type name variable**
  - User can substitute **T** with actual data type later!

Template : Pair Usage Example

```
template <typename T>
class Pair { // definition not shown };

int main() {
    Pair<int> twoInt( -5, 20 );
    Pair<string> twoStr( "Turing", "Alan");

    // You can have pair of any (same) data types for now!
    // ..........
}
```

- During compilation time:
  - Different versions of the **`Pair`** class are generated:
    - With type name **`T`** substituted with actual data type
  - The above code generates **two versions** of the **`Pair`** class automatically

Template: Multiple Type Names

- The class **Pair** can be even more general by allowing different types for its two elements

```
template <typename T1, typename T2>
class Pair {

private:
  T1 _first;  // first value
  T2 _second; // second value

public:
  Pair(T1 a, T2 b) : _first(a), _second(b) {}

  T1 getFirst()  { return _first;  }
  T2 getSecond() { return _second; }
};
```

Template : Pair Usage Example 2

```
template <typename T1, typename T2>
class Pair { // definition not shown };

int main() {
    Pair<string, int> person("Alan Turing", 41);
    Pair<double, string> constant(3.14159, "PI");

    // Now you can have pair of any data types!
    // ..........
}
```

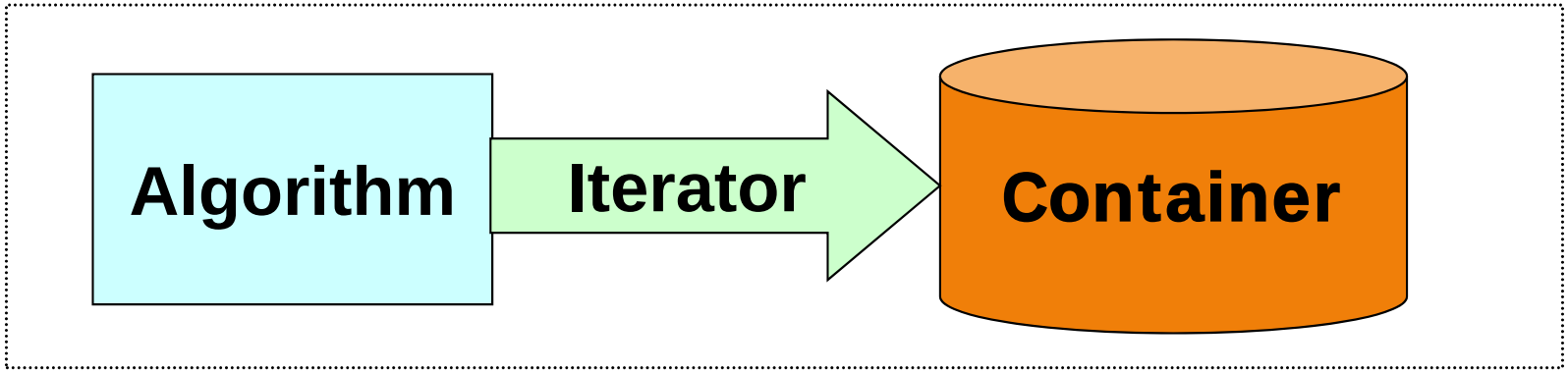
- The **`Pair`** class is indeed very useful!
  - C++ has a built-in version with slight differences
  - You can find it under the **`<utility>`** header

Standard **T**emplate **L**ibrary

STL

Standard Template Library (STL)

- A major part of the C++ standard library
- Consists of three components:
  1. **Container**
  2. **Iterator**
  3. **Algorithm**
- Defined as template classes
- Relationship between STL components:



STL Containers

- **Containers:**
  - Object that contains other objects
  - Represent general **data structures** in computing
  - Most of the data structures covered in this course are available as STL containers ☺
- Main features:
  - Template class
    - Can be used for built-in and user defined data types
  - All containers supports a set of general methods
    - **`size()`** : number of elements
    - **`empty()`** : is the container empty?
    - Etc
  - Specialized methods are defined for individual container classes

STL Containers

- **Vectors**
- **Lists**
- **Double-Ended Queues**
- **Stacks**
- **Queues**
- Priority Queues
- Sets
- Multisets
- Maps
- Multimaps

CS1020E domain

CS2010 domain

STL Vector

- Header File:

```
#include <vector>
```

- Defined as template class

```
vector<int> intVector;
```

- Stores contiguous elements as an array
  - i.e. **object oriented implementation of an array**
- **Advantages:**
  - Fast insertion and removal of at the end of vector
  - Support dynamic number of elements
  - Automatic memory management
- Vector is the simplest STL container class, and in *some* cases the most efficient

