

# **CS100**

# **Introduction to Programming**

**Lectures 14 Templates & STL**

# Today's learning objectives

- Getting to know the Standard Template Library
- Learn advanced solutions for I/O in C++
- Understand the functionality of templates

# Outline

- Introduction to STL
- Strings and Basic I/O
- Reading and writing files
- Introduction to template functions and classes

# The C++ Standard Template Libraries

- In 1990, Alex Stepanov and Meng Lee of HP Laboratories extended C++ with a library of class and function templates which has come to be known as the STL
- In 1994, STL was adopted as part of ANSI/ISO Standard C++

# The C++ Standard Template Libraries

- STL had three basic components:
  - Containers
    - Generic class templates to store data
  - Algorithms
    - Generic function templates to operate on containers
  - Iterators
    - Generalized 'smart' pointers that facilitate use of containers
    - They provide an interface that is needed for STL algorithms to operate on STL containers
- **String abstraction was added during standardization**

# Why use STL?

- STL
  - offers an assortment of containers
  - releases containers' time/storage complexity
  - containers grow/shrink in size automatically
  - provides built-in algorithms to process containers
  - provides iterators that make the containers and algorithms flexible and efficient.
  - is extensible which means that users can add new containers and new algorithms such that
    - algorithms can process STL containers as well as user defined containers
  - User defined algorithms can process STL containers as well as user defined containers

# Outline

- Introduction to STL
- **Strings and Basic I/O**
- Reading and writing files
- Introduction to template functions and classes

# Strings

- In C we used `char*` to represent a string.
- The C++ standard library provides a common implementation of a string class abstraction named `string`



# Hello World example: From C to C++

```
#include <stdio.h>

void main()
{
    // create string 'str' = "Hello world!"
    char *str = "Hello World!";

    // print string
    printf("%s\n", str);
}
```

# Hello World example: From C to C++

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

→ Include header file to use string

→ string is part of a namespace ("std"), which has to be included (we will be learning more about namespace later)

```
int main()
```

```
{
```

```
    // create string 'str' = "Hello world!"
```

```
    string str = "Hello World!";
```

```
    cout << str << endl;
```

```
    return 0;
```

```
}
```

# Different ways to create strings

```
string str = "some text";
```

or

```
string str("some text");
```

Equivalent



or

```
string s1(7, 'a');
```

Initialization with size 7 and only a's



or

```
string s2 = s1;
```

Copy constructor



# string length

- The length of string is returned by its size() function

```
#include <string>
```

...

```
string str = "something";  
cout << "The size of "  
      << str  
      << "is " << str.size()  
      << "characters." << endl;
```

# string length

- In C we had we only had pointers to data
  - Length of string??
- In C++, we have

```
class string {  
    ...  
public:  
    ...  
    unsigned int size();  
    ...  
};
```

# String concatenation

- concatenating one string to another is done by the '+' operator
  - Operator-overloading will be seen later

```
string str1 = "Here ";  
string str2 = "comes the sun";  
string concat_str = str1 + str2;
```

# String comparison

- To check if two strings are equal use the '==' operator

```
string str1 = "Here ";  
string str2 = "comes the sun";
```

```
if ( str1 == str2 )  
    /* do something */  
else  
    /* do something else */
```

# String assignment

- To assign one string to another use the “=” operator.

```
string str1 = "ShanghaiTech";  
string str2 = "SIST";  
str2 = str1;
```

- Now : str2 equals “Sgt. Pappers”



# More string functions

- Can check if string is empty

```
bool isEmpty = str1.empty();
```

- Can access single character like C-style string

```
str2[0] = 'a';
```

- Can access single character like C-style string

```
string substring = str1.substr(0,8);  
// substring will be "Shanghai"
```

# More string functions

- Find a substring inside another string

```
int index = str1.find(substring);
```

- index will be the starting index of the found substring

- Replace a substring with something else

```
str1.replace(  
    index,  
    substring.length(),  
    newStr );
```

# Working with Input/Output in C++

- at top of each file that uses input/output  
`using namespace std;`
- to use streams to interact with user/console,  
must have  
`#include <iostream>`

