**SHARED POINTER:**

// smart pointer with shared ownership

std::shared\_ptr<MyClass> s1 { new MyClass (40000, 1, 1) };

std::shared\_ptr<MyClass> s2 {s1};

// smart pointer with exclusive ownership

std::unique\_ptr<MyClass> u {new MyClass (30000, 1, 1) };

std::unique\_ptr<MyClass> u2 { std::move(u) };

**ARRAYS OF FUNCTION POINTERS:**

void error(std::string s) { cout "e: " << s; };

void warning(std::string s) { cout "w: " << s; };

void trace(std::string s) { cout "t: " << s; };

int main int argc, char \*argv[]) {

void(\*efct[3])(std::string)

= { &error, &warning, &trace };

log("some error", (efct[0]); }

int log(std::string txt, void(\*f)(std::string)) {

...

(\*f)(txt); }

**LAMDA EXPRESSIONS:**

template<typename Func>

int add(int i, Func func) { return func(i); }

int main() {

int k = 4;

cout << add(10, [=](int i) { return i + k; }); }

* **[=](...)** - captures all non-local variables by value
* **[&](...)** - captures all non-local variables by reference
* **[=,x,y](...)** - captures **x** and **y** by reference, all else by value
* **[&,x,y](...)** - captures **x** and **y** by value, all else by reference
* **[x,&y](...)** - captures **x** by value and **y** by reference
* **[this](...)** - captures **this** by value

**EXTERNAL LINKAGE:** must be defined outside of its translation unit

**COMPILATION PROCESS:**

* pre-processing stage , macro expansion
* compilation stage , translate source into binary
* linking, create single binary, finds location of definitions

**CONSTANT EXPRESSIONS:** expression whose return value can be

computed at compile-time

constexpr int fact(int n) {

return (n > 1) ? n\*fact(n - 1) : 1 }

int j = 0;

constexpr k = j + 1; //error, j is not a constant

constexpr k = j + 1; //error, j is not a constant

**POINTERS VS REFERENCES:**

**Pointers:** holds an address in memory, can be null,

Can be reassigned, casting required to assign one pointer type to another, doesn’t have to be assigned at compiled time

**References:** are alias, same address with the referred objects, cannot be null, cannot be reassigned, has to be assigned at compiled time

**LValues vs RValues:** lValues are value that can be on the left side of an expression, can be reassigned, has a identifier, must be initialized. rvalues are anything on the right side, temporary object or value

**ARRAYS:**

int id[10]; //stack allocation

int\* id = id[10]; //heap allocation

ENUMERATIONS:

enum class Colour { red, yellow, green };

class TrafficLight {

Colour currentColor;

public:

TrafficLight() { currentColor = Colour::red; }

void changeSignal(Colour c) { currentColor = c; } };

**CLASS RELATIONSHIPS:**

Association: loosely coupled service relationship – aggregation: “part-of”/“uses” relationship – composition: “has-a" relationship

**STATIC ASSERTTIONS:** The static\_assert() mechanism generates a custom compiler error message if a specified condition has not been met.

constexpr int N = 0;

constexpr int factorial(int i) {

return i > 1 ? i\*factorial(i - 1) : 1;}

int main() {

static\_assert(N > 0, "N <= 0");

static\_assert(N < 20, "N >= 20");

volatile int n{ N };

std::cout << n << "! = " << factorial(n) << std::endl;

system("pause");

return 0; }

**MOVE OPERATORS:**

/\*! copy constructor \*/

Notifications::Notifications(const Notifications& rhs) {

if (this != &rhs) {

m\_messages = nullptr;

\*this = rhs; } }

/\*! copy assignment operator \*/

Notifications&Notifications::operator=(const Notifications& rhs) {

//Check for self-assignment

if (this != &rhs) {

m\_size = rhs.m\_size;

delete[] m\_messages;

m\_messages = new Message[m\_size];

for (int i = 0; i < m\_size; i++) {

m\_messages[i] = rhs.m\_messages[i];}}

return \*this;}

/\*! move constructor \*/

Notifications::Notifications(Notifications&& rhs) {

m\_size = rhs.m\_size;

m\_messages = rhs.m\_messages;

rhs.m\_size = 0;

rhs.m\_messages = nullptr; }

/\*! move assignment operator \*/

Notifications&&Notifications::operator=(Notifications&& rhs) {

//transfer ownership of resources

m\_messages = rhs.m\_messages;

m\_size = rhs.m\_size;

//clean up resources

rhs.m\_size = 0;

rhs.m\_messages = nullptr;

return std::move(\*this); }

**MULTI-THREADINGS:**

**Future - Promise**

1. parent creates promise object (creates a shared location)

2. parent retrieves future object associated with promise

3. child sets promise value passed in

Promise – Future

1. parent creates promise object (creates a shared location)

2. parent retrieves future object associated with promise

3. child blocks until main sets promise

#include <future>

#include <thread>

void task(int i) { std::cout << std::this\_thread::get\_id() << std::endl; }

void taskPromise(std::promise<double>& p) {

p.set\_value(12.34); } // or set\_exception

void taskFuture(std::future<double>& f) {

double x = f.get();

std::cout << "Val = " << x << std::endl; }

int main() {

std::thread t1(task, 1);

std::thread t2(task, 2);

// Or using a lamda style

std::thread t3([=]() {std::cout << std::this\_thread::get\_id() << std::endl; });

t2.join();

t1.join();