STL Vector : Commonly used methods

`vector<T> v`	Construct a vector ***v*** to store elements of type **T**
***size*()**	returns the number of items
***empty*()**	returns **true** if the vector has no elements
***clear*()**	removes all elements
***at*(*n*) or [*n*]**	returns an element at position ***n***
***front*()**	returns a reference to the first element
***back*()**	returns a reference to the last element
***pop_back*()**	removes the last element
***push_back*(*e*)**	add element *e* to the end

STL Vector : Example

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

int main() {
    vector<int> intV;

    cout << "intV size = " << intV.size() << endl;
    for (int ix = 0; ix != 5; ++ix)
        intV.push_back(ix);
    cout << "intV size = " << intV.size() << endl;

    if (!intV.empty()) {
        cout << "intV = [ ";
        for (int ix = 0; ix != intV.size(); ++ix)
            cout << intV[ix] << " ";
        cout << "]" << endl;
    }

    intV.pop_back();
    cout << "intV size = " << intV.size() << endl;
}
```

output:

```
intV size = 0
intV size = 5
intV = [ 0 1 2 3 4 ]
intV size = 4
```

STL Iterator

Accessing items in container

Recap: Operations on Pointer (L1)

```
int a[] = {1,2,3,4,5,6,7,8,9};
int *p;

for (p = a; p != a+9; ++p) {
    cout << *p << endl;
}
```

A pointer can be

1. initialized to point to the **begin** of the container (the array)
2. compared with another pointer to see whether it has come to the **end** of the container
3. incremented (**++**) to point to the next element in the container
4. dereferenced (*****) to access the element in the container

STL Iterator

- **Iterator** is an abstraction:
  - Resembles a *pointer* that points into an *array*
- Iterator can be used to access and manipulate elements in a container:
  - Elements are accessed in a sequence regardless of actual organization
- Allows the programmer to define common operations (algorithm) for container without worrying about the underlying details
  - Some of these common operations are implemented as **STL Algorithm**

Iterator

- All container classes provide their own iterators
  - Declaration:

    ```
    container::iterator iterator_variable;
    ```

  - Example:

    ```
    vector<int>::iterator myIter;
    ```

- All container classes define following methods
  - ***begin()*** returns an iterator that points at the beginning of the container
  - ***end()*** returns an iterator that points at one element pass the end of the container
    - Usually used as a termination condition for loops

STL Vector : Iterator Related Methods

`begin()`

- **Returns**: an iterator to the first element

`end()`

- **Returns**: an iterator ***beyond*** the last element

`insert(iter, item)`

- **Purpose**: inserts element *item* before the element indicated by iterator *iter*
- **Returns**: an iterator to the newly inserted item

`erase(iter)`

- Purpose: removes element indicated by iterator *iter*
- **Returns**: an iterator points to the item **beyond the** removed element.

Operations on Iterator

- Let **`iter`** be an iterator of a container

`*iter`	Accesses the item pointed by the iterator
`iter++` or **`++iter`**	Moves the iterator to point to the next item in the container
`iter--` or **`--iter`**	Moves the iterator to point to the next item in the container
`iter1 == iter2`	Returns true when both iterators point at the same item in the container
`iter1 != iter2`	Returns true when the two iterators do not point at the same item in the container

Iterator : Example

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

void print_vector(vector<int>& iV) { // important, byRef
    vector<int>::iterator iter;
    cout << "V = [ ";
    for (iter = iV.begin(); iter != iV.end(); ++iter)
        cout << *iter << " ";
    cout << "]" << endl;
}

int main() {
    vector<int> intV;
    for (int ix = 0; ix != 5; ix++)
        intV.push_back(ix);
    print_vector(intV);
}
```

The vector:

```
V = [ 0 1 2 3 4 ]
```

output:

```
V = [ 0 1 2 3 4 ]
```

Iterator : Example (cont)

```
...

void print_vector(vector<int>& iV) {...}

int main() {
    vector<int> intV;
    for (int ix = 0; ix != 5; ++ix)
          intV.push_back(ix);
    print_vector(intV);

    vector<int>::iterator myIter = intV.begin();

    intV.insert(myIter, 123); // caution!
    print_vector(intV);
```

output:

```
V = [ 0 1 2 3 4 ]
V = [ 123 0 1 2 3 4 ]
```

Iterator : Example (cont)

```
    // continued from previous slide ...

    myIter = intV.begin(); // Important: reset myIter
    myIter++;
    intV.erase(myIter);
    print_vector(intV);

    myIter = intV.begin(); // Reset!
    cout << *myIter << endl;

    return 0;
}
```

output:

```
... ...
V = [ 123 1 2 3 4 ]
123
```

- Most built-in methods (e.g. ***insert***( ), ***erase***( )) invalidates the iterator after the operation:
  - Make sure you “reset” the iterator before the next usage!
  - Alternatively, use the return result from ***insert***(), ***erase***() methods