# Input/Output in C++

```
#include <stdio.h>
```

```
printf("test: %d\n", x);
```

```
scanf("%d", &x);
```

# Input/Output in C++

```
#include <stdio.h>
```

```
#include <iostream>
```

```
printf("test: %d\n", x);
```

```
scanf("%d", &x);
```

# Input/Output in C++

```
#include <stdio.h>
```

```
#include <iostream>
```

```
using namespace std;
```

```
printf("test: %d\n", x);
```

```
scanf("%d", &x);
```

# Input/Output in C++

```
#include <stdio.h>
```

```
#include <iostream>
```

```
using namespace std;
```

```
printf("test: %d\n", x);
```

```
cout << "test: " << x << endl;
```

```
scanf ("%d", &x) ;
```

# Input/Output in C++

```
#include <stdio.h>
```

```
#include <iostream>
```

```
using namespace std;
```

```
printf("test: %d\n", x);
```

```
cout << "test: " << x << endl;
```

```
scanf("%d", &x);
```

```
cin >> x;
```



# The << Operator

- insertion operator → used along with `cout`
- separate each “type” of thing we print out

```
int x = 3;  
cout << "X is: " << x  
<< "; squared "  
<< x*x << endl;
```

# The >> Operator

- extraction operator → used with `cin`
  - returns a boolean for (un)successful read
- like `scanf` and `fscanf`, skips leading whitespace, and stops reading at next whitespace
- don't need to use ampersand on variables  
`cin >> firstName >> lastName >> age;`

# Outline

- Introduction to STL
- Strings and Basic I/O
- **Reading and writing files**
- Introduction to template functions and classes

# Reading In Files in C++

```
FILE *ifp;
```

- read/write will be specified in call to `fopen()`

# Reading In Files in C++

~~`FILE *ifp;`~~

`ifstream inStream;`

- read specified by variable type
  - `ifstream` for reading
  - Must include header `<fstream>`

# Reading In Files in C++

```
FILE *ifp;
```

```
ifstream inStream;
```

```
ifp = fopen("testFile.txt", "r");
```

- read is specified by "r" in call to fopen

# Reading In Files in C++

```
FILE *ifp;
```

```
ifstream inStream;
```

```
ifp = fopen("testFile.txt", "r");
```

```
inStream.open("testFile.txt");
```

- read is specified by declaration of `inStream` as a variable of type `ifstream`
  - used by `open()`

# Reading In Files in C++

```
FILE *ifp;
```

```
ifstream inStream;
```

```
ifp = fopen("testFile.txt", "r");
```

```
inStream.open("testFile.txt");
```

```
if ( ifp == NULL ) { /* exit */ }
```



# Reading In Files in C++

```
FILE *ifp;
```

```
ifstream inStream;
```

```
ifp = fopen("testFile.txt", "r");  
inStream.open("testFile.txt");
```

```
if (ifp == NULL) { /* exit */ }  
if (!inStream) { /* exit */ }
```

- Check to make sure file was opened

# Writing to Files in C++

- very similar to reading in files
- instead of type `ifstream`, use type `ofstream`
- everything else is the same

# Writing To Files in C++

- `ofstream outStream;`
  - Declare an output file variable
- `outStream.open("testFile.txt");`
  - Open a file for writing
- `if (!outStream) { /* exit */ }`
  - Check to make sure file was opened

# Opening Files

- The `.open()` call for the file stream takes a `char*` (a C-style string)
- If you are using a C++-string variable, you must extract a C-style string
- Calling `.c_str()` will return a C-style string  
`cppString.c_str()`
- Example: `stream.open(cppString.c_str());`

# Using File Streams in C++

- once file is correctly opened, use your `ifstream` and `ofstream` variables the same as you would use `cin` and `cout`

```
inStm >> firstName >> lastName;
```

```
outStm << firstName << " "  
<< lastName << endl;
```

# Advantages of Streams

- does not use placeholders (**%d**, **%s**, etc.)
  - no placeholder type-matching errors
- can split onto multiple lines easily
- precision with printing can be easier
  - once set using **setf()**, the effect remains until changed with another call to **setf()**

