Module 3

Using C++ Standard Library

1 Containers

1.1 Generic Classes

Most applications would probably need to use standard data structures such as binary trees, linked-lists, stacks, queues and so on. It would be tedious for each programmer to have to build all these from scratch even though these kinds of data structures have already been implemented countless of times before. The C++ library comes with a Standard Template Library (STL) that consists of a set of templates to support standard data structures. They are templates since each application may wish to store or retrieve different types of data which can instantiate them to generate classes for the data types used. Templates can be used to implement both generic functions and generic classes as well. The example below uses a template to create a simple stack classes for different data types.

Defining a generic class: Generic1\Stack.h

```
template <typename T>
class CStack {
protected:
    T m_values[256];
    int m_count;
public:
    CStack() { m_count = 0; }
    int GetCount() const { return m_count; }
    void Push(T value) {
           if(m_count == 256) throw "Stack is full.";
           m_values[m_count++] = value;
    }
    T Pop() {
        if(m_count == 0) throw "Stack is empty.";
        return m_values[--m_count];
      }
};
```

No code is generated if templates are not instantiated. To instantiate a class template, use the class within the code. When the template is used, the actual data type must be specified.

Instantiating class templates: main.cpp

```
CStack<char> stack1;
CStack<int> stack2;
CStack<double> stack3;
CStack<char *> stack4;
```

```
stack2.Push(100);
cout << stack2.Pop() << endl;
stack4.Push("Hello Templates!");
cout << stack4.Pop() << endl;
```

You can also declare non-type arguments for a template. This can be used to replace values in the code and not just to replace a data-type. In the following example, we modified the template declaration to allow the class user to determine the maximum size of the stack. You can also assign default values to arguments.

Defining non-type parameters

```
template <typename T = int,int SIZE = 256>
class CStack {
protected:
    T m_values[SIZE];
        :
    void Push(T value) {
           if(m_count == SIZE) throw "Stack is full.";
           m_values[m_count++] = value;
     }
};
```

Instantiating templates with non-type or default parameters

```
CStack<char,128> stack1;    // not using defaults
CStack<> stack2;            // using all defaults
CStack<double> stack3;      // using default SIZE
```

You can use **typedef** to pre-instantiate specific classes from a template. You will not realize that some types are actually generated from templates since the angle braces will not be used for pre-instantiated types.

Using typedefs to pre-instantiate generic types

```
typedef CStack<double> DoubleStack;
typedef CStack<int, 52> CardStack;
```

Using pre-instantiated generic types

```
DoubleStack stack5;
CardStack 6;
```

1.2 STL Elements

STL core elements are **containers**, **iterators** and **algorithms**. Containers determine how data is stored and in what manner the data can be retrieved. Iterators allow you to access the data stored in the container using the concept of pointers. Every time you increment iterators, you can retrieve the next item while decrementing it will allow you to access the previous item. Algorithms are pre-defined type of operations that can be used on any containers, such as sorting. The STL templates are declared in the **std** namespace. Thus you have to prefix templates used from the STL with this namespace. However, you can use the **using namespace** to import the elements from this namespace.

Importing STL templates into the global namespace

```
using namespace std;
```

There are many different types of containers provided in STL. Following shows the **vector** container. The problem with using arrays is that once you allocate the space for an array, the size is fixed. You can only store and access elements in array but you cannot add more elements or remove existing elements from an array. A vector is a STL container to replace the array. It is dynamic as you can choose to add or remove items at any time in the container. Vectors are used to store a list of items. You can treat a vector as a stack, since it allows both push and pop operations. Another nice feature is that you can access items in a vector as an array. To add or remove items in a vector, you can use **push_back** and **pop_back** functions. You can then use a **size** function to find out how many elements are within the vector.