STL Algorithms

Built-in algorithms for manipulating elements in container

STL Algorithm: Sorting

- Header File:

```
#include <algorithm>
```

- Sorting Methods:
  - `void sort(iterator start, iterator end);`
  - Sort the items between start and end in the container in ascending order

  - `void sort(iterator start, iterator end, StrictWeakOrdering cmp);`
  - Sort the items between start and end in the container using `cmp` as comparator function

STL Sorting : Ascending Order

```
#include <algorithm>
#include <vector>
using namespace std;

int main() {
   vector<int> v;
   v.push_back(23);
   v.push_back(-1);
   // add in more items

   sort(v.begin(), v.end());

   cout << "After sorting: ";
   for (unsigned int i = 0; i < v.size(); i++)
      cout << v[i] << endl ;
}
```

sort integers into ascending order

STL Sorting : Descending order

- To sort in *descending* order:

```
sort(v.begin(), v.end(), greater<int>());
```

  - `greater<int>()` is known as **comparison functor**

- Other STL comparison **functors** are:
  - `equal_to<T>`
  - `not_equal_to<T>`
  - `less<T>`
  - `greater_equal<T>`
  - `less_equal<T>`

STL Algorithms: Commonly Used

Sort

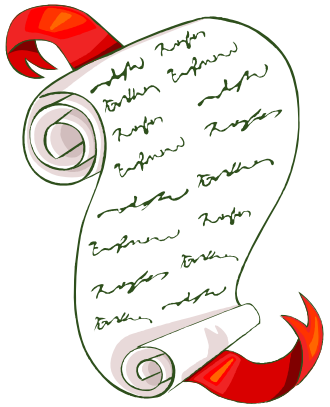
1. `sort`
2. `stable_sort`
3. `partial_sort`
4. `partial_sort_copy`
5. `is_sorted`
6. `nth_element`

Binary search

1. `lower_bound`
2. `upper_bound`
3. `equal_range`
4. `binary_search`

`merge`

`inplace_merge`



Self-study

User Defined: Strict Weak Ordering

- A binary relation (**op**) that has the following properties:
  - For all *x*, it is not the case that (*x* **op** *x*)
  - For all *x* ≠ *y*, if (*x* **op** *y*) then it is not the case that (*y* **op** *x*)
  - For all *x*, *y*, and *z*, if (*x* **op** *y*) and (*y* **op** *z*) then (*x* **op** *z*)
- Examples of such binary relations:
  - **<** (smaller than)
  - **>** (larger than)
- Counter examples:
  - **>=** (larger than or equal to)

User defined sorting requires a function with "strict weak ordering" property to work

Self-study

STL Sorting: User Defined Comparison

```
class BankAcct {
public:
   // unchanged declarations not shown
    // from lecture 2
   double getBalance() { return _balance;}
};
```

An extra accessor method to read the balance from `BankAcct` object

```
bool poorer(BankAcct a, BankAcct b) {
   return a.getBalance() < b.getBalance();
}

bool richer(BankAcct a, BankAcct b) {
    return a.getBalance() > b.getBalance();
}
```

Comparison functions to order two accounts. Both satisfy the "strict weak ordering" property

Self-study

STL Sorting: `BankAcct` Example

```
#include <algorithm>
// other includes are not shown

int main() {
    vector< BankAcct > vba;

    vba.push_back(BankAcct(123,6745.35));
    vba.push_back(BankAcct(678,10.50));
    vba.push_back(BankAcct(102,25.75));
    // ... add more bank acct objects

    sort(vba.begin(), vba.end(), poorer);

    // ... print the bank acct objects to check
    // ... should be sorted by balance amount

    return 0;
}
```

Use the newly defined comparison function to order the objects!

