**Vu Pham StudentID: 129908174**

**Inheritance** is the relationship between classes where one class inherits the entire structure of another class **Polymorphism** refers to the multiplicity of behaviors that an identifier exhibits in an inheritance hierarchy. **Modularity:** Object-oriented solutions lend themselves naturally to modular partitioning of source code.  Modules define classes and their implementations **.** A **block** is a delimited set of program instructions. local scope - the name has been declared within a function/ class scope - the name has been declared as a member of a class/ namespace scope - the name has been declared as a member of a named block/ global scope - the name has not been declared in any one of the above scopes

**Linkage** The same name may refer to the same entity across different scopes.  The name's linkage identifies its connectivity.  A name has a linkage if it can refer to an identical name declared in another scope

**Statically allocated memory**is memory of a size known at compile-time.  The compiler inserts the code to allocate this memory on the stack or data segment.  **Dynamically allocated memory**is memory that may be of variable size and is allocated at run-time.  this memory to the executable in the size requested at that time.  This memory is allocated on the heap, is managed by the freestore manager and needs to be returned to the manager before it can be reused by the executable for some other purpose.

**Definition**is a declaration that associates a meaning with a name.  A declaration may or may not be a definition.  **The scope** is the region of the program throughout which the name is valid; that is, the region where the entity associated with that name is visible. A name declared within the scope of an identical name **shadows** the entity that has the broader scope.  Avoiding the use of identical names improves readability. A **namespace**is a named scope.  Any name declared within a namespace is local to that namespace and invisible outside the scope.

An **lvalue** declaration refers to an accessible region of memory. An **rvalue** declaration refers to: an object near the end of its lifetime

a temporary object or subobject a value not associated with an object

signed char 1 BYTE short int 2 int 4 long int 4 long long int 8 BOOL 1 CHAR

|  |  |
| --- | --- |
| // Generic Pointer Type  // void.cpp  #include <iostream>  int main() { int i; void\* v = &i; int\* j;  j = static\_cast<int\*>(v); // OK - j now holds the address of i  std::cout << &i << std::endl;  std::cout << j << std::endl;} |  |

typedef unsigned long long int ullint;  ullint\* p;

A **generic** pointer type is a pointer type that is not associated with any particular type.  The keyword void identifies a generic pointer type void\* p; //

The **generic** pointer type is distinct from any other pointer type defined in an application:  a pointer of generic type can hold the address of a variable or object without holding its type information.

Converting a pointer of any type into one of generic type and vice versa does not lose its address:

Converting from one pointer type to another requires an explicit cast:

|  |  |
| --- | --- |
| int\* i;  char\* c;  i = c; // ERROR - Incompatible  // Different Pointer Types | int\* i;  char\* c;  i = static\_cast<int\*>(  static\_cast<void\*>(c)); // OK |

void displayVolume(const Shape\* shape) {

if (shape)

std::cout << shape->volume() << std::endl;

else

std::cout << "error" << std::endl;

}

abstract types - their representation is not part of their definition and is unknown

concrete types - their representation is part of their definition and is known

Assosiation: has no owner ship&lifetime dependency

Aggrestion: one own but no depen

Compotion: one own inst and lt of child depen on lt of own insta

Primary postfix prefix unary binary ternary

Cpconstr Student::Student(const Student& src) : Person(src) {

no = src.no;

ng = src.ng;

if (src.grade != nullptr && ng > 0) {

grade = new float[ng];

for (int i = 0; i < ng; i++)

grade[i] = src.grade[i]; }

else grade = nullptr }

Student& Student::operator=(const Student& src) {

if (this != &src) {

// Base class assignment

// 1 - functional expression

// Person::operator=(src);

// 2 - assignment expression

Person& person = \*this; // only copies address

person = src; // call base class operator

delete [] grade;

no = src.no;

ng = src.ng;

if (src.ng > 0) {

grade = new float[ng];

for (int i = 0; i < ng; i++)

grade[i] = src.grade[i] }

elsegrade = nullptr; } return \*this; }

mplate <typename T>

void swap(T& a, T& b) {

T c;

c = a;

a = b;

b = c;

}\

bool Cube::operator==(const Shape& s) const {

return len == s.len; // ERROR because len is not a member of Shape  }

bool Cube::operator==(const Cube& s) const {

return len == s.len; // ERROR because Cube is not concrete  }

Shape\* Cube::clone() const {

return new Cube(\*this); }

template <typename T>

class Array {

T a[50];

unsigned n;

T dummy;

public:

Array() : n{0u}, dummy{0} {}

T& operator[](unsigned i) {

return i < 50u ? a[i] : dummy;

}

};

void hexDump(void\*, int);

// Dump the first n bytes to the address a

void hexDump(void\* a, int n) {

unsigned char\* c =

static\_cast<unsigned char\*>(a);

std::cout.fill('0'); // zero fill

std::cout << std::hex; // hexadecimal output

for (int i = 0; i < n; i++)

std::cout << std::setw(2)

<< (int)c[i] << ' ';

std::cout.fill(' '); // std::cout<<std::dec; // decimal output }

Array(const Array& src) { \*this = src; }

Array& operator=(const Array& src) {

if (this != &src) {

delete [] a;

a = new int[src.n];

for (unsigned i = 0u; i < src.n; ++i)

a[i] = src.a[i];

n = src.n;