Adding and removing elements from a vector: STL1\main.cpp

```
#include <iostream>
#include <vector>

using namespace std;

void main() {
    vector<int> v;
    v.push_back(100);       // append element
    v.push_back(200);
    v.push_back(300);
    cout << v.size() << endl;
    v.pop_back();           // remove element
    cout << v.size() << endl;
}
```

Since a vector can be treated as an array, you can retrieve elements from the vector using the array (**[]**) operator. Use **front** and **back** functions to specifically retrieve the first and last elements in the vector. Since operators and functions return references rather than values, you can use it to modify the elements within the vector. You can use **clear** to erase all elements and **empty** to check whether there are any elements in the container.

Retrieving elements using array operator and functions

```
for(int i=0;i<n.size();i++) cout << v[i] << endl;
cout << v.front() << endl;
cout << v.back() << endl;
```

Modifying elements retrieved

```
v.front() = 99;
v.back() = 66;
```

Clearing the vector

```
v.clear();
if(v.empty()) cout <<
      "No elements!" << endl;
```

A vector by default only allows you to append or remove items from the back. If you need to be able to add and remove items from both back and front of the container you can use a deque (double-ended queue) instead. A double-ended queue container allows access from both sides as it has functions such as **push_back**, **push_front**, **pop_back**, **pop_front** to allow you to add and remove elements from both front and back. Use **front** and **back** functions to retrieve the value just like a vector.

Using deques

```
#include <iostream>
#include <vector>
#include <deque>

using namespace std;

void main() {
        :
    deque<char *> d;
    d.push_front("Hello!");
    d.push_back("Goodbye!");
    cout << d.front() << endl;
    cout << d.back() << endl;
    d.pop_back();
    d.pop_front();
        :
}
```

Iterators allow us to access elements within a container the same way we can use pointers to access elements. You need to declare the iterators through the container template to generate the correct iterator type or just use **auto**. Use **begin** method to retrieve an iterator that points to the first element, or **end** to retrieve iterator for end of the container. You can then use increment and decrement operators to move the pointer through the elements.

Using iterators

```
vector<int>::iterator b1 = v.begin();    // auto b1 = v.begin();
vector<int>::iterator e1 = v.end();      // auto e1 = v.end();
while(b1 != e1) cout << *(b1++) << endl;
```

While **begin** and **end** retrieves a pointer to first and past the last element, you can also obtain a reverse iterator, where data are always accessed in reverse order, even though the same operators are used. Use the **rbegin** and **rend** functions to retrieve reverse iterators.

Using reverse iterators

```
vector<int>::reverse_iterator b2 = v.rbegin();   // auto b2 = v.rbegin();
vector<int>::reverse_iterator e2 = v.rend();     // auto e2 = v.rend();
while(b2 != e2) cout << *(b2++) << endl;
```

To iterate through containers is now much simpler in C++/11. Beforehand we need to explicitly call **begin** function to retrieve the first iterator and **end** function to get the end of the container. The iterator also has to be dereferenced to access the value. The interator is increment to point to the next object until the end is reached. All this code can now be replaced by the following new iterator **for** loop in C++/11.

Iterating through a container in C++/11

```
for(auto n : v) cout << n << endl;
```

Even though certain containers are not efficient when performing certain types of operations, these operations may still be available as long as containers have support for iterators. A vector allows you to insert and delete elements at any position even though it is not efficient for this type of container. Use **insert** and **erase** function to perform such operations by using iterators.

Inserting elements

```
v.insert(v.begin(),160);
v.insert(v.begin() + 1, 480);
v.insert(v.end() - 1,200);
```

Algorithms are generic functions that can be applied to containers to perform specific processing. Algorithms commonly use iterators since most containers have support for them. Following is some example algorithms. A **sort** function is also provided to sort elements in containers that do not have their own sorting support.

You need to provide iterators specifying the range of elements to sort. In the following example, we sort a list of numbers within a vector. If the elements have been sorted, you can use the **binary_search** algorithm function to perform faster searching operation on containers. The following is an example of binary searching. Elements can be reversed or rotated using **reverse** and the **rotate** functions.