# Finding EOF with ifstream – Way 1

- use `>>`'s boolean return to your advantage

```
while (inStream >> x)
{
    // do stuff with x
}
```

# Finding EOF with ifstream – Way 2

- use a “priming read”

```
inStream >> x;

while( !inStream.eof() )
{
    // do stuff with x

    // read in next x
    inStream >> x;
}
```



# Using File Streams in C++

- What if there is multiple lines, and we want to read line-by-line?
- Use getline

```
std::string oneLine;  
std::getline( inStm, oneLine );
```

# Using File Streams in C++

- Example of reading line-by-line

```
while( inStm.good() ) {  
    std::string oneLine;  
    std::getline( inStm, oneLine );  
}
```

# Using File Streams in C++

- What if we don't know how many elements there are in one line?

# Using File Streams in C++

- Example of reading line-by-line, and element-by-element

```
while( inStm.good() ) {  
    std::string oneLine;  
    std::getline( inStm, oneLine );  
  
    std::stringstream lineStm(oneLine);  
    while( lineStm.good() ) {  
        std::string copy;  
        lineStm >> copy;  
        std::cout << copy << " ";  
    }  
    std::cout << "\n";  
}
```

# What about stringstream?

- Like ofstream and cout, but streaming into a string rather than the console or a file!

# Outline

- Introduction to STL
- Strings and Basic I/O
- Reading and writing files
- Introduction to template functions and classes

# C++ Templates

- Support generic programming
  - develop reusable software components (e.g. function, class)
- Template uses generic data type T
  - Replaced by concrete type at compile time
  - Enables “on-the-go” construction of a member of a family of functions and classes that perform the same operation on different data types
    - functions → function templates
    - classes → class templates

# Function Templates

- For functions of considerable importance which have to be used frequently with different data types
- Simple solution:
  - Many functions each operating on one data type only
- Better solution:
  - Defining one function template (i.e. generic function)
- Syntax:

```
template <class T, ... >
returntype function_name (arguments)
{
    /* Body of function */
}
```



# Example: Swapping functions

```
void swap(char &x, char &y) {  
    char t;  
    t = x; x = y; y = t;  
}  
void swap(int &x, int &y) {  
    int t;  
    t = x; x = y; y = t;  
}  
void swap(float &x, float &y) {  
    float t;  
    t = x; x = y; y = t;  
}
```

```
void main()  
{  
    char ch1, ch2;  
    cout << "\n Enter values  : ";  
    cin >> ch1 >> ch2;  
    swap(ch1,ch2) ;  
    cout << "\n After swap ch1 = "  
        << ch1 << " ch2 = " << ch2;  
    int a, b;  
    cout << "\n Enter values  : ";  
    cin >> a >> b;  
    swap(a,b);  
    cout << "\n After swap a = "  
        << a << " b = " << b;  
    float c, d;  
    cout << "\n Enter values  : ";  
    cin >> c >> d;  
    swap(c,d);  
    cout << "\n After swap c  = "  
        << c << " d = " << d;  
}
```

# Example:

## Swapping functions

- Output:

Enter values : R K

After swap ch1 = K ch2 = R

Enter values : 5 10

After swap a = 10 b = 5

Enter values : 20.5 99.3

After swap c = 99.3 d = 20.5

# Generic swapping function

```
#include <iostream.h>
template<class T>
void swap(T &x, T &y) {
    T t;
    t = x; x = y; y = t;
}
```

- Output: same as previous example!

```
void main()
{
    char ch1, ch2;
    cout << "\n Enter values  : ";
    cin >> ch1 >> ch2;
    swap(ch1,ch2);
    cout << "\n After swap ch1 = "
         << ch1 << " ch2 = " << ch2;

    int a, b;
    cout << "\n Enter values : ";
    cin >> a >> b;
    swap(a,b);
    cout << "\n After swap a = "
         << a << " b = " << b;

    float c, d;
    cout << "\n Enter values : ";
    cin >> c >> d;
    swap(c,d);
    cout << "\n After swap c = "
         << c << " d = " << d;
}
```

# Function and Function Template

- Function templates may not be suitable for all data types
- If required, override with normal functions for specific types