}

return \*this;

}

Array(Array&& src) { \*this = std::move(src); }

Array& operator=(Array&& src) {

if (this != &src) {

delete [] a;

a = src.a;

n = src.n;

dummy = src.dummy;

src.a = nullptr;

src.n = 0u;

src.dummy = 0;

}

return \*this;

} b = std::move(a);

KVList(KVList&& other) {

if (this != &other) {

m\_list = other.m\_list;

numofele = other.numofele;

currentele = other.currentele;

other.m\_list = nullptr;

other.numofele = 0; other.currentele = 0;}}

static unsigned noHorses;

unsigned Horse::noHorses = 0;

An aggrestion is a composition that does not manage the creation or destruction of the obj that it uses

void reverse (double\* a, int size){

for (double\* end = a+size-1; a < end; ++a, --end){

double temp = \*a;

\*a = \*end;

\*end = temp; }

cout << size[] << endl;}

template<typename F>

void display(std::ostream& os, F f) const { os << std::left << std::setw(fieldWidth) << m\_key << " " << ":" << " " << std::right <<

The thread class defines an object that represents a single thread of execution. A thread object is either joinable or not-joinable. A joinable object represents an actual thread of execution with a unique thread id. A non-joinable object represents a potential thread of execution.

void divide(double a[], int i, int n, double divisor) {

if(i < 0 || i >= n)

throw "Outside bounds";

else if(i == n / 2)

throw i;

else if(divisor == 0)

throw divisor;

else

a[i] = i / divisor;

}

int main() {

bool keepdividing = true;

double a[] = {1.1,2.2,3.3,4.4,5.5,6.6}, divisor;

int i, n = sizeof a / sizeof a[0];

do {try { std::cout << "Index: "; std::cin >> i;

std::cout << "Divisor: ";std::cin >> divisordivide(a, i, n, divisor);

std::cout << "a[i] = " << a[i] << std::endl;

} catch(const char\* msg) {

std::cout << msg << std::endl;

keepdividing = false;

} catch(int& value) {

std::cout << "Index is " << value << std::endl;

std::cout << "a[i] = " << a[i] << std::endl;

} catch(...) {

std::cout << "Zero Division!" << std::endl;

std::cout << "a[i] = " << a[i] << std::endl;

} } while (keepdividing) }

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

list() - default constructor - creates an empty container

list(int n) - creates a container with n elements

list(int n, const T& l) - creates a container with n elements, each initialized to value l

list(const list& l) - copies the contents of v into the current object

~list() - destroys the container

list& operator=(const list& l) - copies the contents of l into the current object

size\_t size() const - returns the number of elements in the current object

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

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\_back(const T& t) - adds element t after the last element in the container

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

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

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

iterator insert(iterator position, const T& t) - adds element T at the iterator position

iterator erase(iterator position, const T& t) - remove element T at the iterator position

void clear() - removes all elements from the container

stack - last in, first out (LIFO) context

queue - first in, first out (FIFO) context

priority\_queue - first element is always the greatest

|  |  |
| --- | --- |
|  |  |

std::setw(fieldWidth) << m\_val << std::setw(fieldWidth) << f(m\_val) << std::endl; }

The equivalent of static storage duration for a set of concurrently executing threads is **thread\_local** storage duration.

**Resource Acquisition** Is Initialization: specifies resource acquisition occurs at initialization time

**List** stores elements at non contiguous memory Whereas, **vector** stores elements at contiguous memory locations like an array i.e.

list<int> L({ 2,3,5 });

list<int>::iterator a;

for (a = L.begin(); a != L.end(); a++) {

cout << \*a << " ";

}

advance(a,3);

L.insert(a, 4);

L.push\_back (6);

L.push\_front (1);

cout <<endl;

for (a = L.begin(); a != L.end(); a++) {

cout << \*a << " ";

}

cout <<endl;

A **copy** constructor is used to initialize a previously uninitialized object from some other object's data.An **assignment** operator is used to replace the data of a previously initialized object with some other object's data.In the **move** constructor, assign the class data members from the source object to the object that is being constructed:In the **move** assignment operator, add a conditional statement that performs no operation if you try to assign the object to itself.

A promise object creates or acquires a shared state in which it can store a value.The **promise** class template provides a simple set\_value() counterpart to the get() member function of the future class template A provider object complements a future object. One of the following templates can instantiate a provider object:std::promise class template//std::packaged\_task class template//std::async() function template

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

for (int i = 0; i < argc; i++) { cout << argv[i][i] << " ";}cout << endl;

return 0;}

if the program is executed as follows:

prg one two three <ENTER>]

p n o e

vector<int> v = {1, 3, 5, 7, 9};for (auto x : v)

cout << x << ' ';

std::thread t1(task);// spawn another child threa

std::thread t2(task);// synchronize - IMPORTANT!

t2.join();t1.join();

template <typename T>const T& Deque<T>::operator[](size\_t index)const {index %= size();Qnode<T>\* cur = m\_head;

for (size\_t i = 0; i < index; i++, cur = cur->m\_next);

return cur->m\_data;}