Sorting a vector

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

void main() {
    vector<int> v(4);
    v[0] = 99;
    v[1] = 62;
    v[2] = 110;
    v[3] = 103;
    sort(v.begin(),v.end());
    for(auto n : v) cout << n << endl;
}
```

Other algorithms

```
sort(v.begin(),v.end());
if(binary_search(v.begin(),v.end(),110))
    cout << "Found!" << endl;
reverse(v.begin(),v.end());
rotate(v.begin(),v.begin()+2,v.end());
```

1.3 String

In C/C++, strings are actually character arrays that contain a NULL character to mark the end of the string. In modern programming languages, strings are implemented as objects. In STL there is a **string** class to create objects to store and process ANSI strings. For Unicode strings, use **wstring** class instead. However you will also need to use **wcout** and **wcin** for console input and output of unicode strings even though the console window can only support ANSI strings.

Using strings: String1\main.cpp

```
#include <iostream>
#include <string>
using namespace std;
void main() {
      string s1 = "Hello!";
      s1.append(" Goodbye!");
      cout << s1.size() << endl;
      s1.replace(7, 7, "Ciao");
      s1.erase(0, 7);
      s1.insert(0, "Sayonara! ");
      cout << s1.c_str() << endl;
      wstring s2 = L"Hello!";
      wcout << s2.c_str() << endl;
}
```

There is no built-in support in the Standard C++ Library to write code that can either be compiled to ANSI or Unicode but but you can still provide support by defining your own TCHAR macros by checking for the **UNICODE** symbol.

Supporting both ANSI and Unicode: Unicode.h

```
#pragma once

#ifdef UNICODE
    #define tstring wstring
    #define tcin    wcin
    #define tcout   wcout
    #define _t(str) L##str
#else
    #define tstring string
    #define tcin    cin
    #define tcout   cout
    #define _t(str) str
#endif
```

Sample program using the above macros

```
#include <tchar.h>
#include <string>
#include <iostream>
using namespace std;

#include "Unicode.h"


void main() {
    TCHAR buffer[256];
    tcout << _t("Enter your name: ");
    tcin.getline(buffer, sizeof(buffer) / sizeof(TCHAR));
    tstring name(buffer);
    tcout << _t("Hello ") << name << _t('!') << endl;
}
```

1.4 Bitset

The **bitset** is a specialized type of container where it only stores a list of bits where each bit can either be 0 or 1. When instancing the template, you provide the number of bits required rather than the data type. It contains functions to manipulate the bits within the set. However C/C++ already has a complete set of bitwise-operators that works specifically with bits so this is not a useful container unless you do not wish to use operators.

Manipulating a 4-bit number: Bitset1\main.cpp

```
#include <iostream>
#include <bitset>
using namespace std;
void main() {
    bitset<4> v;
    v[0] = 1; v[1] = 0;
    v[2] = 0; v[3] = 1;
    v.set(1); v.reset(3); v.flip(2);
    cout << v.to_ulong() << endl;
}
```

The following shows how to use C++ bitwise-operators to work with bits which will be more efficient but much more complex than using the **bitset** class.

Using bitwise operators

```
int n = 9;          // assign bit 0 and 1  (0b1001)
n = n | 2;          // set bit 1    (0b0001)
n = n & ~8;         // reset bit 3  (0b1000)
if (n & 4)          // check bit 2  (0b0100)
    n = n & ~4;     // reset bit 4  (0b0100)
else n = n | 4;     // set bit 4    (0b0100)
cout << n << endl;
```

1.5 Maps & Sets

These containers are associative containers that allows you to use a key to associate with elements. You can then fetch the element by using the key. The main difference between **map** and **multimap** is that **map** requires unique keys while **multimap** allow duplicate keys. A pair template is required to construct the element that will consist of the key and value to add into map containers. Use **insert** function to add paired elements into a map. You can then use the **find** method to locate the element using the key that returns an iterator to the paired element. From a paired element, use the **first** attribute to access the key, and **second** to access the value. In a **multimap**, you can continue to increment the iterator to retrieve the next matching item until you reach one that does not match by checking the iterator against the one retrieved by **upper_bound**.

A **set** is like a **map** except the value of the element will also be used as the key to locate the element. This can be used to determine whether a value is part of a **set** or not. Elements in a **set** must be unique while it can be repeated in a **multiset**.

Using maps: Maps1\main.cpp