File Stream

For your own reading

Self-study

File Stream

- **Header File:**
  - `#include <fstream>`
- **`ifstream`**
  - Input file stream
- **`ofstream`**
  - Output file stream
- Opening a file stream:

```
ifstream inFile("input.txt");
```

OR

```
ifstream inFile;
inFile.open("input.txt");
```

  - Similar for `ofstream`
- Closing a file stream:

```
inFile.close();
```

Similar for `ofstream`

Self-study

File Stream : Example 1

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

int main() {
  ifstream readFile("in.txt") ;
  ofstream writeFile("out.txt");
  int x;

  while (readFile >> x)
    writeFile << x << "*";
  writeFile << endl;

  readFile.close();
  writeFile.close();

  return 0;
}
```

in.txt:

```
1 2 3
 4 5
6      7      8
```

out.txt:

```
1*2*3*4*5*6*7*8
```

- Observe:
  - The behavior of the extractor **(">>")** operator as summarized in slide 18
  - Behavior of **cout** (standard output strem) is similar

Self-study

File Stream : Example 2

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

int main() {
  ifstream readFile("in.txt") ;
  ofstream writeFile("out.txt");
  char x;

  while (readFile >> x)
    writeFile << x << "*";
  writeFile << endl;

  //...other code remain unchanged
  return 0;
}
```

in.txt:

```
1 2 3
 4 5
6      7      8
```

out.txt:

```
1*2*3*4*5*6*7*8
```

- The >> extractor skips over white spaces even when reading for **characters**
- To read every characters including white spaces
  - Use the **`get()`** method

Self-study

File Stream : Example 3

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

int main() {
  ifstream readFile("in.txt") ;
  ofstream writeFile("out.txt");
  char x;

  while (readFile.get(x))
    writeFile << x  << "*";
  writeFile << endl;

  //...other code remain unchanged
  return 0;
}
```

in.txt:

```
1 2 3
 4 5
6      7      8
```

out.txt:

```
1* *2* *3*
* *4* *5*
*6* * * *7* * * *8*
*
```

- Be careful when reading:
  - Make sure you use the correct operation

Self-study

File Stream : Example 4

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

int main() {
  ifstream readFile("in.txt") ;
  ofstream writeFile("out.txt");
  string x;

  while (readFile >> x)
    writeFile << x << "*";
  writeFile << endl;

  //...other code remain unchanged
  return 0;
}
```

in.txt:

```
1 2 3
 4 5
6      7      8
```

out.txt:

```
1*2*3*4*5*6*7*8
```

- As mentioned:
  - Extractor read only a **single word** for string object
- If the whole sentence is needed:
  - Use **`getline()`** method

Self-study

File Stream : Example 5

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

int main() {
  ifstream readFile("in.txt") ;
  ofstream writeFile("out.txt");
  string x;

  while (getline(readFile, x))
    writeFile << x  << "*" << endl;
  writeFile << endl;

  //...other code remain unchanged
  return 0;
}
```

in.txt:

```
1 2 3
 4 5
6      7      8
```

out.txt:

```
1 2 3*
 4 5*
6      7      8*
```

- Note:
  - Newline character at the end of the sentence is **not** stored in the string object

Self-study

File Stream : Example 6

```
//... Similar to previous example

int main() {
  ifstream readFile("test.txt") ;
  int i;
  string x;

  readFile >> i;
  getline(readFile, x);

  cout << "i: " << i << endl;
  cout << "x: " << x << endl;
  //... Similar to previous example
}
```

test.txt:

```
1
4 5 6
```

output:

```
i: 1
x:
```

- Be careful when mixing **>>** and **`getline()`**
  - **>>** reads only required data, newline character is left untouched!
  - **`getline()`** picks up anything on a line, even if it is just a single new line character

Self-study

File Stream : Example 6 (corrected)

```
//... Similar to previous example

int main () {
  ifstream readFile("test.txt") ;
  int i;
  string x;

  readFile >> i;
  getline(readFile, x);
  getline(readFile, x);

  cout << "i: " << i << endl;
  cout << "x: " << x << endl;
  //... Similar to previous example
}
```

test.txt:

```
1
4 5 6
```

output:

```
i: 1
x: 4 5 6
```

- Simple remedy:
  - Use additional **`getline()`** to discard left over newline characters from the previous extractor operation

Summary

C++ Elements

Built in classes:
- String Class
- Stream Classes
  - Standard input/output
  - String Stream

Template:
- Generic Programming
- Template class with single/multiple type name

Standard Template Library
- Container Class
  - Vector
- Iterator Class
- Algorithm Functions
  - Sorting

References

- **[Carrano]** Chapter 8, Appendix C, E
- **[Elliot & Wolfgang]** Chapter 4
- SGI STL Programmer's Guide
  - http://www.sgi.com/tech/stl/
- C++ Online Reference
  - http://www.cplusplus.com/reference/
- C++ Quick Reference Sheet
  - http://www.sourcepole.ch/sources/programming/cpp/cppqref.html