ADVANCED CPP (Part 2)

SMART POINTERS

A smart pointer is a template class which uses operator overloads to provide the functionality of a pointer, while providing additional features to support improved memory management and safety.

Essentially, they are a wrapper around a standard C pointer. Smart pointers are defined in the <memory> header and were introduced in C++ 11. There are 3 types; unique, shared and weak.

Unique

A unique pointer cannot be copied.

std::unique\_ptr<type> obj(*expression to create)*

class Test {}

std::unique\_ptr<Test> test(new Test); // use the *test* obj as normal

|  |  |
| --- | --- |
| **Common methods** | **Description** |
| Test.reset(); | // destroy (calls destructor) |
| Test.reset(new Test); | // destroy and create another in its place |
| Auto copy = move(test); | // moved – no constructor or destructor is called |

As a unique pointer is unique, it cannot be copied: *auto copy = test*; this will fail.

You can also use the built-in make\_unique function to create a unique pointer which is recommended:

Make\_unique<Test>(//args); // if no args, the parenthesis are still needed

Shared

Similar to a unique pointer, but copies can be made.

Shared\_ptr<Test> test(new Test);

Shared pointers can also be made using the make\_shared function:

Make\_shared<Test>();

Make copies using:

Auto copyTest = test;

|  |  |
| --- | --- |
| **Common methods** | **Description** |
| Test.use\_count(); | count number of copies including self |

Weak

A weak pointer is a special kind of shared pointer, except it is not counted in a shared pointers reference. It is useful in cases where you need a pointer that doesn’t affect the lifetime of the resource that it points to. A weak pointer is created from a shared pointer.

Auto a = make\_shared<Test>(); // make shared pointer

Auto c1 = a; // make copy

Auto c2 = weak\_ptr<Test>(a); // make weak pointer from shared ptr

Cout << a.use\_count(); // 2 – weak ptr not included

Before you can call methods from a weak pointer, you need to call the lock method.

Auto b = c2.lock();

b->methodName(); // method name from Test class

Weak pointers are useful when you need a pointer to something that may or may not exist. Commonly used to prevent circular references, which is a situation in which 2 values reference each other.

OPERATOR OVERLOADING

Allows you to use operators, such as; +, -, \*, / - and change the way that they’re used.

For example, if you have a class, and you want a way to add 2 objects together:

Class Sally {

Public:

Int num;

Sally() {} // constructor

Sally(int a) :num(a) {} // constructor with arg

Sally operator+ (Sally aso) { // operator overload +

Sally brandNew;

brandNew.num = num + aso.num;

return brandNew;

}

}

Int main() {

Sally a(34);

Sally b(21);

Sally c = a + b; // use overloaded operator

Cout << c.num << endl; // 55

}

UNIT TESTING

Use the ‘Google Test’ library – look online on how to install.

Main.cpp

#include <gtest/gtest.h>

Int main(int argc, char \*\*argv) {

testing::InitGoogleTest(&argc, argv);

return RUN\_ALL\_TESTS();

}

// to compile: g++ -Wall -g -pthread main.cpp -lgtest\_main -lgtest -lpthread

Unit test

*Example 1*

TEST(testGroup, testDefinition) {} // note arguments are not strings or variables

Struct BankAccount {

Int balance = 0;

BankAccount() {}

BankAccount(const int balance) :balance(balance) {}

};

TEST(accountTest, bankAccountStartsEmpty) {

BankAccount account;

EXPECT\_EQ(0, account.balance);

}

*Example 2*

Int add(int n, int o) {return n + o;}

TEST(add, addsCorrectly) {

EXPECT\_EQ(4, add(2, 2));

}

Test fixture

*Using struct BankAccount example in example 1* – create a struct/class to test BankAccount.

Class BankAccountTest : public testing::Test { // google test class

Public:

BankAccount \*account;

BankAccountTest() { account = new BankAccount; }

~BankAccountTest() { delete account; }

}

TEST\_F(BankAccountTest, BankAccountStartsEmpty) { // arg 1 name same as test class

EXPECT\_EQ(0, account->balance);

// no need to create instance

}

Instead of creating account instance and deleting it in the class test constructor and destructor, you can override the SetUp() and TearDown() methods (note capitalisation), from the testing::Test class.

TEMPLATES AND THE STL (STANDARD TEMPLATE LIBRARY)

Template specialisation

When you invoke a template with a specific type, the compiler creates a specialisation of the template with that specific type.

Template<typename T>

T maxOf(const T a, const T b) { return (a > b ? a : b); }

Int main() {

Int a = 7;

Int b = 9;

Cout << “max is “ << maxOf(a, b);

}

When compiled, as integers where used as the type, the compiler creates a specialisation of the maxOf function which will look like:

Int maxOf(const int a, const int b) { return (a > b ? a : b);}

STL containers

**Vector**

(see advanced cpp part 1)

**List**

Essentiality a doubly linked list. Lists are sequence containers that allow non-contiguous memory allocation. Compared to a vector, lists have slow traversal, but once a position has been found, insertion and deletion are quick.