```
#include <iostream>
#include <map>
using namespace std;
void main() {
    map<int,string> m;
    m.insert(pair<int,string>(100,"Can of coke"));
    m.insert(pair<int,string>(200,"Bottle of ketchup"));
    m.insert(pair<int,string>(300,"Pack of noodles"));
    map<int,string>::iterator p = m.find(200);
    if(p == m.end()) {
        cout << "Element not found!" << endl;
        return;
    }
    cout << "Key: " << p->first << endl;
    cout << "Value : " << p->second << endl;
}
```

Using sets: Sets1\main.cpp

```
#include <iostream>
#include <set>
using namespace std;
void main() {
    set<string> s;
    s.insert("Rice Cooker");
    s.insert("Food Blender");
    s.insert("Television");
    set<string>::iterator p;
    p = s.find("Television");
    if(p == s.end()) cout << "Not found!" << endl;
}
```

1.6 STL Objects

The STL templates can be instantiated to store any type including objects. If you are using a container that can sort or search for data such as a priority queue, maps and sets, you need to overload the less than operator that will be called by container for such operations. Following example demonstrates storing and searching for objects in a set.

Implementing STL compatible objects: STLObjects1\main.cpp

```
#include <set>
#include <string>
#include <iostream>

using namespace std;
```

```
class CContact {
    string m_name;
    string m_email;
public:
    CContact(const char *name, const char *email)
        : m_name(name), m_email(email) { }
    CContact(const string& name, const string& email)
        : m_name(name), m_email(email) { }
    string GetName() const { return m_name; }
    string GetEmail() const { return m_email; }
    friend bool operator < (const CContact& c1, const CContact& c2) {
        return c1.m_name < c2.m_name; }
};
```

The above is the format required for overloading less than operator. You must follow the format as shown to work with STL. Overload globally but mark it friend function if you need to access the private members from the function. Note that in our example, we only make use of the contact name for sorting and searching.

Storing and searching objects

```
void main() {
    multiset<CContact> contacts;
    contacts.insert(CContact("Phillip","the_visualizer@yahoo.com"));
    contacts.insert(CContact("Phillip","visualizer@domaindlx.com"));
    contacts.insert(CContact("Sally","sasa88@hotmail.com"));
    char name[64]; cout << "Enter name:";
    cin.getline(name,sizeof(name));
    CContact contact(name,"");
    multiset<CContact>::iterator p = contacts.find(contact);
    if(p != contacts.end()) {
        multiset<CContact>::iterator e =
                contacts.upper_bound(contact);
        do {
                cout << p->GetName() << ','
                   << p->GetEmail() << endl;
        } while (++p != e);
    }
}
```

1.7 Smart Pointers

Even though it is safer to not use dynamic objects, the standard C++ has support to ensure that there is no memory leaks if you do use them. Since the compiler does not generate code to destroy dynamic objects, you will need to remember to call **delete** when you no longer need the object anymore. You should also invalidate the pointer by assigning a null address.

Using dynamic objects

```
CContact *c1 = new CContact("Phillip", "tango@symbolicon.com.my");
cout << c1->GetName().c_str() << endl;
delete c1; c1 = nullptr;
```

The issue is developers may forget to delete the dynamic object or accidentally try to delete it more than once. A C++ standard library now provides a set of templates to create smart pointers. Smart pointers objects are not dynamic but they store pointers to dynamic objects. Because the smart pointer is not a dynamic object a compiler will generate code to destroy the smart pointer. In turn the smart pointer will destroy the dynamic object. You can access the dynamic object thru the smart pointer by using the same **->** operator. You can use **unique_ptr<>** or **shared_ptr<>** template to create a smart pointer object. The difference is unique_ptr does not allow more than one smart pointer to access the object so you cannot make a copy of a unique_ptr while shared_ptr does allow it.

Using the unique_ptr smart pointer

