Visual Studio

CPP301-B

Module 4

Multi-Threading & Asynchronous Objects

1 Multi-Threading

1.1 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();
}
```

1.2 Synchronization

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;
    }
        :
}
```

1.3 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;
}
```

1.4 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;
}
```

2 Asynchronous Objects

2.1 Base Asynchronous Class

Even though thread support is provided by the C++ standard library, threading is still function based where threads call functions instead of running objects. You can create your own class to implement the concept of asynchronous objects. This will simplify multi-threading even for those that are not familiar with the concept. If every object has its own thread, then it can run independently from other objects. Following is an example of a base class that will create one thread for each object. A thread cannot call object functions so we provide a static **callback** function that the thread will call. This function will then call the **run** function in the object that owns the thread. Since different objects will perform different tasks, the **run** function is abstract. This class is generic to allow you to customize what is passed to the **run** function as each task will require different input and output. For example a task that processes a file would then require the filename.

A base asynchronous object class: MyTypes.h

```
#include <thread>

template<typename T>
class async_object {
protected:
    std::thread m_thread;
private:
    static void callback(async_object *obj, T state) {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        obj->run(state);
    }
public:
    virtual void run(T state) = 0;
    async_object(T state) : m_thread(callback, this, state) { }
    void wait() { m_thread.join(); }
};
```

2.2 Asynchronous Objects

You can then extend the base class to implement custom asynchronous tasks. Specify what is passed to the **constructor** and **run** function and then override **run** function to write the code for the task.

Extending the base class

```
#include <Windows.h>
#include <tchar.h>

class async_msgbox : public my::async_object<LPCTSTR> {
public:
    async_msgbox(LPCTSTR message) : my::async_object(message) { }
    void run(LPCTSTR message) override {
        MessageBox(NULL, message, _T("Message"), MB_OK); }
};
```

For example the above class will display a message box and the **run** function will then be passed the message to display. If you only have one thread you can only display one message at one time. Now you can easily display multiple messages.

Displaying multiple messages at the same time

```
void main() {
    async_msgbox b1(_T("Hello!"));
    async_msgbox b2(_T("Goodbye!"));
    async_msgbox b3(_T("Sayonara!"));
    b1.wait();
    b2.wait();
    b3.wait();
}
```

3 Other C++ Features

3.1 Initializer List

Previously only arrays and simple structures can be initialized using initializer lists. In C++/11, now everything can be initialized including basic types. To initialize to default value, use an empty list. The assignment operator is optional. If the list has only one value, C++ automatically find a constructor that accepts the arguments.

Initializing basic types: Initializers1\main.cpp

```
int v1 {};      // initialize to default value for int (0)
short v2 {};    // initialize to default value for short (0)
bool  v3 {};    // initialize to default value for bool (false)
bool v4 {true}; // initialize to true
auto v5 {123};  // initialize int to 123
```

C++ selects the appropriate constructor

```
class A {
public:
    A(int n) { std::cout << n << std::endl; }
    A(double d) { std::cout << d << std::endl; }
    A(int n, double d) { std::cout << n << ',' << d << std::endl; }
};

void main() {
    A obj1{ 123 };
    A obj2{ 1.99 };
    A obj3{ 123, 1.99 };
}
```

You can also have a constructor that can accept a specific **initialize_list** container. It will allow you to pass in a variable number of initialization values. For example we can add the constructor to **CContact** class to take values from this container instead.

Supporting initializer list container

```
CContact(initializer_list<const char *> values) {
    if (values.size != 2) exception("not enough initializers");
    auto ptr = values.begin();
    m_name = *(ptr++);
    m_email = *ptr;
}
```

Creating CContact objects with initializer list

```
void main() {
    CContact c1 { "Sally", "sasa99@hotmail.com" };
    CContact c2 { "Phillip", "xnamp@hotmail.com" };
}
```

3.2 Lambda Expressions

It is possible to store the address of a function in a variable and then use the variable to call the function. We normally call this as a virtual function. We can use **typedef** to define the function pointer type. If you are using a C++/11 or later compiler you no longer will need to define pointer type and use **auto** instead. The **auto** keyword lets the C++ compiler detect the variable type automatically.

Declaring function pointer type Lamba1\main.cpp

```
// typedef int (*myfuncptr)(int, int);
typedef int myfuncptr(int, int);
```

Different functions but following the same signature

```
int add(int x, int y) { return x + y; }
int sub(int x, int y) { return x - y; }
int mul(int x, int y) { return x * y; }
```

Assigning function pointers and calling functions through pointer

```
void main() {
    myfuncptr p;
//  p = add; cout << (*p)(10,20) << endl;
//  p = sub; cout << (*p)(10,20) << endl;
//  p = mul; cout << (*p)(10,20) << endl;
    p = add; cout << p(10,20) << endl;
    p = sub; cout << p(10,20) << endl;
    p = mul; cout << p(10,20) << endl;
}
```

Since a pointer is just a value, you can pass functions to other functions as arguments using function pointers. Functions can then be called through the pointers as shown below.

Accepting function pointer as argument

```
void display(myfuncptr p) {
//  cout << (*p)(10, 20) << endl;
    cout << p(10, 20) << endl;
}
```

```
void main() {
    display(add);
    display(sub);
    display(mul);
}
```

Not declaring pointer type

```
void display(int f(int, int)) {
    cout << f(10, 20) << endl;
}
```

Using auto

```
auto f = add; cout << f(10,20) << endl;
f = sub; cout << f(10,20) << endl;
f = mul; cout << f(10,20) << endl;
display(add);
display(sub);
display(mul);
```

Since functions can be called through pointers, in C++/11 you can now directly assign code to variables in the format shown below, called as lambda expressions. These are functions that can be called through the variables that you assign to. Return type can be auto-detected and left out. Capture list is used to capture additional local variables so they are available to the lambda expression during execution. Use = to capture all local variables by value or & to capture by reference.