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

```
#include <string.h>
```

```
template <class T>
```

```
T max(T a, T b) {  
    if(a>b) return a;  
    else return b;  
}
```

```
char * max(char * a, char * b) {  
    if(strcmp(a,b)>0) return a;  
    else return b;  
}
```

```
void main() {  
    char ch,ch1,ch2;  
    cout << "\n Enter two chars : ";  
    cin >> ch1 >> ch2;  
    ch = max(ch1,ch2);  
    cout << "\n max value " << ch;  
    int a,b,c;  
    cout << "\n Enter two ints: ";  
    cin >> a >> b;  
    c = max(a,b);  
    cout << "\n max value : " << c;  
    char str1[20],str2[20];  
    cout << "\n Enter two strings : ";  
    cin >> str1 >> str2;  
    cout << "\n max value : "  
    cout << max(str1,str2);  
}
```

# Function and Function Template

- Output :

Enter two chars : A Z

Max value : Z

Enter two ints: 12 20

Max value : 20

Enter two strings: Shanghai Beijing

Max value : Shanghai

- In the absence of a specialized function for **char \***
  - **max(str1, str2)** executed, but not producing desired result
  - would compare memory addresses instead of string contents

→ logic for comparing strings or other point data types is different

→ requires “normal” function specialized for **char \***

→ both template and normal function can live in parallel

# Overloaded Function Templates

- Overloading by template function (different number of parameters)

```
#include <iostream.h>
template <class T>
void print(T data) {
    cout << data << endl;
}
template <class T>
void print(T data, int ntimes) {
    for( int i = 0; i < ntimes; i++ )
        cout << data << endl;
}
void main() {
    print(1);
    print(1.5);
    print(520,2);
    print("OOP is Great\n", 3);
}
```

- Output:

```
1
1.5
520
520
OOP is Great
OOP is Great
OOP is Great
```

# Class Templates

- Class template
  - generalized to hold/operate on different data types
- Syntax:

```
template <class T1, class T2, ....>
```

```
class class_name
```

```
{
```

```
...
```

```
    T1 m_data1;    // data items of template type
```

```
    // functions of template argument
```

```
    void func1 (T1 a, T2 & b);
```

```
    T1 func2 (T2 * x, T2 * y);
```

```
}
```

# Example:

## Stack of elements

```
class charstack
{
    char array[25];
    unsigned int top;
public:
    charstack();
    void push( const char & elem);
    char pop();
    unsigned int getsize() const;
};
```

```
class doublestack
{
    double array[25];
    unsigned int top;
public:
    doublestack();
    void push( const double & elem);
    double pop();
    unsigned int getsize() const;
};
```

```
class intstack
{
    int array[25];
    unsigned int top;
public:
    intstack();
    void push( const int & elem );
    int pop();
    unsigned int getsize() const;
};
```



# Generic stack of elements

- Rather than having one stack class for each and every data types, one template class is enough!

```
template<class T>
class datastack
{
    T array[25];
    unsigned int top;
public:
    datastack();
    void push( const T & elem);
    T pop();
    unsigned int getsize() const;
};
```

# Inheritance of Class Template

Through one of the following techniques:

- Derive a class template from a base class, which is a template class (more template parameters may be added)

```
template <class T1, ...>
class derivedclass : public baseclass<T1,...> {
    // member data and functions
};
```

- Derive a class from a base class, which is a template class and restrict the template feature, so that the derived class and its derivatives do not have the template feature

```
class derivedclass : public baseclass<T1,...> {
    // member data and functions
};
```

# Where to put templates?

- Templates are no concrete implementations!
- They are just a template!
- Concrete implementations are derived on demand at compile (in the background)

→ Put templates into a header-files!

# Standard Template Library

- Uses template mechanism for generic ...
  - ... containers (classes)
    - Data structures that hold anything
    - Ex.: `list`, `vector`, `map`, `set`
  - ... algorithms (functions)
    - handle common tasks (searching, sorting, comparing, etc.)
    - Ex.: `find`, `merge`, `reverse`, `sort`, `count`, `random shuffle`, `remove`, `nth-element`, `rotate`, ...