```
unique_ptr<CContact> p1(new CContact("Phillip", "tango@symbolicon.com.my"));
cout << p1->GetName().c_str() << endl;
unique_ptr<CContact> p2(p1);    // unique_ptr cannot be copied
```

You can copy and pass a shared_ptr by value

```
shared_ptr<CContact> p2(new CContact("Phillip", "tango@symbolicon.com.my"));
shared_ptr<CContact> p3(p2);// making a copy of the smart pointer
cout << p2->GetName().c_str() << endl << p3->GetName().c_str() << endl;
```

However even though you can have multiple **shared_ptr** objects, they all share the same dynamic object. You do not have to worry about destroying the same dynamic object more than once as **shared_ptr** keeps an internal counter to guarantee that the dynamic object is not destroyed until all smart pointers sharing the same dynamic object is destroyed.

1.8 Exceptions

Even though there is no standard built-in exception class in C++, the standard library does provide an exception class. All the C++ containers iterators and templates throw objects from this class. You can access the error message through the **what** member function that returns a c-style string. A basic exception can only store a message but you can extend the class too store additional information. We cover extending classes in the next module.

Throwing and catching exception objects

```
try { throw exception("throwing C++ exception."); }
catch (exception& e) { cout << "Exception caught : " << e.what() << endl; }
```

2 Object IO & Persistence

2.1 Streams & Operators

The C++ input and output system operate on streams. Streams are objects that can produce or consume information and linked to a physical device such as the screen, keyboard, file or printer. C++ provide I/O operator's called as inserters and extractors (**<<** and **>>**) for input or output data to stream objects, which are objects created from the subclasses of the abstract **istream** and **ostream** classes. The stream objects have only been pre-programmed to output only native data types of C++. What we can do is to overload the global operators for input and output for our class.

Overloading I/O operators: Time.cpp

```
ostream& operator << (ostream& out,CTime& time) {
    out << time.m_hour << endl
        << time.m_minute << endl
           << time.m_second << endl;
    return out;
}

istream& operator >> (istream& inp,CTime& time) {
    inp >> time.m_hour >> time.m_minute >> time.m_second;
     return inp;
}
```

Declaring friend stream I/O operator functions: Time.h

```
#include <iostream>
using namespace std;
class CTime {
        :
    friend ostream& operator << (ostream& out,CTime& t);
    friend istream& operator >> (istream& in,CTime& t);
};
```

Rather than storing objects in text format, we may also store objects in binary format which is more compact and faster I/O. However you should not overload inserters and extractors for binary data. A **read** function and a **write** function may be exposed to load and save object data into a stream object by using the **read** and **write** method of the stream object.

Declaring binary persistence functions

```
class CTime {
        :
    void Read(istream &in);
    void Write(ostream &out);
};
```

Implementing binary persistence functions

```
void CTime::Read(istream &in) {
    in.read((char *)&m_hour,sizeof(m_hour));
    in.read((char *)&m_minute,sizeof(m_minute));
    in.read((char *)&m_second,sizeof(m_second));
}

void CTime::Write(ostream &out) {
    out.read((char *)&m_hour,sizeof(m_hour));
    out.read((char *)&m_minute,sizeof(m_minute));
    out.read((char *)&m_second,sizeof(m_second));
}
```

2.2 File Streams

By overloading operators, we can implement persistence where objects can be saved and retrieved from temporary and fixed storage. In actual fact, we are just saving and retrieving the data within objects rather than the object itself even though this will be transparent to the class user. By overloading inserters and extractors, we can save and load objects in text format.

In the standard C++ library, two stream objects are available for you to perform input and output on console output device (**cout**), or console input device (**cin**). To persist to files, file stream objects can be created from the file stream classes **ofstream** and **if stream** instead. Include **fstream** header file to use these classes.

Persisting objects in text format: Stream1\main.cpp