//Using future

std::promise<double> p;

std::future<double> f = p.get\_future();

std::thread t(taskPromise, std::ref(p));

std::cout << "Val= " << f.get() << std::endl;

t.join();

//Using promise

std::promise<double> p1;

std::future<double> f1 = p.get\_future();

std::thread t1(taskFuture, std::ref(f));

p.set\_value(12.34);

t1.join(); }

**SEQUENTIAL CONTAINERS**

**array** - contiguous storage of fixed size

* **vector** - contiguous storage of variable size
* **deque** - non-contiguous storage of variable size, double-ended queue
* **forward\_list** - non-contiguous storage of variable size, singly linked list
* **list** - non-contiguous storage of variable size, doubly linked list

**#include <vector>**

**#include <deque>**

**size\_t capacity() const** - returns the current capacity of the current object

**bool empty() const** - returns true if the current object has no elements

**T& operator[](size\_t i)** - returns a reference to element i

**const T& operator[](size\_t i) const** - returns an unmodifiable reference to element i

**T& at(size\_t i)** - returns a reference to element i and checks bounds - throwing an exception

**T& front()** - returns a reference to the first element

**const T& front() const** - returns an unmodifiable reference to the first element

**T& back()** - returns a reference to the last element

**const T& back() const** - returns an unmodifiable reference to the last element

**void push\_front(const T& t)** - adds element t before the first element in the container (DEQUE)

**void pop\_back()** - removes the last element from the container

**void pop\_front()** - removes the first element from the container(DEQUE)

**void clear()** - removes all elements from the container

**#include <stack>**

**T& top()** - returns a ref to the top element of the stack

**const T& top() const** - returns an unmodifiable reference to the top element of the stack

**void push(const T& t)** - adds elem t to the top of the stack

**void pop()** - removes the top element from the stack

**#include <queue>**

**void push(const T& t)** - adds element t to the top of the queue

**void pop()** - removes the first element from the queue

**T& front()** - returns a reference to the first element of the queue

**const T& front() const** - returns an unmodifiable reference to the first element of the queue

**T& back()** - returns a reference to the last element of the queue

**const T& back() const** - returns an unmodifiable reference to the last element of the queue

**STL ALGORITHMS:**

#include <algorithm>

#include <iostream>

int main() {

std::array<int, 11> a = { 1, 12, 4, 5, 8, 9, 12, 13,

16, 18, 12 };

int n = std::count(a.begin(), a.end(), 12);

std::cout << "12 occurs " << n << " times" << std::endl; //12 occurs 3 times

int a[] = { 1, 2, 4, 5, 8, 9, 12, 13, 16, 18, 22 };

int n = std::count\_if(a, a + 11, [](int i)

{ return !(i & 1); });

std::cout << "Even Numbers = " << n << std::endl; //Even Numbers = 7

std::vector<double> v(4, 10.34);

std::vector<double> c(4, 20.68);

std::copy(v.begin(), v.begin() + 2, c.begin() + 1);

for (auto i : c)

std::cout << i << std::endl; //20.68 10.34 10.34 20.68

std::vector<int> v = { 1, 2, 4, 5, 7, 8, 10, 13, 17, 21, 43 };

std::vector<int> c(15);

std::copy\_if(v.begin(), v.begin() + 10, c.begin(),

[](int i) -> bool { return i % 2; });

for (auto i : c)

if (i) std::cout << i << std::endl; //1 5 7 13 17 21

std::vector<int> v = { 1, 2, 4, 5, 7, 8, 10, 13, 17, 21, 43 };

std::vector<int> c(11);

std::transform(v.begin(), v.end(), c.begin(),

[](int i) { return 3 \* i; });

for (auto i : c)

std::cout << i << std::endl; //3 6 12 15 21 24 30 ....

std::vector<int> a = { 1, 2, 4, 5, 7, 8, 10, 13, 17, 21, 43 };

std::vector<int> b = { 2, 1, 0, 1, 2, 3, 16, 23, 21, 17, 32 };

std::vector<int> c(11);

std::transform(a.begin(), a.end(), b.begin(), c.begin(),

std::plus<int>());

for (auto i : c)

std::cout << i << std::endl; //3 3 4 6 9 11 26 36 ...

int a[] = { 3, 2, 4, 1 };

std::sort(a, &a[4], [](int i, int j) { return i > j; });

for (int i : a)

std::cout << i << std::endl; //4 3 2 1

int a[] = { 3, 2, 4, 1 }, s;

s = std::accumulate(a, &a[4], (int)0);

std::cout << "sum = " << s << std::endl; //sum = 10

s = std::accumulate(a, &a[4], (int)0,

[](int x, int y) { return x + 2 \* y; });

std::cout << "2 \* sum = " << s << std::endl; //2 \* sum = 20 }

void regression(T& slope, T& y\_intercept) const {

T sumOfXY = 0, sumOfX = 0, sumOfY = 0, sumOfX2 = 0;

sumOfX = std::accumulate(begin(m\_dataX), end(m\_dataX), sumOfX);

sumOfY = std::accumulate(begin(m\_dataY), end(m\_dataY), sumOfY);

sumOfX2 = std::accumulate(begin(m\_dataX), end(m\_dataX), sumOfX2, [](T total, T value) {return total += value\*value; });

std::vector <T> dataXY(m\_numRecs);

std::transform(begin(m\_dataX), end(m\_dataX), begin(m\_dataY), begin(dataXY), std::multiplies<float>());

sumOfXY = std::accumulate(begin(dataXY), end(dataXY), sumOfXY); }