ADVANCED CPP

CLASS VS STRUCTS

The only difference between a class and a struct are; in a struct, members default to public, whereas in a class, members default to private, if not specified.

Struct Test {

Int age; // public

Void printAge() {cout << age << endl;} // structs can also use methods

Private:

Int dob;

}

Class Test {

Int age; // private

Public:

Int dob;

}

C++ classes are based off C structs. You can use inheritance with structs, call constructs/destructors, etc…

As structs use public inheritance, you do not need to use the *public* keyword when inheriting:

Struct Child : Parent {}

Class Child : public Parent {}

It is best practice to use structs when only properties are needed, and classes when methods are also needed.

NAMESPACES

As more classes are created, name conflicts become more possible. This is where namespaces come in.

Namespace me {

Int age = 28;

Class MyClass {};

};

Std::out << Me::age << std::endl;

Me::MyClass obj;

MULTIPLE CONSTRUCTORS

It is possible to have multiple constructors which are called based on the number of parameters given when instantiating.

Class Test {

String name;

Int age;

Public:

Test() { cout<< “constructed!” << endl; };

Test(string n, int a) :name(n), age(a) {}

};

Int main() {

Test test; // calls the first constructor with no args

Test another(“Ali”, 28); // calls the second constructor

}

ALLOCATING OBJECT MEMORY

The *new* and *delete* operators are used to allocate and de-allocate memory. This is only needed when the object is needed beyond the lifetime of a function or a block, i.e. you want to use it in different scopes.

Std::bad\_alloc is one of the many built in c++ exception classes which throws an error when the allocation of memory fails.

Int main() {

Try {

MyClass \*obj = new MyClass;

delete MyClass;

} catch (std::bad\_alloc & ba) {

Std::cerr << ba.what() << endl; // what() is from base exception class

}

}

ARRAY OF OBJECTS

MyClass \*obj = new MyClass[5]; // 5 objects created

Obj[1].methodName; // second object calls method

Delete [] obj; // must use this syntax when deleting

FRIENDSHIP

Sometimes you want to grant access to a classes private members to an object (or group of objects), this is accomplished with a friend declaration.

Class Parent {

Private:

Void test() { cout << “Test” << endl;};

Friend class Child; // give access to Child class

}

Class Child : public Test {

Public:

Child() {Parent::test();} // private method called

}

// note that objects of child will not be able to call the private method of Parent directly, only the class itself. Unless friendships are made between the individual objects.

Generally friendship is not a good idea as it goes against the rules of encapsulation.

MULTIPLE INHERITANCE

Class Child : public Grandparent, public Parent {}

Objects are constructed in order of inheritance, the class being instantiated constructed last.

In the example above, it will be; Grandparent > Parent > child.

STRING LIBRARY

#include <string>

Concatenate

String str1 = “Hello”;

String str2 = “World”;

String str2 = str1 + str2; // + operand

Length

String str1 = “Ali Issaee”;

Cout << str1.length() << endl; //10

Append

String str1 = “Hello”;

Cout << str1.append(“ World”) << endl; // Hello World

Substr

String str1 = “Hello”;

*Str1.substr(start, end);*

Str1.substr(2); // llo

Str1.substr(2, 1,) // l

Compare

*String.compare(stringToCompare)*

Returns 0 if equal, -1 if not equal

Find index

String str1 = “Ali”;

Int index = str1.find(“i”); // 2

TYPEID OPERATOR

Class Test {}

Class Test1 : public Test {}

Test test

Test1 test1;

If (typeid(test) == typeid(test1)) {} // these 2 are not the same (Test and Test1)

Int num = 1;

Int num1 = 2;

If (typeid(num) == type(num1)) {} // both are the same type (int)

TEMPLATES

Templates are special functions that can operate with generic types. Similar to function overloading but saves you having to create a new function with just different data types based on the parameters…

You cannot define a template within a block, i.e. must be in global scope.

Template <typename T> // where T is the name

T half(T n) { return n / 2; };

Int main() {

Int n = 9;

Double n1 = 9;

Cout << half(n) << endl; // 4 (as int)

Cout << half(n1) << endl; // 4.5 (as double)

}

Template <typename MyType>

Class Test {

MyType money;

Public:

MyType getMoney() { return money; }

Void addMoney(MyType m) {money = m;}

};

Int main() {

// specify type when instantiating in <>

Test<int> test; // <dateType> to replace *template* *type*

}

ERROR HANDLING

Cout << errno << endl; // if 0 then no errors

Remove(“foo.bar”); // remove non-existent file

Cout << errno << endl; // 2 (error number for non-existent file/dir)

Perror(“cannot erase file”); // cannot erase file: no such file or directory

Perror() > outputs message to standard error stream.

VECTOR

Vectors are similar to arrays. Vector occupy much more memory in exchange for the ability to manage storage and grow dynamically.

#include <vector>

Int main() {

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

Cout << list.size() << endl; // 5

Cout << list.front() << endl; // 5 – first element

Cout << list.back() << endl; // 25 – last element

Cout << list.at(1) << endl; // 10

Cout << list[1] << endl; // 10

List.push\_back(30); // push another element to end

List.pop\_back(); // remove last element

}

EXCEPTIONS

#include <exception>

Try {

Throw exception(); // does not take arg

} catch (exception & e) {

Cerr << e.what() << endl; // ‘std::exception’ (base class message)

}

To make exception handling more effective, you can inherit from the base exception class to make custom exceptions.

Class E : public exception {

Private:

Const char \* msg;

E();

Public:

E (const char \*s) : message(s) {} // \*\*\*

Const char \*what() { return msg; }

};

Throw E(“This is an error”);

We are now able to provide a bunch of exception objects:

Const E e\_ouch(“ouch!”);

Const E e\_bad(“bad!”);

Const E e\_worse(“worse!!!”);

Throw e\_ouch;

\*\*\*The *throw()* in the function signature: *E(const char \*s)* ***throw()*** *: message(s) {}* indicates that the function cannot throw any exceptions! If supplied with an argument of an exception then this means this method can throw an exception of this type. They are generally bad practice and you should avoid them as they are weakly enforced and therefore don’t actually accomplish much.\*\*\*

QUIZ

PART 1

1. What is the difference between a class and a struct?
2. When would you use a struct over a class?
3. Give an example of a namespace
4. Create a class with multiple constructors
5. When would you use the *new* and *delete* keywords?
6. What exception should you catch when creating objects with *new*
7. How do you create/destroy an array of objects?

PART 2

1. Give an example of the *friend* feature
2. Give example of multiple inheritance
3. Using the <string> library:
   1. Concatenate 2 strings together
   2. Append to a string
   3. Find the length of a string
   4. Create a substring from a string
   5. Compare 2 strings
   6. Find the index of a letter in a string
4. What operator is used to compare types between 2+ variables
5. What is a template – give example

PART 2

1. What built-in variable can you use to determine whether an error has happened?
2. What built-in function can you call to view the output of an error – give example of trying to remove a non-existent file
3. What is a vector – give example ensuring to:
   1. Get the size of the vector
   2. Getting the first element in a vector
   3. Getting the last element in a vector
   4. Pushing on top of a vector
   5. Removing the last element from a vector
   6. Finding a specific element in a vector based on index (2 ways)
4. Extend the base exception class and create objects from this class and give example throwing and catching these custom objects/exceptions