```
CTime time1(10,20,30);
ofstream file1("C:\\test.txt");
file1 << time1;
file1.close();

CTime time2;
ifstream file2("C:\\test.txt");
file2 >> time2;
file2.close();
```

For more control, you should use **fstream** object instead. This allows you to specify flags to determine how the file should be opened. The following are a list of flags that can be used as the mode argument.

Mode flags

`ios::app`	*Open the file for append mode*
`ios::ate`	*Seek to the end of the file*
`ios::trunc`	*Truncate all data from file*
`ios::in`	*Open file for input*
`ios::out`	*Open file for output*
`ios::binary`	*Open file in binary mode*

In binary mode, you can make use of **get** and **put** function to read and write single bytes while **read** and **write** can be used to read and write a single block of data. You can use the **seekg** and **seekp** to position read and write pointer to any part of the file before reading and writing the data.

ios seek members

`ios::beg`	*position relative to beginning of file*
`ios::end`	*position relative to end of file*
`ios::cur`	*new position relative to current position*

Persisting objects in binary format

```
void main() {
    CTime time1(10,20,30);
    fstream stream("\\test.bin",
        ios::in | ios::out | ios::trunc | ios::binary);
    time1.write(stream);
    stream.flush();

    CTime time2;
    stream.seekg(0,ios::beg);   // go to beginning of file
    time2.read(stream);
    stream.close();
}
```

To read text files you can use inserters and extractors instead. However the inserter for text will only read a word. To read a line, you should use the **getline** function instead. Use **eof** function to determine the end of the file has been reached. Following example shows how to write and read lines from a text file.

Text file I/O

```
#include <iostream>
#include <fstream>

void main() {
    fstream stream("\\example.txt", ios::in | ios::out | ios::app);
    stream << "This is line 1." << endl;
    stream << "This is line 2." << endl;
    stream << "This is line 3." << endl;
    char buffer[256];
    stream.seekg(0,ios::beg);
    while(!stream.eof()) {
        stream.getline(buffer,sizeof(buffer));
        cout << buffer << endl;
    }
    stream.close();
}
```

3 Multi-Threading

3.1 Thread

The standard C++/11 library now provides support for multi-threading. Use a **thread** class to create a new thread and provide the function address and the arguments to pass to the function. The function will be called immediately on the new thread. Each thread has a unique id that can be retrieved using the **get_id** function. You can wait for a thread to complete using the **join** function. You can place the thread into a wait state by using **sleep_for** or **sleep_until**. Duration can now be specified using special duration objects in **chrono** namespace such as **hours**, **minutes**, **seconds** and also **milliseconds**. To run operations on the current thread, use **this_thread** as a scope. Following example shows a display function called by three threads.

Multi-threading in C++/11: Thread1\main.cpp

```
#include <iostream>
#include <thread>

using namespace std;

void display(const string& message) {
    cout << message.c_str() << " is running..." << endl;
    this_thread::sleep_for(chrono::seconds{ 4 });
    cout << message.c_str() << " is completed." << endl;
}

void main() {
    thread t1{ display, "Thread #1" };
    thread t2{ display, "Thread #2" };
    thread t3{ display, "Thread #3" };
    cout << "Thread #1 id = " << t1.get_id() << endl;
    cout << "Thread #2 id = " << t2.get_id() << endl;
    cout << "Thread #3 id = " << t3.get_id() << endl;
    t1.join();
    t2.join();
    t3.join();
}
```

When you run the program, the messages appearing on the screen will be messed up since all threads are trying the display the string at the same time. To make sure only one thread can output to the console at one time, we can use a **mutex** as a lock.

Synchronize code to one thread with mutex