#include <list>

List <int> l1 = {1, 2, 3, 4, 5};

// similar methods used for vectors can be used with lists

// size(), front(), back(), etc…

In order to insert and erase elements, you need to use iteration:

List <int>::iterator it1 = l1.begin();

While((\*++it1 != 5) && (it1 != l1.end()));

if (it1 != l1.end()) {

cout << “insert 112 before 5” << endl;

l1.insert(it1, 112);

}

// print list

For (int & i : l1) cout << i << “ “;

// this for loop is a range, and is the same as:

For(int i = l1.begin(); i < l1.end(); ++i)

**Array**

A container for constant sized arrays.

#include <array>

Array<int, 5> a1 = {1, 2, 3, 4, 5};

a1.size(); // 5 – always the size constructed at

a1.at(1); // 2

// loop through and print array elements

int \*ip = a1.data();

for (size\_t i = 0; i < a1.size(); i++) {

cout << \*ip++ << endl;

}

**Map**

Key value pairs

#include <map>

*Map<keyType, valueType>*

Map<string, string> mapstr = {

{“name”, “Ali Issaee”},

{“placeOfBirth”: “Beverley”}

};

Cout << mapstr[“name”]; // Ali Issaee

Mapstr.at(“name”); // Ali Issaee

Mapstr.size(); // 2

Mapstr.insert({“key”, “value”}); // add new key/value pair

Maps do not allow for duplicates so it is good practice to check insertion has worked.

Auto rp = mapstr.insert({“name”, “Sha Issaee”});

If (rp.second) {

Cout << “success”;

} else {

Cout << “fail”; // will fail as name key exists

}

// to overwrite value

Mapstr[“name”] = “New value”;

C++ THREADS

A thread pf execution is a sequence of instructions that can be executed concurrently with other such sequences in multithreading environments, while sharing the same address space.

#include <thread>

Void threadFn() {

Cout << “Inside thread function”;

}

Int main() {

Thread t1(threadFn); // can also use {} syntax, which takes function as first arg, and function arguments, e.g. {threadFn, arg1, arg2, …}

Cout << “Test 1”;

T1.join();

Cout << “Test 2”;

Return 0;

}

// *output:* Test 1 Inside thread function Test 2

Thread::join(); - blocks the current thread, meaning it will not continue running, until the thread the join is called upon will finish.

LAMBDA FUNCTION/EXPRESSION

A lambda function, or expression, is an anonymous function with the ability to refer to identifiers outside of its own scope. Sometimes confused with a closure, although similar, a lambda is an anonymous function which creates a closure. A closure is an encapsulated block of code.

[*capture*](*args*) {*code*}

Capture – list of variables in outside scope to give access to. Wildcards (=, &) where = gives access to all variables in outside scope by value, and & gives access to all variables in outside scope by reference.

Auto getNum = []() -> int {return 10;};

Cout << getNum(); // 10

*Example 1:*

Auto addCurry = [](auto a) {

Return [a](auto b) {

Return [a,b](auto c) {

Return a +b + c;

};

};

};

Auto add1 = addCurry(1);

Auto add2 = add1(2);

Auto add3 = add2(3);

Cout << add3; // 6

*Example 2:*

#include <algorithm> // get access to for\_each

#include <vector>

Int main() {

Vector<int> list = {5, 10, 15, 20, 25};

For\_each(list.begin(), list.end(), [] (int & i) {

Cout << i << endl;

});

}

Passing function as parameter

Void test(void (\*fn)()) { // return type of passed function, pointer to function, params

fn();

}

// pass lambda as argument to test

Test([]() {

cout << “Test”}

);

*Example:*

Void test(void(\*f)(int)) {

f(1);

};

Void printNum(int x) {

Cout << x << endl;

}

Int main() {

Test(printN);

}

TEST

PART 1

1. Give an example (using the common methods), of:
   1. Smart pointer
   2. Shared pointer
   3. Weak pointer
2. What is the method used to count the number of copies a shared pointer has?
3. Give an example of operator overloading
4. Give an example of multithreading in c++
5. Create a lambda function for currying
6. Using the for\_each method, add a lambda function as the 3rd arg to iterate over a vector and double each element and print the value to the screen
7. Create a function, which takes another function as an argument

PART 2

1. Create a:
   1. Vector
   2. List
   3. Array
2. What is the difference between a vector, list and array and when would you use them?
3. Create a map and:
   1. Update an existing value
   2. Insert a new key/value pair ensuring to check previous existence
   3. 2 ways to print a value to the screen
   4. Find the size of the map
4. Using Google test, add a basic test to a basic function and run all tests
5. Using google test, add a test fixture for a class, ensuring to create and remove instances after each test, and test some methods in the class