Lambda expression syntax

```
[capture_list](arguments) -> return_value_type { statements; }
```

Using lambda expressions

```
auto add = [](int x, int y) -> int { return x + y; };
auto sub = [](int x, int y) { return x - y; };
auto mul = [](int x, int y) { return x * y; };
```

Directly passing functions as arguments

```
display([](int x, int y) { return x + y; });
display([](int x, int y) { return x - y; });
display([](int x, int y) { return x * y; });
```

Capturing local variables

```
int v = 0;
auto add = [&v](int x, int y) { return v += x + y; };
auto sub = [&v](int x, int y) { return v += x - y; };
auto mul = [&v](int x, int y) { return v += x * y; };
```

```
cout << add(10, 20) << endl;// 30
cout << sub(10, 20) << endl;// -10 = 20
cout << mul(10, 20) << endl;// +200 = 220
cout << v << endl;
```

3.3 Constant Expressions

Functions are called at runtime so they are not allowed in C++ expressions that has to be evaluated at compilation time. For example a constant value or expression will be required to allocate a non-dynamic array. You cannot call any function including a generic function to obtain the value since the size is required at compilation time and functions are evaluated at runtime.

Certain operations require compile time values: ConstExpr1\main.cpp

```
#include <cstdint>

template<typename T>
int sizeInBytes(int count) { return sizeof(T) * count; }

void main() {
    uint8_t array1[sizeof(int) * 10];
    uint8_t array2[sizeInBytes<int>(10)];  // function call not allowed
}
```

You can now mark functions as constant expressions using **constexpr**. However such functions are limited to one single return statement and it must be guaranteed that if the function is called using the same arguments, the return value must always be the same.

A constant expression function

```
template<typename T>
constexpr int sizeInBytes(int count) { return sizeof(T) * count; }
```

3.4 Variadic Templates

Variadic templates allow support of a variable number of generic arguments in generic classes and functions. For example you want to implement a function that can display any number of arguments and the arguments can also be of any type. Specify a **...** in the template and the code to represent variable number of types and arguments. The following shows a generic function that can the first and remaining arguments. It will also be necessary to have a separate function that processes the end since arguments can be empty or exhausted.

Generic function to process variable number of arguments:: Variadic1\main.cpp

```
template<typename T, typename... Ts>
void out_all(const T& head, const Ts&... rest) {
    std::cout << head << std::endl; // process the first item
    out_all(rest...);   // repeatedly calls same function to process the rest
}
```

Function to process the end of argument list

```
void out_all() { std::cout << "<-end->" << std::endl; }
```

Call function with variable number and type of arguments

```
void main() {
    out_all(10, 20, 30, 40, 50, 60);
    out_all('A', 10, 1.1, "Hello!");
}
```

3.5 Regular Expressions

The C++/11 library now has support for regular expressions. Since regular expression usually contain backslashes and sometimes double quotes, to avoid having to escape these characters in a literal string you can use a raw string which allows embedding of these characters. A raw string must have a **R** prefix and all the characters within have to be placed inside braces. The braces will not be part of the output. For Unicode you can use **LR** or **UR**.

Using raw strings: Regex1\main.cpp

```
void main() {
    cout << "\"C:\\CPPDEV\\DOC\\01.ODT\"" << endl;    // normal string
    cout << R"("C:\CPPDEV\DOC\01.ODT")" << endl;      // raw string
}
```

Use **regex** class to construct a regular expression pattern. Then call **regex_match** to match an entire string against the pattern. To do data extraction using groups use the **regex_search** function instead and provide a **smatch** object to capture the results. Use **wregex** and **wsmatch** instead to match and capture results using Unicode string and pattern.

Matching against a regular expression

```
#include <regex>
#include <iostream>

using namespace std;
```

```
void main1() {
    string zipcode;
    regex rx(R"(\d{5})");
    do {
        cout << "Enter zip code:"; cin >> zipcode;
        if (regex_match(zipcode, rx)) break;
        cout << "Invalid zip code. Please re-enter." << endl;
    } while (true);
    cout << "Zipcode:" << zipcode << endl;
}
```

Extracting matched data using groups

```
void main2() {
    smatch data;
    regex rx(R"((\d{2})/(\d{2})/(\d{4}|\d{2}))");
    do {
        string date;
        cout << "Enter date (dd/mm/yyyy):"; cin >> date;
        if (regex_search(date, data, rx)) break;
        cout << "Invalid date. Please re-enter." << endl;
    } while (true);
    cout
        << data[0] << endl  // default group
        << data[1] << endl
        << data[2] << endl
        << data[3] << endl;
}
```

To match multiple times and extract each matched value use **sregex_iterator** to get an iterator that can accessed each matched value. Use **wsregex_iterator** to match against Unicode string and pattern.

Extracting multiple match values

```
void main() {
    string text("11,22,33,44,55,66");
    regex rx(R"(\d+)");
    auto b = sregex_iterator(text.begin(), text.end(), rx);
    auto e = sregex_iterator();
    while (b != e) {
        smatch m = *(b++);
        cout << m.str() << endl;
    }
}
```