```
#include <mutex>

mutex m1;

void display(const string& message) {
    m1.lock(); cout << message.c_str() << " is running..." << endl;
    m1.unlock(); this_thread::sleep_for(chrono::seconds{ 4 });
    m1.lock(); cout << message.c_str() << " is completed." << endl;
    m1.unlock();
}

void main() {
        :
    m1.lock();
    cout << "Thread #1 id = " << t1.get_id() << endl;
    cout << "Thread #2 id = " << t2.get_id() << endl;
    cout << "Thread #3 id = " << t3.get_id() << endl;
    m1.unlock();
        :
}
```

To do automatic locking or unlocking, you can use **lock_guard**. This object will unlock for you when it is destroyed; goes out of scope. In the passing example we create the object to help us lock and unlock the mutex.

Auto-lock and unlock with lock_guard

```
void display(const string& message) {
    {
        lock_guard<mutex> g(m1);
        cout << message.c_str() << " is running..." << endl;
    }
    this_thread::sleep_for(chrono::seconds{ 4 });
    {
        lock_guard<mutex> g(m1);
        cout << message.c_str() << " is completed." << endl;
    }
}

void main() {
        :
    {
        lock_guard<mutex> g(m1);
        cout << "Thread #1 id = " << t1.get_id() << endl;
        cout << "Thread #2 id = " << t2.get_id() << endl;
        cout << "Thread #3 id = " << t3.get_id() << endl;
    }
        :
}
```

3.2 Atomic

Concurrency issues such as data races can occur when multiple threads updates the same memory at the same time. Only atomic operations are safe. Atomic means that the operation is compiled into a single instruction. Simple operations like incrementing a variable may not be a atomic operation since it is compiled into a load, update and store instructions. The following shows the possible code that can be compiled from a simple increment operation.

Incrementing a variable in assembly

```
int count;
__asm {
    mov eax, count  // load value into CPU register from memory
    inc eax         // increment value in CPU register
    mov count, eax  // store value from CPU register back into memory
}
```

C++/11 library now provides a custom set of atomic types. Standard operations have to be implemented for each type but guaranteed that the operations will be atomic; where each operation can be considered as a single instruction that is not interupted by other threads. The following code is implementing a thread counter where we track the number of running threads. You will not have to use manual locking as the update is guaranteed to be atomic.

Using an atomic type

```
atomic_int tcount;
void display(const string& message) {
    ++tcount;
        :
    --tcount;
}
```

3.3 Promise & Future

A thread may be used to process information to obtain a result that is required to be used in another thread. You can use a **promise** to pass data between threads. Thread that requires the value to run can obtain a **future** from a **promise** and call **get** to get the value. The thread will block until the value has been set by another thread using the promise.

A promise to return an int in the future

```
promise<int> p1;

void setValue(int min, int max) {
    this_thread::sleep_for(chrono::seconds{ 4 });
    p1.set_value((rand() % max) + min);
}
```

```
void setValue(int min, int max) {
    this_thread::sleep_for(chrono::seconds{ 4 });
    p1.set_value((rand() % max) + min);
}
```

A thread waiting for the future value

```
void main() {
    thread t1{ setValue, 1, 6 };
    future<int> f1 = p1.get_future();
    int n1 = f1.get();
    cout << n1 << endl;
    t1.join();
}
```

To simply multi-tasking where a function that is not build for multi-threading can be called asynchronously using multiple threads you can use **async** function. It creates a thread to run the operation and returns a **future** so that you can get the return value of the operation.

A normal function returning an value

```
int getValue(int min, int max) {
    this_thread::sleep_for(chrono::seconds{ 4 });
    return (rand() % max) + min;
}

void main() {
    int n1 = getValue(1, 6);    // synchronous call
    cout << n1 << endl;

    future<int> f1 = async(getValue, 1, 6);     // asynchronous call
    n1 = f1.get();  // wait for promised value
    cout << n1 << endl;
}
```